Análise de Desempenho de Algoritmos de Decodificação para Códigos LDPC Regulares

Fernanda Pinto Magalhães

Dissertação apresentada ao Instituto Nacional de Telecomunicações, como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Prof. Dr. Geraldo Gil Ramundo Gomes

Santa Rita do Sapucai
2007

Dissertação defendida e aprovada em 11/12/2007, pela comissão julgadora:

Prof. Dr. Geraldo Gil Ramundo Gomes (Engenharia Elétrica - INATEL)

Dr. Fabbryccio Akkazzha Chaves Machado Cardoso (Faculdade de Engenharia Elétrica e Computação - FEEC/UNICAMP)

Prof. Dr. José Marcos Câmara Brito (Engenharia Elétrica - INATEL)

Prof. Dr. Carlos Roberto dos Santos
Coordenador do Curso de Mestrado

"A mente que se abre a uma nova idéia jamais voltará ao seu tamanho original"

(Albert Einstein)

A meus pais Célio e Carmem, pelo incentivo constante.

Ao meu namorado Sandro, pelo carinho e compreensão.

Agradecimentos

Agradeço a Deus e Santa Rita de Cássia pelas bençãos ao transformar a possibilidade em realidade. Sou grata por me permitirem seguir firme e concretizar este trabalho.

Ao meu orientador, Prof. Dr. Geraldo Gil, pelo apoio e atenção durante a produção da dissertação; ao Prof. Dr. Maurício Silveira por me iniciar como pesquisadora nos projetos de Iniciação Científica; ao Prof. Dr. José Santo Guiscafré Panaro pela sugestão do tema desse estudo e a todos os professores que de alguma forma contribuiram para meu crescimento profissional e pessoal.

Ao meu pai Célio, meu namorando Sandro e meu tio Zé, por me motivarem a realizar e finalizar este trabalho. O apoio de vocês foi imprescindível.

À minha família pela paciência, carinho ao ouvirem minhas ladainhas e por, mesmo de longe, estarem rezando e torcendo para o sucesso dessa dissertação. Vocês são meu ponto de apoio e meu refúgio.

Meu muito obrigada aos amigos do mestrado; em especial, Astrid, Daniel Hindemburg, Alexandre, Mauro, Felipe Augusto e Jaime; pelo companheirismo e ajuda incansável. Levarei vocês para sempre comigo.

Ao Prof. Carlos Augusto Rocha, por não medir esforços para auxiliar a mim e demais mestrandos, principalmente junto aos órgãos de fomento.

À FINEP, Financiadora de Estudos e Projetos, pelo suporte financeiro.

Resumo

Essa dissertação tem como objetivo a análise de desempenho de algoritmos de decodificação de códigos de verificação de paridade de baixa densidade, os códigos LDPC. Neste estudo são apresentadas as principais características do código e sua construção e três algoritmos de decodificação, para o código utilizado. Em um segundo momento tem-se a implementação de um simulador no programa Simulink para análise de desempenho dos algoritmos com diferentes tamanhos de palavras-código e os desempenhos obtidos por simulação são comparados.

Palavras-chave: Decodificação LDPC, códigos de blocos,
decodificação iterativa, algoritmos de decodificação LDPC.

Abstract

The purpose of this master degree thesis is to analyze the decoding performance of decoding algorithm for low density parity check codes, the LDPC codes. The main characteristics of the encoding scheme and its construction are presented on this work, as well as, three decoding algorithms for the proposed LDPC encoding technique. A simulator was also developed in Mathlab/Simulink to conduct a performance analysis and comparison of the target decoding algorithm.

Keywords: LDPC decoding, block codes, iterative decoding, LDPC decoding algorithms.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

BSC	*Binary Symetric Channel* - Canal Binário Simétrico
DVB-S2	*Digital Video Broadcasting - Satellite - Second generation* - Transmissão Digital de Vídeo - Satélite - Segunda Geração
LDPC	*Low Density Parity Check* - Verificação de Paridade de Baixa Densidade
MI-SBTVD	Modulação Inovadora - Sistema Brasileiro de Televisão Digital
BP	*Belief-propagation* - Algoritmo Soma-produto
APP	*A Posteriori Probability* - Probabilidade A Posteriori
BPB	*Based on Belief-propagation* - Baseado no algoritmo Soma-produto
LLR	*Log-likelihood Ratio* - Razão de Log-Verossimilhança
BPSK	*Binary Phase Shift Keying* - Chaveamento Binário por Deslocamento de Fase

QAM *Quadrature Amplitude Modulation* - Modulação de Amplitude em Quadratura

AWGN *Additive White Gaussian Noise* - Ruído Branco Aditivo Gaussiano

BER *Bit Error Rate* - Taxa de Erro de Bit

Lista de Símbolos

$\mathbf{H}$	Matriz de verificação de paridade do código LDPC
n	Comprimento da palavra-código
t_c	Peso das colunas da matriz $\mathbf{H}$
t_r	Peso das linhas da matriz $\mathbf{H}$
m	Mensagem de informação
k	Bits de informação da mensagem
R	Taxa de codificação LDPC
h_{mn}	Componentes da matriz $\mathbf{H}$ posicionado na linha m e coluna n
$\mathbf{G}$	Matriz geradora do código LDPC
v_n	Nó de bit do grafo bipartido na posição n
c_m	Nó de verificação do grafo bipartido na posição m
$\mathbf{v}$	Vetor codificado
$\mathbf{r}$	Vetor recebido
$\mathbf{e}$	Vetor erro
$\mathbf{s}$	Vetor síndrome
$\mathbf{b}$	Vetor paridade
$\mathbf{H}_{mn}$	Matriz $\mathbf{H}$ de tamanho m x n
$N(m)$	Conjunto de linhas em que para cada linha m há um conjunto de posições das colunas n para $H_{mn} = 1$
$N(m)\backslash n$	Conjunto $N(m)$ excluindo n
$M(n)$	Conjunto de linhas em que para cada coluna n há um conjunto de posições das linhas m para $H_{mn} = 1$
$M(n)\backslash m$	Conjunto $M(n)$ excluindo m

$\mathbf{x}$	Vetor-código de n posições usado nos algoritmos de decodificação
$\mathbf{x_{mn}}$	Matriz-código de tamanho m $\times$ n usada nos algoritmos de decodificação
q^0_{mn}	Probabilidade de x ser o valor binário 0
q^1_{mn}	Probabilidade de x ser o valor binário 1
f^1_n	Probabilidade de x dado que x é igual a 1
f^0_n	Probabilidade de x dado que x é igual a 0
r^0_{mn}	Probabilidade da verificação m ser satisfeita se o bit n do vetor x for fixo em 0
r^1_{mn}	Probabilidade da verificação m ser satisfeita se o bit n do vetor x for fixo em 1
δq_{mn}	Diferença entre os valores das probabilidades q^1_{mn} e q^0_{mn}
δr_{mn}	Diferença entre os valores das probabilidades r^1_{mn} e r^0_{mn}
$\hat{\mathbf{x}}$	Vetor decodificado
q^0_n	Pseudo-probabilidade de $\hat{x}$ ser o valor binário 1
q^1_n	Pseudo-probabilidade de $\hat{x}$ ser o valor binário 0
α_{mn}	Fator de normalização de q^x_{mn}
α_n	Fator de normalização de q^x_n
$\hat{\mathbf{y}}_\mathbf{n}$	Vetor decodificado por decisão abrupta
$\lvert\mathbf{r_n}\rvert$	Razão de log-verossimilhança de y_n
σ_{mn}	Soma módulo-2 entre os valores dos bits de $\mathbf{x_{mn}}$
σ_n	Soma módulo-2 entre os valores dos bits de $\mathbf{x_n}$
z_n	Vetor resultante do processo iterativo
z_{mn}	Matriz resultante do processo iterativo
$\hat{\mathbf{x}}^\mathbf{r}_\mathbf{n}$	Vetor razão $\hat{x}_n$
E_b	Energia de bit
N_0	Densidade de ruído
dB	Decibéis

Capítulo 1

Introdução

1.1 Motivação

A codificação para correção de erros é uma das ferramentas utilizadas para a transmissão de dados de maneira confiável em sistemas de comunicação. Para canais contaminados com ruídos, o Teorema de Shannon prova que se a informação codificada é transmitida com uma taxa abaixo da capacidade do canal, a probabilidade de erro de decodificação pode tender exponencialmente a zero, dependendo do esquema de codificação utilizado. O uso de códigos de verificação de paridade é uma alternativa de codificação que torna o processo de codificação relativamente simples para se implementar. Além disso, se um código de verificação de paridade de comprimento de bloco longo é usado em um canal binário simétrico (BSC), e se a taxa de codificação encontra-se entre a taxa crítica e a capacidade do canal, então a probabilidade de erro de decodificação poderá ser bem pequena [1]. Por outro lado, a decodificação suave de códigos de verificação de paridade, onde o número de palavras códigos é muito grande, geralmente não é simples de se implementar devido a sua alta complexidade. Neste contexto, uma vantagem dos Códigos de Verificação de Paridade de Baixa Densidade (*Low Density Parity Check Codes - LDPC*) é apresentar características que tornam a sua decodificação suave menos árdua. O foco desse trabalho está justamente nos algoritmos de decodificação para códigos LDPC. [2][6].

1.2 Relevância do Assunto

A atual utilização de técnicas de codificação e decodificação LDPC em sistemas de transmissão e recepção de televisão digital como, por exemplo, no projeto do sistema brasileiro de TV Digital desenvolvido pelo INATEL, UNICAMP, CEFET-

PR e UFSC, denominado MI-SBTVD (Modulação Inovadora - Sistema Brasileiro de Televisão Digital), motivou esse estudo. Além disso, essa solução foi adotada na norma DVB-S2 para transmissão digital de vídeo via satélite, o que demonstra a relevância desse estudo. A necessidade de implementar algoritmos para decodificação LDPC resultou neste estudo cuja pesquisa bibliográfica apresentou os trabalhos de Gallager, 1963 [2], Mackay, 1999 [13], Fossorier, 1999 [6] e Pìrou, 2004 [7]; que representam a linha de investigação mais significativa sobre algoritmos de decodificação para códigos LDPC.

1.3 Histórico

[1962] Robert Gallager realizou os primeiros trabalhos em código LDPC. A teoria ficou esquecida por quase 20 anos devido a impraticidade de implementação.

[1981] Tanner apresentou uma abordagem gráfica recursiva para o desenvolvimento de códigos longos de baixa complexidade.

[1999] Após 18 anos da apresentação do Grafo Bipartido, o LDPC for redescoberto por David Mackay. Foram realizados estudos sobre regularidade e também proposto o algoritmo Belief Propagation para decodificação de códigos LDPC baseado no algoritmo de J. Pearl [3].

A otimização dos códigos LDPC foi possível a partir de estudos sobre códigos irregulares desenvolvidos principalmente por Richardson [**?**] e Luby [5].

1.4 Objetivos

Essa dissertação tem como principal objetivo a análise de desempenho dos algoritmos clássicos de decodificação LDPC onde procurou-se evidenciar, comparativamente, a complexidade e o desempenho existente entre eles. A ferramenta utilizada para a simulação computacional para a avaliação de desempenho dos algoritmos analisados foi o programa Simulink. Para que a implementação do simulador fosse possível, fez-se necessário o estudo do algoritmo de decodificação LDPC proposto por Gallager em [2] e os algoritmos simplicados de decodificação propostos em [6] por Fossorier. A fase seguinte consistiu na implementação de um simulador bem como na análise de desempenho comparativa entre os algoritmos selecionados tanto por códigos curtos quanto para códigos longos.

1.5 Principais Contribuições

Como principais contribuições deste trabalho, tem-se a implementação do simulador LDPC para a avaliação do desempenho dos algoritmos para decodificação de códigos LDPC.

A partir dos resultados de desempenho obtidos os algoritmos foram comparados e resumidos em gráficos e tabelas de forma a identificar as principais diferenças entre eles.

1.6 Organização e Estrutura

O trabalho está organizado em 5 capítulos. Além do primeiro capítulo introdutório, é apresentado no segundo capítulo a teoria sobre codificação LDPC bem como a caracterização do código. No terceiro capítulo é apresentada a técnica utilizada para a construção do código LDPC. O quarto capítulo mostra os algoritmos utilizados para a decodificação LDPC e no quinto é mostrado todo o sistema implementado utilizando as técnicas apresentadas nos capítulos iniciais, bem como a análise de desempenho do sistema proposto. O último capítulo apresenta a conclusão do trabalho, bem como a motivação e sugestões para trabalhos futuros.

Capítulo 2

Código LDPC

2.1 Introdução

Muitos dos estudos em codificação nos últimos 50 anos foram direcionados à construção de códigos com estrutura robusta e grande distância mínima e, que através da manipulação dessas características, fossem apresentados códigos com baixa probabilidade de erro de decodificação [7]. Um código com grande distância mínima garante um bom desempenho do sistema porém, a estrutura robusta torna o processo de decodificação mais complexo. Desde 1993, novas técnicas de codificação tem possibilitado desempenhos próximos ao limite de Shannon, tal qual o uso de códigos turbo ou códigos LDPC, baseados em sistemas de decodificação iterativa.

Os códigos LDPC são códigos caracterizados por uma matriz de verificação de paridade **H** esparsa, ou seja, são códigos gerados através de uma matriz com muitos zeros e um número pequeno de uns. Um código de baixa densidade regular (n , t_c, t_r) é um código onde n é o comprimento do bloco, t_c é o número de 1´s de cada coluna da matriz de verificação de paridade e t_r é o número de 1´s de cada linha da matriz **H** [8][9][10][11].

2.2 Códigos de blocos lineares

Em um código de bloco binário define-se como bloco de mensagem m a seqüência binária de k dígitos de informação. Para o total de k bits existem 2^k mensagens e, assim sendo, existem 2^k palavras-código de comprimento n. Logo, um código de blocos (n,k) é composto basicamente por mensagens de k bits, em blocos codificados de n bits, onde $n > k$, isto é, a palavra código é composta por n bits codificados [12].

Uma característica desejada para códigos de blocos lineares é que as palavras-código sejam apresentadas de forma sistemática, i. e., é possível separá-la em duas partes, a parte de mensagem e a parte de redundância[12]. Assim, os bits de redundância consistem em n-k bits. A taxa de codificação é dada por k/n. Sendo assim, para uma palavra-código com 40 bits codificados e 20 bits de mensagem, a taxa de codificação é igual a $\frac{1}{2}$. Consequentemente, em uma palavra código o número total de bits é igual a soma do número de bits de informação com o número de bits de paridade (ou redundância), ou seja, $n > k$ [13].

2.2.1 Códigos de Verificação de Paridade Binários

Uma palavra-código composta por n bits é formada através da combinação de um bloco de k dígitos binários de informação e um bloco de n - k dígitos de verificação. Cada bit de verificação é uma soma módulo-2 de um conjunto pré-determinado de bits de informação. Essa regra de formação para bits de verificação é representada pela Matriz de Verificação de Paridade, ou matriz **H** [7][8]. Um exemplo de matriz de paridade por ser dado por (2.1). Na prática são utilizadas matrizes em que o número de zeros da matriz é superior ao número de uns.

$$H = (h_{mn})_{4\times 8} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{2.1}$$

O uso de códigos de verificação de paridade torna o processo de codificação relativamente simples de se implementar, sendo que a matriz geradora **G** é uma matriz sistemática. Porém a matriz de verificação de paridade **H** não é gerada de forma sistemática e para determinar a matriz **G** são necessárias algumas manipulações matriciais como poderá ser visto posteriormente.

2.3 Os códigos LDPC

Os códigos LDPC são códigos determinados por uma matriz de verificação de paridade (matriz **H**) esparsa, isto é, com muitos bits zeros e um número pequeno de bits iguais a um [8].

Quando se tem um código LDPC de comprimento longo, a análise torna-se complexa devido ao grande número de palavras-código envolvidas e por isso é comum se adotar uma análise estatística para encontrar um código com baixa

probabilidade de erro de decodificação.

2.4 Grafo de Tanner ou Grafo Bipartido

Um grafo de Tanner ou bipartido é uma forma de visualização de códigos LDPC que facilita a compreensão do processo de decodificação iterativa. A idéia central é usar o gráfico para estruturar as equações de verificação de paridade, facilitando a codificação e decodificação [9]. O grafo bipartido é construído definindo-se os nós de bit (v_n), em quantidades iguais às colunas da matriz **H**. E os nós de verificação (c_m) compostos pelo mesmo número de linhas de **H**. O número de ligações (t_c) que sai de cada nó de bit em direção aos nós de verificação é determinado pelo valor do peso de cada coluna da matriz de paridade, isto é, o número de 1´s em cada coluna. Da mesma forma, o número de ligações (t_r) que sai de cada nó de verificação em direção aos nós de bit é igual ao valor do peso de cada linha da matriz, dado pelo número de 1´s em cada linha de **H**. Logo, as posições dos 1´s da matriz **H** definem as interconexões entre nós de bit e nós de verificação.

Comparando a matriz **H** genérica representada por (2.2) com a matriz de verificação de paridade do código LDPC (8,4) apresentada pela matriz (2.1), e escolhendo-se aleatoriamente um bit de valor 1 da matriz **H**, percebe-se que para a posição h_{36}, tem-se m=3, correspondendo à terceira linha de **H**, e n=6, indicando a sexta coluna de **H**.

$$H = (h_{mn})_{4\times 8} = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} & h_{15} & h_{16} & h_{17} & h_{18} \\ h_{21} & h_{22} & h_{23} & h_{24} & h_{25} & h_{26} & h_{27} & h_{28} \\ h_{31} & h_{32} & h_{33} & h_{34} & h_{35} & h_{36} & h_{37} & h_{38} \\ h_{41} & h_{42} & h_{43} & h_{44} & h_{45} & h_{46} & h_{47} & h_{48} \end{bmatrix} \tag{2.2}$$

Assim, n indica o índice do nó de bit e m indica o índice do nó de verificação. Então, para h_{36} é realizada a ligação do nó de bit de índice 6, v_6 , para o nó de verificação de índice 3, c_3, indicada pela ligação com linha tracejada na Figura 2.1. Logo, para cada bit 1 da matriz **H** haverá uma ligação correspondente no grafo bipartido. Observando-se a matriz **H** de (2.1), tem-se 4 linhas, 8 colunas, t_r=4 e t_c =2, logo, obtém-se um grafo bipartido com 8 nós de bit com 2 ligações de saída de cada nó em direção aos nós de verificação, e 4 nós de verificação, com 4 ligações de saída em direção aos nós de bit, como se pode observar na Figura 2.1.

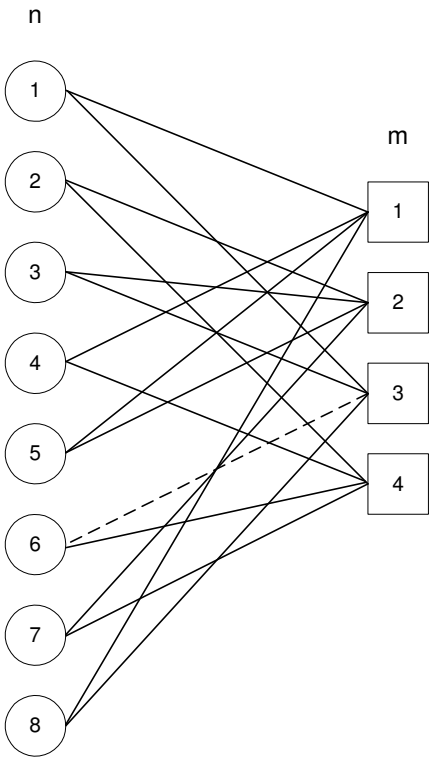

Figura 2.1: *Grafo bipartido do Código LDPC (8,2,4).*

2.5 Códigos Regulares

Os códigos LDPC regulares são códigos que possuem os nós de mesmo tipo com o mesmo peso, ou seja, todos os nós de bit devem ter o mesmo número de ligações de saída, e todos os nós de verificação devem ter o mesmo número de ligações de chegada, conforme foi definido por Gallager em [14].

Em (2.1), o grafo bipartido tem todos os nós de bit com grau 2 e todos os nós de verificação com grau 4, logo, o código LDPC(8,2,4) apresentado é um código regular. A complexidade do algoritmo de decodificação depende diretamente da densidade de ligações da matriz **H** e, como esta é esparsa, o grafo bipartido tem baixa densidade de ligações, portanto, menor a complexidade.

2.6 Códigos Irregulares

Códigos LDPC irregulares são códigos em que nós de mesmo tipo (verificação ou bit) têm graus com valores diferentes [9]. Para um código LDPC (7,4), onde

n=7 e k=4 pode-se ter a matriz de verificação de paridade **H** demonstrada abaixo.

$$H = (h_{mn})_{3\times 7} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{2.3}$$

Através da matriz **H** pode-se gerar o grafo bipartido para um código irregular, da mesma forma que ocorre para um código regular. Assim, tem-se n nós de bit e n-k nós de verificação, isto é, para a matriz **H** anterior serão sete nós de bit e três nós de verificação, porém com graus diferentes, como pode ser analisado na Figura 2.2.

Analisando v_1 e v_2 na Figura 2.2 nota-se a diferença entre os graus dos nós de bit caracterizando um código irregular [15].

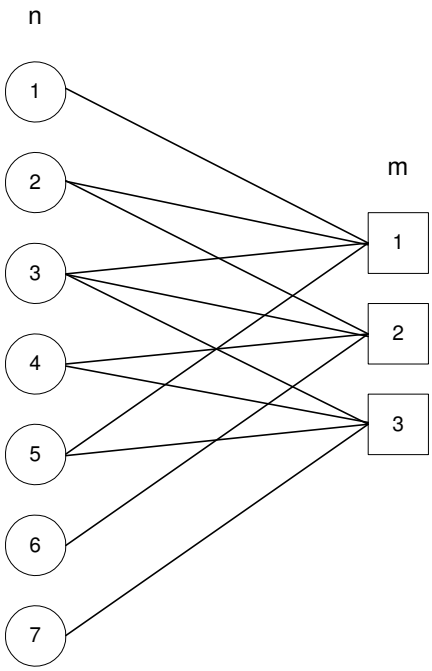

Figura 2.2: *Grafo bipartido para o código LDPC (7, 4) irregular.*

Atualmente os códigos LDPC, juntamente com os códigos turbo, são os códigos corretores de erros com melhor desempenho. Particularmente, os códigos LDPC irregulares são os que mais se aproximam do limite de Shannon [17] [5]

2.7 Cálculo da síndrome

Considerando um código LDPC (n, k) com uma matriz de verificação de paridade $\mathbf{H}$, pode-se definir então o vetor $\mathbf{v}$ como palavra-código e o vetor $\mathbf{r}$ como vetor recebido após o canal de transmissão [12]. Logo, encontra-se a expressão

$$\mathbf{r} = \mathbf{v} + \mathbf{e} \tag{2.4}$$

sendo $\mathbf{e}$ o vetor erro ou padrão de erro. O vetor erro, ou vetor $\mathbf{e}$ é causado devido ao ruído do canal de transmissão. Através do cálculo da síndrome, ou vetor $\mathbf{s}$, é possível verificar se o vetor recebido $\mathbf{r}$ contém ou não erros. Conforme apresentado em [12], através de

$$\mathbf{s} = \mathbf{r}.\mathbf{H}^{\mathbf{T}} \tag{2.5}$$

encontra-se o vetor síndrome, de dimensões $1 \times (n\text{-}k)$, e como o vetor recebido $\mathbf{r}$ é a soma entre o vetor erro e a palavra-código, i.e., $\mathbf{r}$=$\mathbf{v}$+$\mathbf{e}$ pode-se substituir $\mathbf{r}$ em (2.6), e tem-se

$$\mathbf{s} = (\mathbf{v} + \mathbf{e}).\mathbf{H}^{\mathbf{T}} = \mathbf{v}.\mathbf{H}^{\mathbf{T}} + \mathbf{e}.\mathbf{H}^{\mathbf{T}} \tag{2.6}$$

e como $\mathbf{v}.\mathbf{H}^{\mathbf{T}} = 0$, encontra-se o padrão de erro dado por $\mathbf{s} = \mathbf{e}.\mathbf{H}^{\mathbf{T}}$.

Este método de cálculo da síndrome é utilizado como a primeira etapa dos processos de decodificação. Caso a síndrome encontrada seja nula, o vetor recebido $\mathbf{r}$ é considerado uma palavra-código e admitido como vetor decodificado. Em caso contrário, admite-se que o vetor recebido contém erros, inseridos pelo canal de transmissão, e será necessária uma nova etapa de decodificação, como será apresentado no capítulo 4.

Capítulo 3

Construção de Códigos LDPC

3.1 Introdução

Um código LDPC (n , t_c, t_r), de comprimento n, peso t_c de cada coluna da matriz $\mathbf{H}$ e peso t_r de cada linha, com $t_r > t_c$, tem taxa de codificação determinada por:

$$R = 1 - \frac{t_c}{t_r} \tag{3.1}$$

Por definição, a taxa de codificação de um código de blocos é k/n, para que se possa igualar esse valor à (3.1), as linhas da matriz de verificação de paridade $\mathbf{H}$, devem ser linearmente independentes. Diferentemente dos códigos lineares, a matriz de verificação de paridade $\mathbf{H}$ de um código LDPC não é sistemática, isto é, não é possível distinguir diretamente os bits de mensagem dos bits de paridade. Para a codificação, pode-se obter uma matriz geradora $\mathbf{G}$ para códigos LDPC por meio do método de eliminação de Gauss, com operação módulo-2 [8][16]. Inicialmente, o vetor-código, ou palavra-código, que é um vetor linha, é particionado em termos dos vetores $\mathbf{m}$ e $\mathbf{b}$ como:

$$\mathbf{c} = \left[\begin{array}{c|c} \mathbf{b} & \mathbf{m} \end{array} \right] \tag{3.2}$$

onde $\mathbf{m}$ é o vetor mensagem de comprimento k e $\mathbf{b}$ é o vetor paridade de comprimento n-k. De forma equivalente, a matriz de verificação de paridade $\mathbf{H}$ também pode ser fracionada como apresentado em (3.3).

$$\mathbf{H^T} = \left[\begin{array}{c} \mathbf{H_1} \\ \hline \cdots \\ \hline \mathbf{H_2} \end{array} \right] \tag{3.3}$$

Para dar um exemplo de obtenção da matriz geradora **G**, considere a matriz **H** de (3.4) para o código LDPC (10, 5) com representação no grafo dado pela Figura 3.1.

$$H = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{3.4}$$

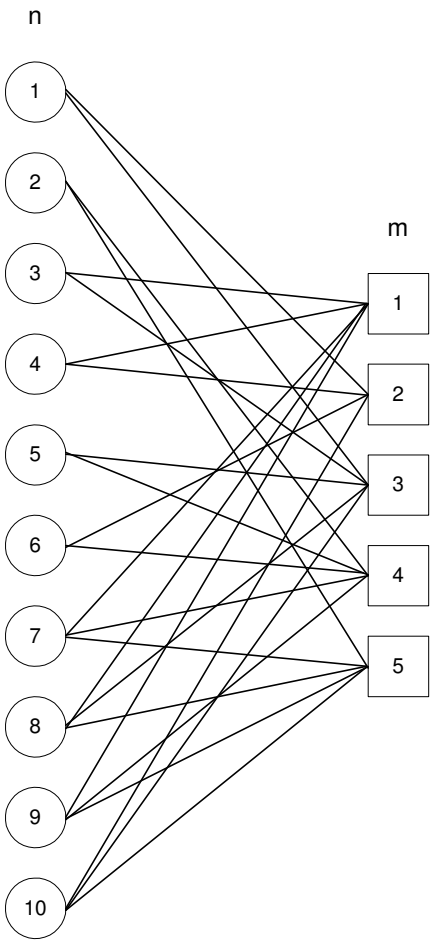

Figura 3.1: *Grafo bipartido do Código LDPC (10,5)*

Particionando a matriz **H** de (3.4) de acordo $\mathbf{H_1}$ e $\mathbf{H_2}$ de (3.3) tem-se:

$$\mathbf{H_1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.5}$$

e

$$\mathbf{H_2} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.6}$$

onde $\mathbf{H_1}$ é uma matriz quadrada de dimensões (n-k) x (n-k); e $\mathbf{H_2}$, uma matriz retangular de dimensões k x n-k. A transposição mostrada em (3.3) é usada no particionamento da matriz $\mathbf{H}$ para facilitar a apresentação e para atender

$$\mathbf{cH^T} = \mathbf{mGH^T} = 0 \tag{3.7}$$

A partir de (3.7) pode-se obter a relação entre os vetores $\mathbf{b}$ e $\mathbf{m}$ onde

$$\mathbf{b} = \mathbf{mP} \tag{3.8}$$

onde $\mathbf{P}$ é a matriz de paridade. Substituindo a Equação (3.8) em (3.7), tem-se $\mathbf{PH_1} + \mathbf{H_2} = \mathbf{0}$ e portanto $\mathbf{P} = \mathbf{H_2} - \mathbf{H_1^{-1}}$, onde $\mathbf{H_1^{-1}}$ é a matriz inversa de $\mathbf{H_1}$ e, para o exemplo tem-se $\mathbf{H_1^{-1}}$ definido por

$$\mathbf{H_1^{-1}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{3.9}$$

Portanto a matriz de paridade é dada por

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{3.10}$$

Assim, a matriz geradora de códigos LDPC é definida por

$$\mathbf{G} = \left[\begin{array}{c:c} \mathbf{P} & \mathbf{I_k} \end{array} \right] = \left[\begin{array}{c:c} \mathbf{H_2 H_1^{-1}} & \mathbf{I_k} \end{array} \right] \tag{3.11}$$

onde $\mathbf{I_k}$ é a matriz identidade k x k.

E, finalmente, a matriz geradora para o código LDPC(10, 5) do exemplo resulta em

$$\mathbf{G} = \left[\begin{array}{ccccc:ccccc} 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array} \right] \tag{3.12}$$

Logo, para gerar o vetor codificado, ou palavra-código, $\mathbf{c}$ faz-se:

$$\mathbf{c} = \mathbf{m}.\mathbf{G} \tag{3.13}$$

finalizando assim o processo de codificação LDPC.

O código LDPC(10, 5) usado como exemplo tem a finalidade de ilustrar o procedimento de geração do código, já que na prática o comprimento n do bloco é maior. Para o projeto MI-SBTVD, citado do Capítulo 1, foi utilizada uma matriz com o comprimento $n = 9792$ bits. No Capítulo 5 serão apresentadas matrizes onde $n > 500$, caracterizadas como matrizes para códigos grandes, bem como o resultado do desempenho de acordo com os diferentes algoritmos de decodificação utilizados [6].

Capítulo 4

Decodificação de Código LDPC

4.1 Introdução

Para muitos códigos de verificação de paridade o número de 0's e 1's na matriz de cheque de paridade **H** é aproximadamente o mesmo. Como mencionado anteriormente, para os códigos LDPC o número de 1's é muito menor se comparado ao número de 0's, logo a matriz **H** tem baixa densidade de 1's. Equivalentemente, o gráfico de Tanner do código tem baixa densidade de ligações. A complexidade do algoritmo de decodificação LDPC está diretamente ligada à densidade, e por esse motivo tem-se o interesse no desenvolvimento dos códigos LDPC tentando manter a densidade a menor possível [17][15]. O número pequeno de ligações entre os nós do grafo bipartido permite que aconteça um processo iterativo de decodificação de forma simples. A decodificação iterativa permite que o vetor recebido seja analisado várias vezes, até que se encontre um vetor considerado decodificado ou que seja declarado um erro, caso seja excedido o número máximo de iterações permitidas. Essa decodificação iterativa acontece com a troca de informações entre os nós de bit e os nós de verificação, através das ligações determinadas durante a construção do grafo de Tanner.

4.2 Algoritmos de Decodificação LDPC

Os algoritmos de decodificação LDPC utilizam o processo de decodificação iterativa e têm como objetivo encontrar o vetor $\mathbf{x} = [x_n]$ para que a operação $\mathbf{Hx} = 0$ seja verdadeira. Conforme definido no Capítulo 2, n representa o comprimento da palavra código, número de coluna de **H**, e m o número de equações de paridade, número de linhas de **H**. Desta forma k=n-m representa a dimensão de código, ou seja, o número de bits de informação e, consequentemente, 2^k é o

número de palavras código possíveis. Seja ainda $N(m)$ o conjunto de nós de bit incidentes sobre o nó de verificação (linhas) de index m. Da mesma forma, seja $M(n)$ o conjunto de nós de verificação que incidem sobre o nó de variável (coluna) de index n. Define-se ainda $N(m)\backslash n$ como o conjunto $N(m)$ excluindo-se o nó de bit de index n ; e $M(n)\backslash m$ como o conjunto $M(n)$ excluindo-se o nó de verificação de index m [6].

4.2.1 Decodificação Belief-propagation

Para o algoritmo Belief-propagation (BP), também conhecido com soma-produto, são definidas probabilidades q^x_{mn} e r^x_{mn} que serão atualizados iterativamente. A variável q^x_{mn} é definida como a probabilidade do bit n do vetor $\mathbf{x}$ assumir o valor x (0 ou 1), dada as informações obtidas pelos nós de verificação, com exceção do nó de verificação m. A variável r^x_{mn} é definida como a probabilidade da equação de paridade (nó de verificação) ser satisfeita fixando o bit n de $\mathbf{x}$ no valor binário xe os demais bits tendo a probabilidades dadas por $\{q_{mn'} : n' \in N(m)\backslash n\}$

A partir da matriz $\mathbf{H}$, constrói-se o grafo bipartido, que será a base da decodificação *Belief-propagation* (BP) , onde os valores das probabilidades serão transmitidos entre os nós de bit e nós de verificação através das ligações pré-determinadas. A decodificação BP é iniciada calculando-se os valores de q^1_{mn} e q^0_{mn} para cada nó de bit. Definindo-se tais valores, inicia-se o processo iterativo ainda nos nós de bit, sendo que primeiro passo é a determinação da matriz δq_{mn}com a diferença entre q^1_{mn} e q^0_{mn}. Esses valores são enviados para cada nó de verificação. Após definida a matriz δq_{mn}, calcula-se r^1_{mn}, r^0_{mn} e δr_{mn}, onde

$$\delta r_{mn} = \prod_{n' \in N(m)\backslash n} \delta q_{mn'} \tag{4.1}$$

Definidos r^1_{mn} e r^0_{mn} e δr_{mn}, enviam-se os valores calculados para os respectivos nós de bit.

O segundo passo é iniciado em cada nó de bit, calculando-se os novos valores de q^1_{mn} e q^0_{mn}, onde

$$\delta q^x_{mn} = \alpha_{mn} . f^x_n \prod_{m' \in M(n)\backslash m} {r^x}_{m'n} \tag{4.2}$$

f^x_n é a probabilidade a priori do n-ésimo bit de $\mathbf{x}$ assumir o valor binário x e α_{mn} em (4.2) é calculado de tal modo que a soma entre q^1_{mn} e q^0_{mn} deve ser igual a 1.

Logo, são calculadas as pseudo-probabilidades a posteriori q^1_n e q^0_n de cada nó

de verificação através da equação

$$q_n^x = \alpha_n . f_n^x \prod_{m' \in M(n)} {r^x}_{m'n} \tag{4.3}$$

sendo que α_n em (4.3) é o fator de normalização para que a soma entre q_n^1 e q_n^0 seja igual a 1.

O terceiro passo é o o passo da decisão do bit. Neste passo a decisão de bit é feita em cada nó, que recebe os valores de q_n^1 através das ligações pré-determinadas. Logo, se o valor de q_n^1 exceder 0,5, o bit n da palavra-código recebida é dado como válido e o valor se mantém; e em caso contrário, o bit n receberá o valor 0. Encontra-se então o primeiro vetor decodificado $\hat{\mathbf{x}}_\mathbf{n}$. Caso a síndrome seja nula, o vetor $\hat{\mathbf{x}}_\mathbf{n}$ é considerado válido e definido com vetor decodificado igualando-se ao vetor transmitido. Caso a síndrome não seja nula, retorna-se ao primeiro passo do processo iterativo. Deve-se definir um número máximo de iterações para que o sistema defina uma falha caso esse valor seja atingido.

A Figura 4.1 apresenta o fluxograma com todas as etapas do algoritmo de decodificação Belief-propagation. De acordo com o fluxograma da Figura 4.1 foi implementado algoritmo e a simulação posteriormente apresentada no Capítulo 5.

Nos próximos itens serão apresentados dois algoritmos simplificados definidos em [6], com objetivo de ter-se a etapa de decodificação implementada de forma mais rápida e eficiente.

Informações detalhadas do algoritmo *Belief-propagation* podem ser obtidas em Mackay [13].

4.2.2 Decodificação A Posteriori Probability

Da mesma forma que no algoritmo Belief-propagation, apresentado na seção 4.2.1, o algoritmo *A Posteriori Probability*, ou algoritmo APP, é iniciado com a construção do grafo bipartido. Além disso $\hat{\mathbf{x}}_\mathbf{n}$ recebe os valores das decisões abruptas de cada bit do vetor recebido $\hat{\mathbf{y}}_\mathbf{n}$ e calcula-se a LLR a priori dada por (4.4), onde

$$|\mathbf{r_n}| = |\mathbf{y_n}| \tag{4.4}$$

O processo iterativo é iniciado quando cada nó de bit envia os valores de $\hat{x_n}$ para os respectivos nós de verificação através das ligações do grafo bipartido. Em cada nó de verificação calcula-se σ_m, dado pela soma módulo-2 entre os valores de x_n que chegam em cada nó de verificação de índice m. Na seqüência, identifica-se o valor mínimo de $|y_{mn}|$, isto é, o menor valor de $|y_n|$ em cada nó de verificação. Cada nó de bit recebe os valores σ_m e $|y_{mn}|_{min}$ através das ligações do grafo

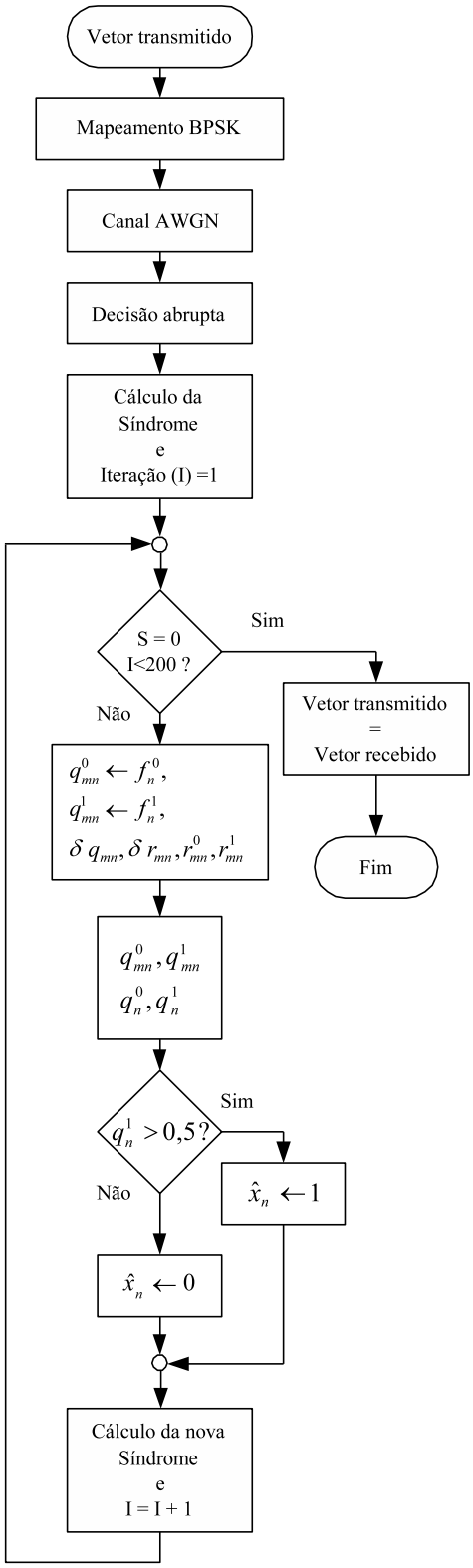

Figura 4.1: *Fluxograma que representa o algoritmo BP.*

bipartido, iniciando-se o segundo passo do processo iterativo. Para cada nó de

bit calcula-se z_n dado por

$$z_n = |r_n| + \sum_{m' \in M(n)} (\bar{\sigma}_m - \sigma_m)|y_{mn}|_{min} \tag{4.5}$$

No terceiro passo da etapa iterativa o valor calculado em (4.5), encontrado em cada nó de bit n, é testado. Caso o valor de z_n seja maior que zero, inverte-se então o bit $\hat{x_n}$ do respectivo nó de bit, i.e.,

$$\hat{x}_n = \hat{x}_n \oplus 1 \tag{4.6}$$

Caso contrário, o valor de $\hat{x_n}$ mantém-se inalterado. Além disso, atualiza-se $|\mathbf{y_n}|$ com o valor de $|\mathbf{z_n}|$. No próximo passo realiza-se o teste do vetor decodificado $\hat{\mathbf{x}}_\mathbf{n}$ calculando-se a síndrome. O mesmo processo de término do algoritmo *Belief-propagation* é utilizado, ou seja, caso a síndrome seja nula, o vetor $\hat{\mathbf{x}}_\mathbf{n}$ é válido e definido como vetor decodificado e transmitido. Caso a síndrome não seja nula, inicia-se novamente o processo iterativo com um número máximo de iterações.

A Figura 4.2 representa o fluxograma do algoritmo de decodificação A Posteriori Probability. No Capítulo 5 o desempenho do algoritmo APP é analisado e comparado com os demais algoritmos apresentados neste capítulo.

4.2.3 Decodificação Based on Belief-propagation

O algoritmo *Based on Belief-propagation*, ou *BP-Based*, como o próprio nome indica, é baseado no algoritmo *Belief-propagation*, porém de forma simplificada já que valores de probabilidades não são utilizados nos cálculos ocorridos nos nós de bit e nós de verificação. Baseado no grafo bipartido, encontra-se os valores das decisões abruptas do vetor recebido $\hat{\mathbf{y}}_\mathbf{n}$ chamado de vetor $\hat{\mathbf{x}}_\mathbf{n}$ e iguala-se $\hat{\mathbf{x}}^\mathbf{r}_\mathbf{n}$ ao vetor $\hat{\mathbf{x}}_\mathbf{n}$. Também são definidos os valores dos vetores *LLR* a priori, por $|\mathbf{r_n}| = |\mathbf{y_n}|$ e $|\mathbf{y_{mn}}| = |\mathbf{y_n}|$. Logo, cada nó de bit de índice n recebe os valores $\hat{x}^r_n$, $\hat{x_n}$, $|r_n|$ e $|y_{mn}|$, iniciando o processo iterativo. Assim, calcula-se o valor de σ_{mn} por

$$\sigma_{mn} = \hat{x}^r_n \oplus \Big(\sum_{n' \in N(m) \backslash n} \hat{x}_{mn}[\text{mod} - 2] \Big) \tag{4.7}$$

e identifica-se o valor mínimo de $|y_{mn}|$, i. e., o menor valor de $|y_n|$ em cada nó de verificação. Encontrados os valores de σ_{mn} e $|y_{mn}|_{min}$, calcula-se em cada nó de bit, os valores de z_n, da mesma forma que apresentado em (4.5) e z_{mn}, sendo

$$z_{mn} = |r_n| + \sum_{m' \in M(n)} (\bar{\sigma}_{m'n} - \sigma_{m'n})|y_{m'n}|_{min} \tag{4.8}$$

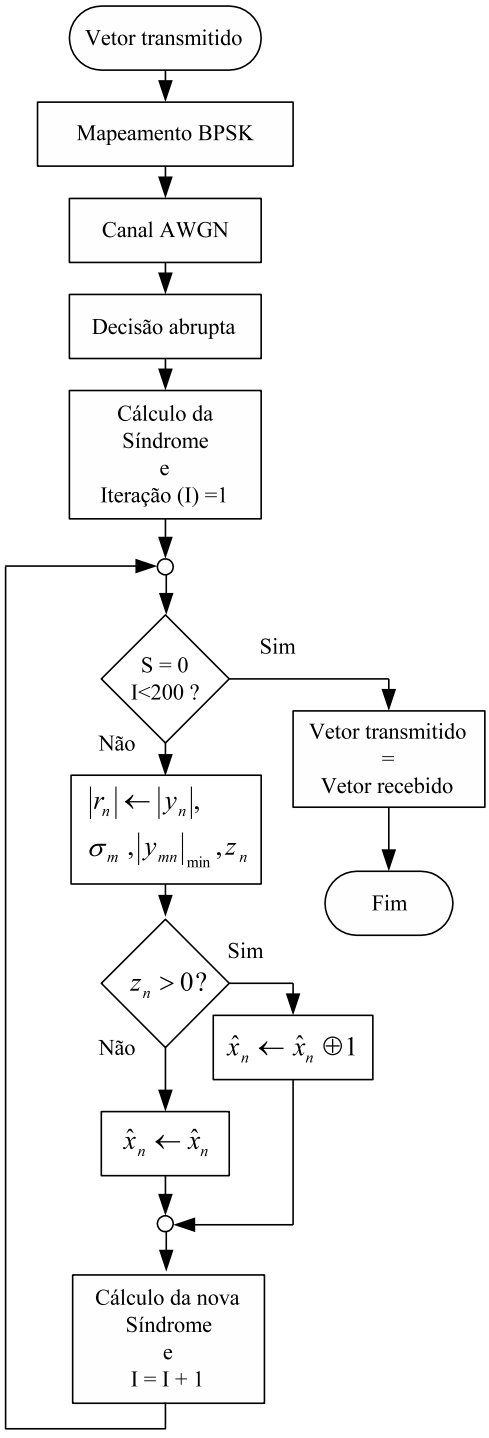

Figura 4.2: *Fluxograma que representa o algoritmo APP.*

A terceira etapa do algoritmo *BP-Based* ocorre quando, para cada nó de bit, o valor de z_n é testado. Quando z_n é menor ou igual a zero, o valor de $\hat{x}_n$ é igualado a $\hat{x}_n = \hat{x}_n^r \oplus 1$ e se z_n for maior que zero, faz-se $\hat{x}_n = \hat{x}_n^r$. Cria-se então o vetor $\mathbf{\hat{x}_n} = [\hat{x}_{mn}]$, sendo que $\hat{x}_{mn} = \hat{x}_n^r$ se z_{mn} for maior do que zero, e $\hat{x}_{mn} = \hat{x}_n^r \oplus 1$ caso contrário.

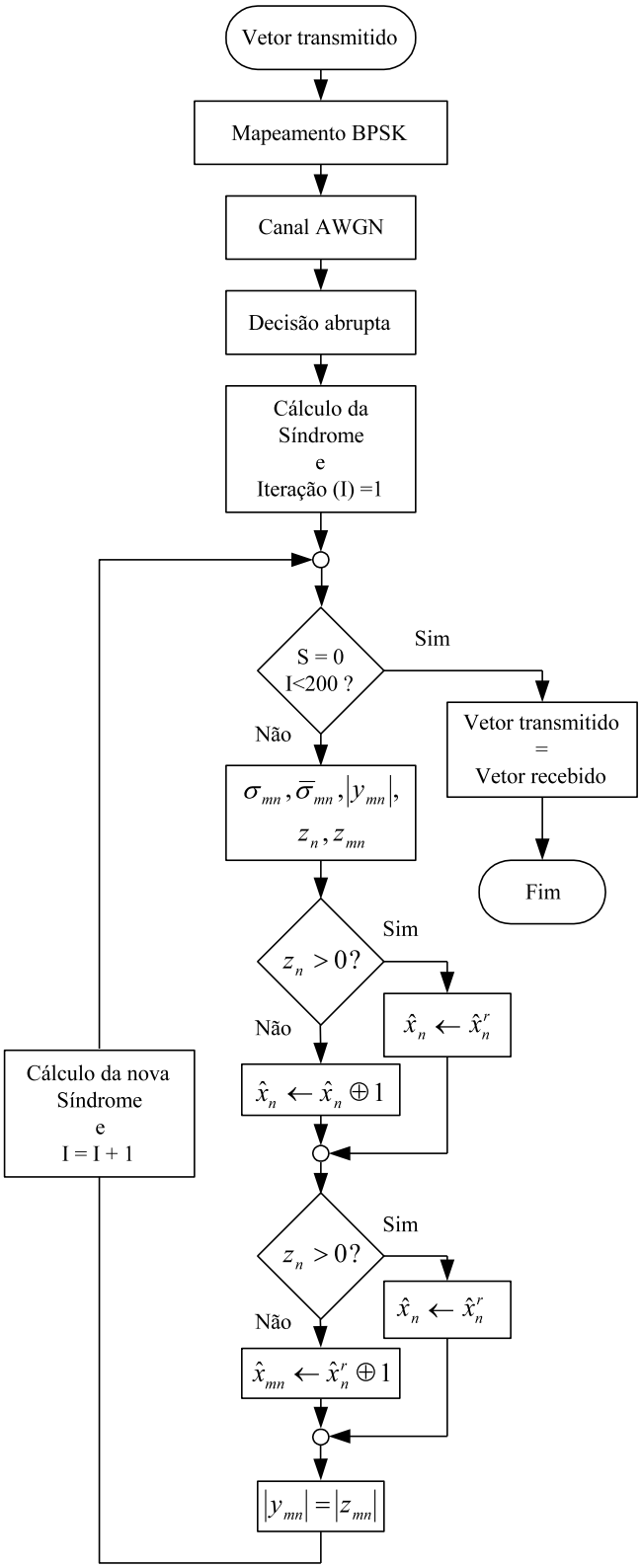

Figura 4.3: *Fluxograma que representa o algoritmo BP-Based.*

A partir da determinação de $\hat{\mathbf{x}}_\mathbf{n}$, calcula-se a nova síndrome por meio da matriz de verificação de paridade $\mathbf{H}$, dando início ao processo de conclusão do algoritmo.

Caso o valor da síndrome seja nulo, i.e., $\mathbf{H}.\hat{\mathbf{x}} = \mathbf{0}$, o vetor $\hat{\mathbf{x}}_\mathbf{n}$ é considerado válido, ou seja, o vetor é considerado decodificado. Caso a síndrome não seja nula, inicia-se novamente processo iterativo, que ocorrerá de acordo com um número máximo de iterações pré-estabelecido.

A Figura 4.3 apresenta o fluxograma com todas as etapas do algoritmo de decodificação BPB. A partir deste fluxograma foram implementadas as simulações computacionais apresentadas no Capítulo 5.

Capítulo 5

Implementação dos Algoritmos e Apresentação de Resultados

5.1 Introdução

O objetivo principal deste capítulo é apresentar o desempenho comparativo dos algoritmos de decodificação para códigos LDPC, apresentados no capítulo anterior. Como este trabalho foi iniciado no âmbito do projeto MI-SBTVD, a ferramenta de simulação deveria apresentar facilidades que permitissem uma rápida implementação do algoritmo de decodificação escolhido direto para o *hardware*. Desta forma, os algoritmos foram implementados diretamente em Matlab, que após algumas adaptações pode ser lido pelo software Simulink, que facilita a concepção, otimização e validação de algoritmos em um ambiente de simulações de sistemas bastante rico em recursos de biblioteca em praticamente todas as áreas do conhecimento, especialmente telecomunicações e processamento de sinais. O Simulink possibilita inclusive, através de bibliotecas disponibilizadas por terceiros, implementação de componentes ou projetos para FPGA. No projeto MI-SBTVD foram utilizados os pacotes System Generator para FPGA Xilinx e DSP Builder para FPGA da Altera. Para a comparação do desempenho dos algoritmos de decodificação foram utilizadas matrizes com pequenos valores de n para facilitar a verificação da confiabilidade dos resultados e em seguida com comprimento n maiores do que 500 bits, para comparação do desempenho.

5.2 Simulador

O sistema é composto basicamente pelos blocos apresentados nas seções seguintes, de acordo com cada algoritmo de decodificação LPDC e também por um bloco de decodificação abrupta, ou decisão *hard*.

5.2.1 Gerador de Bits Aleatórios

O bloco Gerador de Bits Aleatórios é o primeiro bloco do sistema e tem como função a geração de bits uniformemente distribuídos entre $[0, M-1]$ onde, para este caso, M vale 2. Pode-se acrescentar os bits da mensagem na configuração do bloco, no parâmetro *samples per frame*, que será o número de bits m da mensagem de saída.

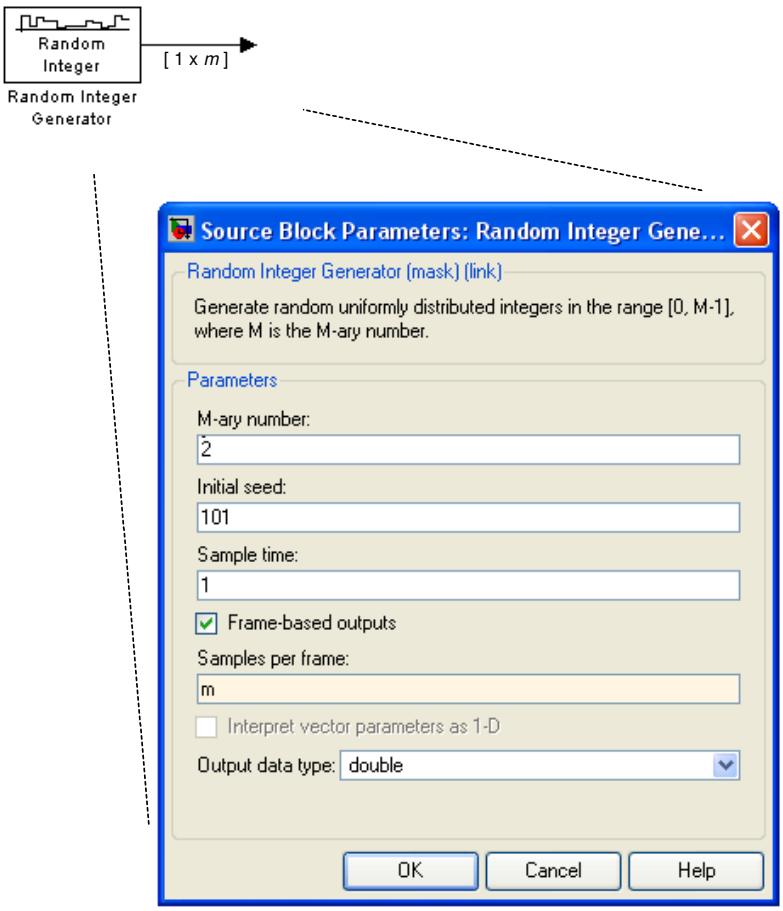

Figura 5.1: *Gerador de bits aleatórios*

5.2.2 Codificador LDPC

O bloco Codificador LDPC foi gerado através da ferramenta *Level-2 M-file S-Function*, que possibilita a criação de novos blocos utilizando a linguagem *M* do software Matlab. Algumas regras iniciais devem ser obedecidas, como indicação do tamanho da mensagem de entrada e saída do bloco, bem como a utilização de variáveis globais do sistema.

Para esse bloco sistêmico implementado e apresentado na Figura 5.2, a mensagem na entrada do bloco tem as dimensões [1 x m], onde m é o tamanho da mensagem e, como saída, o bloco apresenta a palavra-código, de dimensões [1 x n], onde n é o número de bits da palavra código gerada como o bloco implementado de acordo com o Capítulo 3.

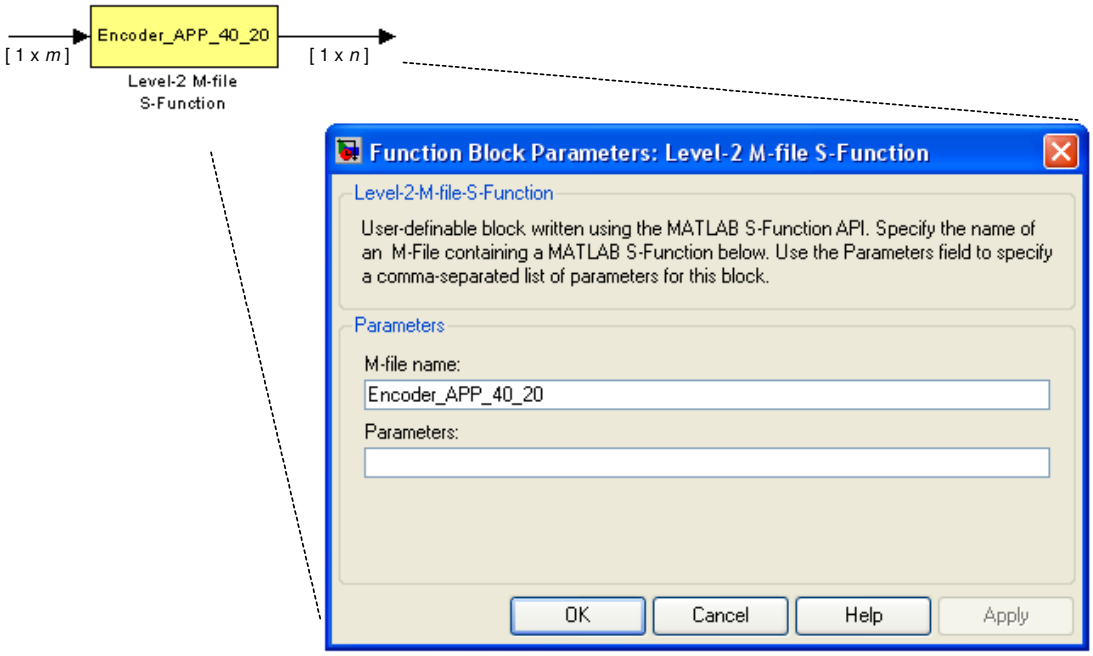

Figura 5.2: *Codificador LDPC*

5.2.3 Mapeamento BPSK

O bloco Mapeador é apresentado na Figura 5.3. Este bloco é usado para mapear o sinal de acordo com a modulação BPSK (Binary Shift Keying). Dentro da máscara do bloco do mapeador encontra-se o bloco de modulação QAM (Quadrature Amplitude Modulation) que modula o sinal em fase e quadratura. Na página de configuração do bloco, pode-se encontrar o parâmetro "Signal Constallation", que define os pontos da constelação, sendo que para o caso do mapeamento BPSK são definidos os pontos -1 e 1 para a constelação.

Como parâmetros de entrada e saída deste bloco tem-se a mensagem codificada em bits e a mensagem codificada mapeada (em -1 e 1), ambas de comprimento iguais a n.

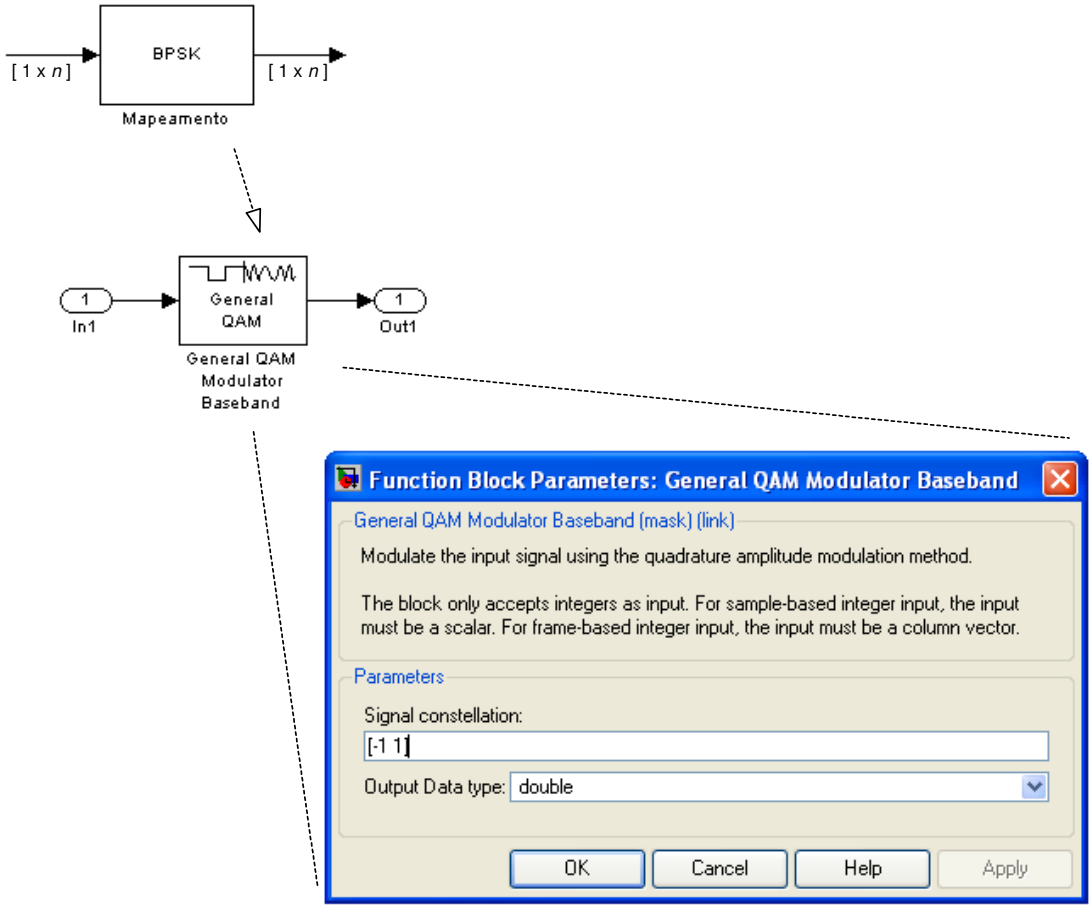

Figura 5.3: *Mapeamento BPSK*

5.2.4 Canal AWGN

O bloco seguinte ao mapeamento BPSK é o bloco que representa o canal de ruído gaussiano branco aditivo ou canal AWGN. O bloco do canal AWGN adiciona ruído gaussiano branco às entradas real ou complexa do sinal [1].

Este bloco possui duas entradas, sendo uma delas a entrada do sinal mapeado, de comprimento [1 x n] e a outra entrada apresenta a variância do ruído gaussiano branco que será adicionado ao sinal. Na Figura 5.4 pode-se notar o parâmetro de modificação do bloco da variância, apresentado pela seguinte expressão:

$$\sigma^2 = \frac{1}{\left(\frac{E_b}{N_0} \times R\right)} \tag{5.1}$$

O valor da variância calculado é utilizado com uma das entradas do bloco do canal AWGN, simulando assim um canal real que degradará o sinal mapeado.

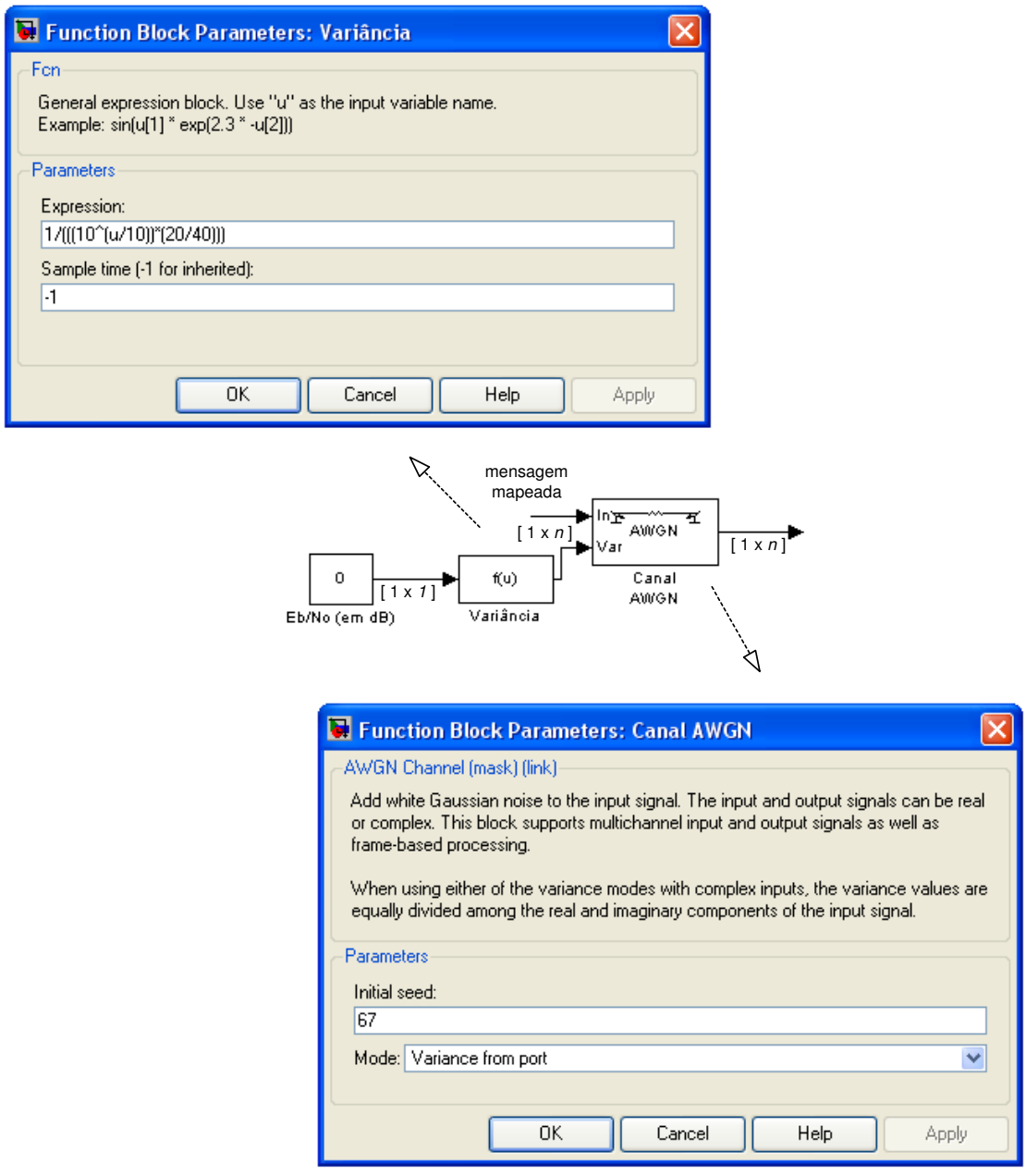

Figura 5.4: *Canal AWGN*

Como sinal de saída do bloco tem-se o vetor recebido, ou vetor **r**, que é o sinal com adição de ruído de tamanho [1 x n] que será enviado à entrada do decodificador LDPC.

5.2.5 Decodificador LDPC

O próximo bloco do sistema representa o decodificador LDPC. Esse bloco possui duas entradas sendo a primeira o sinal codificado, antes de passar pelo bloco do mapeamento BPSK; e a segunda entrada é a entrada do sinal recebido após a saída do canal AWGN.

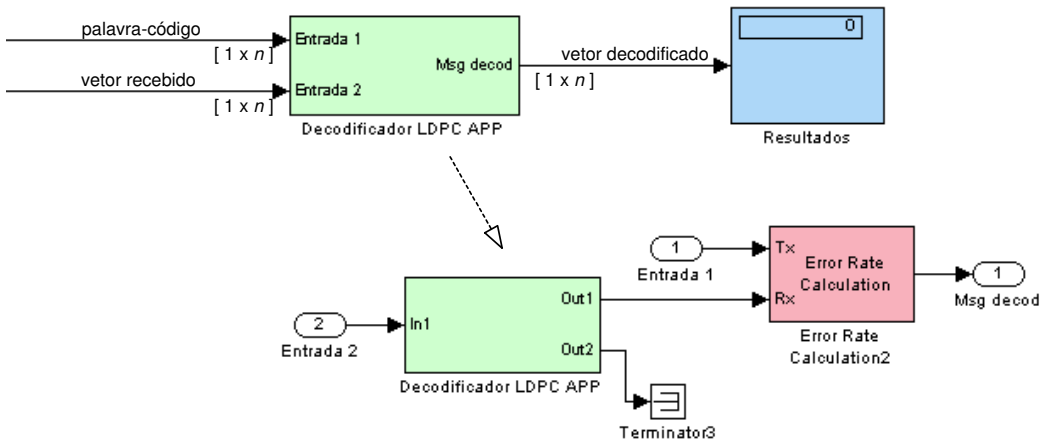

Figura 5.5: *Decodificador LDPC*

Na Figura 5.5 nota-se o bloco do decodificador, e sob a máscara é possível encontrar os blocos "decodificador LDPC"e "Error Rate Calcutator".

O bloco do decodificador LDPC é um bloco gerado da mesma maneira que o bloco do codificador, descrito em 5.2.2, utilizando a ferramenta *Level-2 M-file S-Function* para que o algoritmo implementado em linguagem do Matlab fosse utilizada no software Simulink. Após o bloco do decodificador, tem-se o bloco que fará o cálculo da taxa de erro do sistema, através da comparação entre as duas entradas no bloco Error Rate Calculator, que são respectivamente, a palavra-código, e o vetor de saída do bloco do decodificador LDPC. Assim na tabela de apresentação de resultados serão apresentados o valor da taxa de erro de bit (BER), o número de bits errados entre a palavra-código e o vetor decodificador e o tempo de simulação ajustado para o funcionamento do simulador.

5.3 Ajuste dos parâmetros iniciais e condições de simulação

Após a implementação do simulador foi necessário realizar o ajuste dos parâmetros iniciais, i. e., realizar a inicialização do sistema com as variáveis que são globais para todos os blocos. A inicialização desses parâmetros é realizada no Simulink em File > Model Properties > Callbacks > Model Pre-load functions, onde são incluídos os seguintes parâmetros:

1. **H**, matriz de verificação de paridade, que é adicionada no sistema através de $H = load('H(n).mat')$ onde n é o número de bits da palavra-código
2. M, que é o número de linhas da matriz **H**, calculado por $M = size(H, 1)$
3. N, que é o número de colunas da matriz **H**, calculado por $N = size(H, 2)$
4. K, que representa o número de bits de redundância e que é calculado por $K = N - M$

Esses parâmetros ficam disponíveis no sistema e, conseqüentemente, nas máscaras de todos os blocos do simulador LDPC apresentado na Figura 5.6.

Com a implementação do simulador pôde-se analisar o desempenho sistêmico utilizando cada um dos algoritmos de decodificação do capítulo 3 e, para essa análise, foram utilizadas as matrizes apresentada pela Tabela 5.1.

Tabela 5.1: *Tabela de matrizes utilizadas para análise do sistema implementado*

Matriz	**N**	**K**	$\mathbf{t_c}$	$\mathbf{t_r}$
40 x 20	40	20	3	6
204 x 102	204	102	3	6
504 x 252	504	504	3	6
816 x 408	816	816	3	6
1008 x 504	1008	504	3	6

Além da utilização dos dados dos códigos baseados nas matrizes da Tabela 5.1, o valor de E_b/N_0 é um parâmetro de entrada do sistema já que influencia diretamente no sinal após o canal AWGN. Essa variável de entrada terá no sistema uma gama de valores que variam de 0 $[dB]$ a 7 $[dB]$, com passo de $0,5$ $[dB]$. Logo, variando os valores de E_b/N_0 no sistema é possível obter os valores da BER - *Bit Error Rate* e, conseqüentemente, após simulados todos os valores de E_b/N_0 para cada matriz da Tabela 5.1, apresentá-las em forma de gráfico da BER versus E_b/N_0 em $[dB]$.

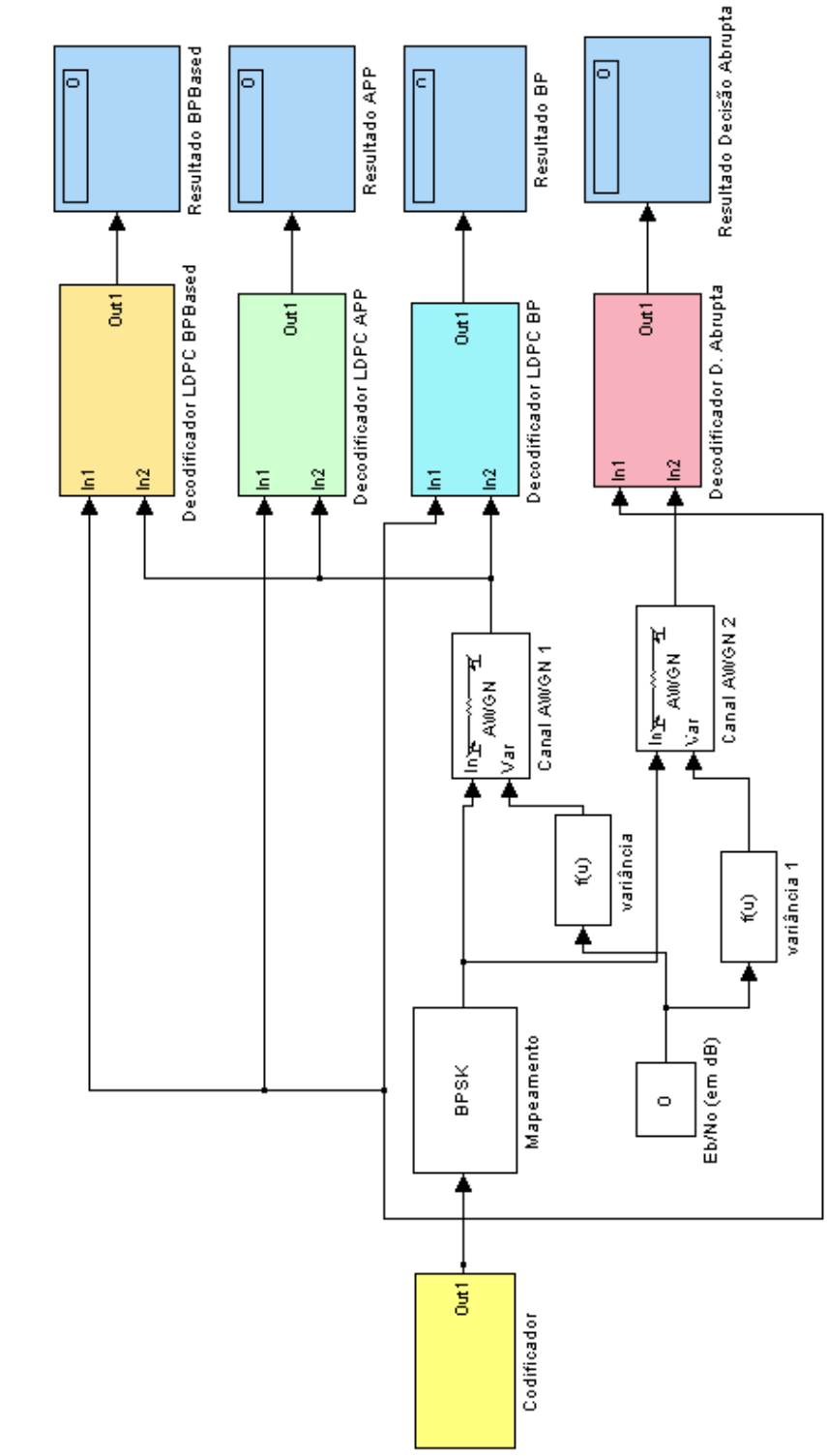

Figura 5.6: *Simulador LDPC*

5.4 Análise de desempenho do sistema

As próximas seções apresentam os resultados obtidos através dos dados gerados pelo simulador da Figura 5.6, desenvolvido para os códigos definidos na Tabela

5.1.

5.4.1 Desempenho baseado no comprimento do código

Primeiramente utilizou-se a matriz **H**(40,20) para a verificação da confiabilidade dos resultados. As características desta matriz encontram-se na primeira linha da Tabela 5.1, e esses dados serão utilizados como parâmetros iniciais do sistema. Assim, a matriz base deste sistema será a matriz $H(40).mat$, que tem o comprimento $n = 40$ e tamanho da mensagem $m = 20$.

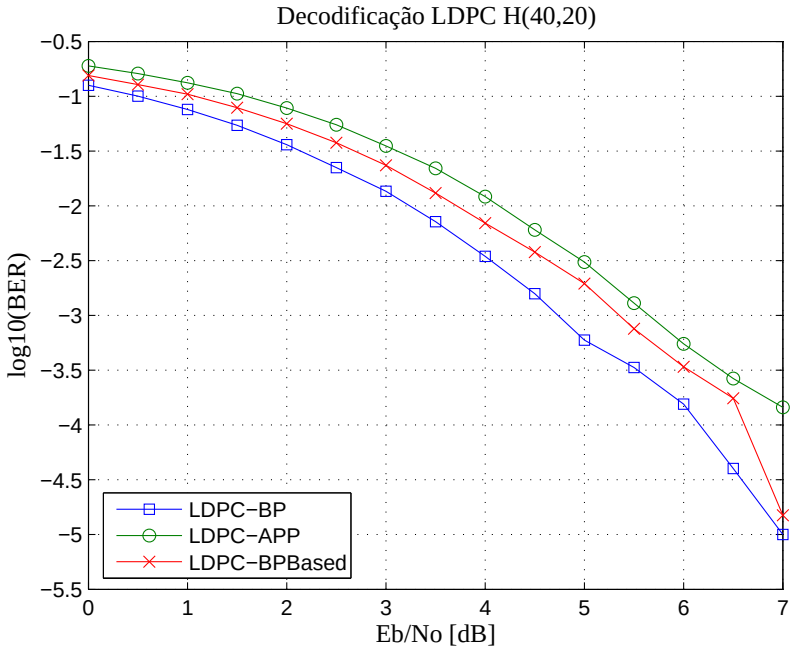

Figura 5.7: *Desempenho do Simulador usando a matriz H(40, 20)*

Para a matriz **H**(40,20), obteve-se o gráfico de desempenho apresentado na Figura 5.7, onde tem-se as curvas geradas utilizando os algoritmos *Belief-propagation - BP*, *A Posteriori Probability - APP* e *Based on Belief-Propagation - BPB*. Analisando as curvas dos algoritmos para o ponto onde $log_{10}(BER)$ vale -2, é possível notar que, para o algoritmo *BP* a curva está $0.5[dB]$ a baixo do mesmo ponto para a curva do algoritmo *BPBased*.

Da mesma forma acontece para o algoritmo *BPB*, que está $0.5[dB]$ a baixo da curva do algoritmo *APP*. Assim, para $log_{10}(BER) = -2$, tem-se os resultados 3.25 $[dB]$, 3.75 $[dB]$ e 4.25 $[dB]$, respectivamente.

Alterando a matriz de verificação de paridade do código para uma matriz onde o comprimento da palavra-código é maior, como a matriz **H**(204,102), chega-se às curvas apresentadas na Figura 5.8.

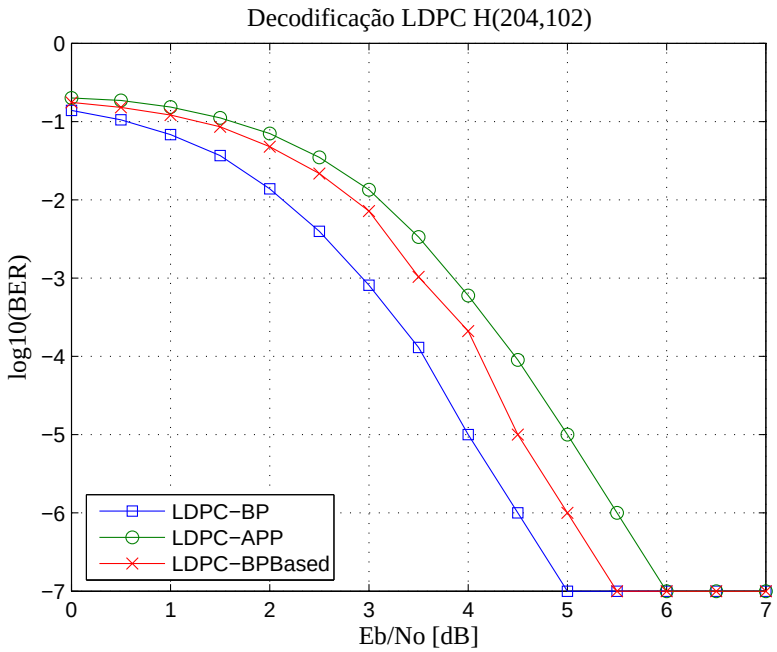

Figura 5.8: *Desempenho do Simulador usando a matriz H(204, 102)*

Da mesma forma que na Figura 5.7, alguns pontos são tomados como base para análise, porém, para as curvas de desempenho na Figura 5.8, serão destacados valores de E_b/N_0. Analisando a curva do algoritmo *BP* para o valor de $E_b/N_0 = 3.25[dB]$, chega-se a $log_{10}(BER) = -3.5$. De forma análoga, faz-se, para o algoritmo *BPB*, utilizando o valor de $E_b/N_0 = 3.75[dB]$, obtém-se $log_{10}(BER) = -3.3$; e para o algoritmo *APP* com o valor de $E_b/N_0 = 4.25[dB]$, tem-se $log_{10}(BER) = -3.5$. Novamente, observa-se diferença de $0,5dB$ entre os algoritmos.

Comparando-se os dados coletados para **H**(40,20) e **H**(204,102) e utilizando o ponto onde $log_{10}(BER) = -4$ como ponto para comparação, chega-se a Tabela 5.2. Pode-se notar que, para o algoritmo *BP*, teve-se uma redução de $0.5[dB]$ comparando-se as duas matrizes geradas. E da mesma forma pode-se comparar os algoritmos *BPBased* e *APP*. Nota-se na Figura 5.8 que para uma mesma matriz, o algoritmo *BP* tem melhor desempenho que os algoritmos simplificados.

Tabela 5.2: *Tabela de comparação entre às matrizes H(40,20) e H(204,102) para* $log_{10}(BER) = -4$

Algoritmo	**H(40,20)**	**H(204,102)**
BP	6.2 [dB]	3.5 [dB]
BPBased	6.7 [dB]	4 [dB]
APP	7.75 [dB]	4.5 [dB]

Assim, faz-se necessário utilizar matrizes de comprimento n maior, como a matriz **H**(504,252), para comprovar que quanto maior o tamanho n do código, melhor o desempenho do sistema apresentado na Figura 5.6, chegando-se assim às curvas de desempenho da Figura 5.9.

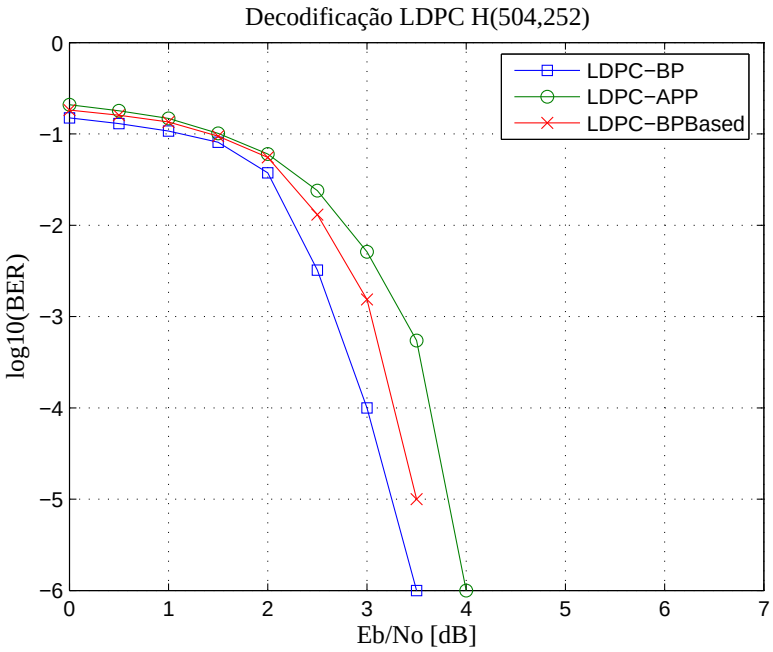

Figura 5.9: *Desempenho do Simulador usando a matriz H(504, 252)*

Observando-se o eixo onde $log_{10}(BER) = -4$, tem-se para os algoritmos *BP, BPBased e APP* os valores de E_b/N_0 com respectivamente $E_b/N_0 = 3[dB]$, $E_b/N_0 = 3.25[dB]$ e $E_b/N_0 = 4.5[dB]$.

De forma análoga às Figuras 5.7, 5.8 e 5.9 foram geradas as curvas de desempenho para os códigos **H**(816,408) e **H**(1008,504), utilizando as matrizes de verificação de paridade $H(816).mat$ e $H(1008).mat$ no simulador. Como resultado foi possível obter os valores da probabilidade de erro na decodificação para a gama de valores de E_b/N_0 entre 0 $[dB]$ e 7 $[dB]$, conforme apresentado nas Tabelas 5.3 e 5.4.

Analisando as curvas de desempenho dos códigos **H**(816,408) e **H**(1008,504) nota-se que o código com maior comprimento n possui o melhor desempenho no sistema simulado.

Na Tabela 5.4 encontram-se os valores gerados pelo simulador para **H**(1008,504), e que também são apresentados na Figura 5.11.

As matrizes utilizadas para a geração das curvas de desempenho para o algoritmo *Belif-propagation* e os algoritmos simplicados *A Posteriori Propagation* e *Based on Belief-propagation* foram baseadas nas matrizes disponibilizadas por Mackay em [18].

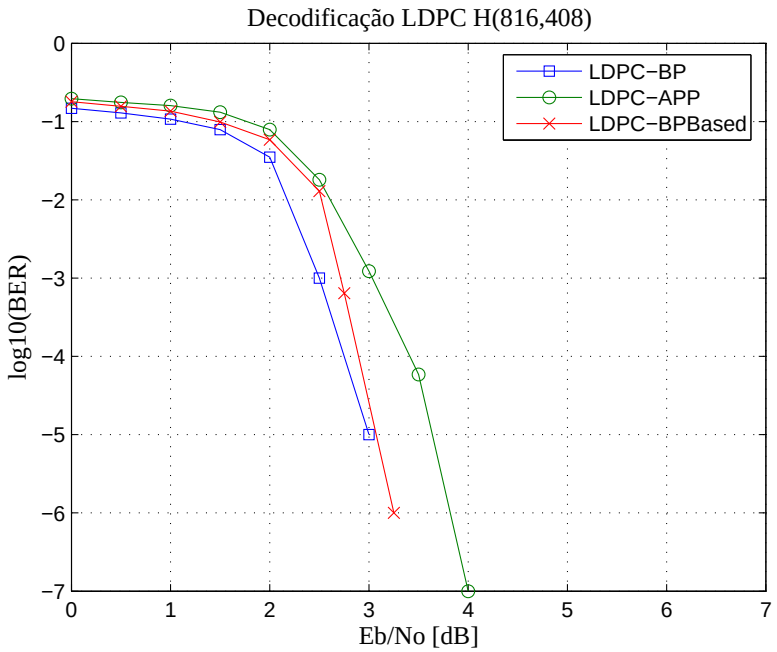

Figura 5.10: *Desempenho do Simulador usando a matriz H(816, 408)*

Tabela 5.3: *Tabela de valores da Taxa de Erro de Bit(BER) para o código H(816,408)*

E_b/N_0	*BP*	*APP*	*BPBased*
0	0,1477	0,1957	0,1797
0,5	0,1292	0,1752	0,1566
1	0,1073	0,1599	0,136
1,5	0,07887	0,132	0,09877
2	0,0351	0,07889	0,05843
2,5	0,0001	0,01804	0,1294
3	0,000001	0,001225	0,0006422
3,5	0	0,00005882	0,0000001
4	0	0,0000001	0

Com posse dos valores das probabilidades de erro encontrados para cada um dos códigos propostos por Mackay em [18] e apresentados na Tabela 5.1, é possível comparar os algoritmos simulados e como o comprimento do código influencia diretamente o desempenho do sistema. Os resultados desta análise são apresentados na Seção 5.4.2.

5.4.2 Desempenho baseado no Algoritmo de Decodificação

Nesta seção são apresentados os gráficos com as curvas de desempenho, levando em consideração o algoritmo utilizado, e comparando as curvas geradas para os

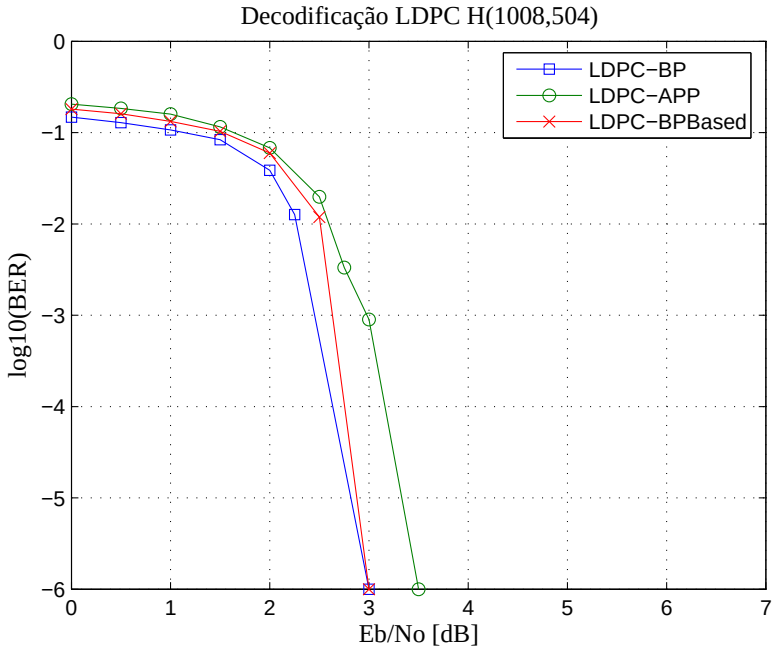

Figura 5.11: *Desempenho do Simulador usando a matriz H(1008, 504)*

Tabela 5.4: *Tabela de valores da Taxa de Erro de Bit(BER) para o código H(1008,504)*

E_b/N_0	*BP*	*APP*	*BPBased*
0	0,1479	0,2047	0,1811
0,5	0,1283	0,1838	0,1607
1	0,1069	0,1596	0,1325
1,5	0,08373	0,1155	0,1034
2	0,3859	0,06815	0,05977
2,5	0,1265	0,1984	0,01186
3	0,000001	0,0008978	0,00001
3,5	0,0000001	0,000001	0,000001

códigos LDPC da Tabela 5.4.

O primeiro algoritmo a ser analisado é o Algoritmo *Belief Propagation*, baseado no grafo de Tanner e onde a troca de valores de probabilidade, entre os nós de bit e nós de verificação, é a principal característica do algoritmo [2].

De acordo com a Figura 5.12, e comparando as curvas simuladas, nota-se que a curva gerada através do código LDPC $\mathbf{H}(40,20)$ é a curva que tem o pior desempenho, comparadas com os demais códigos utilizados. Para o ponto do gráfico onde $log_{10}(BER) = -4$, o valor de E_b/N_0 na curva do código $\mathbf{H}(40,20)$ é de 6 $[dB]$. Já para um código onde n é de 504 bits, tem-se $E_b/N_0 = 3[dB]$, i.e., uma diferença de 3 $[dB]$ no desempenho do sistema. Para a curva utilizada com maior comprimento $n = 1008$ bits, e com o mesmo valor de $log_{10}(BER)$

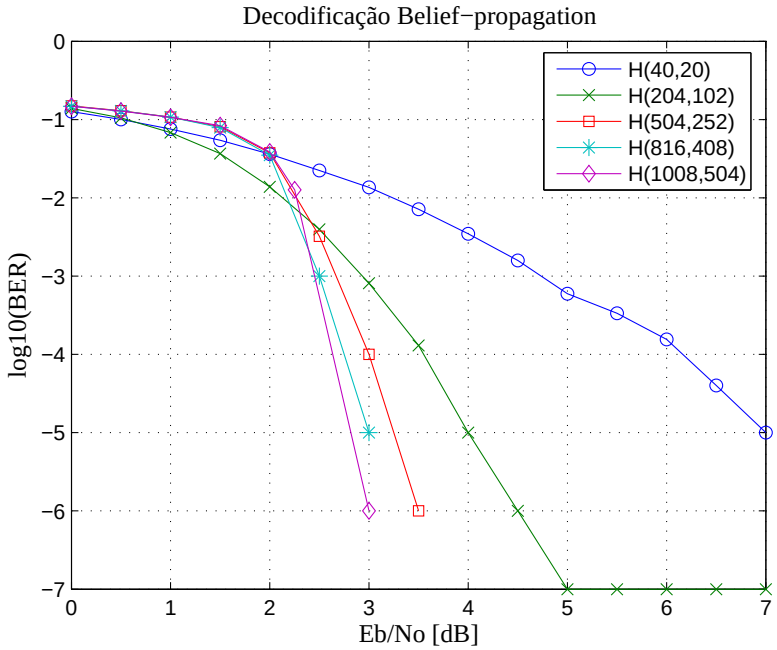

Figura 5.12: *Desempenho do Simulador usando o algoritmo Belief-propagation*

citado acima, chega-se ao valor de $E_b/N_0 = 2.75[dB]$, logo, desempenho de 2.8 $[dB]$ melhor que o menor código utilizado.

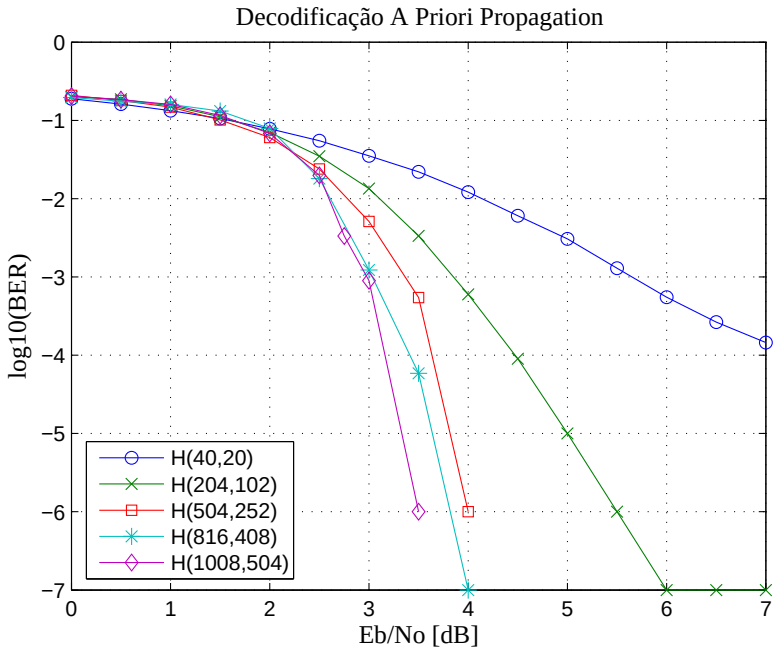

Figura 5.13: *Desempenho do Simulador usando o algoritmo A Priori Propagation*

Nas Figuras 5.13 e 5.14 encontram-se as curvas geradas para os algoritmos simplicados *Based on Belief-propagation* e *A Posteriori Probability*. Conforme as

Seções 4.2.2 e 4.2.3 do Capítulo 4, esses algoritmos simplificados se baseiam na troca de valores binários entre as ligações dos nós de bit e nós de verificação.

Através da análise do desempenho desses algoritmos, pode-se notar que a diferença entre os códigos simplificados e o código base (*Belief − propagation*) é de aproximadamente 0.5 $[dB]$. Levando-se em consideração que para os algoritmos simplificados a análise computacional é mais veloz, a perda de desempenho citada acima pode ser aceitável.

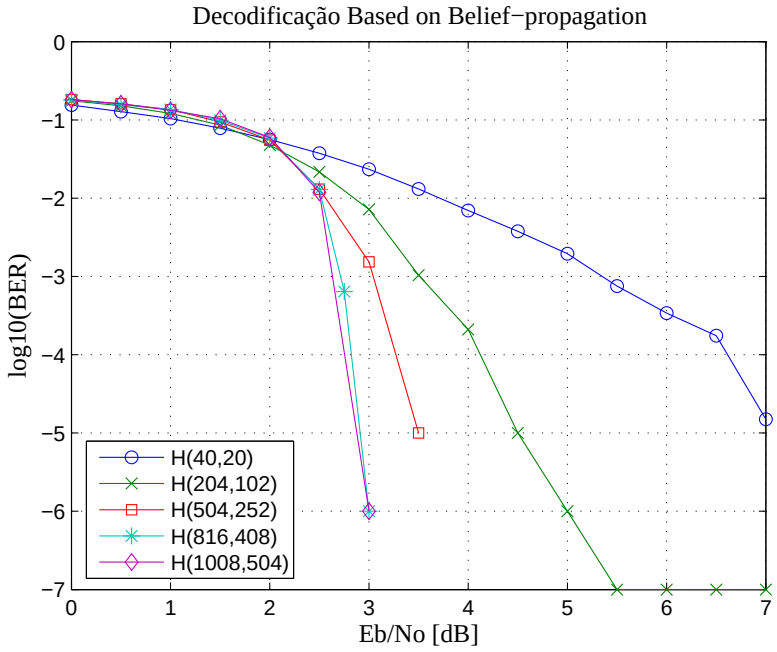

Figura 5.14: *Desempenho do Simulador usando o algoritmo Based on Belief Probability*

Na Figura 5.13 são expostas as curvas de $log_{10}(BER)$ x E_b/N_0 para o algoritmo *A Posteriori Probability* para as diferentes matrizes de verificação de paridade LDPC utilizadas no simulador. Igualmente ocorrido para algoritmo *Belief-propagation* da Figura 5.12, o código com maior comprimento de bloco foi o que teve melhor desempenho para este algoritmo. Para o algoritmo simplificado *Based on Belief-propagation*, onde a troca de bit é realizada entre os nós de bit e nós de verificação, apresentou-se os resultados de desempenho apresentada na Figura 5.14. De acordo com a figura apresentados, a curva gerada para o algoritmo com menor comprimento de bloco, **H**(40,20), é a curva com pior desempenho para o algoritmo simplificado. Conforme o comprimento n é aumentado, o desempenho do sistema melhora, até que, para os códigos **H**(816,408) e **H**(1008,504) as curvas de desempenho estão praticamente sobrepostas.

5.5 Conclusão

Neste capítulo foi apresentado o Simulador Codificador-Decodificador LDPC, onde todos os blocos do sistema foram descritos detalhamente, bem como as suas implementações na ferramenta de simulação Simulink. Este simulador foi baseado nos algoritmos determinados no capítulo 4 e para cada um dos algoritmos foi criado um bloco no simulador. Na segunda parte do Capítulo 5 foi apresentado o desempenho do sistema para códigos LDPC com diferentes comprimentos de palavra-código, possibilitando a análise e comparação baseando-se primeiramente nas matrizes para os algoritmos e, em seguida, comparando para cada algoritmo qual é a matriz mais adequada.

Capítulo 6

Conclusão

6.1 Análise dos Resultados

Para atingir os objetivos propostos neste trabalho, é importante salientar as bases teóricas e matemáticas contidas em trabalhos anteriormente desenvolvidos por Gallager em [2], onde toda a definição dos códigos de verificação de paridade de baixa densidade, ou códigos LDPC, foi definida e apresentada, e por Fossorier no artigo [6], onde apresentou-se os algoritmos simplicados utilizados nesse trabalho. As matrizes utilizadas no desenvolvimento e simulação desse trabalho foram projetadas por Mackay e estão disponíveis em [10] e [18].

Com o Simulador LDPC, desenvolvido neste trabalho, foi possível notar que o primeiro algoritmo desenvolvido para decodificação LDPC, o algoritmo *Belief-propagation*, tem o melhor desempenho tanto para matrizes de códigos pequenos, quanto para matrizes grandes com o comprimento do bloco maior que 500 bits. Porém este algoritmo demanda um tempo de processamento da simulação muito grande tornando-o inviável quando uma matriz de comprimento muito grande é utilizada.

Em contra partida, tem-se os algoritmos simplificados *A Posteriori Probability* e *Based on Belief-propagation*. De acordo com os resultados apresentados no Capítulo 5 esses algoritmos simplificados apresentam uma queda de aproximadamente 0.5 $[dB]$ quando comparados ao algoritmo *Belief-propagation*. A utilização dos algoritmos simplificados no sistema é justificada pela vantagem da execução do processo iterativo nesses algoritmos ser muito superior com relação à velocidade de processamento do algoritmo *Belief-propagation*.

Assim, pode-se optar por ter uma perda de desempenho de 0.5 $[dB]$ que será compensada pelo menor tempo de execução do processo iterativo de decodificação LDPC através dos algoritmos simplificados, podendo viabilizar em muitos casos uma implementação prática para palavras códigos de comprimentos maiores.

6.2 Contribuições

Entre as contribuições deste trabalho podem ser destacadas:

- Desenvolvimento de um simulador para validação e teste de algoritmos de decodificação LDPC;
- Análise comparativa de desempenho entre algoritmos de decodificação rápida, tendo por referência o trdicional algoritmo *Belief-propagation*.
- Finalmente, a própria dissertação que pode ser uma fonte útil para pesquisadores que desejam iniciar estudos sobre decodificação de códigos LDPC.

Desta forma pode-se notar através da Tabela 6.1 que o algoritmo mais simples, BPBased, tem desempenho melhor que o algoritmo APP para códigos de qualquer comprimento. Quando compara-se o algoritmo BPBased ao algoritmo BP percebe-se que o algoritmo simplificado tem desempenho intermediário para matrizes pequenas e que, quando o comprimento do código é aumentado sendo $n > 500$ o desempenho do algoritmo é equivalente ao desempenho do algoritmo BP para valores de E_b/N_0 que correspondem a taxa de erro interior a 10^{-5}.

Tabela 6.1: *Tabela de desempenho comparativo*

	BP	*APP*	*BPBased*
Desempenho para $n < 500$	melhor	pior	intermediário
Desempenho para $n > 500$ e $E_b/N_0 =>$ BER=10^{-5}	melhor	pior	equivalente ao BP
Complexidade	maior	intermediária	menor

6.3 Propostas para trabalhos futuros

Existem algumas sugestões para futuros trabalhos com objetivo de dar continuidade a esta dissertação.

Recomenda-se o estudo dos algoritmos simplificados para matrizes utilizando códigos LDPC irregulares já que a linha de estudo seguida apresenta análise somente para códigos regulares.

Propõe-se a busca de soluções para implementação e melhora no processamento para códigos longos quando faz-se necessária a utilização de algoritmos simplificados.

Anexo A

Sumário Matemático

Este anexo apresenta as principais equações encontradas nesta dissertação.

$$\mathbf{c} = \left[\begin{array}{c|c} \mathbf{b} & \mathbf{m} \end{array} \right] \tag{A.1}$$

$$\mathbf{H^T} = \begin{bmatrix} \mathbf{H_1} \\ \cdots \\ \mathbf{H_2} \end{bmatrix} \tag{A.2}$$

$$\mathbf{cH^T} = \mathbf{mGH^T} = 0 \tag{A.3}$$

$$\mathbf{b} = \mathbf{mP} \tag{A.4}$$

$$\mathbf{G} = \left[\begin{array}{c|c} \mathbf{P} & \mathbf{I_k} \end{array} \right] = \left[\begin{array}{c|c} \mathbf{H_2 H_1^{-1}} & \mathbf{I_k} \end{array} \right] \tag{A.5}$$

$$\mathbf{c} = \mathbf{m}.\mathbf{G} \tag{A.6}$$

$$\delta r_{mn} = \prod_{n' \in N(m) \backslash n} \delta q_{mn'} \tag{A.7}$$

$$\delta q_{mn}^{x} = \alpha_{mn}.f_{n}^{x} \prod_{m' \in M(n) \backslash m} {r^{x}}_{m'n} \tag{A.8}$$

$$q_{n}^{x} = \alpha_{n}.f_{n}^{x} \prod_{m' \in M(n)} {r^{x}}_{m'n} \tag{A.9}$$

$$|\mathbf{r_n}| = |\mathbf{y_n}| \tag{A.10}$$

$$z_n = |r_n| + \sum_{m' \in M(n)} (\bar{\sigma}_m - \sigma_m)|y_{mn}|_{min} \tag{A.11}$$

$$\sigma_{mn} = \hat{x}_n^r \oplus \Big(\sum_{n' \in N(m) \backslash n} \hat{x}_{mn}[\mathrm{mod} - 2] \Big) \tag{A.12}$$

$$z_{mn} = |r_n| + \sum_{m' \in M(n)} (\bar{\sigma}_{m'n} - \sigma_{m'n})|y_{m'n}|_{min} \tag{A.13}$$

$$\sigma^2 = \frac{1}{\left(\frac{E_b}{N_0} \times R\right)} \tag{A.14}$$

Anexo B

Algoritmos Implementados

Este anexo apresenta os algoritmos implementados em Matlab.

```
function Decoder1008_BP(block)
% Level-2 M file S-Function for times two demo.
%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.1.6.1 $

  setup(block);

%endfunction
%==============================================================================
function setup(block)

  %% Register number of input and output ports
  block.NumInputPorts  = 1;
  block.NumOutputPorts = 1;

  %% Setup functional port properties to dynamically
  %% inherited.
  block.SetPreCompInpPortInfoToDynamic;
  block.SetPreCompOutPortInfoToDynamic;

  % Override output ports properties
  block.OutputPort(1).DatatypeID  = 0; % double
  block.OutputPort(1).Dimensions  = [1008 1];
  %block.OutputPort(1).Complexity  = 'Real';
  %block.OutputPort(1).Complexity  = 'Complex';

  % Override input port properties
```

```
  block.InputPort(1).DatatypeID  = 0;  % double
  block.InputPort(1).Dimensions = [1008 1];
  %block.InputPort(1).Complexity  = 'Real';
  %block.InputPort(1).Complexity  = 'Complex';

%   % Override input port properties
%   block.InputPort(2).DatatypeID  = 0;  % double
%   block.InputPort(2).Dimensions = [1 1];
%   %block.InputPort(1).Complexity  = 'Real';
%   %block.InputPort(1).Complexity  = 'Complex';

  %block.InputPort(1).DirectFeedthrough = true;

  %% Set block sample time to inherited
  block.SampleTimes = [-1 0];

  %% Run accelerator on TLC
  block.SetAccelRunOnTLC(true);

  %% Register methods
  block.RegBlockMethod('Outputs',                 @Output);
  block.RegBlockMethod('SetOutputPortDimensions', @SetOutPortDims);
  block.RegBlockMethod('SetInputPortDimensions', @SetInpPortDims);
  block.RegBlockMethod('SetInputPortSamplingMode', @SetInpPortFrameData);

   global Href;
   global Nref;
   global Kref;
   global Mref;

  Href = evalin('base', 'H');
  Nref = evalin('base', 'N');
  Kref = evalin('base','K');
  Mref = evalin('base', 'M');

%endfunction
%================================================================================
function Output(block)

x_recebido = block.InputPort(1).Data;
%variancia = block.InputPort(2).Data;
```

```
T = calcula(x_recebido);

block.OutputPort(1).Data = T;

%================================================================================
% Codificador LDPC

function T = calcula(x_recebido)

global Href; global Nref; global Kref; global Mref;

m=Mref; N=Nref; H=Href; K=Kref;

%--------------------------------------------------------------------------------
% Canal AWGN
%--------------------------------------------------------------------------------
testex=x_recebido; x_recebido=testex'; x = x_recebido;

%--------------------------------------------------------------------------------
% Decisão Hard
%--------------------------------------------------------------------------------

for colunax=1:1:N
    if x(colunax) > 0
        x_hard(colunax) = 1;
    else
        x_hard(colunax) = 0;
    end
end

%x_hard;

%--------------------------------------------------------------------------------
% Cálculo da Síndrome
%--------------------------------------------------------------------------------

for lin=1:1:m
        r(lin,:) = H(lin,:)& x_hard(1,:);
end
```

```
for lin=1:1: m
    col=1;
    R(lin,1)=xor(r(lin,col),r(lin,col+1));
    for col=3 : size(r,2)
         R(lin,1) = xor(R(lin,1),r(lin,col));
    end
end Sindrome = R;

%-------------------------------------------------------------------------
% Processo Iterativo
%-------------------------------------------------------------------------

if max(Sindrome)==0
    %display('vetor recebido = vetor transmitido')
    T=0;
    x_chapeu=x_hard;
    iteracao=0;
    iteracao_final=0;
else

% Inicializacao das variaveis

%-------------------------------------------------------------------------
% Calculo da FDP
%-------------------------------------------------------------------------

    variancia=1.002;
    a = -2.*x/variancia;
    fx0 = exp(a)./(1 + exp(a));             %fdp  fx1 = P(xn = 1) = qmn1
    fx1 = 1 - fx0;                      %fdp  fx0 = P(xn = 0) = qmn0

     for lin=1:1: size(H,1)
                qmn0(lin,: ) = fx0;
                qmn1(lin,: ) = fx1;
     end

%-------------------------------------------------------------------------
% Step 1: Calculo de delta_q, delta r, r1mn e r0mn
%-------------------------------------------------------------------------
        iteracao=0;
        iteracao_final = 0;
```

```
        while ((max(Sindrome) ~= 0) && (iteracao < 50))
%           Calculo de delta_q
            lin2=1:1:m;
            delta_q(lin2,: ) = qmn0(lin2,: ) - qmn1(lin2,: );

%           Calculo de delta_r
            for lin3=1:1:m
                for n=1:1:N
                    multN=1;
                    if H(lin3,n) ~= 0
                        for col=1:1:N
                            if n ~= col && H(lin3,col)~=0
                                multN=multN*delta_q(lin3,col);
                            end
                        end
                    else
                        multN=0;
                    end
                    delta_r(lin3,n)=multN;
                end
            end
            r1mn=(1/2).*(1-delta_r);
            r0mn=(1/2).*(1+delta_r);

%-------------------------------------------------------------------------
% Step 2
%-------------------------------------------------------------------------
            % Calculo do produtorio de r0 e r1 (excluindo m)
            for coluna=1:1:N
                for indice_linha=1:1:m
                    mult0_M=1;
                    mult1_M=1;
                    if H(indice_linha,coluna) ~= 0
                        for linha=1:1:m
                            if indice_linha ~= linha && H(linha,coluna)~=0
                                mult0_M=mult0_M*r0mn(linha,coluna);
                                mult1_M=mult1_M*r1mn(linha,coluna);
                            end
                        end
                    else
                        mult0_M=0;
```

```
            mult1_M=0;
        end
        produtM_r0(indice_linha,coluna)=mult0_M;
        produtM_r1(indice_linha,coluna)=mult1_M;
    end
end
for linhaq=1:1:m
    for colunaq=1:1:N
       if ((produtM_r0(linhaq,colunaq)==0) && ...
               (produtM_r1(linhaq,colunaq)==0))
            alfamn(linhaq,colunaq)=1;
            qmn0(linhaq,colunaq)=0;
            qmn1(linhaq,colunaq)=0;
       else
           divisao1(linhaq,colunaq)=produtM_r0(linhaq,colunaq);
           divisao2(linhaq,colunaq)=produtM_r0(linhaq,colunaq)...
               *fx1(1,colunaq);
           divisao3(linhaq,colunaq)=produtM_r1(linhaq,colunaq)*...
               fx1(1,colunaq);
           divisao4(linhaq,colunaq)=divisao1(linhaq,colunaq)-...
               divisao2(linhaq,colunaq)+divisao3(linhaq,colunaq);
           alfamn(linhaq,colunaq)=1/(divisao4(linhaq,colunaq));
           qmn0(linhaq,colunaq)=alfamn(linhaq,colunaq)*...
               produtM_r0(linhaq,colunaq)*fx0(1,colunaq);
           qmn1(linhaq,colunaq)=alfamn(linhaq,colunaq)*...
               produtM_r1(linhaq,colunaq)*fx1(1,colunaq);
       end
    end
end
%-----------------------------------------------------------------
% Calculo de qn0 e qn1
%-----------------------------------------------------------------
% Calculo do produtorio de r0 e r1
for coluna2=1:1:N
        mult20_M=1;
        mult21_M=1;
    for linha2=1:1:m
        if H(linha2,coluna2) ~= 0
            mult20_M=mult20_M*r0mn(linha2,coluna2);
            mult21_M=mult21_M*r1mn(linha2,coluna2);
        end
```

```
    end
    produt2M_r0(1,coluna2)=mult20_M;
    produt2M_r1(1,coluna2)=mult21_M;
end
for coluna3=1:1:N
   aux3(1,coluna3)=produt2M_r0(1,coluna3);
   auxiliar2(1,coluna3)=produt2M_r0(1,coluna3) - ...
       produt2M_r1(1,coluna3);
   aux4(1,coluna3)= fx1(1,coluna3)*auxiliar2(1,coluna3);
   aux5(1,coluna3) = aux3(1,coluna3) - aux4(1,coluna3);
   alfa_n(1,coluna3)=1/aux5(1,coluna3);
   qn1(1,coluna3)=alfa_n(1,coluna3)*fx1(1,coluna3)*...
       produt2M_r1(1,coluna3);
   if qn1(1,coluna3) == 0
        qn0(1,coluna3)=0;
   else
        qn0(1,coluna3)= 1-qn1(1, coluna3);
   end
   %------------------------------------------------------------
   % Step 3: Calculo de x_chapeu e a nova Sindrome
   %------------------------------------------------------------
qn0;
qn1;
end
for coluna4=1:1:N
    if qn1(1,coluna4)>0.5
        x_chapeu(1,coluna4)=1;
    else
        x_chapeu(1,coluna4)=0;
    end
end
qn1;
x_chapeu;
    for lin2=1:1:m
        for colunas2 = 1:1:N
            result(lin2,colunas2) = H(lin2,colunas2) & ...
                x_chapeu(1, colunas2);
        end
    end
    result;
    for lin2=1:1:m
```

```
                    Result(lin2,1)=xor(result(lin2,1),result(lin2,2));
                    for col2=3 : N
                         Result(lin2,1) = xor(Result(lin2,1),...
                             result(lin2,col2));
                    end
                end
        iteracao = iteracao + 1;
        Sindrome = Result;
         end % do while
         x_chapeu;
         %iteracao_final = iteracao;
end %do primeiro if
iteracao_final = iteracao; T=x_chapeu; T=T';
iteracaofinal=iteracao_final;
%================================================================================
function SetInpPortDims(block, idx, di)

  %block.InputPort(idx).Dimensions = di;
  block.InputPort(1).Dimensions    = [1008 1];
  %block.InputPort(2).Dimensions    = [1 1];

%endfunction

function SetOutPortDims(block, idx, di)

  %block.OutputPort(idx).Dimensions = di;
  block.OutputPort(1).Dimensions  = [1008 1];

function SetInpPortFrameData(block, idx, fd)
  %block.InputPort(idx).SamplingMode = fd;
  if idx == 1
    block.OutputPort(1).SamplingMode  = fd;
  end
  block.InputPort(1).SamplingMode = fd;

function Decoder1008_APP(block)
% Level-2 M file S-Function for times two demo.
%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.1.6.1 $

  setup(block);
```

```
%endfunction
%================================================================
function setup(block)

  %% Register number of input and output ports
  block.NumInputPorts  = 1;
  block.NumOutputPorts = 1;

  %% Setup functional port properties to dynamically
  %% inherited.
  block.SetPreCompInpPortInfoToDynamic;
  block.SetPreCompOutPortInfoToDynamic;

  % Override output ports properties
  block.OutputPort(1).DatatypeID  = 0; % double
  block.OutputPort(1).Dimensions  = [1008 1];
  %block.OutputPort(1).Complexity  = 'Real';
  %block.OutputPort(1).Complexity  = 'Complex';

  % Override input port properties
  block.InputPort(1).DatatypeID  = 0;  % double
  block.InputPort(1).Dimensions = [1008 1];
  %block.InputPort(1).Complexity  = 'Real';
  %block.InputPort(1).Complexity  = 'Complex';

  %block.InputPort(1).DirectFeedthrough = true;

  %% Set block sample time to inherited
  block.SampleTimes = [-1 0];

  %% Run accelerator on TLC
  block.SetAccelRunOnTLC(true);

  %% Register methods
  block.RegBlockMethod('Outputs',                 @Output);
  block.RegBlockMethod('SetOutputPortDimensions', @SetOutPortDims);
  block.RegBlockMethod('SetInputPortDimensions', @SetInpPortDims);
  block.RegBlockMethod('SetInputPortSamplingMode', @SetInpPortFrameData);

   global Href;
   global Nref;
```

```
   global Kref;
   global Mref;

  Href = evalin('base', 'H');
  Nref = evalin('base', 'N');
  Kref = evalin('base','K');
  Mref = evalin('base', 'M');

%endfunction
%================================================================
function Output(block)

x_recebido = block.InputPort(1).Data;
T = calcula(x_recebido);
 block.OutputPort(1).Data = T;

%================================================================
% Codificador LDPC
%H(816)
function T = calcula(x_recebido)

    global Href;
    global Nref;
    global Kref;
    global Mref;

    m=Mref;
    N=Nref;
    H=Href;
    K=Kref;

%----------------------------------------------------------------
% Canal AWGN
%----------------------------------------------------------------
testex=x_recebido; x_recebido=testex'; Y = x_recebido;

%----------------------------------------------------------------
% Decisão Hard
%----------------------------------------------------------------

for colunax=1:1:N
```

```
    if Y(colunax) > 0
        x_hard(colunax) = 1;
    else
        x_hard(colunax) = 0;
    end
end

%-------------------------------------------------------------------
% Cálculo da Síndrome
%-------------------------------------------------------------------
for lin=1:1:m
        r(lin,:) = H(lin,:)& x_hard(1,:);
end for lin=1:1: m
    col=1;
    R(lin,1)=xor(r(lin,col),r(lin,col+1));
    for col=3 : size(r,2)
         R(lin,1) = xor(R(lin,1),r(lin,col));
    end
end
Sindrome = R;
%-------------------------------------------------------------------
% Inicialização
%-------------------------------------------------------------------
% Calculo de modulo_r
for coluna_r=1:1:N
    modulo_r(coluna_r) = abs(Y(coluna_r));
    for linha_y=1:1:m
        modulo_y(linha_y,coluna_r)=abs(Y(coluna_r));
    end
end
 modulo_r; modulo_y;
%-------------------------------------------------------------------
% Processo Iterativo
%-------------------------------------------------------------------
if max(Sindrome)==0
    % display('vetor recebido = vetor transmitido');
    x_chapeu=x_hard;
    iteracao=0;
    iteracao_final=0;
else
    %-------------------------------------------------------------
```

```
% Step 1
%-------------------------------------------------------------
iteracao=0;
iteracao_final = 0;
while ((max(Sindrome) ~= 0) && (iteracao < 50))
    % Calculo de Sigma_m
    for lin1=1:1:m
        i=1;
        a(lin1)=0;
        for col1=1:1:N
            mod_y_n(1,col1)=abs(Y(1,col1));
            if H(lin1,col1)~= 0
                conj_N(lin1,i)=col1;
                i=i+1;
                if conj_N(lin1)~=0
                    a(lin1)=a(lin1)+1;
                else
                    a(lin1)=a(lin1);
                end
            end
        end
        a=a';
        for lin2=1:1:a(lin1)
            if a(lin1)~=1
                teste(lin1,1)=x_hard(conj_N(lin1,1));
                for j=2:1:a(lin1)
                    sigma(lin1,1)=xor(teste(lin1,1),...
                        x_hard(conj_N(lin1,j)));
                    teste(lin1,1)=sigma(lin1,1);
                end
                sigma(lin1,1)=teste(lin1,1);
            else
                sigma(lin1,1)=x_hard(1,conj_N(lin1,1));
            end
        end
    end
    sigma_m=sigma;
    sigma_m_barra=xor(sigma,1);
    lin_conj_N=size(conj_N,1);
    col_conj_N=size(conj_N,2);
    %   Calculo de mod_ymn_min
```

```
for linha=1:1:m
    for n=1:1:N
        k=1;
        if H(linha,n) ~= 0
            for coluna=1:1:N
                if n ~= coluna && H(linha,coluna)~=0
                    A(linha,k)=modulo_y(linha,coluna);
                    k=k+1;
                end
            end
            ymn_min(linha,n)=min(A(linha,:));
        else
            ymn_min(linha,n)=0;
        end
    end
end
ymn_min;
for lin3=1:1:m
    soma_sigma(lin3,1)=sigma_m_barra(lin3,1)-sigma_m(lin3,1);
end
for col2=1:1:N
    soma(1,col2)=0;
    for lin2=1:1:m
        if H(lin2,col2)~=0
            soma(1,col2) = soma(1,col2) + (soma_sigma(lin2,1)...]
                *ymn_min(lin2,col2));
        end
    end
    soma(1,col2)=soma(1,col2);
    zn(1,col2)= modulo_r(1,col2) + soma(1,col2);
end
soma;
modulo_r;
zn;
for col4=1:1:N
    if zn(1,col4)<0
        x_hard(1,col4)=xor(x_hard(1,col4),1);
    else
        x_hard(1,col4)=x_hard(1,col4);
    end
    x_hard;
```

```
            for lin4=1:1:m
                modulo_y(lin4,col4)=abs(zn(1,col4));
            end
        end
        x_hard;
        modulo_y;
        for lin5=1:1:m
            for colunas5 = 1:1:N
                result(lin5,colunas5) = H(lin5,colunas5) & ...
                    x_hard(1, colunas5);
            end
        end
        result;
        for lin6=1:1:m
            Result(lin6,1)=xor(result(lin6,1),result(lin6,2));
            for col6=3 : N
                Result(lin6,1) = xor(Result(lin6,1),result(lin6,col6));
            end
        end
        iteracao = iteracao + 1;
        Sindrome = Result;
        iteracao_final = iteracao;
    end % do while
    x_chapeu=x_hard;
    iteracao_final = iteracao;
end %do primeiro if
iteracao_final = iteracao; T=x_chapeu; T=T';
iteracaofinal=iteracao_final;
%=====================================================================

function SetInpPortDims(block, idx, di)

  %block.InputPort(idx).Dimensions = di;
  block.InputPort(1).Dimensions    = [1008 1];

%endfunction

function SetOutPortDims(block, idx, di)

  %block.OutputPort(idx).Dimensions = di;
  block.OutputPort(1).Dimensions  = [1008 1];
```

```
function SetInpPortFrameData(block, idx, fd)
  %block.InputPort(idx).SamplingMode = fd;
  if idx == 1
    block.OutputPort(1).SamplingMode  = fd;
  end
  block.InputPort(1).SamplingMode = fd;
%endfunction

 function Dec1008_BPBased(block)
% Level-2 M file S-Function for times two demo.
%   Copyright 1990-2004 The MathWorks, Inc.
%   $Revision: 1.1.6.1 $

  setup(block);

%endfunction
%====================================================================
function setup(block)

  %% Register number of input and output ports
  block.NumInputPorts  = 1;
  block.NumOutputPorts = 1;

  %% Setup functional port properties to dynamically
  %% inherited.
  block.SetPreCompInpPortInfoToDynamic;
  block.SetPreCompOutPortInfoToDynamic;

  % Override output ports properties
  block.OutputPort(1).DatatypeID  = 0; % double
  block.OutputPort(1).Dimensions  = [1008 1];
  %block.OutputPort(1).Complexity  = 'Real';
  %block.OutputPort(1).Complexity  = 'Complex';

  % Override input port properties
  block.InputPort(1).DatatypeID  = 0;  % double
  block.InputPort(1).Dimensions = [1008 1];
  %block.InputPort(1).Complexity  = 'Real';
  %block.InputPort(1).Complexity  = 'Complex';

  %block.InputPort(1).DirectFeedthrough = true;
```

```
  %% Set block sample time to inherited
  block.SampleTimes = [-1 0];

  %% Run accelerator on TLC
  block.SetAccelRunOnTLC(true);

  %% Register methods
  block.RegBlockMethod('Outputs',                 @Output);
  block.RegBlockMethod('SetOutputPortDimensions', @SetOutPortDims);
  block.RegBlockMethod('SetInputPortDimensions', @SetInpPortDims);
  block.RegBlockMethod('SetInputPortSamplingMode', @SetInpPortFrameData);

   global Href;
   global Nref;
   global Kref;
   global Mref;

  Href = evalin('base', 'H');
  Nref = evalin('base', 'N');
  Kref = evalin('base','K');
  Mref = evalin('base', 'M');

%endfunction
%======================================================================
function Output(block)

x_recebido = block.InputPort(1).Data;
T = calcula(x_recebido);
block.OutputPort(1).Data = T;

%======================================================================
% Codificador LDPC
%H(40,20)
function T = calcula(x_recebido)

    global Href;
    global Nref;
    global Kref;
    global Mref;
```

```
    m=Mref;
    N=Nref;
    H=Href;
    K=Kref;
%-------------------------------------------------------------------
% Canal AWGN
%-------------------------------------------------------------------
testex=x_recebido; x_recebido=testex'; Y = x_recebido;
%-------------------------------------------------------------------
% Decisão Hard
%-------------------------------------------------------------------
for colunax=1:1:N
    mod_r(colunax)=abs(Y(colunax));
    if Y(colunax) > 0
        x_hard(colunax) = 1;
    else
        x_hard(colunax) = 0;
    end
end
%-------------------------------------------------------------------
% Inicialização
%-------------------------------------------------------------------
% Calculo de x^n, x^mn , xr^n e r^n
for col0=1:1:N
    xn(1,col0)=x_hard(1,col0);
    xrn(1,col0)=xn(1,col0);
    rn(1,col0)=abs(Y(1,col0));
    for lin0=1:1:m
        if H(lin0,col0)~=0
             ymn(lin0,col0)=abs(Y(1,col0));
            if Y(1,col0) > 0
                xmn(lin0,col0) = 1;
            else
                xmn(lin0,col0) = 0;
            end
        else
            ymn(lin0,col0)=0;
        end
    end
end
%-------------------------------------------------------------------
```

```
% Cálculo da Síndrome
%-------------------------------------------------------------------
for lin=1:1:m
        r(lin,:) = H(lin,:)& xn(1,:);
end for lin=1:1: m
    col=1;
    R(lin,1)=xor(r(lin,col),r(lin,col+1));
    for col=3 : size(r,2)
         R(lin,1) = xor(R(lin,1),r(lin,col));
    end
end Sindrome = R;
%-------------------------------------------------------------------
% Processo Iterativo
%-------------------------------------------------------------------
if max(Sindrome)==0
    x_chapeu=x_hard;
    iteracao=0;
    iteracao_final=0;
else
    %-------------------------------------------------------------
    % Step 1
    %-------------------------------------------------------------
    iteracao=0;
    iteracao_final = 0;
    while ((max(Sindrome) ~= 0) && (iteracao < 50))
        %---------------------------------------------------------
        % Calculo de sigma
        %---------------------------------------------------------
        for lin1=1:1:m
            for col1=1:1:N
                a=1;
                if H(lin1,col1)~=0
                    for n=1:1:N
                        if n~=col1 && H(lin1,n)~=0
                            testem(lin1,a) = n;
                            a=a+1;
                        end
                    end
                    testem;
                    acum_sigma(lin1,1) = xor(xmn(lin1, testem(lin1,1))...
                        ,xmn(lin1,testem(lin1,2)));
```

```
            A=size(testem,2);
            for acum=3:1:A
                sigma(lin1,col1) = xor(acum_sigma(lin1,1),...
                    xmn(lin1, testem(lin1, acum)));
                acum_sigma(lin1,1)=sigma(lin1,col1);
            end
        else
            sigma(lin1,col1)=0;
        end
        sigmamn(lin1,col1)=xor(xrn(1,col1),sigma(lin1,col1));
        sigmamn_barra(lin1,col1)=xor(sigmamn(lin1,col1),1);
    end
end
%   Calculo de mod_ymn_min
for lin2=1:1:m
    for n=1:1:N
        k=1;
        if H(lin2,n) ~= 0
            for col2=1:1:N
                if n ~= col2 && H(lin2,col2)~=0
                    B(lin2,k)=ymn(lin2,col2);
                    k=k+1;
                end
            end
            ymn_min(lin2,n)=min(B(lin2,:));
        else
            ymn_min(lin2,n)=0;
        end
    end
end
% ------------------------------------------------------------------
% Step 2
% ------------------------------------------------------------------
% Calculo de zmn
for col3=1:1:n
    for lin3=1:1:m
        mult=1;
        if H(lin3,col3)~=0
            for b=1:1:m
                if b~=lin3 && H(b,col3)~=0
                    mult=mult+(((sigmamn_barra(b,col3)-...
```

```
                        sigmamn(b,col3)))*ymn_min(b,col3));
                end
            end
        else
            mult=1;
        end
        calc(lin3,col3)=mult;
        if H(lin3,col3)~=0
            zmn(lin3,col3)= rn(1,col3)+calc(lin3,col3);
        else
            zmn(lin3,col3)=0;
        end
    end
end
% Calculo de zn
for col4=1:1:N
    multN(1,col4)=1;
    for lin4=1:1:m
        if H(lin4,col4)~=0
            multN(1,col4)=(((sigmamn_barra(lin4,col4)-...
                sigmamn(lin4,col4))*(ymn_min(lin4,col4)))...
                +multN(1,col4));
        else
            multN(1,col4)=multN(1,col4);
        end
    end
    zn(1,col4)=rn(1,col4)+multN(1,col4);
end
zn;
% ------------------------------------------------------------
% Step 3
% ------------------------------------------------------------
% Calculo de x_chapeu
for col5=1:1:N
    if zn(1,col5)>0
        x_chapeu(1,col5)=xrn(1,col5);
    else
        x_chapeu(1,col5)=xor(xrn(1,col5),1);
    end
end
x_chapeu;
```

```
        for lin6=1:1:m
            for col6=1:1:N
                if H(lin6,col6)~=0
                    if zmn(lin6,col6)>0
                        xm_chapeu(lin6,1)=xrn(1,col6);
                        xmn(lin6,col6)=xrn(1,col6);
                    else
                        xm_chapeu(lin6,1)=xor(xrn(1,col6),1);
                        xmn(lin6,col6)=xor(xrn(1,col6),1);
                    end
                    ymn(lin6,col6)=abs(zmn(lin6,col6));
                else
                    ymn(lin6,col6)=0;
                end
            end
        end
        xm_chapeu;
        for lin7=1:1:m
            for col7 = 1:1:N
                result(lin7,col7) = H(lin7,col7) & x_chapeu(1, col7);
            end
        end
        result;
        for lin8=1:1:m
            Result(lin8,1)=xor(result(lin8,1),result(lin8,2));
            for col8=3 : N
                Result(lin8,1) = xor(Result(lin8,1),result(lin8,col8));
            end
        end
        iteracao = iteracao + 1;
        Sindrome = Result;
    end % do while
            x_chapeu=x_chapeu;
            iteracao_final = iteracao;
end %do primeiro if

iteracao_final = iteracao; T=x_chapeu; T=T';
iteracaofinal=iteracao_final;
%====================================================================
function SetInpPortDims(block, idx, di)
  %block.InputPort(idx).Dimensions = di;
```

```
  block.InputPort(1).Dimensions    = [1008 1];
%endfunction
function SetOutPortDims(block, idx, di)
  %block.OutputPort(idx).Dimensions = di;
  block.OutputPort(1).Dimensions  = [1008 1];
function SetInpPortFrameData(block, idx, fd)
  %block.InputPort(idx).SamplingMode = fd;
  if idx == 1
    block.OutputPort(1).SamplingMode  = fd;
  end
  block.InputPort(1).SamplingMode = fd;
%endfunction
```

Referências Bibliográficas

[1] PROAKIS, J. G., Digital Communications, 4th Ed., McGraw-Hill, 2001.

[2] GALLAGER, R.G., Low-Density Parity-Check Codes, IRE Transactions on Information Theory. Cambridge, 1962.

[3] PEARL, J., Probability Reasoming in Intelligent Sistems: Networks of Plansible Interference, San Mateo, CA. Morgan Kauffman, 1988

[4] RICHARDSON, T.J., Design of capacity approaching irregular low-density parity check codes, IEEE Trans on Information Theory, vol. 47, n.2, february, 2001

[5] LUBY, M.G., Improved Low-density parity check codes using Irregular Graphs, IEEE Trans on Information Theory, vol. 47, n.2, february, 2001

[6] FOSSORIER, F. C., Reduced Complexity Iterative Decoding of Low-Density Parity Check Codes Based on Belief Propagation. IEEE Transations on Communications, Vol. 47, No. 5, Maio 1999.

[7] PIROU, F., Introduction of Low-density Parity-Check decoding Algorithm design, Master thesis in Electronics Systems at Linköping Institute of Technology. Linköping, 2004.

[8] HAYKIN, S., Communication Systems, 4 ed. USA, John Wiley & Sons Inc., 2001.

[9] ABRANTES, S. A., Descodificação iterativa de códigos LDPC por transferência de mensagens em gráficos de factores, Departamento de Engenharia Electrotécnica e de Computadores Faculdade de Engenharia, Universidade do Porto Porto, Portugal, Julho 2005.

[10] MACKAY, D. J. C., Information Theory, Inference, and Learning Algorithms, Cambridge University Press, Version 7.2, Março 2005.

[11] GOMES, M. A. C., Códigos Binários definidos por matrizes de teste de paridade esparsas - Algoritmos de Decodificação. Universidade de Coimbra Faculdade de Ciências e Tecnologia Departamento de Engenharia Electrotécnica e de Computadores, Novembro 2003.

[12] SHU, L.; COSTELLO JR., D. J. Error Control Coding: fundamentals and applications. New Jersey: Prentice Hall, 1983.

[13] MACKAY, D. J. C., Good Error-Correcting Codes Based on Very Sparse Matrices, IEEE Transactions on Information Theory, Vol. 45, No. 2, Março 1999.

[14] PANAZIO, C. M., Utilização conjunta de equalização adaptativa e códigos corretores de erros em processamento espacial e temporal, Universidade Estadua de Campinas, Dezembro 2001.

[15] GALLAGER, R. G., Low-Density Parity-Check Codes, IRE Transactions on information theory, Janeiro 1962.

[16] PEGORARO, T. F., et al, Codificação LDPC em Sistemas de Televisão Digital, Revista do Inatel, 2006.

[17] RICHARDSON, T., The Renaissance of Gallager′s Low-Density Parity-Check Codes, IEEE Communications Magazine, Agosto 2003.

[18] http: $//www.inference.phy.cam.ac.uk/mackay/codes/data.html$