CHANNEL CODING FOR THE RELAY CHANNEL

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAIʻI IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

May 2006

By
Danny Yee

Thesis Committee

Anders Høst-Madsen, Chair
N. Thomas Gaarder
Marc Fossorier
We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Master of Science in Electrical Engineering.

THESIS COMMITTEE

[Signatures]

[Names]

Chair
Acknowledgement

I would like to thank my advisor Dr. Anders Høst-Madsen for always helping me with whatever questions I had, as well as giving me a subject matter for my thesis. I would especially like to thank Yige Wang for the LDPC decoder and generator matrices and helping me with all of the questions I had about the code. I would also like to thank Dr. Marc Fossorier for help with erasure codes, irregular codes, and serial concatenation. Finally, I would like to thank Dr. N. Thomas Gaarder for help with some theoretical issues as well as other miscellaneous questions.
Abstract

The relay channel consists of an input, a relay, and a destination. The source transmits $x_{11}$. A relay, which receives that as $y_2$, will decode it and transmit $x_2$ as a function of $x_{11}$. Finally, the destination receives $x_{11}$, $x_{12}$, and $x_2$ as $y_3$, which it decodes. If done correctly, $x_{12}$ and $x_2$ will sum coherently and the resulting $y_3$ will be superior to simply sending the message straight from source to destination at 2x the power.

In this thesis, two cases are considered. The cases are full duplex relaying and half duplex relaying. Full duplex relaying is where the relay can send and receive at the same time. This is implemented using backward decoding, a form of superposition block Markov encoding, as well as QPSK signaling.

The half duplex case is where the relay cannot send and receive on the same channel at the same time. This is implemented using TDD. Code shortening is used to achieve the relaying objective.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>A diagram of a relay channel</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2</td>
<td>The two cuts that are used to prove equation (4)</td>
<td>7</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Superposition Block Markov Encoding</td>
<td>14</td>
</tr>
<tr>
<td>Figure 4</td>
<td>The flow of information through the two paths</td>
<td>15</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Comparison of BER vs. $E_b/N_0$ for Relay and Direct transmission</td>
<td>21</td>
</tr>
<tr>
<td>Figure 6</td>
<td>BER vs. $E_b/N_0$ for the channel where both are operating at peak</td>
<td>26</td>
</tr>
<tr>
<td>Figure 7</td>
<td>BER vs. $\beta$, Trying to find the best value</td>
<td>30</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Another graph trying to find the optimum value of $\beta$</td>
<td>31</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Example of serial concatenation</td>
<td>35</td>
</tr>
<tr>
<td>Figure 10</td>
<td>The theoretical decoder that could not be implemented due to time constraints</td>
<td>36</td>
</tr>
<tr>
<td>Figure 11</td>
<td>An example of an irregular code</td>
<td>40</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Taking every other column of a parity check matrix</td>
<td>41</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Block Error Rate vs. $E_b/N_0$ for the first TDD case</td>
<td>47</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Plot for second case where $\kappa = 0$</td>
<td>50</td>
</tr>
</tbody>
</table>
# Table of Contents

Acknowledgement .............................................................................................................. iii

Abstract .............................................................................................................................. iv

List of Figures ..................................................................................................................... v

Table of Contents ............................................................................................................... vi

1. Introduction ................................................................................................................ 1

2. System Model........................................................................................................... 4

  2.1 Theoretical Capacity for Full Duplex................................................................. 6

    2.1.1 Theoretical Calculation of $\beta$................................................................. 7

  2.2 Theoretical Capacity for Half Duplex................................................................. 9

    2.2.1 Theoretical calculation for $\kappa$............................................................... 10

3. Full Duplex Relaying Implementation ..................................................................... 13

  3.1 Block Markov Encoding and Backward Decoding............................................. 13

  3.2 Implementation Details ....................................................................................... 16

  3.3 Results ............................................................................................................... 19

    3.3.1 First Simulation ......................................................................................... 21

    3.3.2 Second Simulation .................................................................................... 23

    3.3.3 Third Simulation ...................................................................................... 27

    3.3.4 Determination of $\beta$............................................................................. 29

  3.4 Summary ........................................................................................................... 32

4. Half Duplex Relaying Implementation .................................................................... 34

  4.1 Basic Idea ........................................................................................................ 34
1. Introduction

Upon hearing the term relay, one might be tempted to think of the track & field relay, where a runner runs 100m and passes a baton to the next runner, who runs for another 100m. In some ways, the relay channel works similar to this. On one end, there is a source, on the other end there is a destination, and in between there is a relay. The source sends some information, which the relay picks up. Then the relay helps the source send some additional information to the destination. This system of communication, which was first studied by van der Muellen [14] has a wide range of uses, from cellular networks, to ad-hoc networks, to sensor networks. The advantages of these networks are increased capacity (compared to direct transmission) and resistance to fading, shadowing, and path loss [15].

Although there are a lot of papers dealing with the theory and theoretical capacity of the relay channel, there are only a few papers that deal with the implementation of a relay channel [16, 17]. This thesis deals with implementations of the relay channel using LDPC codes. LDPC codes were discovered by Gallager [18] and rediscovered recently by MacKay [19]. They perform very well, in some cases coming within 1 dB of the theoretical capacity.

The general idea of the relay channel is as follows: First, assume that the relay is "closer" to the source than the destination is, and is "closer" to the destination than the source is. The word "closer" is actually referring to the attenuation, or channel gain between source, relay, and destination, which is usually, but not exclusively due to the distance between the three nodes. Now the source transmits a message with enough
power that the relay can decode. Note that both the relay and the destination receive this message. Naturally the relay can decode the message, but the destination, since it is farther away, generally cannot. The relay then sends resolution information to the destination coherently with the source, which combines what it received in the first half with what it receives later to decode the message. The combination of the source and relay sending the common information coherently makes them equivalent to a two-node antenna array, so the advantages of an antenna array apply here.

![Figure 1: A diagram of a relay channel](image)

The challenge in channel coding comes from the fact that none of the current schemes truly extract the potential benefits of cooperating nodes in many cases. In particular, what is called multihop in this thesis does not make use of source-relay cooperation at all. This thesis considers two possible situations: The first situation is when the relay is capable of full duplex, meaning it can send and receive at the same time. The second situation is half-duplex, where it must spend part of the time receiving, and part of the time sending. Even though they are very similar in what they try to do, each situation requires very different approaches.
The full duplex relay method makes use of superposition block Markov encoding (SBME) with backward decoding. Three different situations are considered. One case where the direct transmission should clearly be better than the relay channel method, and two methods where the relay channel method should definitely be better than direct transmission.

The half duplex relay method will not require any sort of block Markov encoding. It will make use of what is known as code shortening for transmission. Two cases will be considered. One case is a typical situation where the relay is situated somewhere between the source and destination. The other case is also one where the relay is between the source and destination, except in this case synchronization between source and relay is not necessary.
2. System Model

This thesis considers the Gaussian relay channel. The Gaussian relay channel is given by Gaussian noise $Z_2$ at the relay and Gaussian noise $Z_3$ at the destination. Each noise is an independent zero mean Gaussian random variable with variance $N$. In other words, it is white Gaussian noise. As a simplification, assume that the noise variance of $Z_2$ and $Z_3$ are the same. Thus the received signals at the relay node and the destination node are given by

$$Y_2 = c_{21}X_1 + Z_2$$
$$Y_3 = c_{31}X_1 + c_{32}X_2 + Z_3$$

(1)

$Z_2$ and $Z_3$ are set to unit variances (after normalization). In this thesis, constant channel gains will be used, which means that $c_{21}$, $c_{31}$, and $c_{32}$ are all non-random.

This thesis considers a technique for full-duplex transmission, and one for half-duplex transmission. Full-duplex transmission means that the relay can receive and transmit at the same time on the same frequency channel. In general, this is difficult to achieve if the receiving and transmitting powers are very different. In the half-duplex case, either time division duplex (TDD) or frequency division duplex (FDD) may be used to fulfill the requirements.

For the TDD method, the time is divided up into two periods that may or may not be equal. The first period shall be called the relay receive period, and the second period shall be called the relay transmit period. In the relay receive period, the sources sends the first part of the message, and the relay and destination both receive that message. The relay is not transmitting at this time. The destination simply stores what it receives, while
the relay receives the encoded message, decodes it and uses the decoded message to generate some resolution information.

The next period will be called the relay transmit period. In this period, the relay and the source may cooperate to transmit the resolution information coherently (in practice this is extremely difficult). The destination receives the resolution information, and uses that in combination with what it received in the first period to resolve the message. In some cases of \( c_{21}, c_{31}, \text{ and } c_{32} \), the best idea would be for the source not to broadcast at the same time as the relay. In that case, synchronization is not necessary.

In comparison, the FDD method will have split up the total bandwidth into two parts. In one part the relay will only receive messages, and in the other part the relay will only send messages. The source transmits over the whole spectrum, and the destination will listen over the whole spectrum. The source must also send coherently with the relay over the part of the spectrum over which they both transmit in common.

In the end, TDD was chosen as the transmission method. The reason TDD was chosen is given in the paper [2], which states: “Clearly, from an information-theoretic point of view, the TD mode and the FD mode are equivalent for the fixed channel gain case. In fading channels, however, the TD mode has an advantage over the FD mode because \( \alpha \) can be adjusted to the instantaneous channel conditions, whereas \( \alpha \) is usually fixed in the FD mode.”

Note that the problem of synchronization is not dealt with in this thesis. In practice, it is extremely difficult to synchronize two separate crystal oscillators [4]. In considering a model that includes a phase difference, one may use the asynchronous
channel model in which there is a random phase offset $\theta[i]$ between the source and the relay. The resulting signal at the destination is [2]

$$Y_3 = c_3, X_1 + e^{j\theta}c_{32}X_2 + Z_3$$

(2)

Note that $\Theta$ is random and ergodic, and uniformly distributed between $[-\pi, +\pi)$.

Although phase differences will not be dealt with in the simulation, note that they do make a large difference in physical implementation.

### 2.1 Theoretical Capacity for Full Duplex

An upper bound on the theoretical capacity of the relay channel with only one relay in-between is

$$C^+ \leq \max_{P(\eta_1,\eta_2)} \min \left\{ I(X_1, X_2; Y_3), I(X_1; Y_3 | X_2) \right\}$$

(3)

The formal proof of this is given in [1, Theorem 14.10.1], which is a result of a max flow min cut algorithm. One can picture a sort of generalization of the proof as follows: The first term in the braces comes from the "cut" going thru $1 \rightarrow 3$ and $2 \rightarrow 3$. The second term in the braces comes from the "cut" going thru $1 \rightarrow 2$ and $1 \rightarrow 3$. The capacities of each term results in the equation

$$C^+ = \max_{0 < \beta < 1} \min \left\{ \frac{1}{2} \log \left( 1 + (1 - \beta)P_1 \left( c_{31}^2 + c_{11}^2 \right) \right), \frac{1}{2} \log \left( 1 + c_{31}^2 P_1 + c_{32}^2 P_2 + 2\sqrt{\beta c_{31}^2 c_{32}^2 P_1 P_2} \right) \right\}$$

(4)

$\beta$ is the fraction of power $P_1$ sent directly to the destination only, and $(1-\beta)$ is the fraction of $P_1$ sent to both relay and destination. Note that the noise variance for each part of the equation is 1. This is the upper bound on the capacity, and is not a realistically
achievable bound, because it assumes that the relay and destination cooperate to decode, and the relay and source cooperate to send all of the message, rather than just part of the message.

If the relay node decodes and then re-encodes the message, then the capacity becomes [3, Theorem 1]

\[
C = \max_{p(x_1,x_2)} \min \left\{ I(X_1;Y_2|X_2), I(X_1,X_2;Y_3) \right\}
\]

\[
C = \max_{0 \leq \beta \leq 1} \min \left\{ \frac{1}{2} \log \left( 1 + (1 - \beta) c_2^2 P_1 \right), \frac{1}{2} \log \left( 1 + c_1^2 P_1 + c_2^2 P_2 + 2\sqrt{\beta c_1^2 c_2^2 P_1 P_2} \right) \right\}
\]

In the simulations, the decode – re-encode method is used, so (6) is applicable to the situation. The proof of this bound is given later. However, it is of the same form as the physically degraded channel in [3]. A justification for this can be seen in [20].

\section{2.1.1 Theoretical Calculation of $\beta$}

In order to simulate the full duplex relay channel, a formula to find the value of $\beta$ must first be derived. From the formula (6), it is seen that both terms within the braces
are convex. Thus, using that formula and the convexity, one could calculate a theoretical value for $\beta$ by setting the first term in the braces equal to the second term. However, that simple formula fails to take into account the requirement that $0 \leq \beta \leq 1$ (\& real) and $0 \leq \sqrt{\beta} \leq 1$ (\& real) are restrictions on what values $\beta$ can take. However, this can be taken care of after doing the calculations. Using simple algebraic manipulation and the quadratic formula, the theoretical value for $\sqrt{\beta}$ turns out to be

$$\sqrt{\beta} = \frac{-c_3c_{32}\sqrt{P_2} + \sqrt{c_3^2c_{32}^2P_2 - c_{21}^2(c_2^2P_2 - c_{21}^2P_1 + c_3^2P_1)}}{c_{21}\sqrt{P_1}}$$

(7)

This is only valid for the cases where $\sqrt{\beta}$ is neither negative nor complex.

In order to determine what to do when $\sqrt{\beta}$ is negative, it is helpful to consider some numbers when it solves to exactly 0 and when it is slightly negative. When the constants are declared in (7) so that $\sqrt{\beta} = 0$, then the power of the received signal at the relay is exactly equal to the received power at the destination, as expected. In this case, when $\beta = 0$, all of the power from the source is being broadcast to the relay and the destination. However, when $\sqrt{\beta}$ goes slightly negative, then $\beta$ becomes a positive number. Since the formula in (6) assumes that $\sqrt{\beta}$ is a positive number, this leads to them no longer being equal. Thus, the method winds up allocating some power for the source $\rightarrow$ destination only signal, even though the source $\rightarrow$ relay path is the limiting path. The source $\rightarrow$ destination only signal is basically the part of the message that the source will be sending to the destination cooperatively with the relay.
If the value of $\sqrt{\beta}$ turns out to be negative, there is nothing one can do to make the received powers at the relay and destination equal to each other. In the case where it is negative, however, the first term in the braces, the source $\rightarrow$ relay path, is the limiting path. Thus if it is negative, then the best that can be done is to set $\sqrt{\beta} = 0$. This also takes care of the complex cases, because when $\sqrt{\beta}$ is complex, it is also negative for all of the ranges of $c_{xy}$ that are realizable. When $\sqrt{\beta}$ is set to 0 instead of going negative or complex, then the source $\rightarrow$ relay is still the term that is limiting capacity, but the source is allocating all of its power through that path, and that is the best that can be done under these circumstances.

2.2 Theoretical Capacity for Half Duplex

The capacity of the TDD relay channel is upper bounded by \[13, \text{Section 4}] \[2, \text{Appendix A}]

\[
C^* = \max_{0 \leq \alpha \leq 1, 0 \leq \kappa \leq 1} \min \{C_1^*, C_2^*\} \tag{8}
\]

with

\[
C_1^* = \alpha \frac{1}{2} \log \left( 1 + (c_{x1}^2 + c_{y1}^2)(1 - \kappa) P_1 \right) + (1 - \alpha) \frac{1}{2} \log \left( 1 + (1 - \beta) c_{x1}^2 \kappa P_1 \right) \tag{9}
\]

\[
C_2^* = \alpha \frac{1}{2} \log \left( 1 + c_{x1}^2 (1 - \kappa) P_1 \right) + (1 - \alpha) \frac{1}{2} \log \left( 1 + c_{x1}^2 \kappa P_1 + c_{x2}^2 P_2 + 2\beta c_{x1}^2 c_{x2}^2 \kappa P_1 P_2 \right) \tag{10}
\]

In these equations $\alpha$ is the fraction of time spent in the relay receive period, and $(1 - \alpha)$ is the fraction of time spent in the relay transmit period. Theoretically, this $\alpha$ could be changed at any time depending on the conditions of the channel, as mentioned earlier.
in the thesis. \( \kappa \) is the fraction of power of \( P_1 \) that the source will use in the relay transmit period, and \( (1-\kappa) \) is the fraction of \( P_1 \) used during the relay receive period. During the relay transmit period, \( (1-\beta) \) is the fraction of \( \kappa P_1 \) 's power that is transmitted directly to the destination as a different part of the message.

The upper bound contains the same terms and has the same situation as that of the full duplex case. The cut for \( C_1^+ \) uses both the relay and the destination to jointly decode what the source sent during the relay receive period. This is obviously impossible. By changing the formula to make use of the decode and forward method, equation (9) becomes

\[
C_i^- = \alpha \frac{1}{2} \log \left( 1 + c_n^2 (1 - \kappa) P_1 \right) + \left( 1 - \alpha \right) \frac{1}{2} \log \left( 1 + (1 - \beta) c_n^2 \kappa P_1 \right) \tag{11}
\]

The theory involves transmitting two different signals to the destination, with one of the signals involving the parameter of \( (1-\beta) \). Thus, unless some methods are used to avoid interference, the destination will have interference from the term with \( (1-\beta) \). To avoid this problem, the interfering signal will not be sent at all by setting \( \beta = 1 \).

### 2.2.1 Theoretical calculation for \( \kappa \)

Now that \( \beta \) has been eliminated as a variable, two parameters are left to solve for, \( \alpha \) and \( \kappa \). It is not so simple to optimize over two parameters with only one equation using both of them. The usual solution to this type of equation is to solve for one parameter in terms of the other parameter. This is not very useful. Practically, in order to simplify the simulations, \( \alpha \) is set to \( \frac{1}{2} \). This has the advantage of making the calculations
slightly easier. In addition, as shall be seen later, it simplifies the method used to achieve TDD transmission.

After the elimination of the other parameters, the only parameter left is \( \kappa \). Thus, equations (11) and (10) become

\[
C_1^- = \frac{1}{4} \log \left( 1 + c_{21}^2 (1 - \kappa) P_1 \right) \quad (12)
\]

and

\[
C_2^- = \frac{1}{4} \log \left( 1 + c_{31}^2 (1 - \kappa) P_1 \right) + \frac{1}{4} \log \left( 1 + c_{32}^2 \kappa P_1 + c_{22}^2 P_2 + 2\sqrt{c_{31}^2 c_{32}^2 \kappa P_1 P_2} \right) \quad (13)
\]

Now, all that needs to be done is set equation (12) equal to equation (13) and solve for \( \sqrt{\kappa} \) in terms of the other variables. Naturally, this does not take into account the limits on \( \sqrt{\kappa} \), where \( 0 \leq \sqrt{\kappa} \leq 1 \) and it must be real.

If solved symbolically, \( \sqrt{\kappa} \) will be found to be the solution to a fourth order polynomial equation. The solution is much too long to reproduce here, and it would not make much sense even if it were here, but the equation to solve for it (symbolically) is reproduced here

\[
\left( c_{31}^2 P_1 + c_{32}^2 P_2 + c_{33}^2 c_{33}^2 P_1 P_2 - c_{21}^2 P_1 \right) + \left( 2c_{31}^2 c_{32} \sqrt{P_1 P_2} + 2c_{32}^2 c_{33} P_1 \sqrt{P_1 P_2} \right) \sqrt{\kappa}
+ \left( c_{31}^4 P_1^2 - c_{32}^4 c_{32} P_1 P_2 + c_{21}^4 P_1^2 \right) \kappa - 2c_{31}^2 c_{32} P_1 \sqrt{P_1 P_2} \kappa^2 - c_{31}^2 P_1^2 \kappa^2 = 0 \quad (14)
\]

In this case, it is best to solve the equation \( ax^4 + bx^3 + cx^2 + dx + e = 0 \). And set \( a, b, c, d, \) and \( e \) equal to the corresponding terms in the above equation.

Now, the limits on \( \sqrt{\kappa} \) must be taken into account. As in the full-duplex \( \sqrt{\beta} \) case, \( \sqrt{\kappa} \) can be negative and/or complex. In addition, it can also be greater than 1.
Unlike in the case of $\sqrt{\beta}$, $\sqrt{\kappa}$ is not necessarily complex when it is negative. Some rules were laid out to try to account for these cases. First, if it is complex, then only its magnitude is taken, and its phase is ignored. The negative case is taken care of by setting any negative value equal to zero, as it was in the full duplex case with $\sqrt{\beta}$. In the case where $\sqrt{\kappa} > 1$, set $\sqrt{\kappa} = 1$. 
3. Full Duplex Relaying Implementation

3.1 Block Markov Encoding and Backward Decoding

For full duplex relaying, the backward decoding method suggested by [2, pg. 4], which comes from [5] based on [6] is used. Here backward decoding is briefly summarized. Backward decoding is a form of a more general Superposition Block Markov Encoding (SBME). First, some things must be set up. Let a message stream be denoted by \( w[1], w[2], \ldots, w[N] \in \{1, \ldots, 2^{nR}\} \). The messages are transmitted in \( N+1 \) blocks each of \( n \) symbols. This gives a rate of \( \frac{N}{N+1} R \), but as \( N \to \infty \) an asymptotic rate of \( R \) is obtained.

The method of SBME is as follows: During block \( b=1,2,\ldots,N \), the transmitters send to the receiver a certain amount of “fresh information”, but with a rate too high to allow for reliable decoding. Because there is ambiguity left for the receiver, the transmitters in the next block cooperate to send “resolution information” to the receiver. Superimposed on this resolution information, the transmitters again send high-rate fresh information.

Decoding will use the idea of backward decoding. The method is as follows: The receiver starts decoding only after all \( N+1 \) blocks have been received. In block \( N+1 \) only resolution information is sent, so it is readily decoded. The resolution information is used to help decode the high rate “fresh information” in block \( N \). Simultaneously, the decoder is able to decode the resolution information in block \( N \), which is used for \( N-1 \). This continues until the message has been finally decoded at block 1.
Figure 3: Superposition Block Markov Encoding

In the proof of this method in [6], they use the idea where \( n \to \infty \), however, since they also use the fact that \( N \to \infty \), they must condition convergence on \( n \) increasing faster than \( N \).

In the case of the full duplex relay, the SBME method is used as follows: The encoder uses two codebooks, each with power 1: \( X_2(w) \) and \( X_1(w_1,w_2) \). The source transmits

\[
x_i[i] = \sqrt{(1-\beta)P}X_i(w[i],w[i-1]) + \sqrt{\beta P}X_2(w[i-1])
\]  

(15)

The relay receives \( y_2[i] = c_{21}x_i[i] + z_2[i] \), but uses only the \( c_{21}\sqrt{(1-\beta)P}X_i(w[i],w[i-1]) \) part of the signal to decode the message.

Assuming a successful decoding of \( w[i-1] \), if

\[
R \leq \frac{1}{2} \log \left( 1 + (1-\beta)|c_{21}|^2 P_1 \right)
\]  

(16)

then the relay can decode \( w[i] \) from \( y_2[i] \). After decoding the relay transmits,

\[
x_z[i+1] = \sqrt{P_2}X_2(w[i])
\]  

(17)
Note that $x_2[i+1]$ is due to the time delay when going from source $\rightarrow$ relay then relay $\rightarrow$ destination.

At this point, the destination node is receiving $y_3[i] = c_{31}x_1[i] + c_{32}x_2[i] + z_3$. This is equal to

$$y_3[i] = c_{31}\sqrt{(1 - \beta)}P_1X_1(w[i], w[i-1]) + \left(c_{31}\sqrt{\beta P_1} + c_{32}\sqrt{P_2}\right)X_2(w[i-1]) + z_3[i] \quad (18)$$

The received power is the magnitude of the two parts. Assuming the destination has successfully decoded $w[i]$, then it can decode $w[i-1]$ from $y_3[i]$ if

$$R \leq \frac{1}{2}\log\left(1 + \left|c_{31}\right|^2P_1 + \left|c_{32}\right|^2P_2 + 2\sqrt{\beta|c_{31}c_{32}|^2P_1P_2}\right) \quad (19)$$

Thus the capacity of the channel is the maximum over all $\beta$ of the smaller of the two rates.

This is a sort of proof of the above equations, specifically equation (6). This encoding method is what is used in the implementation, as all of it was suggested by Prof. Høst-Madsen.

Figure 4: The flow of information through the two paths.
The left cut represents path in (16). The right cut represents the path in (19)
3.2 Implementation Details

In the setup, all of the variables were declared beforehand, specifically $c_{21}, c_{31}, c_{32}, P_1, P_2$, and the noises. From these variables, the theoretically optimum numerical value for $\beta$ is calculated. Using this value of $\beta$ along with an encoded binary message, the source sends the following non-circular QPSK signal

$$x_i[i] = \sqrt{\beta P_i} (-1)^{c_1 w[i-1]} + j \sqrt{(1-\beta) P_i} (-1)^{c_2 w[i-1]} @ w[i]$$

This signal is a realization of equation (15). The relay takes the imaginary part of the signal $y_2[i] = c_{21} x_i[i] + z_i[i]$, and decodes it to obtain $\hat{w}$, which is an estimate of $w[i-1] \oplus w[i]$. It then obtains an estimate of $w[i]$ as $\hat{w}[i] = \hat{w} \oplus \hat{w}[i-1]$, with the exception of $i = 1$, where $\hat{w}[1] = \tilde{w}$. The imaginary part of $y_2[i]$ is

$$\text{Im} \{y_2[i]\} = \text{Im} \{c_{21} x_i[i] + z_i[i]\}$$

$$= c_{21} \sqrt{(1-\beta) P_i} (-1)^{c_1 w[i-1]} @ w[i] + \text{Im} \{z_i[i]\}$$

Note that the received power of this signal follows the power portion of the capacity equation (16).

During block $i$, the relay transmits $x_2[i] = \sqrt{P_2} (-1)^{c_3 w[i]}$. The destination receives

$$y_3[i] = c_{31} x_i[i] + c_{32} x_2[i] + z_3[i]$$

which is

$$y_3[i] = c_{31} \sqrt{\beta P_i} (-1)^{c_1 w[i-1]} + c_{32} \sqrt{P_2} (-1)^{c_3 w[i]}$$

$$+ j c_{31} \sqrt{(1-\beta) P_i} (-1)^{c_1 w[i-1]} @ w[i] + z_3[i]$$
When the $k$-th element is taken and transformed to polar form, the following equation is the result

$$y_3[i]_k = \left( c_{31}\sqrt{\beta P_1} + c_{32}\sqrt{P_2} \right) (-1)^{\theta_0} \exp \left( j\tan^{-1}\left( \frac{c_{31}\sqrt{(1-\beta)P_1}}{c_{31}\sqrt{\beta P_1} + c_{32}\sqrt{P_2}} \right) \right) + z_i[i]_k$$

From this equation, one can see that if $y_3[i]_k$ is rotated by the phase angle given by the exponential, then it becomes a BPSK signal with magnitude given by

$$\sqrt{c_{31}^2P_1 + c_{32}^2P_2 + 2\beta c_{31}^2c_{32}^2P_1P_2}.$$  Thus the received signal for $w[i-1]$ is BPSK with a phase determined by $w[i]$.  Note also that the magnitude of the received signal, when squared, matches the power component of equation (19), thus the theoretical $\beta$ calculated in (7) is expected to be exactly equal, or at least very close, to the experimentally acquired value of $\beta$.

From equation (24), one can see that the formula for the phase difference, which is called $\theta$, is determined solely by the channel constants, $c_{32}$ and $c_{31}$, as well as $P_1$, $P_2$, and $\beta$. Therefore $\theta$ can be calculated beforehand, and at the destination the received signal will be rotated by $\pm \theta$, with the $\pm$ depending on $c_\delta(\theta[i])$, so that it will be a BPSK signal. After rotating and removing the imaginary part of the noise, the received signal can be decoded to find $\hat{\theta}[i-1]$. Thus, as one can see, $\hat{\theta}[i-1]$ depends on $\hat{\theta}[i]$, so this method uses backward decoding for implementation. The steps involved in transmission are summarized below:
First assume that the transmission powers are at least the minimum necessary for good transmission.

1. During block 1, source transmits \( j\sqrt{P_1} (-1)^{c_w[i]} \). The relay is able to decode it because \( c_{21}P_1 > c_{21}(1-\beta)P_1 \), assuming that \( c_{21}(1-\beta)P_1 \) is the minimum power for small \( p(\text{error}) \) at the relay.

2. During block \( i = 2, ..., N \), the source transmits the QPSK signal

\[
x_1[i] = \sqrt{\beta P_1} (-1)^{c_w[i-1]} + j\sqrt{(1-\beta)P_1} (-1)^{c_w[i-1]\oplus w[i]}
\]

The relay receives only the imaginary part of the signal \( j\sqrt{(1-\beta)P_1} (-1)^{c_w[i-1]\oplus w[i]} \).

3. The relay decodes the signal \( \hat{w}[i-1] \oplus \hat{w}[i] \), and \( \oplus \) it with the \( \hat{w}[i-1] \) from the previous block to get \( \hat{w}[i] \). Then it re-encodes the block \( c(\hat{w}[i]) \).

4. In the actual implementation, however, it does not do the re-encoding, because the result of the decoding is \( c(\hat{w}[i-1] \oplus \hat{w}[i]) = c(\hat{w}[i-1]) \oplus c(\hat{w}[i]) \), so doing the addition, \( c(\hat{w}[i-1] \oplus \hat{w}[i]) \oplus c(\hat{w}[i-1]) = c(\hat{w}[i]) \). Thus, no re-encoding.

5. During block \( N+1 \), the source transmits the signal \( \sqrt{P_1} (-1)^{c_w[N]} \).

6. Also, for \( i = 1, ..., N \), the relay transmits \( x_2[i+1] = \sqrt{P_2} (-1)^{c_w[i]} \).
7. The destination receives \( y_3[i] = c_{31}x_1[i] + c_{32}x_2[i] + z_3[i] \). At \( i = N+1 \), the destination decodes it like a BPSK signal. At \( i = 1, \ldots, N \), the destination rotates the received signal by \( \pm \theta \), with \( \theta \) given by (24), and \( \pm \) determined by the previously decoded \( c_k(\tilde{\theta}[i+1]) \). (\( k \) subscript refers to the \( k \)-th element of the codeword) Then the destination takes the real part, thus removing the imaginary part of the noise, and decodes it like a BPSK signal.

It is important to note that in the implementation for the relay, decoding and re-encoding is not actually being done. One of the results of belief propagation decoding is that the whole codeword is the result, and the message bits are derived from that. Thus, rather than discard the original parity bits and re-encode the message bits, the intermediate step may be skipped, and the codeword that is output from the decoder is simply reused. This has the benefit of speeding up the simulation, since the message block is not being multiplied by the Generator matrix at the relay.

The same idea is also used at the destination. The destination requires a decoding and re-encoding of the next block in order to decode the previous block. Using this method, it is no longer necessary to have either the relay or the destination do any multiplication by the Generator matrix. Thus, the requirement that both the relay and the destination need to have the Generator matrix in their memory is relaxed.

### 3.3 Results

For the full duplex relaying with fixed channel gain, simulation is done in Matlab, with the computationally intensive parts written in C. For all of the simulations, \( P_1 = P_2 \).
is used to simplify things, because \( c_{32} \) could just as easily be modified as \( P_2 \), so \( c_{32} \) was chosen. For all of the simulations, the noise was also set at the relay and the destination to be \( N_{02} = N_{03} = 2 \), so that \( N_{0x}/2 = 1 \).

The code being used for this simulation is a rate \( \frac{1}{2} \) LDPC code received from Yige Wang, one of Prof. Fossorier’s students. Upon further research, it was found that this particular code is from Prof. David MacKay’s database on his website [8], in particular the Gallager code of rate \( \frac{1}{2} \) labeled \( \{8000.4000.3.483(N=8000, K=4000, M=4000, R=0.5)\} \). Also, some C source to do Belief Propagation decoding was received from Yige. The author of the original version was Jinghu Chen, also one of Fossorier’s students who wrote the source code for his dissertation [9]. This source was modified so that it could access Matlab variables and a lot of useless code was trimmed. After the modifications to the code were complete, some trial runs were started, to see how long a simulation would take. According to measurements taken, a set of 40,000 blocks would take about 1 day to simulate on a computer with a 1.5 GHz Pentium 4 CPU with 512 MB of reasonably fast RAM.

To convert a parity check matrix (H matrix) into a generator matrix (G matrix) a program was written to put an H matrix into reduced row echelon form, then output it in MacKay’s “alist” format. In addition, a Matlab script by Igor Kosintsev [7] was used to turn files from alist format to Matlab sparse matrix format. Thus, the G matrix was resident in Matlab’s variable memory. Unfortunately, in the end, the use of certain methods meant that it wasn’t even necessary to use the G matrix anywhere. However, for the sake of reference, the use of the source code is included, since the G matrices may be of importance, and the original source did use a G matrix. In addition, the simulation of
the multihop method with full duplex relay (described later) required the use of a \( \frac{1}{4} \) rate G matrix. So, the sources were of use after all.

### 3.3.1 First Simulation

For the first set of simulations, an "equilateral triangle" of sorts was used, by setting \( c_{21} = c_{31} = c_{32} = 0.8 \). With these channel parameters, the limiting path is not the source \( \rightarrow \) destination, but source \( \rightarrow \) relay. Here is a summary of the fixed parameters:

\[
N_{02} = N_{03} = 2 \rightarrow N_0/2 = 1
\]

This will always be true in the simulations.

\[
c_{21} = c_{31} = c_{32} = 0.8, \beta = 0.
\]

![Graph](image.png)

**Figure 5:** Comparison of BER vs. \( E_b/N_0 \) for Relay and Direct transmission
In this case, $\beta = 0$ because all of the power will be sent to the relay, since the source $\rightarrow$ relay section is the limiting part of the channel. Since that is the limiting part, it is possible to reduce the power coming from the relay, $P_2$, and still have the same capacity. However, it was decided earlier to set the transmitting powers $P_1$ and $P_2$ equal to each other, so that situation is ignored.

Unfortunately, the theoretical capacity of the channel for the full duplex case is not calculated and plotted, because this is a QPSK system, and there has not been an analysis of the capacity of QPSK modulation for the full duplex relay channel. Note that since $\beta = 0$, no power is transmitted in the real part of the signal being broadcast from the source.

Within the graph, the simulated relay transmission values are compared with the simulated direct transmission values using QPSK. A detail of importance is that the direct transmission values that are being compared to in the graph are of a $\frac{1}{4}$ rate code being transmitted using QPSK modulation. The reason that the full duplex relay method is compared to the $\frac{1}{4}$ rate QPSK code is because that is the fairest comparison that can be made. The method used in this full duplex case meant that 4000 bits were transmitted over 8000 transmissions. In addition, it used twice the bandwidth of a BPSK signal. In order to match that, a $\frac{1}{4}$ rate code was used, but half of the bits were transmitted over the imaginary part of the signal. Since an analogue cannot be drawn between the original rate $\frac{1}{2}$ code and the rate $\frac{1}{4}$ code, this comparison is not perfect.

From the graph of direct transmission, it is apparent that the relay transmission is much worse than the direct transmission for the particular case of an equilateral triangle. This is unfortunate, but perhaps not unexpected. As was stated earlier, power from the
relay, $P_2$, could have been decreased and the capacity would have remained the same. Thus, it would have been better to compare a direct transmission to full duplex relaying if $P_1 = P_2$ was not one of the conditions.

However, the fact that the direct transmission is over 4 dB better than the relay transmission cannot be explained by the differences in the code, or the fact that the allocated power was not optimum alone. Thus, this method is not ideal for the situation where the direct transmission path is comparably close to the relay paths. The next simulation demonstrates a situation where the relay transmission is much better than direct transmission.

3.3.2 Second Simulation

In the second simulation, a situation where $\beta > 0$ is simulated, so that the source must transmit part of its power to the destination only. The following parameters were used when setting up the program: $c_{21} = 0.8$, $c_{31} = 0.2$, $c_{32} = 0.7$. $\beta$ was calculated to be 0.0625, meaning that most of $P_1$’s power will be used to broadcast to the relay and destination, but a little bit of it will be transmitted only to the destination. Also within the graph, a simulation of multihop transmission is plotted.

3.3.2.1 Multihop

The method of multihop is considered the more “traditional” relaying method. In multihop, the source sends a message to the relay, then the relay transmits that same message to the destination. It is similar to a cascade of two direct transmission channels. Where it is different from the cascade of two channels is in the interference coming from the source to the destination. During $i = 1$, the source transmits $u_1[1]$ which the relay
receives as $c_{21} u_1[1]$. The destination receives $c_{31} u_1[1]$, which it discards. During $i = 2$, the source transmits $u_1[2]$, which the relay receives as $c_{21} u_1[2]$. At the same time, the relay is sending $\tilde{u}_1[1]$. Thus, destination receives $c_{32} \tilde{u}_1[1] + c_{31} u_1[2]$. Evidently, the $u_1[i+1]$ is an interfering signal at the receiver. The ideal case would be for $c_{31} = 0$. In that case there would be no interfering signal, and multihop would be performing at its best.

In the following graph, the plot of multihop does not use simulation values in the strictest sense, in that the simulation involved sending a signal from the source $\rightarrow$ relay, and then sending a signal from the relay $\rightarrow$ destination along with an interfering signal from the source $\rightarrow$ destination. Instead, the simulation makes use of the fact that the transmission from source $\rightarrow$ relay is identical to a direct transmission. Thus, the results from the direct transmission simulation can be reused.

As for the second part, the transmission from relay $\rightarrow$ destination, that is not similar to something that has already been simulated in this thesis. However, it is not so simple to simulate the second part. First, the power from the relay $\rightarrow$ destination must be increased to overcome the interference from the source. Therefore, $P_2 > P_1$. The exact formula is $P_2 = \frac{c_{31}^2 P_1 N_{03}/2 + c_{32}^2 P_1}{N_{03}/2}$. The first ratio before $P_1$ takes care of the channel gain differences between source $\rightarrow$ relay and relay $\rightarrow$ destination. The second ratio after $P_1$ takes care of the interference caused by source $\rightarrow$ destination. It assumes that the source interference is a Gaussian random variable when in fact, it is a Bernoulli random variable. This should not make too much of a difference as long as the interfering term remains small.
Unlike in the second graph, multihop was not compared to the simulated values in the first graph because that graph used an "equilateral triangle", and the channel gains were all equal. Thus, the interference from the next transmission would have required that the relay increase its power to compensate. This will make it perform worse than the direct transmission, so it was left out. In the second graph, the relay channel is compared to the multihop channel as well as the direct transmission channel.

In order to make as fair comparison as possible, the encoding method for multihop used a $\frac{1}{4}$ rate code with QPSK modulation. This is due to the aforementioned fact that the encoding scheme for the relay channel uses real and imaginary parts to send signals. Unfortunately, this also means that a different code will have to be used for comparison. Again, this causes the simulation to not be a completely fair comparison.
Unlike the first situation of an equilateral triangle, in the second case all of the power from the source and relay are required to do a proper decoding. Once again, for this case the theoretical capacity is not calculated or plotted, so that comparison is not made in this graph. This differs from the half-duplex case in section 4, where comparisons between the simulations and the theoretical capacity are made.

Note in the above case that using direct transmission is much worse than using the relay, although it is not a completely fair comparison, since the relay channel uses a different code than the direct transmission channel. It is obvious that the difference is primarily due to the path between source and destination, $c_{31}$, being very bad compared to
the other two paths, so the direct transmission route will require more power to overcome
the attenuation. This means that using the relay to transmit a significant portion of the
transmission in cases similar to this can result in gains of over 5 dB.

The multihop capacity is obviously much better than the direct transmission
capacity, due to the quality of the other paths. What is surprising is that the multihop plot
is better than the relay channel plot. Part of the difference may come from the fact that
the ¼ rate code is slightly closer to its theoretical capacity than the ½ rate code, but that
hardly accounts for the fact that multihop is 1 dB better than the relay channel. The relay
channel should be better than the multihop channel due to the fact that there is no
interference from the source \( \rightarrow \) destination path in the relay method. However, both
methods are still much better than direct transmission.

### 3.3.3 Third Simulation

In the previous simulations, the relay channel method used in this thesis was not
as good as the other methods used, namely, direct transmission and multihop. This next
simulation is one where the channel gain constants are set so that the source and relay are
close together, and the destination is far away. This situation is somewhat like an
antenna relay, so the benefits of an antenna array can be applied directly to the part of the
signal that source and relay transmit coherently. This method is expected to clearly
demonstrate the advantages of the relay channel.
As expected, the graph shows that in this particular case, the relay channel as implemented here is better than direct transmission and multihop methods. According to the graph, it is about 1.5 dB better. This is naturally due to the fact that the source and relay are sending the real part of the QPSK signal coherently, so that part of the signal benefits from the antenna array-like configuration of the relay implementation. As expected, the direct transmission method is better than the multihop method because multihop must use additional power at the relay to cancel out the interference coming from the source → destination path.
3.3.4 Determination of $\beta$

While simulating the above scenario, a discrepancy was noted in the time it was taking to decode at the relay and destination. It seemed that the destination was taking somewhat longer to decode its part of the codeword than the relay, meaning it had a lower SNR. In practice, the discrepancy was not necessary something to be concerned about, but its investigation did lead to a discovery about $\beta$. The value of $\beta$ was tested to see if the optimum $\beta$ was being used. So a simulation was set up with the following parameters:

$P_1 = P_2 = 1.24$

$c_{21} = 0.8, c_{31} = 0.2, c_{32} = 0.7$

Then $\beta$ was varied slightly. The results are as follows:
Figure 7: BER vs. $\beta$. Trying to find the best value

The first point at the far left of the graph is the theoretical value of $\beta$.

The experiment was repeated with a different value for $c_{32}$. The parameters below were tried:

$P_1 = P_2 = 1.48$

$c_{21} = 0.8, c_{31} = 0.2, c_{32} = 0.6$

Then $\beta$ was varied, simulated, and the BER was found. These are the results:
Figure 8: Another graph trying to find the optimum value of $\beta$

Again, at the far left is the theoretical value of $\beta$. According to both of these graphs, the actual $\beta$ is very close to, but not quite equal to the theoretical value. Since the BER of a given code is a function of the received SNR, if a small, but constant bit of power was added to the destination side of equation (6) before solving for $\beta$, then the actual value of $\beta$ may be solved for. In other words, a constant, $\delta$, will be declared and used in equation (6) as follows:

$$C = \max_{0 < \delta < 1} \min \left\{ \frac{1}{2} \log \left( 1 + (1 - \beta) c_2 P_1 \right), \frac{1}{2} \log \left( 1 + c_2^2 P_1 + c_2^2 P_2 + 2 \sqrt{\beta c_2^2 c_3 P_1 P_2} + \delta \right) \right\}$$

(25)

Thus, the new equation for $\beta$ will be
Using the graphs of $\beta$ vs. BER, the estimated value of $\delta \sim 0.02$ to 0.03. There is some evidence that $\delta$ is a function of some of the rest of the variables, and is more than just a constant. However, finding an actual formula for the best value of $\delta$ will be a very time consuming task.

The simulations have not been performed to find the formula for the exact value of $\delta$ for this case. However, it is not critical for the simulation, due to the fact shown in the above graphs that the theoretical value of $\beta$ is very close to the experimentally determined optimum value of $\beta$. So, the non-optimum results obtained in the graphs above should not change a great deal from the ideal results.

\[
\sqrt{\beta} = \frac{-c_{31}c_{32}\sqrt{P_2} + \sqrt{c_{31}^2c_{32}^2P_2 - c_{21}^2(c_{32}^2P_2 - c_{21}^2P_1 + c_{31}^2P_1 + \delta)}}{c_{21}\sqrt{P_1}}
\]  
(26)

3.4 Summary

In the process of implementing the full duplex relay, the method of Superposition Block Markov Encoding is used, which has been shown to achieve capacity in various papers [6]. By using SMBE and a QPSK signaling scheme, what the capacity theory says should be the received power at each node is mirrored exactly. While the relay node decodes the SMBE in the forward direction, the destination decodes its SMBE in the backward direction. In addition, before the destination decodes a block, it should rotate the received QPSK signal by a phase angle with the direction of the angle determined by the previous block. Thus, it turns the QPSK signal into a BPSK signal.

Three situations that use the full duplex channel are simulated and graphed. In the first situation, the relay is the limiting receiver, and so it is possible to reduce the
power to the destination from the relay and still get reliable transmission. Within the graph, there is also a comparison to a $\frac{1}{4}$ rate code transmitted using QPSK. This is the closest comparison that could be simulated that involved direct transmission. In this graph, the relay channel fared very poorly to the direct transmission case.

Another situation is a typical situation where the path from source to destination is very poor. This case is also compared with the $\frac{1}{4}$ rate direct transmission QPSK. In addition, it is also compared with what is known as multihop. Multihop is the traditional form of relaying, where a message is sent from source to relay, and then from the relay to the destination. It is similar to a cascade of two direct transmission channels. In this case multihop was actually a bit better than the relay channel implementation. This is unfortunate, because ideally, the relay channel should be better than multihop.

The third situation is one where the relay channel is better than the other two methods. This is the situation where the source and relay are close together, and far from the destination. In this case, the two transmitting nodes act like an antenna array for the part of the signal that they have in common. The relay implementation is about 1.5 dB better than the next closest method, direct transmission. Direct transmission is better than multihop because multihop must deal with interfering signals from the source to destination path.

Finally, it was noted that the $\beta$ that was being using to simulate blocks was not necessarily the best performing $\beta$. However, in the process of simulation, it was determined that the actual value was very close to the theoretical value. Thus, the values that were recorded were allowed to stand, because they were close enough.
4. Half Duplex Relaying Implementation

4.1 Basic Idea

For the TDD version of the relay, one must return to the idea behind the relay channel. The idea was that the source would broadcast a message that both the relay and the destination would receive. Assuming that the channel gain between source → relay is better than the channel gain between source → destination, the message would be sent at a rate that would be high enough for the relay to decode, but too low for the destination to decode. The relay and source would then cooperate to send resolution information coherently to the destination. The destination would receive this resolution information and combine it with the originally received information to decode the message. For the TDD implementation, an approach that mirrors this basic idea will be used.

Two possible methods will be described that can achieve the TDD implementation. Each of them has their positives and negatives, which will be described. In the end, the one that seemed less likely to have major problems in its implementation was chosen.

4.1.1 Serial concatenation at the encoder

The first method better elucidates the ideas of the relay, but was ultimately not implemented because of a lack of knowledge and inability to analytically determine which method would prove to be better and ease of implementation. This method was a serial concatenation of block codes. It involved using two codes in series, with the output
of one coder put into the input of another coder. An interleaver was placed between the decoders to improve the performance of the overall decoder [12].

The idea was to broadcast the output of the first encoder from the source to the relay and the destination. The rate and power of the signal would be such that the relay would be able to decode it, but the destination wouldn't. Then, since the relay now has the original message, have the relay and source generate the additional parity checks that come from the second encoder. Finally, the relay and source cooperate to send the secondary parity checks coherently. The idea is that the combination of the two block codes and the interleaver is roughly equivalent to a larger code of lower rate. In other words, a message that is encoded with a rate $\frac{1}{2}$ code and then another rate $\frac{1}{2}$ code is like a message encoded with an overall code of rate $\frac{1}{4}$.

<table>
<thead>
<tr>
<th>Message</th>
<th>First Parity Bits</th>
<th>Second Parity Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u^{(t)}$</td>
<td>$c^{(t)}$</td>
<td>$u^i$</td>
</tr>
</tbody>
</table>

![Diagram](image)

**Figure 9: Example of serial concatenation**

The idea is simple enough, and has a certain appeal to it. The source sends an encoded message that the relay would be able to decode, but the destination wouldn't. Then the source and relay cooperate to send additional resolution information in the form of parity bits to the destination. The destination would then use the information sent
during the relay received period along with the resolution information sent during the relay transmit period to decode the message.

The problem with this idea came at the decoder. In order to do soft-decision decoding, a dual soft input soft output decoder with interleavers and feedback would need to be implemented [12]. A program to do the decoding would have to be written from scratch. In addition, there was no guarantee that no unforeseen errors with this method would come up or that this method would even work as planned. Given these facts, and that the decoder for the other method was already written, it was decided to go with the other method.

![Diagram](image)

Figure 10: The theoretical decoder that could not be implemented due to time constraints

### 4.1.2 Code Shortening

The next method, which is the one that is implemented in this thesis, is called code shortening. The idea behind code shortening (which is also known as the method of punctured codes) is to take a code of a given rate and erase a certain number of bits to make it a new code with a new rate. For instance, in the simulations, a rate $\frac{1}{4}$ (10000, 2500) code is taken and only $\frac{1}{2}$ of the bits are transmitted. In effect, it becomes a rate $\frac{1}{2}$
code (5000, 2500). It is important to note, however, that this new shortened rate $\frac{1}{2}$ code will not perform as well as a code that was generated from the start to be a rate $\frac{1}{2}$ code. The method to decode these codewords is to take the transmitted bits and place them in their respective positions in the original codeword, then stuff 0's where no signal was transmitted, and use the original $\frac{1}{4}$ rate parity check matrix to decode the entire codeword. How this is used in the relay channel is as follows:

The source produces a codeword from the message. Then, the source takes $\frac{1}{2}$ of the bits of the codeword and broadcasts them to both the relay and the destination. If the bits were transmitted with sufficient power, then the relay will be able to decode the message, but the destination will not be able to. Then the relay and the source cooperate to coherently send the other $\frac{1}{2}$ of the bits to the destination. Using the bits from the previous transmission, plus the bits from the current transmission, the destination is able to treat it as a single codeword of rate $\frac{1}{4}$, and assuming sufficient power was used in the transmission, should be able to decode it.

This method has the advantage that the decoders written for the full duplex method may be reused in this case. This is in contrast to the case of serial concatenation, where the decoder would have to be written from scratch. From an analytical point of view, it is difficult to determine which of these methods has an advantage over the other one. Therefore, it makes sense to choose the one that is easier to implement. Thus, the code shortening method was chosen.
4.2 Implementation Details

4.2.1 The Problem

There are many ways of implementing the idea of code shortening. Within the scope of this thesis, one of the problems inherent in the relay channel will be solved by the methods that are explained here. This problem comes up in the situation when the source → destination path is much worse than the source → relay and relay → destination paths (a common occurrence in real life). Using a puncturing method similar to the serial concatenation case, after encoding a message, the first block, i.e. the first $n_1$ bits, would be sent to the relay and the destination. Then the source and relay both send the second block, the last $n_2$ bits ($n_2 = n - n_1$), to the destination.

<table>
<thead>
<tr>
<th>$n_1$</th>
<th>$n_2$</th>
</tr>
</thead>
</table>

For the code given, it was discovered that if the $n_1$ bits were sent first, then a very high SNR would be needed in order for the relay to decode. It seems that the code that was in use had to be punctured in a very specific manner to retain good performance. At this point, an attempt was made to send the $n_2$ bits first. It was discovered that this gave much better performance than the first $n_1$ bits at the relay. However, for the case given above, where the source → destination path is very poor, this proved to not work. The problem is that in this case, the source → relay path is sort of a “mirror image” of the relay + source → destination. When the source broadcasts its $n_x$ bits to the relay and destination, the destination receives that particular set of bits on a very poor path, then when it receives the other $n_y$ bits, it receives it on a very good path. This mirrors the relay, which receives the $n_x$ bits on a very good path from the source, but doesn’t receive
any of the \( n_\gamma \) bits at all, which is equivalent to transmitting on the worst possible path with channel gain = 0.

4.2.2 An Attempted Solution

One solution tried in order to solve this problem was to use irregular LDPC codes. Irregular LDPC codes are codes where the columns in the parity check matrix (H matrix) have very different weights. This means that some of the bits are better protected from error than other bits. In general, columns in the H matrix that have higher weight will better protect their corresponding bits. Also, irregular LDPC codes can have better performance than regular LDPC codes [11]. When an irregular code was tried in place of the original code while still keeping the \( n_1 \) and \( n_2 \) block format, it was found that it didn’t make much of a difference. However, in the process of trying irregular codes, a code was discovered that performed better in direct transmission than the \( \frac{1}{4} \) rate regular code given on MacKay’s website [8], the decision was made to use this irregular code in the simulations. This irregular code proved to perform better in the case of relaying than the regular code as well.
4.2.3 A Solution That Worked

Another solution thought out for this problem was to divide the code into two halves in such a way that were approximately equal in performance. Of course, there are a great many ways to divide the code into two halves. One method tried was a sort of "sliding window". A "window" of ½ of all the bits would be selected, and designated as the first half, and the rest of the bits would be the second half. The best division of the bits into two halves would be one where they are approximately equal in terms of error correcting performance, and out of the entire set of divisions that have equal performance, the best one would require the least power.

It was realized that it was not necessary for the bits to be clumped together in blocks, and that perhaps it would be better for them not to be together. So, other methods to split up the codeword were tried. One novel idea was that if every other column of the parity check matrix were taken, and combined into a new matrix, then it would appear to
be normal parity check matrix. So, the message was divided into two separate bits by
taking all of the odd numbered bits as one half of the codeword, and all the even
numbered bits as the other half of the codeword. After some trial simulations, it was
found that the two halves of the parity check matrix were indeed approximately equal in
their error correcting ability. In addition, each halves' overall performance as a \( \frac{1}{2} \) rate
code was fairly good, but not quite as good as a "true" \( \frac{1}{2} \) rate code. However, they were
good enough to use in simulations.

Figure 12: Taking every other column of a parity check matrix
leads to two matrices of approximately equal performance.

4.2.4 Recalculating \( \kappa \)

Much like in the full duplex case, the \( \kappa \) that the equation (14) gave to divide the
power at the source was not optimum. However, unlike in the full duplex case, in this
case the difference in the theoretical \( \kappa \) and the actual \( \kappa \) was quite large. The source of
this difference comes from the point stated earlier, that "It is important to note, however,
that this new shortened rate \( \frac{1}{2} \) code will not perform as well as a code that was generated
from the start to be a rate \( \frac{1}{2} \) code." In other words, the shortened code will need more
power than a good rate \( \frac{1}{2} \) code to decode the same message. In contrast, the destination
does not need any more power to decode what it receives, because it is an original \( \frac{1}{4} \) rate
code with good performance. So, equation (14) needs to be modified to include a constant power going to the relay called \( \delta \).

\[
\left( \delta + c_3^2P_1 + c_3^2P_2 + c_{31}^2c_{32}^2P_1P_2 - c_{21}^2P_1 \right) + \left( 2c_{31}c_{32}\sqrt{P_1P_2} + 2c_3^2c_{32}P_1\sqrt{P_1P_2} \right) \sqrt{ \kappa }
\]

\[
+ \left( c_{31}^4P_2^2 - c_{31}^2c_{32}^2P_2P_1 + c_{21}^2P_1 \right) \kappa - 2c_{31}^2c_{32}P_1\sqrt{P_1P_2} \kappa \chi - c_{31}^4P_1^2 \kappa^2 = 0
\]  

(27)

Although not proven graphically, through trial and error an approximate value for \( \delta \) of 0.1 was estimated for this case. Again, for this case, the value of \( \delta \) is very likely in fact a function of some of the other variables. It would most likely be very difficult to find a formula for \( \delta \) as a function of the other variables.

The steps in transmission are summarized below:
First assume that the transmission powers are at least the minimum necessary for good transmission.

1. Have the source encode a message. Then have it take every other bit (every odd numbered bit or even numbered bit) and broadcast that to the relay and destination.

2. The relay should be able to decode the message by stuffing 0's where no bit was transmitted, and using the original parity check matrix to decode. Then it re-encodes the estimate of the message. The SNR of the received bits from this first part are not enough for the destination to reliably decode the message.

3. Now the relay and source can coherently send the other bits (the even numbered bits or odd numbered bits) to the destination. However, if the relay detects that it did not correctly decode the message, then it will not send any bits at all. Assuming that the relay was able to decode correctly, it will send the same bits as the source, and they will constructively interfere.

4. The destination uses both halves of the codeword along with the parity check matrix to decode the message.

Note that there is also an additional condition when the relay is decoding. If the relay cannot decode the message, then it is very likely that the relay will at least be able to detect an error in the decoding, i.e. the syndrome is not all zeros. So, if the relay detects that the decoder did not correctly decode the message, meaning the relay calculates a syndrome that is not all zeros, then the relay will not forward the rest of the information. It does this because the message will be wrong, and it will destructively interfere with the (correct) source transmission. It is most probable that the destination
will still not be able to decode the transmission from the source transmission itself, but the chances of decoding are much higher than if the relay had transmitted incorrect (interfering) bits. Another minor advantage of this is that it saves a small (nearly negligible) amount of power.

As with the full duplex case, re-encoding is not actually being done at the relay. After the decoding process, the whole codeword is simply taken and reused during the relay transmit period. Again, this speeds up the simulation slightly. Also, as with the full duplex case, this means that the Generator matrix is only needed at the source, rather than at the source and relay.

4.3 Results

In the simulation, exactly the same setup was used as for the full duplex case. The simulation environment was primarily Matlab, for declaring variables, calculating the constant $\kappa$, and doing the matrix multiplication. As in the full duplex case, the only parts written in C were the decoding parts, which were the most time consuming parts of the code. Since code shortening was chosen as the implementation method, the source code that was used for the full duplex case was reused for the half duplex case with a few modifications. The simulations were using the irregular code of rate $\frac{1}{4}$ that was previously derived as described in section 4.2.2, as that turned out to give better performance than the rate $\frac{1}{4}$ code that was downloaded from Dr. David MacKay's website. The program to generate this irregular code was originally written by one of Prof. Fossorier's students, Juntan Zhang for his dissertation [10]. The simulation of this $\frac{1}{4}$ rate code for the TDD relay took approximately twice as long as that of the $\frac{1}{2}$ rate code.
in the full duplex relay case. The reference computer, the aforementioned 1.5 GHz Pentium 4, took 1 day to simulate only 20 000 blocks.

Again, as in the full duplex case, the parity check matrix (H matrix) was converted into a generator matrix (G matrix) using the previously written C program. Then, a Matlab script was used to turn the file from “alist” to Matlab sparse matrix format. Thus, the G matrix was in Matlab’s variable memory. Unfortunately in the end, as in the full duplex case, the use of certain methods meant that the use of the G matrix wasn’t even necessary anymore. However, the G matrix ended up being used in the simulation of the ¼ rate multihop transmission that was compared to the full duplex relay earlier in section 3.3.2.

4.3.1 Theoretical Capacity Calculation

Before giving the simulation results, a calculation of the theoretical capacity of the channel needs to be done. The absolute theoretical capacity given in equations (8), (9), and (10) cannot be done, because the calculation of optimum values of $\kappa, \beta$, and $\alpha$ all depend on each other. Thus, the equation would have to be optimized over three different variables. However, the capacity of equation (8) can be calculated using equations (12) and (13). Those set of equations, as stated earlier, assume that $\beta = 1$ and $\alpha = ¼$, so the capacity is calculated in terms of $\kappa$ only. So, the actual theoretical capacity is larger than what will be calculated here.

Using a rate ¼ code, the calculation is
Assuming that the theoretically best $\kappa$ is being used, then the two formulas in the braces are equal to each other. Thus the equation becomes

$$\frac{1}{4} = \min \left\{ \frac{1}{4} \log \left(1 + c_{21}^2 (1 - \kappa) P_1 \right), \right.$$  
$$\frac{1}{4} \log \left(1 + c_{31}^2 (1 - \kappa) P_1 \right) + \frac{1}{4} \log \left(1 + c_{32}^2 \kappa P_1 + c_{32}^2 P_2 + 2\sqrt{c_{31}^2 c_{32}^2 \kappa P_1 P_2} \right) \right\}$$  

(28)

One could have also set it equal to the other term, but this term is easier to calculate.

Solving this equation for $E_b/N_0$ yields

$$\frac{E_b}{N_0} = \frac{2}{c_{21}^2 (1 - \kappa)}$$  

(30)

Now, in order to solve this equation, all that is needed is the value of $c_{21}$, which is declared beforehand, and the value of $\kappa$, which is calculated from $P_1$, $P_2$, $c_{21}$, $c_{31}$, and $c_{32}$.

In the case the graphs below, for the case of theoretical capacity, the value of $\kappa$ that does not include the $\delta$ value that was used, so calculations assume that at least the encoders and decoders are optimum, rather than suboptimum, code-shortened systems.

The graphs below all use this formula for $\kappa$ to find the theoretical capacity for a given set of channel gains.

4.3.2 First Simulation
This graph is one where both the relay and destination are operating near their maximum error correcting capability, meaning there is an almost equal probability of an error coming from the relay decoding, as the destination decoding. From the graph, the simulated results are approximately 5 dB worse than the theoretical capacity. Besides the theoretical capacity and the simulated half duplex relay values, two additional lines are in the graph. The direct transmission line is much worse than the relay channel implementation. This is natural because the path between the source and destination is very poor in this case. The other line is multihop.
In the case of a half duplex relay, the procedure for multihop is slightly different, and indeed, simpler than in the full duplex case. During the relay receive period, the source sends a codeword to the relay of sufficient power and rate that the relay decodes it successfully. The destination ignores what it receives in this period. During the relay transmit period, the relay sends what it has decoded to the destination at such a rate and power that it is able to decode the codeword successfully. Thus, there is no interfering signal in any case. This case of multihop is in fact a true cascade of two separate channels. In fact, the plot above is derived from the simulations of the \( \frac{1}{2} \) rate code used previously in the full duplex case with a BPSK channel.

In the case given in the graph above, multihop is only slightly worse than relay. This is no surprise, since the path between source and destination is very poor compared to the relay \( \rightarrow \) destination and source \( \rightarrow \) relay paths. The worse the source \( \rightarrow \) destination path becomes, the closer the relay plot will move to the multihop plot. In the case where the source \( \rightarrow \) destination path is extremely comparatively poor, it would in fact be better to use multihop, since that method uses "true" codes that were originally designed for the channel, rather than code shortened codes.

It is important to state that this is not a completely fair comparison. The performance curve of the \( \frac{1}{2} \) rate code used in this simulation is slightly farther from its theoretical capacity curve than the \( \frac{1}{4} \) rate code that was used in the above experiment is from its theoretical capacity curve. However, this is the best comparison that could be made without having to evaluate the performance of a large set of \( \frac{1}{2} \) rate LDPC codes.
4.3.3 The Second Simulation

The next set of data points comes from the condition in which $\kappa = 0$. In this situation, all of the power of $P_1$ is used in the relay receive period, and thus none in the relay transmit period. This situation is interesting because the source is not transmitting anything during the relay transmit period and thus, there is no chance of interference, either constructive or destructive. Unfortunately, the channel constants can only take certain values for this to take place.

The channel gain constants for this particular simulation are as follows: $c_{21} = 0.8$, $c_{32} = 0.6$, and $c_{31} = 0.36$. The channel constants were not chosen so that either the relay or the destination would be the limiting decoder, however, for this particular case, the relay did have slightly more errors than the decoder. The value of $c_{31}$ was chosen so that it would change as little as possible from the first simulation in 4.3.2 and the rest of the channel constants would be the same. This was so that the new value for $\delta$ will not be significantly different from the old value.
Figure 14: Plot for second case where $\kappa = 0$

From this graph, the difference in decibels between the theoretical capacity and the simulated values is approximately the same for this set of channel gain constants as the set of channel gain constants given in the first graph. Note that the minimum in equation (28) refers only to the first term in the braces, since the limiting receiving node is the relay.

Once again, the direct transmission plot is much worse than the relay simulation plot. However, the distance between the two plots is not as great as it is in the first simulation (section 4.3.2). This is because the path between source $\rightarrow$ destination is
somewhat better than in the first simulation, so the power requirements for transmitting over that path are somewhat less.

This is in contrast to the case of the multihop plot in the above situation. In this case, the multihop curve is not as close to the relay curve as in the first simulation in section 4.2.2 due to the fact that the information gotten from the source → destination is of much better quality than in the first situation. In fact, the multihop plot is identical in the two situations, because the channel gain constants of the source → relay and relay → destination paths are identical in both cases. It is not that the multihop capacity decreased, but that the relay capacity and direct transmission capacity increased. In general, as the path between source → destination improves, the capacity of the relay channel method will improve, as will the capacity of direct transmission.

4.4 Summary

For the half duplex case of the relay channel, the choice between TDD and FDD was decided in favor of TDD. Since TDD and FDD are equivalent, it didn't matter which one was picked. TDD at the relay meant that the time needed to be divided into two parts. One part was called the relay receive period, and another part was called the relay transmit period. For the actual implementation of the TDD case, the basic idea of a relay was referred to. One method was that of a serial concatenation of codes. This used an inner code and an outer code with an interleaver in between. The inner code would be transmitted during the relay received period and the outer code would be transmitted during the relay transmit period.
In the end, a method that involved code shortening was used. The way code shortening was used in the relay channel was that half of the bits of a low rate codeword were broadcast to both the relay and the destination during the relay receive period, whereupon the relay decodes the codeword. Then, during the relay transmit period, the relay and source cooperate to send the other half of the codeword to the destination, which uses the information from both periods to decode the codeword. This was picked as the method due to the fact that it reused most of the code from the full duplex case and did not involve writing a brand new decoder.

In order to achieve a good result on the relay channel, both halves of the codeword had to perform equally when the other half was erased. A number of methods were tried to solve this problem. The problem was solved by the idea of taking every odd bit as the first half, and every even bit as the second half. Applying this method to the parity check matrix resulted in two parity check matrices that appeared similar to each other. This use of code shortening meant that the halves of code did not perform as well as independently generated codes of that rate, so the formula for $\kappa$ had to be modified to account for this discrepancy.

Before plotting the simulation results, the theoretical capacity formula had to be changed from one that optimized over three different variables to one that optimized over only one. After that, $E_b/N_0$ was calculated. The first graph was simulated over a common case where the source → destination channel gain was much worse than the other two channel gains. In this case, the source and relay had to be in sync in order to transmit the other half of the codeword coherently. The plot was also compared to the direct transmission simulation and a simulation of the multihop method. The relay
channel in this cases performed much better than direct transmission and moderately better than multihop.

In the second case, the channel gain from source \(\rightarrow\) destination was modified so that the value for \(\kappa=0\). This means that coherency will not be required for implementation. Besides the relay channel simulation, there was also a plot of the direct transmission simulation and multihop simulation. The margin between multihop and relay plot in this case became larger than in the first case. Notice that the graphs for each situation are quite far (5 dB) from the theoretical limit.
5. Conclusions, Possible Improvements or Future Works

5.1 Conclusions

In this thesis, two different situations for the relay channel were considered. The first situation was the full duplex channel. In this situation, the method already laid out by Dr. Høst-Madsen was used to implement the encoding method. Three cases were simulated. The setup, using SBME and QPSK, seemed to follow naturally from the formula for the theoretical capacity. Unfortunately, it cannot yet be compared to the theoretical capacity, because it is the capacity of a QPSK relay channel signal, and a formula has yet to be derived for that.

The simulated full duplex relay channel was compared to the direct transmission case and the multihop case. When the path between source and destination was very poor, the result naturally came to be that the relay channel fared much better than the direct transmission channel. However, compared to multihop, the relay channel fared slightly worse. This is unexpected, because the relay channel should be an improvement of multihop. A third case was simulated, where the source and relay were close together and the destination was equally far from both of them. In this case, the relay channel implementation proved to be better, as expected.

For the half duplex case, the TDD method was used for the relay. Implementation of the TDD channel at the relay means that there are two periods, the relay receive period, and the relay transmit period. The way the TDD relay was implemented was using code shortening, which is related to the erasure channel. The source would transmit half of the
code during the relay receive period, at which point the relay would take the code
shortened signal and recover the original message. Then, during the relay transmit period,
the relay and the source would coherently send the other half to the destination. The
destination would be able to decode the message from both halves.

The result of the simulations of two cases was that the method of implementation
proved to be about 5 dB away from the modified theoretical capacity. When it was
compared to the direct transmission and the multihop case. It was found that the relay
channel is slightly to moderately better than the multihop, depending on the channel gain
from source \(\rightarrow\) destination. For the values of the direct channel gain given, which were
small, the direct transmission case fared rather poorly.

\section*{5.2 Possible Improvements}

A possibility for sharpening of the results would be to calculate the theoretical
capacity of a full duplex QPSK relay channel, and then compare it to what was simulated
in this thesis. Given that the received power at each node is so close to the theoretical
value, one would expect that the difference between theoretical capacity and simulated
values of the full duplex relay channel to be comparable to the difference between the
theoretical capacity and simulated values of direct transmission. However, if there is a
big difference then there is something wrong in the method.

Although the equations seemed to follow from the formula, the actual simulated
optimum value of \(\beta\) did not precisely follow the theoretical value. It may be of interest to
find the actual value of \(\beta\). In some simulations, it was found that the \(\delta\) difference

55
between the theoretical and actual values seems to be a function of the other variables, the transmit powers and the channel gains.

Concerning the full duplex method, while finishing simulating most of the values for the graph, the author was also busy trying to find a solution to the TDD problems. In the course of code-writing efforts, it was realized that slightly better performance could be obtained out of the full duplex method if the method from the TDD channel was used. That method is optional forward at the relay. If the error detection ability of LDPC decoding was used to determine if the decoded message should be forwarded, then the relay would avoid sending erroneous data, and save itself some power. Of course, all of those values could not be simulated again, and it would hardly have made a noticeable difference, so it was left alone. However, this is an area that could be improved in this thesis.

For the half-duplex method, one thing in particular could be improved. Again, as it was in the full duplex case, the equation for $\kappa$ had to be modified in order to get close to the best-simulated value of $\kappa$. Also, as in the full duplex case, experiences during the simulation led to an assumption that the value of $\kappa$ is a function of all of the powers and channel gain constants. However, unlike the full duplex case, the best value and the theoretical value differ by a large amount. So, it would be advantageous to find the formula for the optimum value of $\kappa$. For the case of the values above, a constant was already found that serves as a good estimate at certain values of $P_x$ and $c_{xy}$. Note that the constant is an approximate value for $\kappa$ for this particular code only, and does not apply to other LDPC codes.
In addition, the value of $\kappa$ also depends on the method used to split up the codeword and parity check matrix. If a more analytical method was invented to determine the performance of two halves of a parity check matrix, then that would result in a significant improvement. This would require a deep understanding of LDPC codes.

Furthermore on the subject of $\kappa$, the value of $\sqrt{\kappa}$ in these simulations is limited to the real values only. Situations where $\sqrt{\kappa}$ was complex were avoided, because it was not known what to do in those cases. In the full duplex case, $\sqrt{\beta}$ needed to be real to be positive, so all of the cases where $\sqrt{\beta}$ was negative and complex could be ignored. However for $\sqrt{\kappa}$ this is not the case. Thus, a situation could be considered where it is complex, and figure out the best value of $\sqrt{\kappa}$ for that case.

5.3 Future Work

5.3.1 Full Duplex

Unfortunately, the full duplex implementation for the relay channel used in this thesis seems to perform worse than what is expected. For instance it is worse than multihop in some of the cases when it should be better than multihop. From an information theory point of view, the relay channel should always be better than multihop. For the case of the equilateral triangle, the relay channel method should not be that much worse than the direct transmission method. Most likely, this has something to do with the implementation in this thesis.

The theory for the relay undoubtedly expects all of the information being used in the method to be sent along a single BPSK channel. This, in turn, makes the relay into a
true full duplex relay, with the relay sending and receiving on the same path, rather than having the relay receive on the imaginary path and send on the real path. This does not seem to be possible, due to the fact that two different signals are being sent at the same time, and they would interfere if not for QPSK. Any implementation of the full duplex relay channel will have to ensure that the relay is sending and receiving codewords on the same channel at the same time.

5.3.2 Half Duplex

It would also be interesting to see an implementation of the idea of using two serially concatenated LDPC codes for the half duplex relay channel. From another perspective, it seems like a better idea than code shortening. However, there were a lot of misgivings about whether this method would really work, if there would arise some insurmountable obstacle that would have prevented the completion, like the problem mentioned in section 4.2.1. Any implementation of that decoder would need to be similar to turbo decoders, which are very similar to that decoder structure.

Besides that, this thesis only involves optimizing over \( \kappa \) when \( \alpha = \frac{1}{2} \). It would be interesting to see how to optimize over both \( \alpha \) and \( \kappa \) and how to implement the different rates for \( \alpha \). It would also involve splitting up the parity check matrix and codeword in different ways. A more analytical or at least methodical method to achieve this seems like an area of investigation. Using the original equation, optimizing over \( \alpha, \kappa \), and the constant \( \beta \) would prove to be very difficult, especially since the term that uses \( \beta \) must not interfere with the other signals.
Finally, in the TDD case, a situation was simulated where the relay and source did not need to be in sync. Since it is difficult to implement a full duplex node, and it is extremely difficult to synchronize two separate transmitters, that situation would be an ideal candidate for implementation. The only condition is that the experimenter would have to measure and calculate the average channel gain between the three nodes, and then arrange them in such a manner that the optimum value is $\kappa = 0$. Another option is to ignore the channel gains altogether and always assume that $\kappa = 0$. In fact, using the capacity formula, if the parameter $\beta = 1$ and $\kappa = 0$, then the capacity formula could be solved for just $\alpha$ for the half duplex case without synchronization. Since the values $\alpha$ can take are very code dependent, it would require something like rate compatible codes.
6. References


7. Igor Kosintsev. Matlab programmes for encoding and decoding of LDPC codes over GF(2^m).(600 KB) http://www.kozintsev.net/soft.html
   (alist2sparse.m in the archive)


16. B. Zhao, M. C. Valenti; "Distributed turbo coded diversity for relay channel."


7. Appendix A: Source Code

/* h2g.c
   Program to convert the parity check H file in Alist format to reduced row echelon form.

Revision 1.0
Written by Danny Yee
2005.11.15
*/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

/*************************
  Gaussian elimination  
*************************/

int Gaussiln_elimination(unsigned char *H_matrix[], unsigned short int nnn, unsigned short int rrr)
{
    unsigned short int first_check_row, operating_column, independent_col, target_row;
    unsigned short int i, j;
    unsigned char * temp_row = (unsigned char *)malloc(sizeof(char)*nnn);
    unsigned short int * independent_pos = (unsigned short int *)malloc(sizeof(short)*rrr);

    first_check_row = 0;
    /* Initialize first_check_row (record the first row for searching 1) */
    operating_column = 0;
    independent_col = 0;

    while((first_check_row<rrr)&&(operating_column<nnn))
    {
        target_row = first_check_row;
        /* Search for the first 1 in the operating_column-th column. */
        while((H_matrix[target_row][operating_column]!=1)&&(target_row<(rrr-1)))
            target_row++;

        /* Gaussian elimination */
        for(j=0; j<nnn; j++)
            temp_row[j] = H_matrix[first_check_row][j];
        for(i=first_check_row+1; i<rrr; i++)
            for(j=0; j<nnn; j++)
                H_matrix[i][j] -= temp_row[j]*H_matrix[first_check_row][j];
        independent_pos[operating_column] = (unsigned short int)first_check_row;
        first_check_row++;
        operating_column++;

        for(j=0; j<nnn; j++)
            free(temp_row[j]);
    }
    free(temp_row);
    free(independent_pos);
    return rrr;
}
if((target_row==target_row)\&(\&(H\_matrix[target_row][operating_column]==0))
  operating_column++;
else
{
  for(i=0; i<rrr; i++)
    if(H\_matrix[i][operating_column]==1)\&(\&(i!=target_row))
      for(j=0; j<nnn; j++)
        H\_matrix[i][j] = H\_matrix[target_row][j] \& H\_matrix[i][j];

  if(target_row!=first_check_row)
  {
    for(j=0; j<nnn; j++)
      {
        temp_row[j] = H\_matrix[first_check_row][j];
        H\_matrix[first_check_row][j] = H\_matrix[target_row][j];
        H\_matrix[target_row][j] = temp_row[j];
      }
  }

  independent_pos[independent_col++] = operating_column;
  first_check_row++;
  operating_column++;
}

/* printf("Current first check row is %d\n", first_check_row);
*/

free(temp_row);
free(independent_pos);
return independent_col;
}

/******Reduced row echelon form ******/
/* If there is no 1 on the diagonal, then we swap columns until there is one */
/* The returned int c is the number of swapped columns. */

short int reduced_row_echelon_form(unsigned short int *swap_cols, const short int N,
  const short int M, const short int rank, unsigned char **HH)
{
  short int i,j,d,temp,c=0;
for (i=0; i<M; i++) swap_cols[i] = 0;

for (d=1; d < rank; d++)
{
    if (HH[d][d] != 1)
    {
        j=d;
        while ((HH[d][j] != 1) && (j<N)) j++;
        if (j != N)
        {
            for (i=0; i<M; i++)
            {
                temp = HH[i][d];
                HH[i][d] = HH[i][j];
                HH[i][j] = temp;
            }
            swap_cols[c] = d;
            c++;
            swap_cols[c] = j;
            c++;
        }
    }
}

for (i=d-1; i >= 0; i--)
{
    /* For every row above the diagonal, starting at one, if find a one in the column d in that row, we xor row d with that row */
    if (HH[i][d] == 1)
    {
        for (j=d; j<N; j++)
            HH[i][j] = HH[i][j] ^ HH[d][j];
    } /* printf("Processed row %d.\n", d+1); */
}

return c;

}

int main(int argc, char *argv[])
{
65
FILE *in, *out, *out2;
short int i,j,c;
unsigned short int d,max_row_weight,max_col_weight;
unsigned short int temp,rank,M,N,K;
unsigned char **HH;
unsigned char **G;
unsigned short int *G_row_weight;
unsigned short int *G_col_weight;
unsigned short int *swap_cols;

unsigned short int **row_entries;
unsigned short int **col_entries;

/* We use short int and char to reduce the memory requirements. */

if ((in=fopen(argv[1],"r"))==NULL)
{
    printf("Cannot open the input file.\n");
    exit(0);
}

fscanf(in,"%d",&N);
fscanf(in,"%d",&M);
K = N-M;

fscanf(in,"%d",&max_col_weight);

/* The next number in the file is not necessary to reconstruct the matrix. */
/* Dump it into temp, and don't use it */
fscanf(in,"%d",&temp);

/*************** Allocate memory *******************/

HH = (unsigned char **)malloc(M*sizeof(char *));
G = (unsigned char **)malloc(K*sizeof(char *));

for (i=0; i<M; i++)
    HH[i] = (unsigned char *)malloc(N*sizeof(char));

for (i=0; i<K; i++)
    G[i] = (unsigned char *)malloc(N*sizeof(char));

G_row_weight = (unsigned short int *)malloc(K*sizeof(short int));
G_col_weight = (unsigned short int *)malloc(N*sizeof(short int));

66
swap_cols = (unsigned short int *)malloc(M*sizeof(short int));

row_entries = (unsigned short int **)malloc(K*sizeof(short int *));
col_entries = (unsigned short int **)malloc(N*sizeof(short int *));

/******* Finished allocating ********/

for (j=0; j<N; j++)
    for (i=0; i<M; i++)
        HH[i][j] = 0; /* initialize all to 0 */

/* The following two lines in the file are not necessary to reconstruct the matrix */
/* Dump the numbers into temp and don't use them. */

for (i=0; i<M+N; i++)
    fscanf(in, "%d", &temp);

/* Now read in the H matrix. */

for (j=0; j<N; j++)
    for (i=0; i<max_col_weight; i++)
    {
        fscanf(in, "%d", &temp);
        if (temp != 0)
            HH[temp-1][j] = 1;
    }

close(in);

printf("Read the H matrix.\n");

/* Perform gaussian elimination on HH to put it in row echelon form */
rank = Gaussian_elimination(HH, N, M);

printf("Put H matrix in row echelon form. Rank = %d\n", rank);

/* Now that the HH matrix is in row echelon form */
/* we have to put it in reduced row echelon form */
/* If there is no 1 on the diagonal, then we swap columns until there is one */

c = reduced_row_echelon_form(swap_cols,N,M,rank,HH);

printf("Put H matrix in reduced row echelon form.\n");

if ((out2=fopen("swapped_columns.txt","w+"))==NULL)
{
    printf("cannot open the output file.\n");
    exit(0);
}

for (i=0; i<2; i+=2)
    fprintf(out2, "%d %d\n", swap_cols[i]+1, swap_cols[i+1]+1);

fclose(out2);

/** Now we change the matrix from H to G using the fact that
    H = [I | A] and
    G = [A' | I]
*/

for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        G[i-M][i] = HH[i][j];

for (i=0; i<K; i++)
    for (j=0; j<K; j++)
        if (i==j)
            G[i][j+M] = 1;
        else
            G[i][j+M] = 0;

printf("Converted H matrix to G matrix.\n");

/** Now we have to output the resulting matrix. */
/** Output in Alist format */
/** First we need some statistics about the array */

for (i=0; i<M; i++) G_row_weight[i]=0;
for (j=0; j<N; j++) G_col_weight[j]=0;

for (i=0; i<K; i++)
    for (j=0; j<N; j++)
        if (G[i][j] == 1)
            { 
                G_row_weight[i]++;
                G_col_weight[j]++;
            }

max_row_weight=G_row_weight[0];
max_col_weight = G_col_weight[0];

for (j=1; j<N; j++)
  if (G_col_weight[j] > max_col_weight)
    max_col_weight = G_col_weight[j];

for (i=1; i<K; i++)
  if (G_row_weight[i] > max_row_weight)
    max_row_weight = G_row_weight[i];

/* Now we write to the file */

if((out=fopen("Gmatrix.txt","w+"))==NULL)
{
    printf("cannot open the output file\n");
    exit(0);
}

fprintf(out, "%d %d\n", N, K);
fprintf(out, "%d %d\n", max_col_weight, max_row_weight);

for (j=0; j<N; j++)
{
    fprintf(out, "%d ", G_col_weight[j]);
    col_entries[j] = (unsigned short int *)malloc(max_col_weight*sizeof(short int));
}

fprintf(out, "\n");

for (i=0; i<K; i++)
{
    fprintf(out,"%d ", G_row_weight[i]);
    row_entries[i] = (unsigned short int *)malloc(max_row_weight*sizeof(short int));
}

fprintf(out, "\n");

for (i=0; i<K; i++)
  for (j=0; j<max_row_weight; j++)
    row_entries[i][j] = 0;

for (j=0; j<N; j++)
  for (i=0; i<max_col_weight; i++)
    col_entries[j][i] = 0;
for (i=K-1; i>=0; i--)
    for (j=N-1; j>=0; j--)
        if (G[i][j] == 1)
            {
                G_col_weight[j]--;
                col_entries[j][G_col_weight[j]] = i+1;
                G_row_weight[i]--;
                row_entries[i][G_row_weight[i]] = j+1;
            }
for (j=0; j<N; j++)
    {
        for (i=0; i<max_col_weight-1; i++)
            fscanf(out,"%d ", col_entries[j][i]);
        fscanf(out, "%d\n", col_entries[j][max_col_weight-1]);
    }
for (i=0; i<K; i++)
    {
        for (j=0; j<max_row_weight-1; j++)
            fscanf(out,"%d ", row_entries[i][j]);
        fscanf(out, "%d\n", row_entries[i][max_row_weight-1]);
    }
fclose(out);

printf("Wrote G matrix to file.\n");
}
% Procedure file fullduplexrelay.m
% Self explanatory from the title, it simulates the full duplex relay channel.
% It uses the algorithm described by Dr. Host-Madsen
%
%load G4000x8000          % this is the G (Generator) matrix
%
% I have taken out the G matrix because the method I have used (using
% all zeros), means that I don't need to multiply anything by G.

randn('state',sum(100*clock));        % this is to seed the random number generator
%
% Some channel constants
c21 = 0.8;
c31 = 0.2;
c32 = 0.6;

N_02 = 2;
N_03 = 2;

P1 = 1.42;
P2 = P1;

delta = 0.03;                    % this is just my numerical estimate of delta gotten thru trial and
error.

sqrtBeta = (-c31*c32*sqrt(P2) + sqrt(c31^2*c32^2*P2 - ...
c21^2*(c32^2*P2+c31^2*P1-c21^2*P1 + delta))/(c21^2*sqrt(P1));

for i = find(sqrtBeta < 0),
    sqrtBeta(i) = 0;
end;

% As stated in the paper, negative values for sqrt(beta) must be corrected.

beta = sqrtBeta.^2;
clear sqrtBeta;

theta = atan2( c31 * sqrt((1-beta)*P1) , c31 * sqrt(beta*P1) + c32 * sqrt(P2) );

% theta is dependent only on the channel constants.
% I needed to use atan2() rather than atan()

E_s1 = c21^2*(1-beta)*P1;
\[ E_{s2} = c_{31}^2 P_1 + c_{32}^2 P_2 + 2c_{31}c_{32}\sqrt{P_1 P_2 \beta}; \]

\% \( E_{s1} \) and \( E_{s2} \) are the received powers (or energies, since \( P = E/t \)) they
\% should be approximately equal when \( \beta \) is not equal to 0. This is just
\% a sanity check. It is not used in the code.

\textbf{format compact;}

\textbf{kk=4000;} \hspace{1em} \% message block length
\textbf{nn=8000;} \hspace{1em} \% codeword length

\% Generate the sequence to be transmitted
\% We assume that the discrete input has already been source
\% coded (compressed) so that it is uniformly distributed.

\textbf{M = 10000;} \hspace{1em} \% number of blocks (MUST be a factor of \( N \))
\textbf{N = 100;}  
\textbf{blk_errors = 0;}

\% Transit all zeros \% Transit all zeros

\% Instead of randomly generating blocks, use the all zero block. That way
\% we speed things up and negate the necessity of the G matrix.

\textbf{X1(1,:) = j*sqrt(P1)*ones(1,nn);}
\textbf{X1(2:N,:) = sqrt(\( \beta \)P1)*ones(N-1,nn) + j*sqrt((1-\( \beta \)P1)*ones(N-1,nn);}
\textbf{X1(N+1,:) = sqrt(P1)*ones(1,nn);}

\textbf{for i=1:M/N,}
\textbf{  sprintf('Processing blocks %d - %d', (i-1)*N+1, i*N)}
\textbf{  W = round(rand(N,size(G,1)));}
\% \% Encode the blocks with a Generator matrix \( G \)
\textbf{  W1 = W(1:N-1,:);}
\textbf{  C1 = mod(W1*G, 2);}
\textbf{  W2 = W(2:N,:);}
\textbf{  C2 = mod(xor(W1,W2)*G, 2);}
\textbf{  clear W1 W2;}

\% modulate the blocks

\% for the first block we transmit only to the relay.
\textbf{  X1(1,:) = j*sqrt(P1)*(-1).^mod(W(1,:)*G,2);}
\[
\begin{align*}
X_1(2:N,:) &= \sqrt{\beta P_1}(1-\beta)P_1(-1)^{C_1} + j\sqrt{(1-\beta)P_1}(-1)^{C_2}; \\
X_1(N+1,:) &= \sqrt{P_1}(1)^{-1} \cdot \mod(W(N,:)*G,2); \\
\end{align*}
\]

\% Source transmit over the noisy channel
Y_2 = c21*X_1(1:N,:) + wgn(N, nn, N_02/2, 'linear', 'complex');
\% The wgn is linear because I am using linear units for my calculations.

\% Relay decode
sprintf('Relay decoding')
tic
Cdelta = LDPC_decode(imag(Y_2), N_02/2/sqrt(E_s1));
toc
clear Y_2;
C_hat = mod(cumsum(Cdelta),2);
\% Since Wdelta is the xor of W_hat[i-1] and W_hat[i], we have to sum
\% the Wdelta[i] with the previous W_hat[i-1] to get W_hat[i]. cumsum
\% is a quick way to get that for all of the codewords. Also note that
\% C(W_hat[i-1] xor W_hat[i]) = C(W_hat[i-1]) xor C(W_hat[i])
clear Cdelta;

sprintf('Relay errors = %i', length(find(sum(C_hat,2))))

\% Relay encode
\% C_hat = mod(W_hat*G,2);
clear W_hat;
\% Relay encoding is no longer necessary because we are using the codeword
\% output from the decoder.

\% Relay modulate
X_2 = sqrt(P_2)(-1)^{C_hat};
clear C_hat;
\% Relay transmit
\% First we pad the first row of X_2 with zeros to simulate a time delay
\% of one unit.
X_2 = [zeros(1,nn); X_2];
% Destination receive
Y3 = c31*X1 + c32*X2 + wgn(N+1, nn, N_0.03/2, 'linear', 'complex');
clear X2;

% Destination decode
sprintf('Destination decoding')
tic
W_hat2 = fullduplex_dest_decode(Y3, N_0.03/2/sqrt(E_s2), theta);
toc
clear Y3;
% The destination needs to have theta as an input so that it can rotate
% the signals by theta before decoding.
blk_errors = blk_errors + length(find(sum(W_hat2,2)))
clear W_hat2;

end;
clear M N i X1;
format:
/* File LDPC_decode.c
Original version written by Jinghu Chen

This file does belief propagation decoding with a Matlab interface to variables. */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mex.h"

#defme nn 8000 // Length of the code
#define kk 4000 // No. of information bits
#define rr 4000 // No. of checks
#define IMAX 200 /* maximum iteration times*/
// #define IMAX 500

void update_right_tanhZZ(int xx, double *ZZ[], int *row_distribution_index[], int *column_distribution_index[], double *right_tanhZZ[])

// to update the matrix of right_tanhZZ[] according to BIT xx
// uu is the weight of each column and hh is the weight of each row.
{
    int u,k, row, column;

    for(u=0; u<=(column_distribution_index[xx][0]-1); u++)
    {
        row = column_distribution_index[xx][u+1];

        for(k=0; k<=(row_distribution_index[row][0]-1); k++)
        {
            if(xx==row_distribution_index[row][k+1])
                column = k;
        }

        if(column==(row_distribution_index[row][0]-1))
            *(right_tanhZZ[row]+column)=1.0;
        else
            75
*(right_tanhZZ[row]+column)=(*(right_tanhZZ[row]+column+1))\cdot\tanh((*(ZZ[row]+column+1))/2);
}
}

/****************************update left_tanhhZ[]] according to Bit xx ****************************/

void update_left_tanhZZ(int xx, double *ZZ[], int *row_distribution_index[], int *column_distribution_index[], double *left_tanhZZ[])
{
    int u,k, row,column;

    for(u=0;u<=(column_distribution_index[xx][0]-1);u++)
    {
        row = column_distribution_index[xx][u+1];

        for(k=0; k<=(row_distribution_index[row][0]-1); k++)
        {
            if(xx=row_distribution_index[row][k+1])
                column = k;
        }

        if(column==0)
            *(left_tanhZZ[row]+column)=1.0;
        else
            *(left_tanhZZ[row]+column)=(*(left_tanhZZ[row]+column-1))\cdot\tanh((*(ZZ[row]+column-1))/2);
    }
}

/****************************update Ebsilong_Matrix[][] according to BIT xx ****************************/

void update_Ebsilong_Matrix(int xx, int *row_distribution_index[], int *column_distribution_index[], double *left_tanhZZ[],double *right_tanhZZ[],double *EEbsilong_Matrix[])
{
    int u,k, row,column;

    double tem, tem1;

    for(u=0;u<=(column_distribution_index[xx][0]-1);u++)
    {

row = column_distribution_index[xx][u+1];

for(k=0; k<=(row_distribution_index[row][0]-1); k++)
{
    if(xx==row_distribution_index[row][k+1])
        column = k;
}

tem1=(*(left_tanhZZ[row]+column))*(right_tanhZZ[row]+column));

tem=log((1+tem1)/(1-tem1));

if(tem>=10.0)
    tem=10.0;

if(tem<=-10.0)
    tem=-10.0;

*(EEbsilon~Matrix[row]+column)=tem;
}

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

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

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

void update_Ebsilong(int xx, int *row_distribution_index[], int
*column_distribution_index[], double *EEbsilon~Matrix[],double EEbsilong[])
{
    double tem=0.0;

    int u,k, row,column;

    for(u=0;u<=(column_distribution_index[xx][0]-1);u++)
    {
        row = column_distribution_index[xx][u+1];

        for(k=0; k<=(row_distribution_index[row][0]-1); k++)
        {
            if(xx==row_distribution_index[row][k+1])
                column = k;
        }
}
tem = tem + (*(E_Ebsilong_Matrix[row]+column));
}
E_Ebsilong[xx] = tem;

/** update E[i][j] according to BIT xx *************/
void update_E(int xx, int *row_distribution_index[], int *column_distribution_index[], double E_Ebsilong[], double *E_Ebsilong_Matrix[], double *E_E[])
{
    int u, k,
    row, column;
    for(u=0; u<=(column_distribution_index[xx][0]-1); u++)
    {
        row = column_distribution_index[xx][u+1];
        for(k=0; k<=(row_distribution_index[row][0]-1); k++)
        {
            if(xx == row_distribution_index[row][k+1])
                column = k;
        }
    }
}

/** update ZZ[i][j] according to Bit xx *************/
void update_ZZ(int xx, int *row_distribution_index[], int *column_distribution_index[], double *YY[], double *E_E[], double *ZZ[])
{
    int u, k,
    row, column;
    for(u=0; u<=(column_distribution_index[xx][0]-1); u++)
    {
        row = column_distribution_index[xx][u+1];
        for(k=0; k<=(row_distribution_index[row][0]-1); k++)
        {
            if(xx == row_distribution_index[row][k+1])
                column = k;
        }
    }
}
column = k;
}

*(ZZ[row]+column)=*(EE[row]+column)+*(YY[row]+column);
}

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

/* This is the primary function that does all of the work. It takes the constants
 n0, mrows, and the received signal matrix yy and decodes them using the BP
 functions above. */

void ldpc_decode(const double n0, const double *yy, double *W_hat, const int mrows)
{
    FILE *in1,*in2;

    double *y;

    // int *decoded_message;

    /**************************************************************************/
    // items used in BP Decoding
    /**************************************************************************/
    double *r0;

    double **Z;
    double **Y;
    double **E;
    double **Ebsilong_Matrix;

    int ** row_distribution_index;
    int ** column_distribution_index;

    double *Ebsilong;

    int *z;

    int syndrom, iteration, n, h;

    double **right_tanhZ;
    double **left_tanhZ;

    int i, j, k, p,q,t,temp6;

    double tem;
    int temp_weight;
/******************Allocate memory***********************

y = (double*) mxMalloc(sizeof(double)*nn);

// decoded_message = (int*) m_malloc(sizeof(int)*kk);

r0 = (double*) mxMalloc(sizeof(double)*nn);

Z = (double **) mxMalloc(sizeof(double*)*rr);
Y = (double **) mxMalloc(sizeof(double*)*rr);
E = (double **) mxMalloc(sizeof(double*)*rr);
Ebsilong_Matrix = (double **) mxMalloc(sizeof(double*)*rr);

row_distribution_index = (int **) mxMalloc(sizeof(int*)*rr);
column_distribution_index = (int **) mxMalloc(sizeof(int*)*nn);
Ebsilong = (double*) mxMalloc(sizeof(double)*nn);
z = (int *) mxMalloc(sizeof(int)*nn);

right_tanhZ = (double **) mxMalloc(sizeof(double*)*rr);
left_tanhZ = (double **) mxMalloc(sizeof(double*)*rr);

/********************get H_index***************************/

// read in the H_index
if ((in1=fopen("New_row_weight_distribution.txt","r"))==NULL)
{
    mexErrMsgTxt("cannot open the input file1 ");
}

if ((in2=fopen("New_col_weight_distribution.txt","r"))==NULL)
{
    mexErrMsgTxt("cannot open the input file2 ");
}

/* The H matrix in these two files is in a special form. It is very
similar to MacKay's alist format for his H matrices, but instead of
one file, there are two files, one for row and one for column. The
first entry of each line represents the number of 1's in the
row|column. The rest of the line tells us where in the row|column
the 1's are. Note that the first column|row in the H matrix is
numbered starting from 0 as in all C arrays, while in MacKay's
alist format, all indices in the H matrix start from 1.
*/

80
for(i=0;i<=(rr-1);i++)
{
    fscanf(in1,"%d", &temp_weight);

    if (( row_distribution_index[i] = (int *) mxMalloc((temp_weight+1)*sizeof(int))) == NULL)
        mexErrMsgTxt("Error allocating array row_distribution_index!");

    row_distribution_index[i][0] = temp_weight;

    for(j=1;j<=row_distribution_index[i][0];j++)
    {
        fscanf(in1,"%d",&row_distribution_index[i][j]);
        // the file have the index begin from 0
    }
}

fclose(in1);

for(i=0;i<=(nn-1);i++)
{
    fscanf(in2,"%d", &temp_weight);

    if (( column_distribution_index[i] = (int *) mxMalloc((temp_weight+1)*sizeof(int))) == NULL)
        mexErrMsgTxt("Error allocating array column_distribution_index!");

    column_distribution_index[i][0] = temp_weight;

    for(j=1;j<=column_distribution_index[i][0];j++)
    {
        fscanf(in2,"%d",&column_distribution_index[i][j]);
        // the file have the index begin from 0
    }
}

fclose(in2);

/************ Initialization ****************/
left_tanhZ[i]=(double *) mxMalloc(sizeof(double)*row_distribution_index[i][0]);
right_tanhZ[i]=(double *) mxMalloc(sizeof(double)*row_distribution_index[i][0]);
Z[i]=(double *) mxMalloc(sizeof(double)*row_distribution_index[i][0]);
Y[i]=(double *) mxMalloc(sizeof(double)*row_distribution_index[i][0]);
E[i]=(double *) mxMalloc(sizeof(double)*row_distribution_index[i][0]);
Ebsilong_Matrix[i]=(double *) mxMalloc(sizeof(double)*row_distribution_index[i][0]);

for(j=0; j<=(row_distribution_index[i][0]-1); j++)
{
    *(left_tanhZ[i]+j)=1.0;
    *(right_tanhZ[i]+j)=1.0;
    *(Z[i]+j)=1.0;
    *(Y[i]+j)=1.0;
    *(E[i]+j)=1.0;
    *(Ebsilong_Matrix[i]+j)=1.0;
}
} /**************finish initialization*************/

/*************** start processing rows ***************

for (n=0; n<mrows; n++)
{
    /* Here I make use of the way that Matlab stores a matrix. In short, Matlab stores a 2-D array, like the one we are using here, as a 1-D array of very long length. It stores the array column-wise. In other words, the first entry in the yy array is the first entry of the first row of the original Matlab matrix. The second entry in yy is the first entry in the second column of the Matlab matrix. This continues until all of the Matlab matrix is done. If you would like a
better description, I suggest you go use Matlab's help facilities. They have a very descriptive picture.
/*
for (h=0; h<nn; h++)
y[h] = yy[h*mrows + n];

/****************************BP Decode**************************/

/************initialize r0[], Z[][], Y[][]*************/

for (i=0;i<=(nn-l);i++)
{
    r0[i]=(4/n0)*y[i];
}

for (i=0;i<=(rr-l);i++) // rr is the number of checks
{
    for (j=0;j<=(row_distribution_index[i][0]-l);j++)
    {
        // Use the received sequence to initialize
        tem=y[(row_distribution_index[i][j]+1)]*(4/n0);

        *(Y[i]+j)=tem;

        *(Z[i]+j)=tem;
    }
}

iteration=1;
syndrom=1;

/* If the syndrome is 1, then the codeword has errors. Thus, we continue to decode. However, sometimes, we cannot decode, so we use a maximum iteration to ensure that we do not continue forever.*/
while((syndrom==1)&&(iteration<=IMAX))
{
    for (i=(nn-1); i>=0; i--)
    {
        update_right_tanhZZ(i,Z,row_distribution_index,column_distribution_index,right_tanhZ);
    }

83
for (i=0; i<=(nn-1); i++)
update_left_tanhZZ(i,Z,row_distribution_index,column_distribution_index,lefCtanhZ);
for (i=0; i<=(nn-1); i++)
update_Ebsilong_Matrix(i,row_distribution_index,column_distribution_index,lefCtanhZ, right_tanhZ,Ebsilong_Matrix);
for (i=0; i<=(nn-1); i++)
update_Ebsilong(i, row_distribution_index, column_distribution_index, Ebsilong, Matrix,Ebsilong);
for (i=0; i<=(nn-1); i++)
update_EE(i,row_distribution_index,column_distribution_index,Ebsilong, Ebsilong_Matrix,E);
for (i=0; i<=(nn-1); i++)
update_ZZ(i,row_distribution_index,column_distribution_index,Y,E,z);
for (q=0; q<=(nn-1); q++)
{
    if((rO[q]+Ebsilong[q])>=0)
        z[q]=0;
    else
        z[q]=1;
}
/*
for(p=(nn-kk); p<=(nn-1); p++)
{
    if((r0[p]+Ebsilong[p])>=0)
        decoded_message[p-(nn-kk)]=0;
    else
        decoded_message[p-(nn-kk)]=1;
}
*/
/* I don't return the decoded_message variable because in this case the codeword is more valuable to me. I can use it and avoid doing another encoding later. So, I don't bother with this variable. */

/********************check whether z*H is 0********************/
syndrom=0;
for(q=0; q<=((rr-1); q++)
{
    temp6=0;

    for(t=0; t<=((row_distribution_index[q][0]-1); t++)
        temp6 = temp6 ^ z[(row_distribution_index[q][t+1])];

    if(temp6==1)
        syndrom=1;
}

iteration++;

} /*end of BP iteration while((syndrom==1)&&(iteration<=IMAX))********/ /* This for loop uses the same principle as the yy array. It is stored in Matlab's internal format. */

for (h=0; h<nn; h++)
    W_hat[h*mrows + n] = (double)z[h];

} //for (n=0; n<mrows; n++)

--------------------------------------------------------------------------------------

Free all of the memory */

mxFree(r0);
for (k=0; k<rr; k++)
    mxFree(Z[k]);
mxFree(Z);
for (k=0; k<rr; k++)
    mxFree(Y[k]);
mxFree(Y);
for (k=0; k<rr; k++)
    mxFree(E[k]);
mxFree(E);
for (k=0; k<rr; k++)
    mxFree(Ebsilong_Matrix[k]);
mxFree(Ebsilong_Matrix);
for (k=0; k<rr; k++)
mxFree(row_distribution_index[k]);
mxFree(row_distribution_index);

for (k=0; k<nn; k++)
    mxFree(column_distribution_index[k]);
mxFree(column_distribution_index);

mxFree(Ebsilong);
mxFree(z);

for (k=0; k<rr; k++)
    mxFree(right_tanhZ[k]);
mxFree(right_tanhZ);
for (k=0; k<rr; k++)
    mxFree(left_tanhZ[k]);
mxFree(left_tanhZ);

// mxFree(decoded_message);

    mxFree(y);

/* The mex function
   The inputs should be n0 and the received message Y
   Since N0 and sqrt(Es) are used as a ratio throughout
the code, we incorporate sqrt(Es) into N0 and use that
scalar instead. In other words, n0 is actually N0/sqrt(Es).
The output is the decoded message W_hat.

    W_hat = LDPC_decode(Y, n0);
*/

/* If you would like to understand all of these declarations, then you
will have to read the Matlab help files on mex functions.
*/
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double n0;
    double *yy;
    double * W_hat;
    int i,j,mrows;

    mxFree(z);
    mxFree(right_tanhZ);
    mxFree(left_tanhZ);

    for (k=0; k<rr; k++)
        mxFree(right_tanhZ[k]);
    mxFree(right_tanhZ);
for (k=0; k<rr; k++)
    mxFree(left_tanhZ[k]);
mxFree(left_tanhZ);

    mxFree(decoded_message);

    mxFree(y);

}
check for proper number of arguments. */
if (nrhs != 2)
    mexErrMsgTxt("Two inputs required.");

/* Check that the second input is a scalar */
if (!mxIsDouble(prhs[1]) || mxIsComplex(prhs[1]) ||
    mxGetN(prhs[1]) * mxGetM(prhs[1]) != 1)
    mexErrMsgTxt("Input n0 must be a scalar.");

if (mxGetN(prhs[0]) != nn)
    mexErrMsgTxt("Y does not have the correct code length.");

n0 = mxGetScalar(prhs[1]);

/* mrows is the number of blocks being simulated for this particular instance. */
mrows = mxGetM(prhs[0]);

yy = mxGetPr(prhs[0]);

/* Create a matrix for the return argument. */
plhs[0] = mxCreateDoubleMatrix(mrows, nn, mxREAL);

W_hat = mxGetPr(plhs[0]);

ldpc_decode(n0, yy, W_hat, mrows);

}
/* File dest_decode.c
 * Based on LDPC_decode.c
 * Based on ldpc.cpp written by Jinghu Chen

This file is for the destination decoding of the full duplex method
this means that it uses phase angle rotation by theta and previous
codeword to decode the next codeword.
*/

The functions to do BP decoding are the same as in the LDPC_decode.c file, so I left
them out. Also, everything else is almost, but not quite the same, so I only highlight the
differences in bold.

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

*/ This is the primary function that does all of the work. It takes the constants
n0, mrows, the received signal matrices yr (real part) and yi (imaginary), and
the constant theta and decodes them using the BP functions above.
*/

void dest_decode(const double n0, const double *yr, const double *yi, double
*W_hat, const int mrows, const int theta)
{
    FILE *in1,*in2;/*, *Generator_fp;

    int ** Generator_matrix = (int **) mxMalloc(sizeof(int*)*nn);
    *
    /* I no longer need the Generator matrix, since I am using the whole
       output of the decoder instead of just the message, and then
       re-encoding.
     */

    double *y;
    int *decoded_message;

    /******************************************************************************
    ....some code....

    if ((in2=fopen("New_col_weight_distribution.txt","r"))==NULL)
        mexErrMsgTxt("cannot open the input file2");

    88
if (Generator_fp=fopen("Generator_matrix.txt","r")==NULL)
  mexErrMsgTxt("cannot open the Generator_matrix");

/* The H matrix in these two files is in a special form. It is very

....some code....

cfclose(in2);

/*
  for(i=0; i<nn; i++)
  {
    fscanf(Generator_fp,"%d", &temp_weight);

    if (( Generator_matrix[i] = (int *) mxMalloc((temp_weight+1)*sizeof(int))) ==
        mexErrMsgTxt("Error allocating array Generator_matrix!”));

    Generator_matrix[i][0] = temp_weight;

    for(j=1; j<=Generator_matrix[i][0]; j++)
      fscanf(Generator_fp,"%d",&(Generator_matrix[i][j])); /* the file have the
          index begin from 0
  }

cfclose(Generator_fp);
*/

/* Let me reiterate one more time, that I no longer need the Generator matrix. */

/****************************  Initialization  *******************************/

....some code....

Matlab’s help facilities. They have a very descriptive picture.

/*
  if (n==mrows-1)
    for (i=0; i<nn; i++)
      y[i] = yr[i*mrows + n];
*/

/* In this if ... else statement, we see that if the block is the last
  block to be received, then it does not need to be rotated by theta.
  If it is not the last block, then it does need to be rotated, with
the direction of rotation determined by the previous block. This comes from the paper.
*/

else
  for (i=0; i<nn; i++)
  {
    /*
     * z[i] = 0;
     * for(j=1; j<=Generator_matrix[i][0]; j++)
     *   z[i] ^= decoded_message[(Generator_matrix[i][j])];
     */
    /* Once again, generator not needed. */

    y[i] = sqrt(pow(yr[i*mrows + n], 2) + pow(y[i*mrows + n], 2));
    if (z[i]==1)
      y[i] = y[i]*cos(atan2(yr[i*mrows + n], yr[i*mrows + n]) + theta);
    else
      y[i] = y[i]*cos(atan2(yr[i*mrows + n], yr[i*mrows + n]) - theta);
  }

/************BP Decode*************/

....some code....

for (q=0; q<=(nn-1); q++)
{
  if((r0[q]+Ebsilog[q])>=0)
    z[q]=0;
  else
    z[q]=1;
}

for(p=(nn-kk); p<=(nn-1); p++)
{
  if((r0[p]+Ebsilog[p])>=0)
    decoded_message[p-(nn-kk)]=0;
  else
    decoded_message[p-(nn-kk)]=1;
}

/************check whether z^H is 0*************/

....some code....
/* This for loop uses the same principle as the yy array. It is stored in Matlab's internal format. However, unlike at the relay, the maximum length of each block is limited to kk. */

    for (i=0; i<k; i++)
        W_hat[i*(mrows-1) + n - 1] = decoded_message[i];

} //for (n=mrows-1; n>=1; n--)

....some code....

    mxFree(y);

/*

    for (k=0; k<n; k++)
        mxFree(Generator_matrix[k]);
    mxFree(Generator_matrix);

*/
}

/* The mex function
   The inputs should be N0 and the received message Y and the constant theta
   The output is the decoded message W_hat

   W_hat = dest_decode(Y, n0, theta);
*/

/* If you would like to understand all of these declarations, then you will have to read the Matlab help files on mex functions. */

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double n0, theta;
    double * yr, * yi;
    double * W_hat;
    int i,j,mrows;

    /* Check for proper number of arguments. */
    if(nrhs != 3)
        mexErrMsgTxt("Three inputs required.");

    /* Some code...*/
/* Check that the second and third inputs are scalar */
if (!(mxIsDouble(prhs[1]) || mxIsComplex(prhs[1]) ||
    mxGetN(prhs[1]) * mxGetM(prhs[1]) != 1 ))
    mexErrMsgTxt("Input n0 must be a scalar.");
if (!(mxIsDouble(prhs[2]) || mxIsComplex(prhs[2]) ||
    mxGetN(prhs[2]) * mxGetM(prhs[2]) != 1 ))
    mexErrMsgTxt("Input theta must be a scalar.");

if (mxGetN(prhs[0]) != nn)
    mexErrMsgTxt("Y does not have the correct code length.");

n0 = mxGetScalar(prhs[1]);

/* theta is the constant that was calculated earlier in the Matlab script */
theta = mxGetScalar(prhs[2]);
mrows = mxGetM(prhs[0]);

/* Real part of the received signal. */
yr = mxGetPr(prhs[0]);

/* Imaginary part of the received signal. */
yi = mxGetPi(prhs[0]);

/* Create a matrix for the return argument. */
plhs[0] = mxCreateDoubleMatrix(mrows-1, kk, mxREAL);

W_hat = mxGetPr(plhs[0]);
dest_decode(n0, yr, yi, W_hat, mrows, theta);

}
% Procedure File er_td_relay_channel.m
% Is a procedure to simulate the TDD relay channel with Erasure bits
%
%load G2500x10000
% The use of all zeros for the message as well as a lack of need for
% re-encoding means that the Generator matrix (G) is no longer needed.

randn('state',sum(100*clock)); % seed the random number generator

% Some channel constants
c21 = 0.8;
%c31 = 0.2;
c31 = 0.36;
c32 = 0.6;

N_02 = 2;
N_03 = 2;

%P1 = 2.78;
P1 = 2.5;
P2 = P1;

delta = 0.1;
% This is the value of delta that I got through trial and error.

a = -c31^4*P1^2;
b = -2*c31^3*c32*P1*sqrt(P1*P2);
c = c31^4*P1^2 - c31^2*c32^2*P1*P2 + c21^2*P1;
d = 2*c31*c32*sqrt(P1*P2) + 2*c31^3*c32*P1*sqrt(P1*P2);
e = c31^2*P1 + c32^2*P2 + c31^2*c32^2*P1*P2 - c21^2*P1 + delta;
sqrtKappa = very long equation deleted due to pointlessness

% I got this formula for sqrt(kappa) by using the symbolic math functions
% within matlab. Of course, I had to use a b c d e variables because if I
% had used the regular P1 P2 c31 c32 c21 variables, then the equation
% length would have overflowed the equation buffer and would not be
% reproducible here. The method to get sqrtKappa is as follows:
%
% syms a b c d e x
% X=solve('a*x^4+b*x^3+c*x^2+d*x+e=0');
%
% Then you have to try out values for the four solutions to get the right one.
for i = find(sqrtKappa < 0),
    sqrtKappa(i) = 0;
end;

for i = find(sqrtKappa > 1),
    sqrtKappa(i) = 1;
end;

kappa = abs(sqrtKappa^2);

P1_1 = (1-kappa)*P1;
P1_2 = kappa*P1;

alpha = 0.5;

cap1 = c21^2*P1_1;
cap2 = (1 + c31^2*P1_1) * (1 + c31^2*P1_2 + c32^2*P2 + 2*c31*c32*sqrt(P1_2*P2)) - 1;

% cap1 and cap2 are the overall received powers at the relay and the
% destination. cap1 represents the power received by the relay during the
% relay transmit period. cap2 represents the power received by the
% destination during the relay receive and relay transmit periods. These aren't really
% necessary, they are just a way to check my values.

format compact;

kk=2500;
nn=10000;

sqrtE_s1 = c31*sqrt(P1_1);
sqrtE_s2 = c31*sqrt(P1_2) + c32*sqrt(P2);
% sqrtE_s1 is the received power at the destination during the relay
% receive period
% sqrtE_s2 is the received power at the destination during the relay
% transmit period

% Generate the sequence to be transmitted
% We assume that the discrete input has already been source
% coded (compressed) so that it is uniformly distributed.
M = 10000; % number of blocks (MUST be a factor of b)
b = 100;

% Here we use ceil so that the number of bits is always a whole number.
We transmit all zeros to speed up the process.

As with the full duplex method, we transmit all zeros so that the G matrix becomes unnecessary and the encoding is faster.

\[ X_{1,1} = \sqrt{P_{1,1}} \times \text{ones}(b, nn*alpha); \]
\[ X_{1,2} = \sqrt{P_{1,2}} \times \text{ones}(b, nn*(1-alpha)); \]

% Since we transmit the all zero sequence, all of the parity bits are zero

\[
\text{blk\_errors} = 0; \\
\text{bit\_errors} = 0; \\
\text{for } i = 1: \text{M}/b, \\
\text{block} = i*b \\
\text{Y2} = c21 \times X_{1,1} + \text{wgn}(b, nn/2, N_0/2, 'linear', 'real'); \\
\%	ext{ wgn is linear because that is the standard we are using} \\
\%	ext{ Now we pad Y2 with zeros to accomplish code shortening} \\
Z = \text{zeros}(b, nn); \\
Z(:,1:2:nn) = Y2; \\
\text{clear Y2;} \\
\%	ext{ Relay decode} \\
\text{sprintf('Relay decoding')} \\
\text{tic} \\
[C\_hat, err] = \text{ir\_relay\_decode}(Z, N_0/(c21*sqrt(P_{1,1}))); \\
\text{toc} \\
\text{clear Z;} \\
\%	ext{ Now we send the other half of the bits coherently with the X1_2 of} \\
\%	ext{ the source (if X1_2 exists)} \\
\%	ext{ C\_hat = mod(W\_hat*G(:,2:2:nn),2);} \\
\%	ext{ clear W\_hat;} \\
\%	ext{ Modulate the codeword} \\
X2 = \sqrt{P_{2}} \times (-1) \times C\_hat(:,2:2:nn); \\
\text{clear C\_hat;}
% This is the part where we decide if we are really going to forward % the message. Since we are only going to forward if there is no % detectable error, we save a little bit of power and possibly improve % the decoding probability.
S = find(err);
for d = 1:length(S),
    X2(S(d),:) = zeros(1,nn/2);
end;
clear S d;

Y3_1 = c31*X1_1 + wgn(b, nn/2, N_03/2, 'linear', 'real');

Y3_2 = c31*X1_2 + c32*X2 + wgn(b, nn/2, N_03/2, 'linear', 'real');
clear X2;

% Y3_1 is the bits received during the relay receive period          
% Y3_2 is the bits received during the relay transmit period
Y3(:,1:2:nn) = Y3_1;
Y3(:,2:2:nn) = Y3_2;
clear Y3_1 Y3_2;

% In order to do decoding of blocks where half of them have one SNR and % the other have another SNR, we need the received power at each % period. Thus, the destination decoder uses sqrtE_s1 and sqrtE_s2. sprintf('Destination decoding')
tic
W_hat2 = ir_dest_decode(Y3, N_03, sqrtE_s1, sqrtE_s2);
toc
clear Y3;

% bit_errors = bit_errors + sum(sum(W_hat2))
blk_errors = blk_errors + length(find(sum(W_hat2,2)))
clear W_hat2;
end;
clear M b i Relay_errors kk nn block clear X1_1 X1_2 a b c d e;
format;
/* File ir_relay_decode.c 
   Based on LDPC_decode.c 
   Based on ldpc.cpp written by Jinghu Chen 

   This file is for the relay decoding of the half duplex method. */

/***************************************************************************/
#include <math.h> 
#include <stdio.h> 
#include <stdlib.h> 
#include "mex.h"

//**************************LDPC**************************

#define nn 10000    // Length of the code
#define kk 2500     // No. of information bits
#define rr 7500     // No. of checks
#define IMAX 200    /*maximum iteration times*/
//#define IMAX 500

All of the functions to do belief propagation decoding are the same for this file as for the previous files, so I cut them out.

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

/*****MAIN*************

/* This is the primary function that does all of the work. It takes the constants n0, mrows, the received signal matrix yy and decodes them using the BP functions above. Then it stores the codewords in C_hat and says whether there is an error or not in err. */

void ldpc_decode(const double n0, const double *yy, double *C_hat, const int mrows, double *err)
{

   ....some code....

   } /*end of BP iteration while((syndrom==1)&&(&iteration<=IMAX))******/
err[n] = (double)syndrom;

/* This for loop uses the same principle as the yy array. It is
   stored in Matlab's internal format. */

....some code....

/* The mex function
   The inputs should be the noise variance No and the received message Y
   The output is the decoded message C_hat

   [ C_hat, err ] = ir_relay_decode(Y, n0); */

/* If you would like to understand all of these declarations, then you
   will have to read the Matlab help files on mex functions. */

....some code....

int i,j,mrows;
double *err;

....some code....

plhs[1] = mxCreateDoubleMatrix(1, mrows, mxREAL);

/* err corresponds to the block errors that the decoding algorithm
   detected. A 1 in position X of the array corresponds to an error
   detected in row X of the blocks. */
err = mxGetPr(plhs[1]);

ldpc_decode(n0, yy, C_hat, mrows, err);

}
/* File ir_dest_decode.c
   Based on LDPC_decode.c
   Based on ldpc.cpp written by Jinghu Chen

This file is for the destination decoding of the half duplex method
this means that it needs to know the received powers of each half of
the received codeword, hence sqrtEs1 and sqrtEs2.
*/

All of the functions to do belief propagation decoding are the same for this file as for the
previous files, so I cut them out.

MAIN

/* This is the primary function that does all of the work. It takes the
   constants n0, mrows, the received signal matrix yy, and the received
   powers at the destination sqrtEs1 and sqrtEs2 and decodes them using
   the BP functions above. */

void ldpc_decode(const double n0, const double *yy, double *W_hat, const int mrows,
                 const double sqrtEs1, const double sqrtEs2)
{

....lots of code....

initialize r0[], Z[], Y[]

/* This is the first place where we need to apply the received
   energy at the destination. The ML ratio for Gaussian channels
   requires that we multiply 4*sqrt(Es)/N0. Since the two
   halves are received with different energy, we have to multiply
   each of them by a different sqrt(Es).
   */

/* Note that sqrtEs1 corresponds to the half of the code that is
   all of the odd numbered columns, i.e. column 1,3,5. It is
   important that if you were to somehow change the division that
   you change it here and in the next place.
   */

for (i=0; i<n0; i+=2)
    r0[i]=(4*sqrtEs1/n0)*y[i];
/* sqrtEs2 corresponds to the even numbered columns, i.e. column 2,4,6. If you change the way you divide the codeword, you need to change it here and the place below this. */

for (i=1; i<nn; i+=2)  
    r0[i]=(4*sqrtEs2/n0)*y[i];

for (i=0;i<=rr-1;i++) // rr is the number of rows and hh is the weight of rows
{
    for (j=0;j<=row_distribution_index[i][0]-1;j++)
    // Use the received sequence to initialize

    /* Here is the next place where you need to change the way the sqrt(Es) is used if you change it in the codeword outside of this source code. I'm sure you can figure out which is which. */

    if (row_distribution_index[i][j+1]%2 == 0)
        tem=y[(row_distribution_index[i][j+1])]*(4*sqrtEs1/n0);
    else
        tem=y[(row_distribution_index[i][j+1])]*(4*sqrtEs2/n0);

    *(Y[i]+j)=tem;
    *(Z[i]+j)=tem;
}

....some code....

/* This for loop uses the same principle as the yy array. It is stored in Matlab's internal format. However, unlike at the relay, the maximum length of each block is limited to kk because that is the message length. */

for (h=0; h<k; h++)
    W_h[h*mrows + n] = (double)decoded_message[h];

} //for (n=0; n<mrows; n++)

....some code....
/* The mex function
The inputs should be the noise variance n0 and the received message Y and
the received energies sqrtEs1 and sqrtEs2
The output is the decoded message W_hat

W_hat = unbalanced_LDPC_decode(Y, n0, sqrtEs1, sqrtEs2);
*/

/* If you would like to understand all of these declarations, then you
will have to read the Matlab help files on mex functions.
*/

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double n0;
    double sqrtEs1, sqrtEs2;
    double * yy;
    double * W_hat;
    int i,j,mrows;

    /* Check for proper number of arguments. */
    if(nrhs != 4)
        mexErrMsgTxt("Four inputs required.");

    /* Check that the second input is a scalar */
    if (!(mxIsDouble(prhs[1]) || mxIsComplex(prhs[1]) ||
         mxGetN(prhs[1])*mxGetM(prhs[1]) != 1 )
        mexErrMsgTxt("Input n0 must be a scalar.");

    /* Check that the third input is a scalar */
    if (!(mxIsDouble(prhs[2]) || mxIsComplex(prhs[2]) ||
         mxGetN(prhs[2])*mxGetM(prhs[2]) != 1 )
        mexErrMsgTxt("Input sqrt(Es1) must be a scalar.");

    /* Check that the fourth input is a scalar */
    if (!(mxIsDouble(prhs[3]) || mxIsComplex(prhs[3]) ||
         mxGetN(prhs[3])*mxGetM(prhs[3]) != 1 )
        mexErrMsgTxt("Input sqrt(Es2) must be a scalar.");

    if (mxGetN(prhs[0]) != nn)
        mexErrMsgTxt("Y does not have the correct code length.");

    n0 = mxGetScalar(prhs[1]);

    /* Received energy of the first half */

sqrtEsl = mxGetScalar(prhs[2]);

/* Received energy of the second half */
sqrtEs2 = mxGetScalar(prhs[3]);

/* mrows is the number of blocks being simulated for this particular instance. */
mrows = mxGetM(prhs[0]);

yy = mxGetPr(prhs[0]);

/* Create a matrix for the return argument. */
plhs[0] = mxCreateDoubleMatrix(mrows, kk, mxREAL);

W_hat = mxGetPr(plhs[0]);

ldpc_decode(n0, yy, W_hat, mrows, sqrtEs1, sqrtEs2);

}