ANTONIO GORTAN

OTIMIZAÇÃO DE ALGORITMOS DE DECODIFICAÇÃO DE CÓDIGOS DE BLOCO POR CONJUNTOS DE INFORMAÇÃO VISANDO SUA IMPLEMENTAÇÃO EM HARDWARE

Dissertação apresentada ao Programa de Pós Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de "Mestre em Ciências" – área de concentração: Telemática.

Orientador: Prof. Dr. Walter Godoy Junior

Curitiba

2011

Dados Internacionais de Catalogação na Publicação

G765 Gortan, Antonio

Otimização de algoritmos de decodificação de códigos de bloco por conjuntos de informação visando sua implementação em hardware / Antonio Gortan. — 2011.

210 f. : il. ; 30 cm

Orientador : Walter Godoy Junior.

Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. Área de concentração: Telemática. Curitiba, 2011.

Bibliografia: p. 208-210.

1. Arranjos de lógica programável em campo. 2. Programação (Computadores). 3. Quantização. 4. Engenharia elétrica – Dissertações. I. Godoy Junior, Walter, orient. II. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. III. Título.

CDD (22. ed.) 621.3

Biblioteca Central da UTFR, Campus Curitiba

Título da Dissertação Nº 583:

"Otimização de Algorítmos de Decodificação de Códigos de Bloco por Conjuntos de Informação Visando sua Implementação em Hardware"

por

Antonio Gortan

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM CIÊNCIAS – Área de Concentração: Telemática, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI – da Universidade Tecnológica Federal do Paraná – UTFPR – Câmpus Curitiba, às 10h do dia 09 de dezembro de 2011. O trabalho foi aprovado pela Banca Examinadora, composta pelos professores:

Prof. Walter Godoy Junior, Dr.
(Orientador - UTFPR)

Prof. Jaime Portugheis, Dr.
(UNICAMP)

Prof. Volnei Antônio Pedroni,Dr.
(UTFPR)

Prof. Emilio Carlos Gomes Wille, Dr.
(UTFPR)

Visto da coordenação:

Prof. Fábio Kurt Schneider, Dr
(Coordenador do CPGEI)

Aos meus pais, Roberto e Irma...

À minha esposa, Vera...

AGRADECIMENTOS

Agradeço a minha esposa, Vera, pela paciência e compreensão durante esses anos em que teve que dividir minhas atenções e até mesmo prescindir delas, em favor de mais este empreendimento em minha vida que foi o mestrado.

Agradeço também ao Prof. Walter Godoy Junior, meu orientador, que, mesmo nos horários mais singulares, esteve sempre pronto e disposto a auxiliar, sugerir e conduzir, sem jamais impor.

Finalmente, não poderia deixar de expressar minha gratidão aos colegas do LME – Laboratório de Microeletrônica da UTFPR (Universidade Tecnológica Federal do Paraná), particularmente ao professor Volnei A. Pedroni e seu orientando, o doutorando Ricardo P. Jazinski, cujos questionamentos, sugestões e utilização de muitos resultados de meu trabalho, permitiram conferir também um sentido prático a este empreendimento.

A todos, meu muito obrigado...

Über allen Gipfeln
Ist Ruh,
In allen Wipfeln
Spürest du
Kaum einem Hauch;
Die Vögelein schweigen im Walde.
Warte nur, balde
Ruhest du auch.

Johan Wolfgang von Goethe
Wandres Nachtlied – Ein Gleiches
06-09-1780

RESUMO

GORTAN, Antonio. Otimização de algoritmos de decodificação de códigos de bloco por conjuntos de informação visando sua implementação em *hardware*. 2011. 210 f. Dissertação – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2011.

Este trabalho tem como finalidade realizar uma análise teórica dos processos envolvidos na decodificação de códigos de bloco lineares por meio de conjuntos de informação visando otimizar esses procedimentos para viabilizar sua implementação em *hardware* de forma eficiente através do uso de FPGAs (do inglês *Field Programmable Gate Array*). Em especial, quatro contribuições são apresentadas com essa finalidade: uma versão modificada do algorítimo de Dorsch, um conjunto de algoritmos para determinar as candidatas mais prováveis e dimensionar sua quantidade de acordo com o ganho de codificação desejado aproximando seu desempenho ao do decodificador de máxima verossimilhança, uma versão implementável em *hardware* do critério de parada BGW (das iniciais dos autores: Barros, Godoy e Wille) e a obtenção de critérios para o dimensionamento da quantidade de intervalos de quantização a utilizar.

Palavras-chave: códigos de bloco lineares, decodificação por decisão suave, conjuntos de informação, critérios de parada, quantização, *hardware* programável, FPGAs.

ABSTRACT

GORTAN, Antonio. Optimization of information set block codes decoding algorithms targeting their hardware implementation. 2011. 210 f. Dissertação – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2011.

The purpose of this work is to undertake a theoretical analysis of the processes involved in soft-decision decoding of linear block codes using the information set approach aiming at an efficient hardware implementation in FPGAs (*Field Programmable Gate Arrays*). Accordingly, four contributions to this goal are presented: a modified version of the Dorsch algorithm, a set of algorithms to determine the most reliable candidates and to gauge their quantity according desired coding gain, approaching its performance to the maximum likelihood decoder, a hardware implementable version of the BGW (from the authors initials: Barros, Godoy e Wille) stop rule and the attainment of design criteria for the number of quantization intervals to apply.

Key-words: linear block codes, soft-decision decoding, information sets, stop rules, quantization, programmable hardware, FPGAs.

LISTA DE SIGLAS

ARQ	–	Automatic Retransmission Request
BGW	–	Critério de parada de Barros, Godoy e Wille
BPSK	–	Binary Phase Shift Keying
CI	–	Conjunto de Informação
CITEC	–	Centro de Inovação Tecnológica
CMC	–	Colunas Mais Confiáveis
FEC	–	Forward Error Control
FPGA	–	Field Programmable Gate Array
LD	–	Linearmente Dependentes
LI	–	Linearmente Independentes
LME	–	Laboratório de Micro-Eletrônica
LUTs	–	Look-up tables
MDS	–	Maximum Distance Separable Codes
MLD	–	Maximum Likelihood Decoding
PQN	–	Pseudo Quantization Noise
SMC	–	Símbolos Mais Confiáveis
SNR	–	Signal to Noise Ratio
UTFPR	–	Universidade Tecnológica Federal do Paraná
VHDL	–	VHSIC Hardware Description Language
VHSIC	–	Very High Speed Integrated Circuit

LISTA DE SÍMBOLOS

$[+]$	Soma híbrida
$\binom{n}{k}$	Combinação de n elementos tomados k a k.
n	Comprimento (em símbolos) do código.
k	Comprimento (em símbolos) da mensagem.
R	Taxa do código – relação entre k e n.
$d_H(\mathbf{v},\mathbf{v'})$	Distância de Hamming entre os vetores **v** e **v'**.
d, d_{Hmin}	Distância mínima de Hamming de um código.
$d^{\perp}$, $d_{Hmin}{}^{\perp}$	Distância mínima de Hamming do código dual.
$w_H(\mathbf{v})$	Peso de Hamming do vetor **v**.
$d_E(\mathbf{v},\mathbf{v'})$	Distância Euclidiana entre os vetores **v** e **v'**.
d_{Emin}	Distância Euclidiana mínima de um código.
$d_E{}^2(\mathbf{v},\mathbf{v'})$	Distância Euclidiana quadrática entre os vetores **v** e **v'**.
$\langle \mathbf{x},\mathbf{y} \rangle$	Produto interno entre os vetores **x** e **y**.
$\|\|\mathbf{x}\|\|$	Módulo ou norma do vetor **x**.
$W(\mathbf{v})$	Peso analógico de um vetor **v**.
$V(\mathbf{x})$	Região de Voronoi do vetor **x**.
$C(n,k)$	Código de comprimento n para mensagens de comprimento k.
$C(n,k,d)$	Código comprimento n para mensagens de comprimento k, com distância mínima de Hamming d.
$\mathbf{G}$	Matriz geradora de um código.
$\mathbf{m}$	Mensagem a ser codificada.
$\mathbf{c}$	Palavra-código – $\mathbf{c} = \mathbf{m} \times \mathbf{G}$.
E_b	Energia dispendida por símbolo para transmissão pelo canal.
N_o	Densidade espectral de ruído do canal.
$f(x)$	Função densidade de probabilidade da variável aleatória x.
$F(x)$	Função de distribuição cumulativa de probabilidade da variável aleatória x.
$f_{n:m}$	Função distribuição de probabilidade do m-ésimo elemento de um conjunto de n elementos ordenados em ordem crescente

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Dentro da teoria da informação e da codificação, no que diz respeito a códigos corretores de erros, os algoritmos dedicados aos códigos de bloco e sua decodificação, particularmente aquele utilizando técnicas de conjuntos de informação, já vêm sendo abordados de longa data.

Prange (1962), Dorsch (1974), Coffey e Goodman (1990) e, mais recentemente, Fossorier (1994, 1995, 1998, 2002), dedicaram excelentes trabalhos a esse tema. Entretanto, foi apenas na última década que a tecnologia para sua implementação em *hardware* digital se tornou razoavelmente acessível para justificar a implementação de sistemas desse tipo em *hardware* de médio a baixo custo, abrindo novas perspectivas e possibilidades de sua utilização.

Também foi apenas nos últimos anos que a tecnologia de computadores pessoais e *softwares* de simulação de sistemas colocou à disposição dos pesquisadores a capacidade computacional necessária para analisar o desempenho desses sistemas de forma eficiente.

Assim, considerada a nova realidade tecnológica, uma revisão do assunto se justifica, com o fim de investigar possíveis adaptações e otimizações desses algoritmos para sua implementação em *hardware*, mais especificamente em circuitos integrados programáveis como FPGAs (do inglês *Field Programmable Gate Arrays*).

1.2 OBJETIVOS

O objetivo deste trabalho é realizar uma análise matemática das possibilidades e limitações dos algoritmos de decodificação suave de códigos de blocos por meio de conjuntos de informação e propor melhorias e alterações nos mesmos de maneira a otimizá-los para viabilizar sua implementação em *hardware* de forma efici-

ente.

Com base na análise teórica, quatro contribuições importantes ao tema da decodificação por conjuntos de informação serão apresentadas. A primeira consiste em apresentar todo o processo de decodificação sob forma de um conjunto mínimo de operações matriciais binárias passíveis de uma implementação eficiente em uma linguagem de programação para circuitos integrados programáveis como por exemplo VHDL(do inglês VHSIC *Hardware Description Language*, sendo VHSIC as iniciais de *Very High Speed Integrated Circuit*) .

A segunda diz respeito à determinação das palavras candidatas mais confiáveis de acordo com sua probabilidade de conter erros nos símbolos pertencentes à mensagem, assim como o dimensionamento da quantidade dessas candidatas a serem utilizadas de forma a atingir um determinado ganho de codificação tão próximo do ganho de codificação do decodificador por máxima verossimilhança quanto se queira.

A terceira contribuição diz respeito a uma modificação do critério de parada BGW, desenvolvido por Barros, Godoy e Wille (1997), com a finalidade de também permitir sua implementação em *hardware* programável.

O critério de parada BGW original exige uma ordenação dos símbolos da palavra código recebida de acordo com sua polaridade, o que inviabiliza sua utilização em algoritmos baseados em conjuntos de informação, visto que o processo de ordenação acaba sendo mais dispendioso que a busca exaustiva no subconjunto de palavras candidatas que o critério busca reduzir. Neste trabalho será proposta uma modificação desse critério, a qual, apesar de reduzir ligeiramente sua eficiência, elimina a necessidade da ordenação, viabilizando sua utilização em conjunto com o algorítimo de decodificação por conjuntos de informação.

Finalmente, como última contribuição ao tema de adaptação de algoritmos para implementação em *hardware*, serão investigados os efeitos da quantização dos sinais recebidos no desempenho dos algoritmos propostos, um tema em geral relegado a um segundo plano nos estudos sobre o assunto, mas que tem uma grande influência na quantidade de recursos a serem alocados em *hardware* programável.

As diversas análises teóricas realizadas foram consubstanciadas por simulações utilizando o *software* MATLAB. A viabilidade da implementação dos algorit-

mos propostos foi confirmada por implementações realizadas no laboratório de microeletrônica (LME) no CITEC (Centro de Inovação Tecnológica)/ UTFPR (Universidade Tecnológica Federal do Paraná), sob a coordenação do prof. Dr. Volnei A. Pedroni, com participação do doutorando Ricardo P. Jasinski. O resultado dessas implementações foram apresentados em diversos congressos (GORTAN et. al. 2010), (JASINSKI et. al. 2010), (GORTAN, et. al. 2012 – submetido).

1.3 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está organizada em 5 capítulos e 9 anexos.

O capítulo 2 realiza uma análise teórica detalhada das possibilidades e limitações da decodificação suave por conjuntos de informação, além de fornecer definições para os vários conceitos utilizados no restante da dissertação.

O capítulo 3 descreve os algoritmos propostos, que viabilizam uma implementação em *hardware* programável, mostrando como essas contribuições representam um avanço em relação aos algoritmos existentes.

O capítulo 4 relata brevemente os resultados práticos obtidos na aplicação dos conceitos desenvolvidos nesse trabalho à implementação em *hardware* dos decodificadores de códigos de bloco.

Finalmente, o capítulo 5 apresenta as conclusões finais e discute propostas para trabalhos futuros baseadas nos resultados alcançados até o momento.

Os anexos contêm todas as listagens de *scripts* Matlab e programas em linguagem C utilizados nas simulações para validar os resultados obtidos, assim como deduções e demonstrações matemáticas que, pela sua extensão, interfeririam no fluxo normal do texto, dificultando sua leitura.

2 CONCEITOS, ANÁLISE E FUNDAMENTOS TEÓRICOS

Nesta seção serão apresentados os conceitos e definições subjacentes ao desenvolvimento do restante do trabalho.

2.1 DEFINIÇÕES PRELIMINARES E CONCEITOS BÁSICOS

2.1.1 Modelo básico para estudo

O estudo da codificação de mensagens é baseado em um modelo composto de uma fonte de informações, um destino e um canal. O canal representa o meio de transmissão, geralmente imperfeito e sujeito portanto a interferências, que tendem a distorcer e podem mesmo adulterar as informações transmitidas. Conforme ilustrado na Figura 1, as informações a serem transmitidas necessitam ser processadas de maneira a fazer a melhor adaptação possível da fonte e do destino ao canal, minimizando dessa maneira os efeitos das interferências a que o mesmo está sujeito.

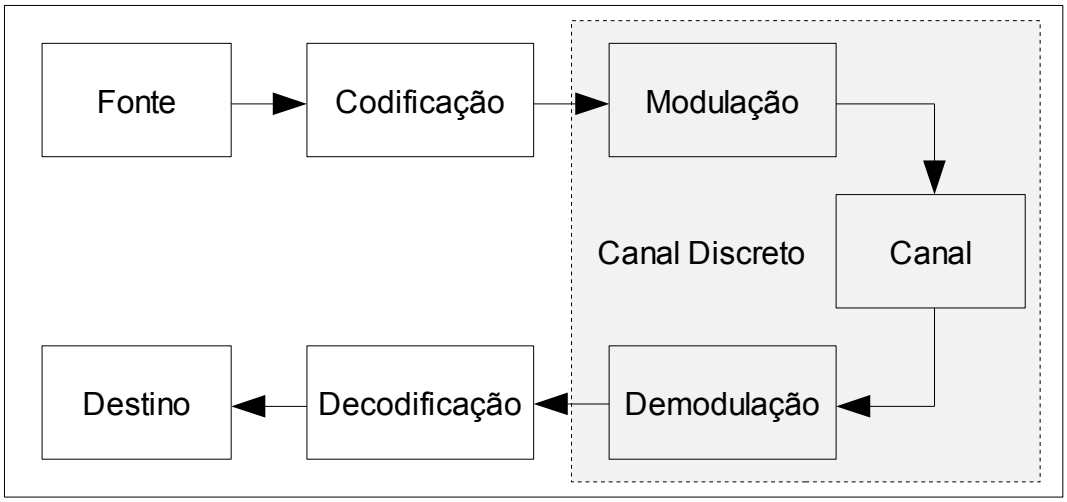

Figura 1 – Modelo utilizado para transmissão de informações

As informações raramente são geradas na forma puramente digital. Embora a maioria das mídias atuais armazene seus dados digitalmente, não o fazem considerando as necessidades de um canal de transmissão de informações. O blo-

co indicado na Figura 1 como “Fonte” inclui a digitalização, conformação e possível compactação dos dados que contêm a informação. Da mesma forma, dificilmente as informações são consumidas em sua forma digital oriunda da transmissão pelo canal. Assim, o bloco indicado na Figura 1 como “Destino” contém a correspondente descompactação e transformação dos dados digitais na forma necessária à sua utilização final. Esse agrupamento foi feito de forma a melhor isolar o objeto principal deste estudo, representado pelos demais blocos da Figura 1.

Os blocos indicados na Figura 1 como “Codificação” e “Decodificação” representam, respectivamente, o processo de adição e retirada de redundância do fluxo de dados. Isso é feito com a finalidade de permitir detectar e mesmo corrigir possíveis interferências e distorções causadas pelo canal.

O meio pelo qual transita o fluxo de dados – na Figura 1 indicado como “Canal” – utiliza-se de processos analógicos como variações de ondas eletromagnéticas, ou de intensidade ou fase de pulsos de luz, para realizar sua função. Para adaptar o fluxo de dados digitais a esse meio são então utilizados os blocos de “Modulação” e “Demodulação”.

O conjunto constituído pelo modulador, canal e demodulador, pode ser visto como um canal digital com possíveis perdas. Para os usuários que enviam e recebem dados através desse canal não interessam, em princípio, os vários processos e transformações a que os dados são submetidos no trajeto interno desse canal. O conjunto se comporta como um “canal discreto” possivelmente sujeito a perdas e retardos.

2.1.2 Estratégias para minimização de erros na comunicação de dados

Basicamente, duas estratégias são utilizadas na comunicação de dados para maximizar sua integridade: ao detectar possíveis erros, simplesmente solicitar a retransmissão da mensagem – também conhecida como protocolos do tipo ARQ, ou retransmissão automática de dados, da sigla em inglês “Automatic Retransmission Request” – ou, detectar e, com base na redundância da codificação, corrigir os erros detectados, também conhecida como protocolos tipo FEC, ou de correção de erros no destino, da sigla em inglês “Forward Error Correction”.

Os protocolos tipo ARQ são mais utilizados, sempre que a aplicação per-

mitir, por sua maior simplicidade. É mais simples apenas detectar erros que detectá-los e corrigi-los. Determinadas aplicações porém não dispõem da possibilidade de retransmissão de dados. É o caso de aplicações em tempo real, ou ainda de transmissões a distâncias interplanetárias, onde uma retransmissão é inviável. Nestes casos os protocolos tipo FEC assumem grande importância. Existe finalmente a possibilidade de uma operação híbrida. Utiliza-se então um protocolo FEC sub-optimo, em que grande parte dos erros detectada pode ser corrigida dentro de um prazo limite, e, ultrapassado esse prazo sem que o protocolo FEC tenha conseguido corrigir o erro detectado, passa-se então a solicitar uma retransmissão.

2.1.3 Códigos de bloco

De uma forma geral e simplificada, pode-se classificar os códigos em códigos de bloco e códigos convolucionais (LIN, 2004, p. 3). Neste trabalho será abordada apenas a classe dos códigos de bloco.

As mensagens a serem transmitidas pelo canal são constituídas de símbolos, os quais são agrupados em palavras ou blocos, formando assim os blocos a serem transmitidos. Ao passar pelo processo de codificação, esses blocos recebem símbolos adicionais, que constituem uma redundância de informação, formando assim novos blocos, denominados palavras-código. A Figura 2 esquematiza essa configuração.

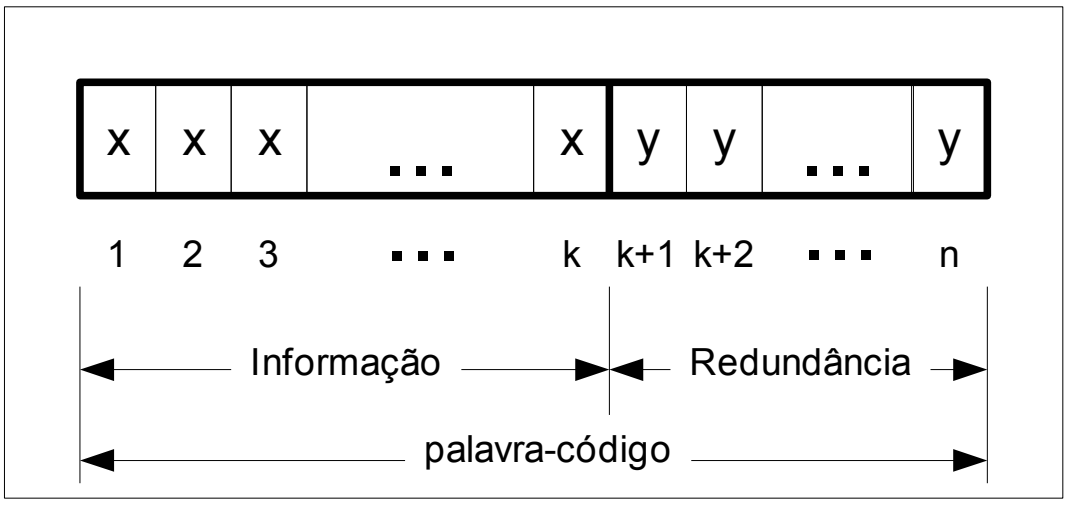

Figura 2 – Estrutura de um código de blocos

No exemplo, tem-se uma palavra-código constituída por *n* símbolos, sendo que destes, *k* constituem a informação propriamente dita, simbolizada por *xxx* … *x* e *n* – *k* constituem a redundância, simbolizada por *yy* … *y*.

Com base nessa representação, costuma-se utilizar a notação $C(n,k)$ para designar códigos de bloco com comprimento de n símbolos para a palavra-código e k símbolos para a palavra de informação.

Outras notações igualmente utilizadas para designar códigos de bloco são $C(n,k,d)$ e $C(q,n,m,d)$, onde:

- q é a cardinalidade do alfabeto, ou seja, o número de valores que um símbolo pode assumir. Neste trabalho se ocupará exclusivamente de códigos binários, em que se tem $q = 2$.
- k é o número de símbolos da palavra de informação, também conhecida como mensagem a ser codificada.
- m é a quantidade de palavras códigos possíveis para o código em questão.
- n é o número de símbolos da palavra-código, resultante da codificação da mensagem.

2.1.4 Códigos de bloco lineares

Um código de bloco é dito linear quando qualquer combinação linear de suas palavras-código resulta também em uma palavra pertencente a esse código.

2.1.5 Peso de Hamming

O peso de Hamming $w_H(\mathbf{v})$ de um vetor $\mathbf{v}$ com componentes do alfabeto binário {0,1} corresponde à quantidade de posições não nulas desse vetor. Por exemplo, para a palavra-código 10100011 de comprimento $n = 7$ do alfabeto binário {0,1}, o peso de Hamming será 4. Formalmente, pode-se expressar o peso de Hamming como:

$$w_H(\mathbf{v}) = \sum_{i|\mathbf{v}_i \neq 0} 1 \tag{2.1}$$

2.1.6 Distância de Hamming

A distância de Hamming $d_H(\mathbf{v},\mathbf{v'})$ entre dois vetores quaisquer $\mathbf{v}$ e $\mathbf{v'}$ de mesmo comprimento corresponde à quantidade de componentes em que esses dois vetores diferem. Por exemplo, para as palavras-código 11202 e 21201 de comprimento $n = 5$ do alfabeto ternário {0,1,2} a distância de Hamming será 2. Formalmente, pode-se expressar a distância de Hamming como:

$$d_H(\mathbf{v},\mathbf{v}') = \sum_{i|\mathbf{v}_i \neq \mathbf{v}'_i} 1 \tag{2.2}$$

Pode ser facilmente verificado que o peso de Hamming de um determinado vetor corresponde à distância de Hamming desse vetor para o vetor nulo, ou seja:

$$w_H(\mathbf{v}) = d_H(\mathbf{v},\mathbf{c}_0) \ , \ \mathbf{c}_0 = (0,\ldots 0) \tag{2.3}$$

2.1.7 Peso mínimo de Hamming

O peso mínimo de Hamming w_{Hmin} de um determinado código $C(q,n,m,d)$ é definido como sendo o menor peso de Hamming de todas as m palavras, excetuando-se a palavra-código nula, constituída de n posições nulas.

2.1.8 Distância mínima de Hamming

A distância mínima de Hamming d_{Hmin} de um determinado código $C(q,n,m,d)$ é definida como a menor distância de Hamming entre duas palavras quaisquer pertencentes a esse código.

Pode-se demonstrar (BLAHUT 2003, cap. 3, teorema 3.2.3, p. 90) que para um código linear, a distância mínima de Hamming d_{Hmin} é igual a seu peso mínimo de Hamming w_{Hmin}.

2.1.9 Decodificação por decisão abrupta e por decisão suave

O objetivo da decodificação é retirar e analisar a redundância dos dados com a finalidade de detectar e, possivelmente, corrigir eventuais erros introduzidos pelo canal.

O decodificador realiza essa tarefa operando sobre os dados fornecidos pelo demodulador. O demodulador, por sua vez, fornece sua melhor estimativa dos dados que recebe através do canal analógico.

Quando apenas essa estimativa é fornecida pelo demodulador ao decodificador diz-se que o sistema opera com “decisão abrupta” (HUFFMAN, 2003 cap. 15), (GODOY, 1991). Quando, entretanto, o demodulador fornece ao decodificador informações complementares sobre a confiabilidade de suas estimativas, as quais são consideradas pelo decodificador no processo de decodificação, diz-se então que

o sistema opera com “decisão suave” (HUFFMAN, 2003 cap. 15), (GODOY, 1991).

Diversos trabalhos (BARROS, 2000; GODOY, 1991; CLARK; CAIN, 1981) mostraram que a eficiência da decodificação pode ser melhorada quando o algoritmo de decodificação pode levar em conta adicionalmente as informações sobre confiabilidade das estimativas do demodulador.

2.1.10 Produto interno entre dois vetores

O produto interno entre dois vetores **x** e **y** $\in \mathbb{R}^n$ é o escalar definido como a soma dos produtos de suas componentes de mesmo índice:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i \tag{2.4}$$

2.1.11 Soma híbrida (operador [+])

A soma híbrida, representada pelo operador [+], foi definida por Godoy (GODOY, 1981) como a operação entre um vetor analógico **y** $\in \mathbb{R}^n$ e um vetor binário **c** $\in \mathbb{F}^n_q$, resultando em outro vetor analógico **y’** $\in \mathbb{R}^n$ com as seguintes características:

$$\beta_i' = |y_i'| = \beta_i = |y_i|$$

significando que as confiabilidades das componentes de **y** são preservadas, e

$$\mathbf{y_a'} = \mathbf{y_a} \ [+] \ \mathbf{c}$$

onde $\mathbf{y_a'}$ e $\mathbf{y_a}$ são os vetores binários pertencentes a $\mathbb{F}^n_q$ cujas componentes são obtidas por decisão abrupta das componentes de **y’** e **y** respectivamente.

Para q = 2, em que as componentes de **c** assumem valores do alfabeto {– 1,+1}, obtém-se facilmente as componentes y_i' de **y'** através de:

$$y'_i = -c_i \times y_i \tag{2.5}$$

2.1.12 Distância Euclidiana

A distância euclidiana $d_E(\mathbf{x},\mathbf{y})$ entre dois vetores **x** e **y** $\in \mathbb{R}^n$ é definida (BLAHUT, 1983) como:

$$d_E(\mathbf{x},\mathbf{y})=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2} \tag{2.6}$$

2.1.13 Distância Euclidiana Mínima

A distância euclidiana mínima de um código modulado $\mathcal{X}$ é definida como a menor distância euclidiana entre dois vetores quaisquer **x** e **x'** $\in \mathcal{X}$.. Como por definição as componentes de **x** e **x'** só podem assumir os valores +1 ou –1, é fácil verificar que:

$$d_E(\mathbf{x},\mathbf{x}')=2\sqrt{d_H(\mathbf{c},\mathbf{c}')} \tag{2.7}$$

onde **c** e **c'** são os vetores binários correspondentes a **x** e **x'.**

Pelo mesmo raciocínio tem-se que:

$$d_{Emin} = 2\sqrt{d_{Hmin}} \tag{2.8}$$

2.1.14 Distância Euclidiana Quadrática

A distância euclidiana quadrática é definida por:

$$d_E^2(\mathbf{x},\mathbf{y})=\sum_{i=1}^{n}(x_i-y_i)^2 \tag{2.9}$$

a qual pode igualmente ser expressa como:

$$\begin{aligned} d_E^2(\mathbf{x},\mathbf{y})&=\sum_i x_i^2-2\sum_i x_i y_i+\sum_i y_i^2 \\ d_E^2(\mathbf{x},\mathbf{y})&=\sum_i x_i^2-2\langle \mathbf{x},\mathbf{y}\rangle+\sum_i y_i^2 \end{aligned} \tag{2.10}$$

expressão essa que indica que caso se deseje comparar as diversas distâncias euclidianas de um certo vetor **y** com vários vetores-código **x**, apenas o segundo termo do segundo membro variará, pois o primeiro é constante e igual a *n* e o último tem

sempre o mesmo valor para o mesmo **y**. Assim, minimizar a distância euclidiana quadrática – e portanto também a distância euclidiana – equivale a maximizar o produto interno $\langle \mathbf{x},\mathbf{y} \rangle$:

$$d_E^{\,2}(\mathbf{x},\mathbf{y})_{min} \;\Rightarrow\; \langle \mathbf{x},\mathbf{y} \rangle_{max} \tag{2.11}$$

Neste ponto é interessante observar, que, em determinadas circunstâncias, a comparação pode se reduzir à soma dos módulos das diferenças entre as componentes correspondentes dos dois vetores. Particularmente, no caso do uso da soma híbrida, pode-se simplesmente minimizar a soma das componentes da soma híbrida, como mostrado no ANEXO VIII.

2.1.15 Peso analógico de um vetor.

Godoy (1991) definiu o peso analógico $W(\mathbf{v})$ de um vetor **v** com componentes moduladas em BPSK (do inglês *Binary Phase Shift Keying*) como sendo a distância euclidiana entre esse vetor e a palavra-código toda nula $\mathbf{x_0}$:

$$W(\mathbf{v}) = d_E(\mathbf{v},\mathbf{x}_0) = \sqrt{\sum_i (v_i - (-1))^2} = \sqrt{\sum_i (v_i + 1)^2} \tag{2.12}$$

2.1.16 Região de Voronoi de uma palavra código modulada

A região de Voronoi $\mathcal{V}(\mathbf{x})$ de um determinado vetor $\mathbf{x} \in \mathcal{X}$ constitui-se no conjunto de todos os vetores $\mathbf{y} \in \mathbb{R}^n$ mais próximos de **x** do que de qualquer outro vetor $\mathbf{x'} \in \mathcal{X}$.. Formalmente, esse conjunto é expresso como:

$$\mathcal{V}(\mathbf{x}) \;=\; \{ y \in \mathbb{R}^n \mid d_E^{\,2}(\mathbf{y},\mathbf{x}) \le d_E^{\,2}(\mathbf{y},\mathbf{x}') \;\; \forall \; \mathbf{x}' \in \mathcal{X} \} \tag{2.13}$$

2.1.17 Delimitante da região de Voronoi

Também conhecido como critério de parada ou regra de parada, é uma condição que permite testar se um determinado vetor $\mathbf{y} \in \mathbb{R}^n$ pertence ou não à região de Voronoi de um determinado vetor $\mathbf{x} \in \mathcal{X}$.

Teoricamente, para se testar se $\mathbf{y} \in \mathcal{V}(\mathbf{x})$ seria necessário comparar a distância euclidiana de **y** a **x** com todas as demais distâncias euclidianas de **y** aos de-

mais vetores-código pertencentes a $\mathcal{X}$.

Sem o critério de parada, um vetor-código **x** só pode ser declarado como correto para a decodificação de **y** após terem sido calculadas todas as distâncias euclidianas de **y** para os demais vetores-código pertencentes a $\mathcal{X}$ e constatado que essas são superiores à distância euclidiana entre **y** e **x**.

O critério de parada permitiria interromper essa longa seqüência de testes uma vez que o vetor-código **x** correto fosse encontrado.

Infelizmente, os critérios de parada descobertos e desenvolvidos até o presente não delimitam com perfeição a região de Voronoi. Em geral delimitam uma sub-região, contida dentro da região de Voronoi. Com isso, é possível que um determinado critério de parada, quando aplicado, não indique que $\mathbf{y} \in \mathcal{V}(\mathbf{x})$, quando na realidade isso ocorre.

2.1.18 Utilização da soma híbrida para mudança da região de Voronoi

Godoy (1991, p. 30) mostrou que dada uma palavra código **c** pertencente a um código C, mapeada símbolo a símbolo no alfabeto binário {– 1,+1} um dado vetor analógico **y** pertencerá à região de Voronoi de **c** se e somente se a soma híbrida **y'** de **y** com **c** pertencer à região de Voronoi da palavra código com todos os símbolos nulos $\mathbf{c_0}$. Formalmente, essa relação pode ser escrita como:

$$\mathbf{y} \in \mathcal{V}(\mathbf{c}) \Leftrightarrow (\mathbf{y}' = \mathbf{y}\,[+]\,\mathbf{c}) \in \mathcal{V}(\mathbf{c}_0) \qquad (2.14)$$

e sua importância está em permitir que todas as comparações de distâncias entre um vetor analógico recebido e uma possível palavra-código candidata podem ser transformadas em uma comparação com distâncias à palavra-código toda nula $\mathbf{c_0}$. Para ilustrar essa transformação, a Figura 3 mostra a palavra-código **c** = [1 0 1 0 0 1 1], pertencente a código $C(7,4,3)$, com símbolos mapeados no alfabeto binário {– 1, +1}, e um vetor analógico **y**, recebido através do canal com ruído. As componentes c_i assumem valores ±1 e as distâncias entre essas e as componentes y_i estão indicadas por $d_i = | c_i - y_i |$. Portanto, a distância euclidiana quadrática

entre **y** e **c** será dada por:

$$d_E^{\,2}(\mathbf{y},\mathbf{c}) \;=\; \sum_i (y_i - c_i)^2 \;=\; \sum_i |y_i - c_i|^2 \;=\; \sum_i d_i^{\,2} \qquad (2.15)$$

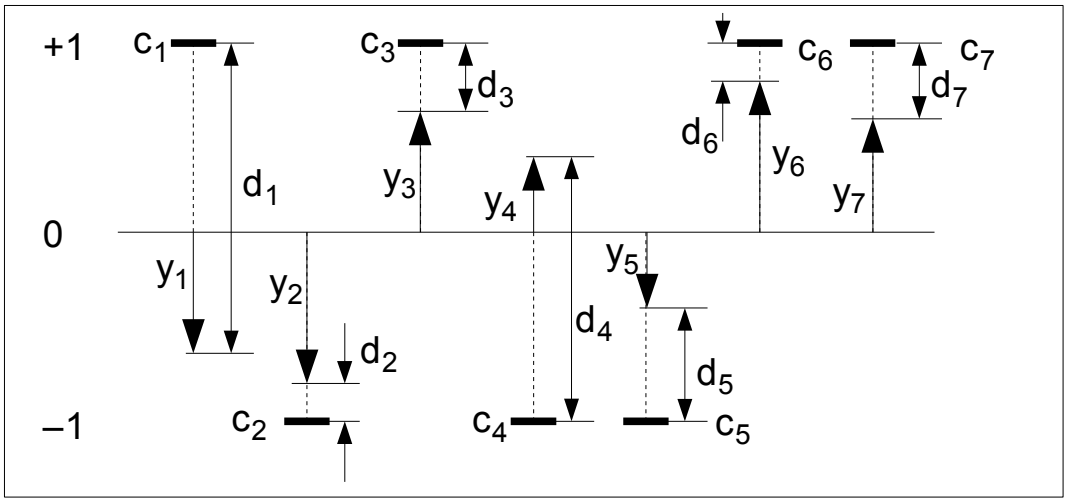

Figura 3 – Distâncias entre o vetor recebido **y** e a palavra candidata **c**

Na Figura 4 são mostradas as componentes y'_i resultantes da soma híbrida de **y** com **c**, obtidas de acordo com a equação (2.5). Nessa figura foram indicadas adicionalmente o valor absoluto das distâncias d_i entre as componentes y'_i e 0_i do vetor todo nulo $\mathbf{c_o}$.

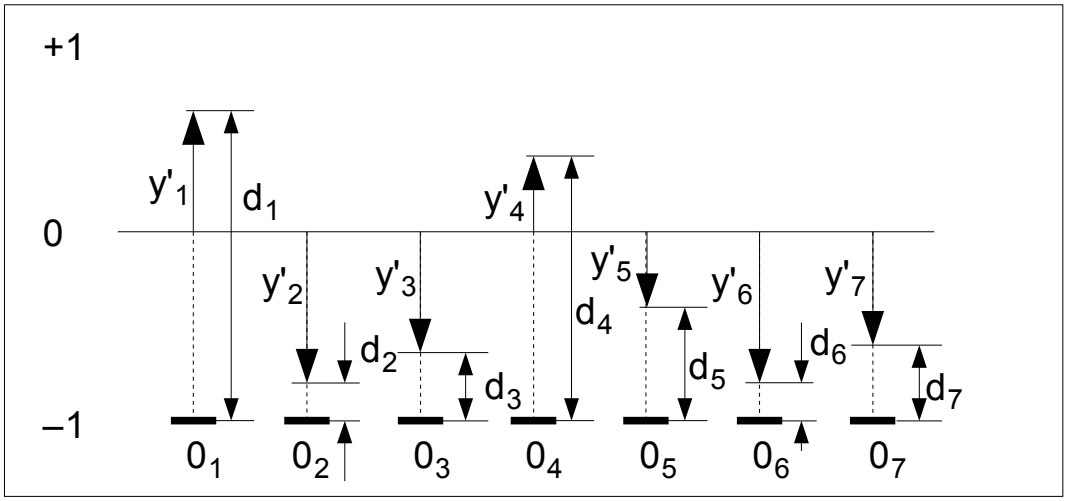

Figura 4 – Distâncias entre componentes da soma híbrida **y'** = **y** [+] **c** e o vetor $\mathbf{c_0}$

Como pode ser notado comparando-se as figuras 3 e 4 acima, as distâncias d_i permanecem, por construção, iguais em ambas as figuras. Percebe-se assim, de uma forma visual e geométrica, a validade da relação (2.14), cuja dedução formal pode ser obtida da bibliografia anteriormente citada. Além disso, a transição

da Figura 3 para a Figura 4 mostra também uma forma geométrica e rápida, ainda que destituída de formalismo, de se visualizar as componentes y_i' da soma híbrida entre um vetor analógico **y** e uma palavra-código candidata **c** modulada em BPSK (do inglês *Binary Phase Shift Keying*): basta inverter de 180º todos os eixos y das componentes em que as componentes c_i da palavra-código candidata são positivas.

A importância da relação (2.14) está em permitir que toda verificação de pertinência ou não a uma determinada região de Voronoi possa ser transformada em uma comparação de distâncias euclidianas à palavra-código toda nula. Essa transformação por sua vez simplificará o enunciado e verificação do critério de parada BGW como será visto adiante na subseção 3.3.4.

A análise da Figura 4 permite também concluir que se $\mathbf{y} \in \mathcal{V}(\mathbf{c})$ e portanto $\mathbf{y'} = \mathbf{y}\ [+]\ \mathbf{c} \in \mathcal{V}(\mathbf{c_0})$, então a soma das componentes de **y'** deve ser a mais negativa possível. Como consequência, dado um vetor recebido **y**, a busca da palavra código **c**, a cuja região de Voronoi **y** pertence, pode ser transformada na busca da palavra código **c** tal que a soma das componentes de $\mathbf{y'} = \mathbf{y}\ [+]\ \mathbf{c}$ seja a mais negativa possível.

O resultado do parágrafo anterior pode ser formalizado da seguinte maneira: se $\mathbf{y'} \in \mathcal{V}(\mathbf{c_0})$ então, de acordo com a definição 2.1.15, o peso analógico de **y'**, denotado por W(**y'**), será o menor possível, e no ANEXO VIII está demonstrado que minimizar o peso analógico de um vetor equivale a minimizar a soma de suas componentes. Porém é importante observar que, como mostrado no ANEXO VIII, essa conclusão somente é válida para um vetor **y** (ou **y'**) cujas componentes y_i (ou y'_i) tenham sido normalizadas, de maneira que $|y_i| \leq 1$ (ou $|y'_i| \leq 1$). Formalmente, portanto, pode-se escrever:

$$\begin{aligned} &\mathbf{y} \in \mathcal{V}(\mathbf{c}) \quad \Leftrightarrow \quad \mathbf{y}' \in \mathcal{V}(\mathbf{c}_0), \qquad \mathbf{y}' = \mathbf{y}[+]\mathbf{c} \\ &\mathbf{y}' \in \mathcal{V}(\mathbf{c}_0) \ \Leftrightarrow \ \mathbf{W}(\mathbf{y}')_{\min} \\ &\mathrm{W}(\mathbf{y}')_{\min} \quad \Leftrightarrow \ \left(\sum_i y'_i\right)_{\min} \end{aligned} \tag{2.16}$$

em palavras a (2.16) pode ser expressa como:

Minimizar a distância euclidiana entre **y** e **c** equivale a minimizar a soma das componentes da soma híbrida de **y** com **c**.

2.1.19 Conjuntos de Informação

Dado um código $C(n,k)$, define-se um conjunto de informação (CI) para esse código como um conjunto qualquer de k posições de suas palavras-código, tais que possam ser especificadas de forma independente (CLARK, 1981). Uma vez definidos os valores para essas k posições, as demais $n - k$ posições da palavra-código ficam automaticamente definidas.

Observe-se que, de uma forma geral, nem todo conjunto de k posições de um determinado código constitui um conjunto de informação. Isso só ocorre para uma classe especial de códigos denominados MDS – do inglês “Maximum Distance Separable Codes” (BARROS, 2000; HUFFMAN, 2003, cap. 2, teorema 2.4.3, p. 71).

Os conjuntos de informação têm sua importância para os algoritmos de decodificação de códigos de bloco com decisão suave como um meio de se gerar palavras-código candidatas à decodificação de uma seqüência recebida. Fixadas as k posições pertencentes a um determinado conjunto de informação, cujos valores foram obtidos por decodificação abrupta da seqüência recebida, as demais $n - k$ posições ficam definidas pela estrutura do código e obtém-se assim uma palavra-código válida – isto é, pertencente ao código – a qual será submetida como candidata provável à decodificação.

2.2 POSIÇÕES DE UMA PALAVRA-CÓDIGO QUE CONSTITUEM UM CI

Os códigos de bloco lineares podem sempre ser definidos por sua matriz geradora **G**. A Figura 5 mostra, como exemplo, a matriz geradora $\mathbf{G}_{157}$ para o código $C(15,7,5)$:

$$G_{157}=\begin{bmatrix}
1&0&1&0&0&0&1&1&1&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&1&0&0&0&1&1&1&0&1&0&1&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&1&0&0&0&1&1&1&1&0&0&1&0&0&0&0&0&0&0&0&0\\
1&0&1&1&0&1&0&0&0&1&1&1&0&0&0&1&0&0&0&0&0&0&0&0\\
1&1&0&1&1&0&1&0&0&0&1&1&0&0&0&0&1&0&0&0&0&0&0&0\\
1&1&1&0&1&1&0&1&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0\\
0&1&1&1&0&1&1&0&1&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&1&1&1&0&1&1&0&1&0&1&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&1&1&1&0&1&1&0&1&1&0&0&0&0&0&0&0&0&1&0&0&0\\
1&0&0&0&1&1&1&0&1&1&0&1&0&0&0&0&0&0&0&0&0&1&0&0\\
0&1&0&0&0&1&1&1&0&1&1&1&0&0&0&0&0&0&0&0&0&0&1&0\\
1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{bmatrix}$$

Figura 5 – Matriz geradora para código $C(15,7,5)$

A matriz $\mathbf{G_{157}}$ acima está na forma chamada sistemática à direita, na qual pode ser separada em duas submatrizes justapostas, uma de paridade ($\mathbf{P}_{n-k}$) e outra identidade ($\mathbf{I}_k$) de acordo com a expressão:

$$\mathbf{G}_{157}=\left[\mathbf{P}_{n-k} \;\middle|\; \mathbf{I}_k\right] \tag{2.17}$$

onde no exemplo acima tem-se n = 15 e k = 7.

Dada uma mensagem **m** binária qualquer, de comprimento k, a correspondente palavra-código **c**, de comprimento n, será obtida por meio da multiplicação de **m** pela matriz geradora **G** (em GF(2)), como a seguir, onde foram acrescentados índices indicativos das dimensões de cada um dos vetores e matrizes envolvidos:

$$\mathbf{c}_n \;=\; \mathbf{m}_k \times \mathbf{G}_{k\times n} \;=\; \mathbf{m}_k \times \left[\mathbf{P}_{n-k} \;\middle|\; \mathbf{I}_k\right] \;=\; \left[\mathbf{p}_{n-k} \;\middle|\; \mathbf{m}_k\right] \tag{2.18}$$

Neste caso, como $\mathbf{G}_{k\,x\,n}$ é sistemática à direita, vê-se que os últimos k símbolos de $\mathbf{c_n}$ correspondem exatamente à mensagem $\mathbf{m}_k$, enquanto que os primeiros $n - k$ símbolos constituem a redundância acrescentada e são linearmente dependentes (LD) dos k símbolos da mensagem, ou seja, conhecidos os k símbolos da mensagem a redundância fica automaticamente definida. Portanto, de acordo com 2.1.19 tem-se que as últimas k posições de $\mathbf{c}_n$ constituem um conjunto de informação. Esse fato indica também que as últimas k colunas da matriz $\mathbf{G}_{k\,x\,n}$ são linear-

mente Independentes (LI). Essas últimas *k* posições porém não constituem o único conjunto de informação possível. Embora nem todo conjunto de *k* colunas da matriz $\mathbf{G}_{k\,x\,n}$ seja LI, diversos o são, o que torna possível, como será visto nas seções adiante, determinar o valor correto de símbolos da mensagem a partir de símbolos da redundância ou paridade.

2.2.1 Conjuntos de posições que não constituem conjuntos de informação

Embora o objetivo final desta seção seja mostrar maneiras de determinar se um determinado conjunto de índices constitui ou não um conjunto de informação, será inicialmente mostrado nesta subseção como determinar quando um determinado conjunto de *k* colunas **não** constitui um conjunto de informação. Uma maneira imediata é formar uma matriz *k* x *k* com as *k* colunas em questão e então calcular o determinante (em GF(2)). Se o determinante for nulo, então as *k* colunas serão LD e não constituirão um CI. O cálculo do determinante porém é atualmente considerado uma das operações mais computacionalmente intensivas da álgebra linear e por esse motivo será introduzida outra opção, a qual apresenta algumas vantagens, principalmente para se determinar todos os conjuntos de informação possíveis de um determinado código. Esse outro método exige o conhecimento das palavras-código do código em questão. Dado um conjunto de índices S de colunas de $\mathbf{G}_{k\,x\,n}$, caso exista uma (ou mais) palavra(s)-código com zeros nas posições especificadas por esses índices, então o conjunto será LD. A justificativa pode ser entendida da seguinte forma: dada uma palavra-código qualquer $\mathbf{c}_i$ com *z* posições nulas, onde $z \geq k$, existirá uma correspondente mensagem $\mathbf{m}_i$ não nula tal que:

$$\mathbf{c}_i = \mathbf{m}_i \times \mathbf{G}_{k \times n} \tag{2.19}$$

Se a seguir for formado o vetor rearranjado $\mathbf{c}_{ri}$, rearranjando as posições de $\mathbf{c}_i$ de forma a se ter todas as posições nulas nas primeiras posições, iniciando por aquelas especificadas no conjunto S de índices – formando o sub-vetor $\mathbf{c}_{zi}$ – e as demais a seguir, formando o sub-vetor $\mathbf{c}_{nzi}$, e a matriz $\mathbf{G}_{k\,x\,n}$ for rearranjada de acordo com o mesmo critério formando a matriz $\mathbf{G}_{rk\,x\,n}$, pode-se escrever:

$$\mathbf{c}_{ri} = \left[\mathbf{c}_{zi} \mid \mathbf{c}_{nzi}\right] = \mathbf{m}_i \times \mathbf{G}_{r\,k \times n} = \mathbf{m}_i \times \left[\mathbf{A}_z \mid \mathbf{B}_{nz}\right] \tag{2.20}$$

Na equação (2.20), $\mathbf{c}_{zi}$ é um vetor nulo de comprimento *z*, que pode ser obtido como:

$$\mathbf{c}_{zi} = \mathbf{0}_z = \mathbf{m}_i \times \mathbf{A}_z \qquad (2.21)$$

e, no caso, $\mathbf{A}_z$ é uma matriz de dimensões *k* x *z*, formada por no mínimo todas as colunas especificadas pelo conjunto de índices S e, possivelmente, mais algumas, dependendo da quantidade de zeros da palavra-código $\mathbf{c}_i$ selecionada. Ora, o fato de existir o vetor $\mathbf{m}_i$ não nulo, o qual multiplicado por $\mathbf{A}_z$ produz um vetor nulo, significa que existe uma combinação não nula das *k* linhas de $\mathbf{A}_z$ cuja soma resulta zero, e portanto o posto[1] linha de $\mathbf{A}_z$ é inferior a *k*. Como o posto[1] linha de uma matriz é igual a seu posto coluna, isso significa que o posto coluna de $\mathbf{A}_z$ é inferior a *k* e portanto quaisquer conjuntos de *k* colunas de $\mathbf{A}_z$, inclusive aquele especificado pelo conjunto de índices S, será LD.

Portanto, para determinar se um determinado conjunto de colunas de uma matriz geradora de um código de blocos linear é LD basta encontrar uma palavra-código desse código com zeros nas correspondentes colunas. Esse fato permite determinar **todos** os conjuntos LD de colunas de uma matriz geradora de um código, apenas inspecionando suas palavras-código válidas, ou a distribuição de pesos das palavras do código.

A título ilustrativo, será feita uma análise para o código $C(15,7,5)$, cuja matriz geradora foi apresentada na Figura 5, visando verificar quantos, dos possíveis $\binom{15}{7}$ conjuntos de 7 colunas, são LD (sendo que, consequentemente, os demais serão LI).

A Figura 6 mostra a distribuição de pesos para todas as $2^7 = 128$ palavras pertencentes ao código. A distribuição de pesos de um código pode ser facilmente determinada com um *script* semelhante ao Script 1, contido no ANEXO I.

1 em inglês "rank"

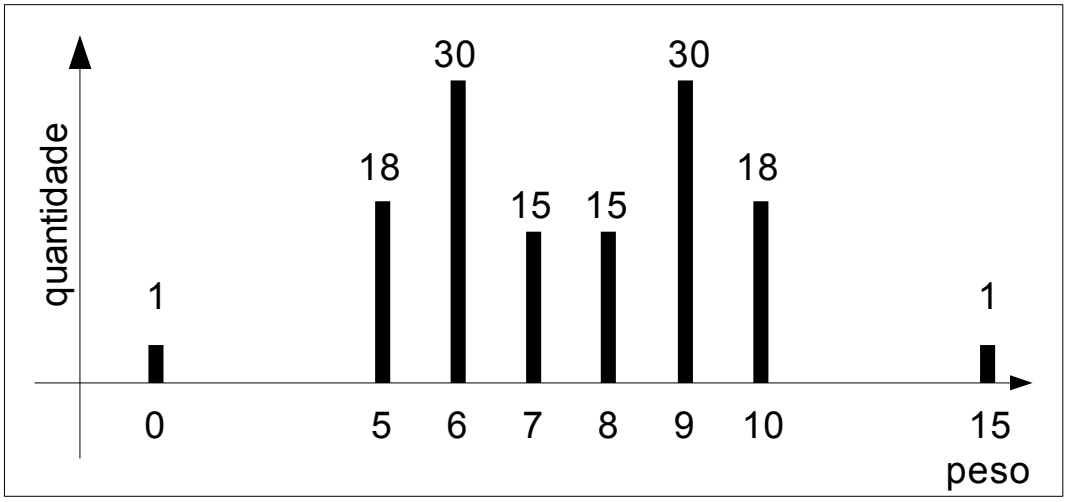

Figura 6 – Distribuição de pesos das palavas do código $C(15,7,5)$

Neste exemplo serão enumeradas todas as palavras-código com 7 ou mais zeros. Serão, portanto, as palavras de peso 8,7,6 e 5 (a palavra de peso zero não conta, pois é resultado do produto de um vetor nulo pela matriz geradora e está se buscando situações em que se tem o produto de um vetor *não nulo* pela matriz geradora fornecendo um sub-vetor nulo).

– As 15 palavras de peso 8 contribuem cada uma com um conjunto de 7 zeros.
– As 15 palavras de peso 7, por conterem 8 zeros, contribuem cada uma com $\binom{8}{7}$ diferentes conjuntos de 7 zeros.
– As 30 palavras de peso 6, por conterem 9 zeros, contribuem cada uma com $\binom{9}{7}$ diferentes conjuntos de 7 zeros.
– As 18 palavras de peso 5, por conterem 10 zeros, contribuem com $\binom{10}{7}$ conjuntos de 7 zeros cada.

O total de conjuntos de palavras contendo 7 zeros elencados até aqui, portanto, vale:

$$15\times\binom{7}{7}+15\times\binom{8}{7}+30\times\binom{9}{7}+18\times\binom{10}{7} = 3375 \quad conjuntos \tag{2.22}$$

desse total, porém, deve-se descontar conjuntos que são comuns. Para tanto é necessário examinar qual a possibilidade das várias combinações consideradas terem elementos comuns. A Figura 7 mostra, do lado esquerdo, a tabela de possibilidades de combinações e, do lado direito, um exemplo de como

uma palavra de peso 6, com 7 zeros em comum, e outra palavra de peso 5, podem ser somadas produzindo uma segunda palavra de peso 5 com os mesmos 7 zeros em comum (não mostrados na figura).

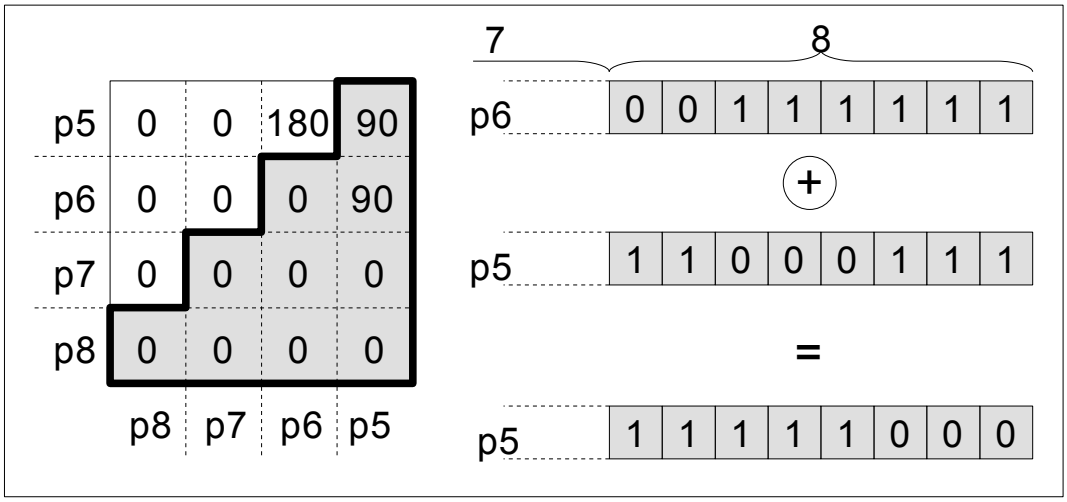

Figura 7 – Palavras-código com 7 posições zero comuns para código $C(15,7,5)$

No lado direito da Figura 7 vê-se que tomando uma palavra-código de peso 6 e outra de peso 5, ambas com 7 posições nulas em comum (não mostradas na figura), restam 8 posições onde deverão ser acomodados os seis 1's da palavra de peso 6 e os cinco 1's da palavra de peso 5, de tal maneira que, quando essas duas palavras forem somadas, o resultado será uma palavra-código pertencente ao código, uma vez que o mesmo é linear. No exemplo, o resultado da soma tem também peso 5, de forma que pode pertencer ao código. Embora essa operação não prove que existam pares de palavras-código de pesos 6 e 5 com 7 zeros em comum, ela sugere que isso é possível e que se deve investigar se, e quantos, desses conjuntos existem. Se, por outro lado, se tentasse implementar o mesmo exemplo com uma palavra de peso 6 e outra de peso 7, perceberia-se que todas possibilidades de acomodar os seis 1's de uma e os 7 1's da outra produziriam somas de peso 3 ou menor, que não poderiam pertencer ao código, levando à conclusão que não podem existir pares de palavras-código de pesos 6 e 7 com 7 posições nulas em comum. Se tais pares existissem, o código deveria abrigar também palavras de peso 3, o que não é o caso. Seguindo raciocínios análogos, pode-se completar todas as posições nulas da tabela mostrada no lado esquerdo da Figura 7. Como a tabela da Figura 7 é simétrica em relação à diagonal, apenas 10 de

suas $4 \times 4 = 16$ posições precisam ser examinadas. Essas 10 posições foram realçadas na figura. Completado o exame das 10 posições vê-se que, no caso, apenas as combinações de palavras-código de peso 6 com palavras-código de peso 5 e das palavras de peso 5 entre si precisam ser examinadas. Por outro lado, como a soma de duas palavras de peso 5 com 7 zeros em comum irá produzir sempre uma palavra de peso 6 com os mesmos 7 zeros em comum, vê-se que é suficiente examinar uma das duas possibilidades, ou das combinações de palavras de peso 5 com palavras de peso 6 ou das de peso 5 entre si. A outra possibilidade ficará, neste caso, automaticamente determinada. Tendo encontrado as duplicatas de um caso, terão sido encontradas as do outro. Uma busca exaustiva, sobre todas as possíveis combinações das 18 palavras-código de peso 5 entre si, num total de $9 \times 17 = 153$ combinações, realizada rapidamente com o auxílio do MATLAB, como mostrado no Script 3 contido no ANEXO I, mostrou que existem 90 repetições. Com isso pode-se reescrever a equação (2.22), descontando as 90 contagens em dobro para as palavras de peso 6 e as 90 para as palavras de peso 5, indicando a quantidade total de conjuntos de colunas LD:

$$15\times\binom{7}{7}+15\times\binom{8}{7}+30\times\binom{9}{7}-90+18\times\binom{10}{7}-90=3195 \ \ conjs\ LD \tag{2.23}$$

e conclui-se assim que todos os demais conjuntos serão LI:

$$\binom{15}{7} - 3195 \ = \ 6435 - 3195 \ = \ 3240 \ \ \ conjuntos\ LI \tag{2.24}$$

2.2.2 Determinação de conjuntos de posições que constituem um CI

Na subseção anterior foi visto como determinar se um determinado conjunto de posições de uma palavra-código não constitui um conjunto de informação. Utilizando esse conhecimento seria possível concluir, por exclusão, quando um determinado conjunto de posições é LI, bastaria percorrer uma tabela, constituída de todas as palavras com *k* ou mais posições nulas e verificar se as posições desejadas coincidem com as posições nulas de alguma das palavras da tabela. Esse método entretanto exige a inspeção de, senão todas, muitas das palavras-código e pode tornar-se muito ineficiente para códigos longos, com muitas palavras-código.

Outra maneira, já antecipada na subseção anterior, é realizar o cálculo do determinante da matriz formada pelas colunas correspondentes às posições desejadas, o que também, como foi aventado, não é eficiente.

Uma forma bastante eficiente de realizar a verificação é formar a matriz composta pelas colunas da matriz geradora **G** correspondentes às posições que se deseja examinar e então obter a sua forma reduzida escalonada por linhas, através de operações elementares sobre a matriz, método também conhecido como redução de Gauss-Jordan.

A álgebra linear (MEYER, 2000, cap. 3 p. 131) define três tipos de operações elementares sobre uma matriz, operações essas que não alteram a relação de dependência linear entre suas colunas transformando a matriz em uma matriz equivalente:

- operações do tipo I, consistem em trocar duas linhas (colunas) entre si.
- operações do tipo II consistem em multiplicar uma linha (coluna) por um escalar $\alpha \neq 0$.
- operações do tipos III consistem em somar à j-ésima linha (coluna) um múltiplo $\alpha \neq 0$ da i-ésima linha (coluna).

  Operando em GF(2), apenas operações do tipo I e III fazem sentido, uma vez que em GF(2) o único escalar $\alpha \neq 0$ disponível é 1.

O teste por meio da redução de Gauss-Jordan consiste em aplicar operações elementares às linhas da matriz formada pelas *k* colunas em exame, de maneira a transformar a matriz na matriz identidade. Se isso não for possível então as colunas serão LD; caso contrário serão LI. O processo será ilustrado utilizando o código de Hamming *C*(7,4,3) que, por ser um código curto, permite visualizar mais facilmente os princípios envolvidos. A Figura 8 mostra a matriz $\mathbf{G}_{74}$ desse código assim como todas suas 16 palavras-código. Na figura vê-se que as palavras-código número 2,4,5,7,9,11 e 14, identificadas com realce, contêm *k* = 4 zeros e portanto indicam todos os conjuntos de colunas que são LD. Para ilustrar a operação da redução Gauss-Jordan extrai-se da $\mathbf{G}_{74}$ dois conjuntos de 4 colunas, formando as matrizes $\mathbf{M}_0$ e $\mathbf{M}_A$ como indicado na figura. De antemão sabe-se que as colunas 1,2,5 e 7, que constituem a matriz $\mathbf{M}_0$, são LD, pois a palavra-código 4 contém zeros nessas posições. Já as colunas 4,5,6 e 7, que formam a $\mathbf{M}_A$, são LI, pois inspecionando to-

das as palavras-código não foi encontrada nenhuma com essas quatro posições contendo zeros.

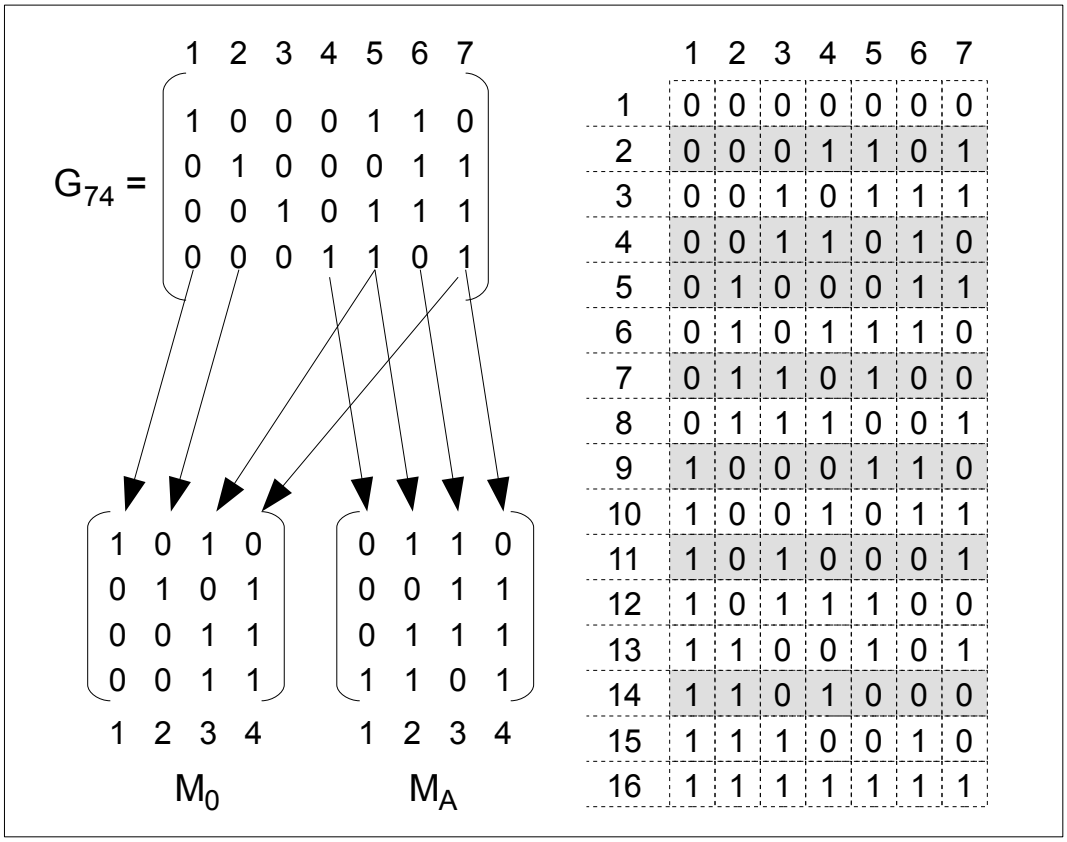

Figura 8 – Código *C*(7,4,3) e suas palavras-código

A seguir serão implementados os passos da redução Gauss-Jordan para a matriz M0 como mostrado na Figura 9.

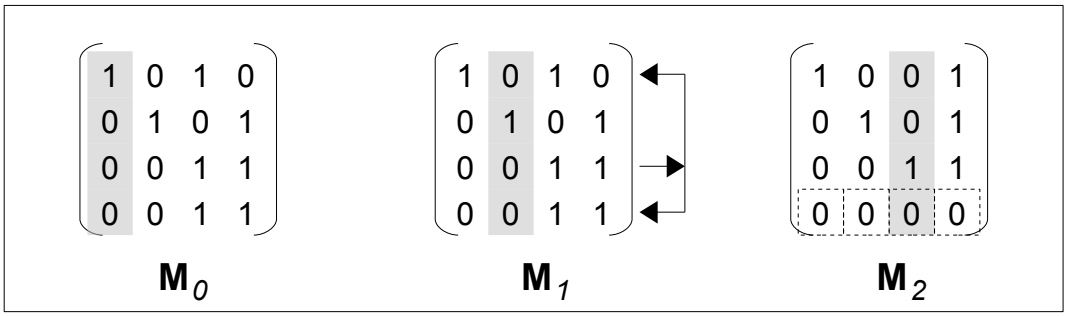

Figura 9 – Processo de redução Gauss-Jordan para a matriz $\mathbf{M}_0$

Na figura vê-se que em $\mathbf{M}_0$ já se tem a coluna 1 igual à coluna 1 da matriz identidade e portanto não há nada a fazer. A seguir será examinada a coluna 2, em $\mathbf{M}_1$ e vê-se que esta também já corresponde à coluna 2 da matriz identidade. Passa-se então para a coluna 3 e vê-se que para transformá-la em coluna da matriz

identidade é necessário substituir as linhas 1 e 4 pela sua soma (ou exclusivo em GF(2)) com a linha 3. O resultado produz a matriz $\mathbf{M}_2$, com uma linha toda nula, indicando que o posto dessa matriz é inferior a *k* e que portanto suas colunas são LD e não formam um conjunto de informação. A coluna onde o processo teve que ser interrompido, no caso a coluna 4 da matriz $\mathbf{M}_2$, traz um informação adicional: suas posições não nulas especificam sua relação de dependência com as demais colunas, tanto na matriz $\mathbf{M}_2$ como na matriz original $\mathbf{M}_0$. No exemplo em pauta, inspecionando a coluna 4 de $\mathbf{M}_2$ é possível aduzir, a partir de suas posições não nulas nas linhas 1,2 e 3, que essa coluna é o resultado da soma das colunas 1, 2 e 3. A mesma relação de dependência é também válida para a matriz $\mathbf{M}_0$, como pode ser facilmente verificado.

A seguir será visto o que ocorre quando se aplica o processo à matriz $\mathbf{M}_A$, cujas colunas sabe-se, de antemão, formarem um conjunto de informação. O processo está ilustrado na Figura 10 :

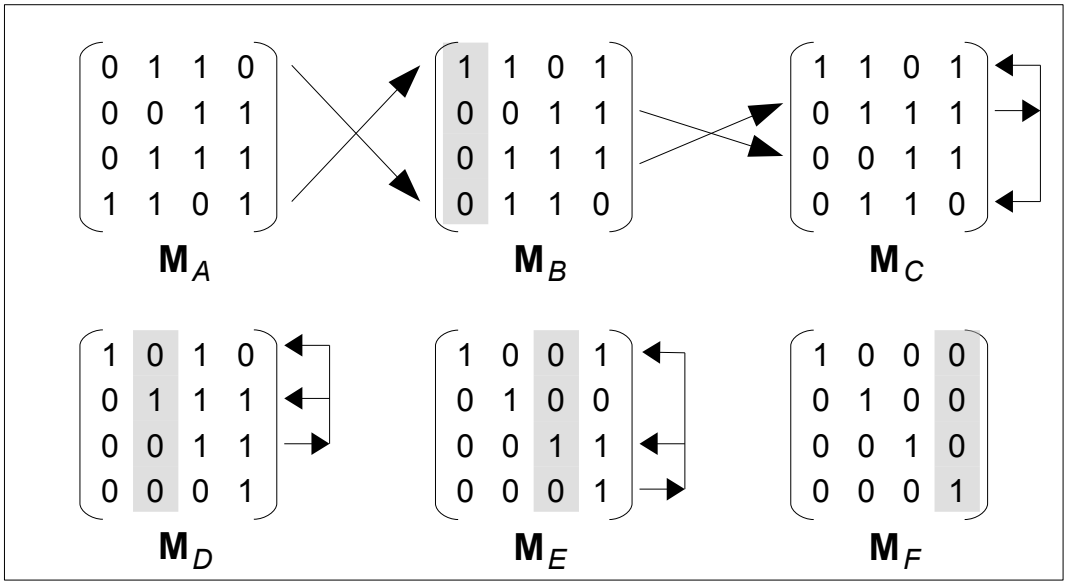

Figura 10 – Processo de redução Gauss-Jordan para a matriz $\mathbf{M}_A$

Na Figura 10 vê-se que para transformar a primeira coluna de $\mathbf{M}_A$ na coluna correspondente da matriz identidade é necessário trocar as linhas 1 e 4 de $\mathbf{M}_A$ entre si, o que irá produzir a matriz $\mathbf{M}_B$. A posição não nula de cada cada coluna da matriz identidade é geralmente chamada de pivô da coluna e doravante será assim referenciada. Na matriz $\mathbf{M}_B$ vê-se que todas as posições da primeira coluna abaixo do pivô já são nulas e portanto nenhuma operação elementar adicional é necessária para essa coluna. Passa-se então à segunda coluna. Nesse caso, para obter o pivô

é necessário trocar as linhas 2 e 3 de $\mathbf{M}_B$ entre si, obtendo a matriz $\mathbf{M}_C$. Na matriz $\mathbf{M}_C$ vê-se que há o pivô da segunda coluna, mas para transformar essa coluna na correspondente coluna da matriz identidade é necessário ainda executar operações elementares do tipo III entre a linha 2 de $\mathbf{M}_C$ e suas linhas 1 e 4, como indicado na figura. Essas operações consistem em substituir a linha 1 de $\mathbf{M}_C$ pela combinação ou-exclusivo da linha 1 com a linha 2, e substituir a linha 4 de $\mathbf{M}_C$ pela combinação ou-exclusivo da linha 4 com a linha 2. Ambas as operações entretanto podem ser resumidas em uma única operação de multiplicação – em GF(2) – de uma matriz elementar $\mathbf{E}_2$ pela matriz $\mathbf{M}_C$. como mostrado na Figura 11. A matriz elementar $\mathbf{E}_2$ é obtida a partir da matriz identidade, substituindo-se sua segunda coluna pela correspondente coluna de $\mathbf{M}_C$.

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

$\mathbf{M}_D$ $\mathbf{E}_2$ $\mathbf{M}_C$

Figura 11 – Operação elementar do tipo III para obtenção da matriz $\mathbf{M}_D$

De uma forma geral, a matriz elementar $\mathbf{E}_n$ é obtida a partir da matriz identidade, substituindo-se sua *n*-ésima coluna pela *n*-ésima coluna da matriz sobre a qual deseja-se aplicar a operação elementar do tipo III. Assim, as demais operações indicadas na Figura 10 são obtidas fazendo-se $\mathbf{M}_E = \mathbf{E}_3 \times \mathbf{M}_D$ e $\mathbf{M}_F = \mathbf{E}_4 \times \mathbf{M}_E$.

A vantagem de expressar as operações elementares dessa forma é que, quando implementadas em *hardware*, p. ex. em uma FPGA (do inglês *Field Programmable Gate Array*), a operação de multiplicação matricial em GF(2) pode ser efetuada de maneira eficiente em um único pulso do sinal de relógio. Portanto, embora o número máximo de adições de linhas de matriz – em GF(2) – seja k colunas $\times$ k – 1 adições por coluna (no pior caso), o que resulta em uma dependência quadrática do número de operações com a dimensão da matriz, em *hardware* é possível executar todas as k – 1 adições simultaneamente, por meio do produto matricial, o que resulta em apenas uma dependência linear do número de operações com a dimensão da matriz. A Figura 12 mostra como a primeira linha de uma matriz 3 x 3 é

multiplicada em GF(2) por outra matriz 3 x 3 obtendo a primeira linha da matriz 3 x 3 resultante, utilizando apenas lógica combinacional. Um exame mais detalhado da figura permite verificar facilmente que para realizar uma multiplicação de duas matrizes de dimensões *k* x *k* serão necessárias k^3 portas E e $k^2(k\text{-}1)$ somadores binários ou ou-exclusivos.

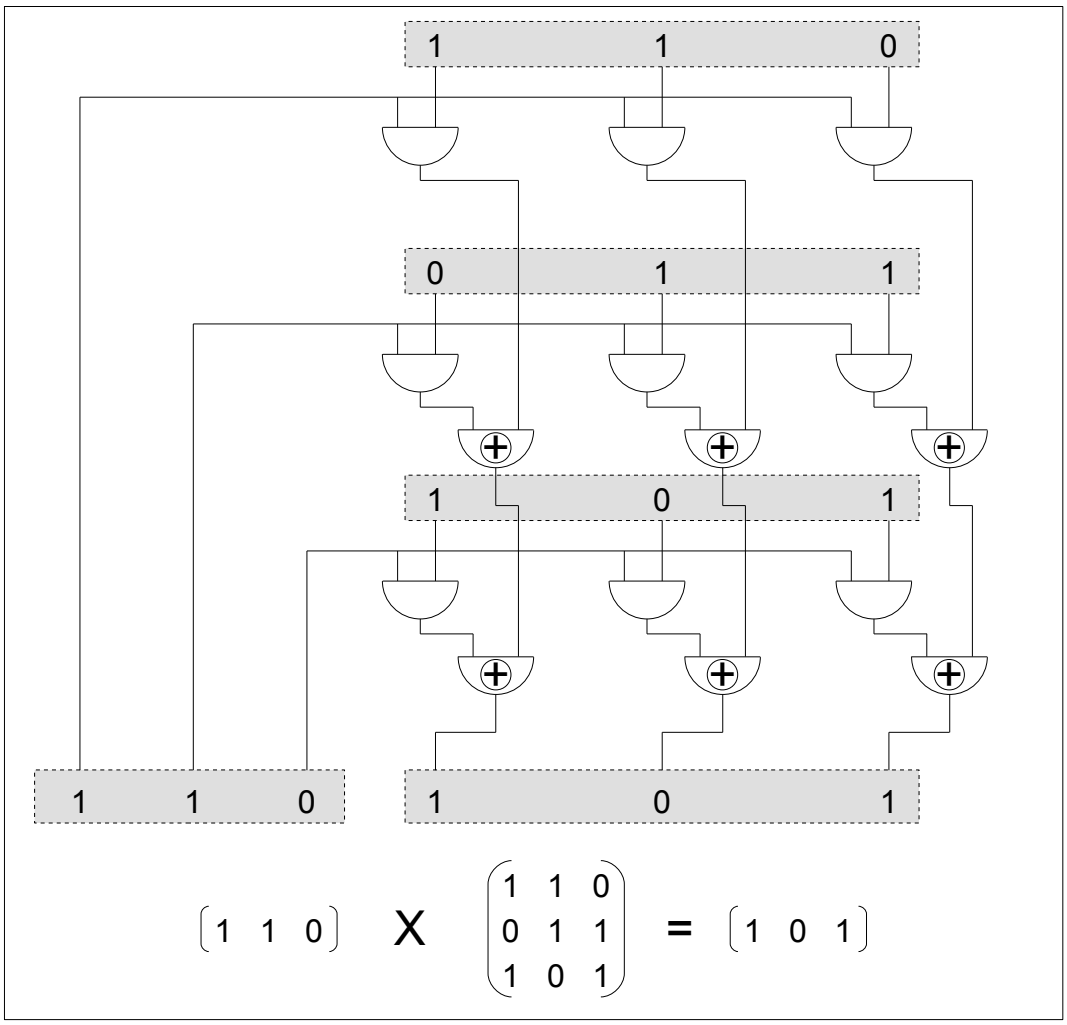

Figura 12 – Lógica combinacional para multiplicação vetorial em GF(2)

2.2.3 A redução de Gauss-Jordan modificada

É importante observar que a troca de linhas indicada na subseção anterior, na Figura 10, p. 37, é supérflua. A verificação da independência linear de um conjunto de colunas não necessita que a redução de Gauss-Jordan seja feita de tal forma a obter uma matriz identidade ao final. Qualquer permutação da matriz identidade é suficiente para demonstrar a independência linear. Com isso pode-se selecionar um pivô em qualquer posição da coluna que está sendo processada, desde

que seja uma posição na qual ainda não foi obtido um pivô. A Figura 13 mostra como a verificação resulta simplificada prescindindo das trocas de linhas. Nessa figura foi introduzida a máscara **m**, que armazena as posições de pivô já obtidas. A cada passo a busca de um novo pivô é feita mascarando-se as posições já obtidas, indicadas por 1's na máscara. Mais uma vez as operações de soma em GF(2) indicadas podem ser realizadas via multiplicações por matrizes elementares. As matrizes elementares a serem utilizadas em cada passo porém precisam ser especificadas por dois índices: **E***mn* é a matriz elementar do tipo III obtida substituindo-se sua *m*-ésima coluna pela *n*-ésima coluna da matriz sobre a qual deseja-se aplicar a operação elementar do tipo III. No caso, o índice *m* corresponde ao índice da linha onde foi encontrado o pivô.

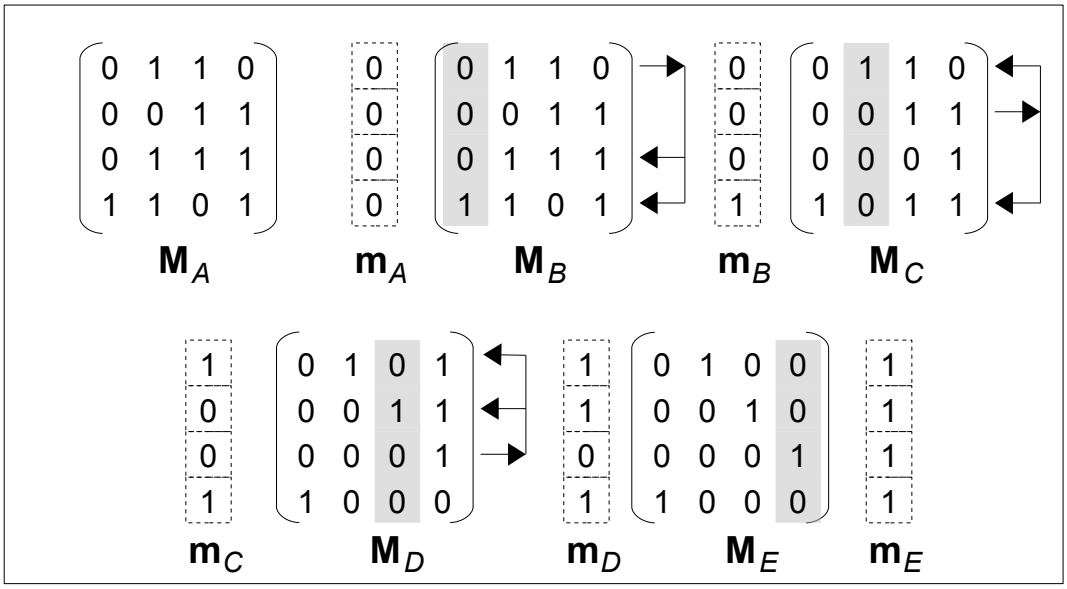

Figura 13 – Redução de Gauss-Jordan modificada, sem troca de linhas

Tomando por base as operações efetuadas na Figura 13 tem-se que $\mathbf{M}_B = \mathbf{E}_{41} \times \mathbf{M}_A$ porém, como $\mathbf{E}_{41}$ é a própria matriz identidade, vê-se que no primeiro passo a matriz permanece inalterada e $\mathbf{M}_B = \mathbf{M}_A$. No segundo passo tem-se $\mathbf{M}_C = \mathbf{E}_{12} \times \mathbf{M}_B$, no terceiro $\mathbf{M}_D = \mathbf{E}_{23} \times \mathbf{M}_C$ e, finalmente, no último, $\mathbf{M}_E = \mathbf{E}_{34} \times \mathbf{M}_D$.

O processo será interrompido e o conjunto de colunas especificado será declarado LD se não for possível encontrar nenhum pivô em alguma das colunas examinadas. Em cada passo a busca do pivô só é realizada nas posições não mascaradas pela máscara **m**. Um pequeno *script* para MATLAB, que realiza essa verificação de acordo com esses princípios, está listado no Script 4, contido no ANEXO I.

Para finalizar esta subseção cabe ainda observar um resultado que será útil mais à frente, na definição do algorítimo de decodificação por decisão suave com conjuntos de informação. Ao concluir a verificação, como mostrado na Figura 13, p. 40, termina-se com uma matriz $\mathbf{M}_E$, que não é uma matriz identidade. Se, por algum motivo, for necessário transformar essa matriz em identidade, ou seja, permutar suas colunas (ou linhas) de tal forma a obter uma matriz identidade, isso pode ser feito de forma simples, multiplicando $\mathbf{M}_E$ pela sua transposta. De fato, como $\mathbf{M}_E$ é ortogonal, sua inversa é igual à sua transposta e pode-se escrever:

$$\mathbf{M}_E \times \mathbf{M}_E^{\mathrm{T}} = \mathbf{M}_E^{\mathrm{T}} \times \mathbf{M}_E = \mathrm{I} \tag{2.25}$$

Esse método constitui uma redução de Gauss-Jordan em que as operações elementares do tipo I, ou seja, a troca de linhas, são desconsideradas. Esse método será referenciado mais adiante como redução de Gauss-Jordan modificada ou RGJM.

2.3 TÉCNICAS PARA DECODIFICAÇÃO DE CÓDIGOS

O objetivo da decodificação é retirar e analisar a redundância dos dados com a finalidade de detectar e, possivelmente, corrigir eventuais erros introduzidos pelo canal.

O decodificador realiza essa tarefa operando sobre os dados fornecidos pelo demodulador. O demodulador, por sua vez, fornece sua melhor estimativa dos dados que recebe através do canal analógico.

Como já citado em 2.1.9, quando apenas essa estimativa é fornecida pelo demodulador ao decodificador diz-se que o sistema opera com “decisão abrupta” (HUFFMAN, 2003 cap. 15), (GODOY, 1991). Quando, entretanto, o demodulador fornece ao decodificador informações complementares sobre a confiabilidade de suas estimativas, as quais são consideradas pelo decodificador no processo de decodificação, diz-se então que o sistema opera com “decisão suave” (HUFFMAN, 2003 cap. 15), (GODOY 1991).

2.3.1 Decodificação suave por máxima verossimilhança (MLD)

A decodificação suave por máxima verossimilhança, ou MLD, da sigla em inglês para “Maximum Likelihood Decoding”, consiste em se utilizar a informação de confiabilidade dos símbolos recebidos, fornecida pelo demodulador, para selecionar, dentre as palavras-código pertencentes ao código, aquela que maximize a probabilidade de que tenha sido ela a palavra transmitida.

De acordo com Shu Lin, em (LIN, 2004, cap. 1, seção 1.4), dada uma palavra-código **v** transmitida, e uma sequência **r** recebida, a decodificação por máxima verossimilhança consiste em se selecionar uma palavra código $\bar{\mathbf{v}}$ tal que se maximize a probabilidade P($\bar{\mathbf{v}}$ = **v** | **r**). Ou seja, consiste em se escolher como decodificação $\bar{\mathbf{v}}$ para a sequência **r** a palavra-código **v** tal que maximize:

$$\mathrm{P}(\mathbf{v}|\mathbf{r}) = \frac{\mathrm{P}(\mathbf{r}|\mathbf{v}) \cdot \mathrm{P}(\mathbf{v})}{\mathrm{P}(\mathbf{r})} \tag{2.26}$$

No caso de todas as palavras-código **v** serem igualmente prováveis, ou seja, P(**v**) é sempre o mesmo, independente de **v**, maximizar a expressão (2.26) equivale a maximizar P(**r** | **v**). E no caso ainda de um canal discreto e sem memória, em que as sequências **r** recebidas dependem exclusivamente das sequências **v** transmitidas, a probabilidade P(**r** | **v**), que se deseja maximizar, pode ser colocada como:

$$\mathrm{P}(\mathbf{r}|\mathbf{v}) = \prod_i \mathrm{P}(r_i \mid v_i) \tag{2.27}$$

e, utilizando a função *log x,* que é uma função que cresce monotonicamente, pode-se tomar o *log* de ambos os membros da (2.27) obtendo:

$$\log(\mathrm{P}(\mathbf{r} \mid \mathbf{v})) = \sum_i \log \mathrm{P}(r_i \mid v_i) \tag{2.28}$$

e um decodificador que busque maximizar o produtório do segundo membro da (2.27) ou o somatório do segundo membro da (2.28) é chamado de decodificador por máxima verossimilhança ou decodificador MLD.

Neste ponto é importante observar que a suposição, adotada acima, de sequências **v** equiprováveis raramente é verdadeira na prática, sendo que as probabilidades P(**v**) dependerão da distribuição de pesos do código utilizado e das características da fonte de informação. Apesar desse fato, e como as propriedades esta-

tísticas das fontes de informação em geral ou não são conhecidas ou são variáveis, a suposição é considerada válida e utilizada na prática.

Em termos práticos, para um canal com sinalização BPSK (do inglês *Binary Phase Shift Keying*), e submetido à interferência de ruido gaussiano branco aditivo (AWGN), Marc Fossorier mostrou em (LIN, 2004, cap. 10, seção 10.1), que maximizar a expressão (2.28) equivale a minimizar a distância euclidiana entre a sequência recebida **r** e a palavra-código **v** dada como a correta decodificação para **r**.

A decodificação por MLD, por sua própria definição, irá produzir a palavra código mais provável de estar correta, ou com a menor probabilidade de erro. A sua implementação porém envolve uma grande complexidade computacional pois para um código binário com mensagem de comprimento k existem 2^k palavras-código válidas, o que envolve a determinação de 2^k distâncias euclidianas e $2^k - 1$ comparações. Por esse motivo técnicas alternativas de decodificação foram desenvolvidas, sempre com o objetivo de reduzir a complexidade computacional em relação ao MLD, buscando, entretanto, obter a mesma capacidade de correção do MLD ou tão próxima desta quanto possível.

Uma dessas alternativas corresponde aos algoritmos de Chase (1972), os quais fazem uso da informação adicional sobre confiabilidade dos símbolos recebidos para gerar um conjunto reduzido de palavras-código candidatas através da permutações dos símbolos menos confiáveis. Chase (1972), propôs três variantes de seu algorítimo, diferenciadas basicamente pela quantidade de símbolos menos confiáveis a serem permutados para gerar palavras-código alternativas.

Outra alternativa, apresentada na seção seguinte, são os algoritmos de decodificação suave baseados em conjuntos de informação.

2.3.2 Decodificação suave por conjuntos de informação

A decodificação por conjuntos de informação assemelha-se, em parte, aos algoritmos de Chase já citados, por se utilizar da informação de confiabilidade dos símbolos para reduzir o conjunto de palavras-código candidatas à decodificação para uma quantidade bem inferior às 2^k palavras-código utilizadas na decodificação por MLD.

Diferentemente dos algoritmos de Chase, porém, são os símbolos com a maior confiabilidade que são utilizados para gerar um conjunto reduzido de palavras-

código através de permutações de seus valores.

O princípio da decodificação por conjuntos de informação é basicamente simples: Dado um vetor binário recebido e conhecida a confiabilidade relativa de seus símbolos, selecionam-se os *k* símbolos mais confiáveis e cujas posições são linearmente independentes. De acordo com a definição na seção 2.1.19, esses *k* símbolos formam um conjunto de informação, de onde deriva o nome dado a esse tipo de algorítimo.

Como os *k* símbolos foram selecionados de maneira a formarem um conjunto de informação, os demais *n* – *k* símbolos podem ser obtidos a partir destes por serem linearmente dependentes deles.

Obtém-se, dessa forma, uma primeira palavra-código candidata, com alta probabilidade de ser a decodificação correta para o vetor recebido.

Candidatas adicionais podem então ser obtidas permutando-se os valores dos *k* símbolos LI selecionados e obtendo os *n* – *k* restantes a partir da permutação dos primeiros.

São três os problemas a resolver na implementação do processo descrito acima:

- Como obter os *k* símbolos mais confiáveis que sejam LI, ou seja, que constituam um conjunto de informação.
- Como recodificar os demais *n* – *k* símbolos a partir dos *k* mais confiáveis e LI obtidos no item anterior.
- Como determinar quais e quantas permutações realizar sobre os *k* símbolos mais confiáveis com o objetivo de gerar palavras-código alternativas.

Os algoritmos baseados em conjuntos de informação existentes diferenciam-se na solução que dão a cada um dos problemas citados acima. Neste trabalho será apresentada uma nova solução que se caracteriza por:

- Obter os *k* símbolos mais confiáveis e LI de forma sistemática, sem recorrer a tabelas ou métodos de tentativas e erros.
- Recodificar os demais *n* – *k* símbolos a partir dos *k* mais confiáveis e LI de uma forma que é eficiente para a implementação em *hardware.*
- Determinar quais e quantas permutações realizar sobre os *k* símbolos

mais confiáveis com base nas probabilidades de incidência dos possíveis padrões de erro aplicados à candidata mais confiável.

Na próxima seção, os conceitos apresentados na seção 2.2 serão aplicados à solução dos três problemas citados aqui, para o desenvolvimento de um novo algorítimo baseado em conjuntos de informação, denominado BP (iniciais para "*Best Practice*"), que se adequa à implementação em *hardware* programável.

3 ALGORÍTIMO PROPOSTO

Como foi visto na seção 2.3, na decodificação por decisão suave o demodulador fornece ao decodificador, além de sua melhor estimativa para o valor dos símbolos recebidos, também uma informação adicional sobre o nível de confiabilidade de cada estimativa para cada símbolo. Torna-se portanto possível realizar a decodificação de uma palavra código pertencente a um dado código $C(n,k,d)$, selecionando os k símbolos mais confiáveis os quais formam um conjunto de informação e então, supondo seus valores corretos, obter os demais a partir deles. Na seção 2.2.2, na p. 34, foi mostrado como determinar se um determinado conjunto de símbolos de um certo código constitui ou não um conjunto de informação. Adiante será mostrado como utilizar uma técnica similar para determinar rapidamente o conjunto de informação mais confiável possível. Tentativas anteriores de resolver este problema basearam-se em preprocessamento do código e preparação de listas de posições mais prováveis de conterem um conjunto de informação, por ex. (GODOY, 2010a), (GODOY, 2010b) Em contrapartida, a técnica apresentada neste trabalho é sistemática e determinística. Entretanto, mesmo o conjunto de informação mais confiável está sujeito a erros e o algorítimo deve considerar a possibilidade de que um ou mais dos k símbolos LI mais confiáveis (SMC's) esteja incorreto. Esse fato é contemplado gerando-se candidatas alternativas por meio de modificação de um ou mais dos k SMC's. Quais e quantos SMC's deverão ser alterados será também objeto de análise nas próximas seções. Embora alguns trabalhos como (GORTAN, 2010a) e (BRANTE, 2011) mostrem que, em determinadas situações, k + 1 SMC's seja uma boa escolha – ou seja, o próprio conjunto de informação e cada uma das permutações de uma de suas posições – a quantidade ideal de SMC's a alterar varia bastante com o comprimento do código (GORTAN, 2010b) , e um método de determinação dessa quantidade em função do ganho de codificação desejado será apresentado.

3.1 O algorítimo BP

O algoritmo BP (iniciais do inglês de *Best* for *Practice*) é um algoritmo de decodificação por decisão suave, baseado em conjuntos de informação que permite obter praticamente o mesmo desempenho da decodificação por máxima verossimilhança (MLD, do inglês *Maximum Likelihood Decoding*), porém com uma eficiência muito superior, pois permite reduzir substancialmente o número de comparações envolvidas. O algoritmo foi otimizado para implementação em *hardware* programável, de onde derivam suas iniciais "**B**est for **P**ractice".

Enquanto que na decodificação por máxima verossimilhança com decisão suave compara-se a distância euclidiana da palavra-código recebida a todas as demais palavras pertencentes ao código, no algorítimo BP a comparação é feita apenas com um pequeno subconjunto de palavras-código, o qual tem todavia uma alta probabilidade de conter a correta decodificação para a palavra-código recebida. A essência do algorítimo portanto está em encontrar, de forma eficiente, esse subconjunto.

Para melhor compreensão e facilidade de análise, o algoritmo BP pode ser dividido em três etapas distintas:

1. A partir da informação de confiabilidade dos símbolos recebidos, determinar o conjunto de informação mais confiável possível.
2. A partir do conjunto de informação mais confiável, gerar o subconjunto de palavras-código com alta probabilidade de conter a decodificação correta para a palavra-código recebida.
3. Determinar, por meio da comparação das distâncias euclidianas, a palavra-código do subconjunto com maior probabilidade de ser a decodificação correta para a palavra-código recebida.

Nesse último item, pode-se ainda melhorar o desempenho e eficiência do algoritmo através da utilização de um critério de parada. Um critério de parada é um critério que permite identificar se a candidata em análise é a melhor possível, permitindo encerrar prematuramente a série de comparações.

A seguir cada um dos itens acima será abordado, ilustrando sua utilização com um exemplo, baseado em um código curto para facilitar a visualização dos princípios envolvidos. Serão também discutidos possíveis critérios de parada, analisando sua eficiência e maior ou menor dificuldade de implementação.

3.1.1 Determinação das posições mais confiáveis que constituem um CI

O item 1 da seção anterior seria trivial se qualquer conjunto de símbolos da palavra-código recebida constituísse um conjunto de informação. Infelizmente, como foi visto na seção 2.2, para cada código $C(n,k,d)$ apenas uma certa porcentagem de todos os conjuntos possíveis de k símbolos constitui um conjunto de informação.

Várias soluções foram propostas anteriormente para esse problema, entre elas o cálculo do determinante da matriz formada pelas k colunas da matriz geradora correspondentes às k posições mais confiáveis – que serão chamadas aqui de colunas mais confiáveis ou CMC's, ainda que a expressão em si não tenha sentido, uma vez que a confiabilidade é uma medida atribuída aos símbolos da palavra recebida e não às colunas da matriz – e o uso de tabelas baseadas em análises estatísticas, contendo os conjuntos mais prováveis, como em (GODOY, 2010a), (GODOY, 2010b) e (FOSSORIER, 2002). Dorsch (1974) propôs uma solução semelhante à que será proposta a seguir, porém baseada na redução da matriz **H,** de verificação de paridade do código, para a forma reduzida escalonada por linhas. Esse método foi igualmente utilizado por Fossorier, em (FOSSORIER,1994) e (FOSSORIER,1995). Todos esses trabalhos, porém, não estavam voltados para a implementação em *hardware*.

O algoritmo BP utiliza para essa etapa a redução de Gauss-Jordan modificada, como apresentada na subseção 2.2.3, na p. 39. Como será visto no exemplo a seguir, esse processo permite determinar, de forma inequívoca e sem necessidade de repetições de tentativas, o melhor conjunto de informação possível conhecida a confiabilidade relativa dos símbolos. Supondo portanto, a título de exemplo, que para o código $C(7,4,3)$ o demodulador tenha fornecido a confiabilidade relativa de cada um dos 7 símbolos recebidos de uma determinada palavra-código e que após a ordenação das confiabilidades se tenha obtido um vetor **S** de índices de confiabilidade como abaixo:

$$\mathbf{S}=[7\,5\,2\,1\,3\,4\,6] \tag{3.1}$$

significando que a posição do 7º símbolo é a mais confiável, seguida da 5ª posição e assim por diante. A RGJM será portanto realizada processando-se as colunas da matriz geradora nessa ordem, como já exposto na subseção 2.2.3. Diferentemente, porém, do que foi ali exposto, o algorítimo não será interrompido caso não seja pos-

sível encontrar um pivô em determinada coluna, mas sim será dado prosseguimento utilizando a próxima coluna mais confiável, de acordo com a sequência ditada pelos índices contidos em **S**. A sequência de operações necessárias está ilustrada na Figura 14.

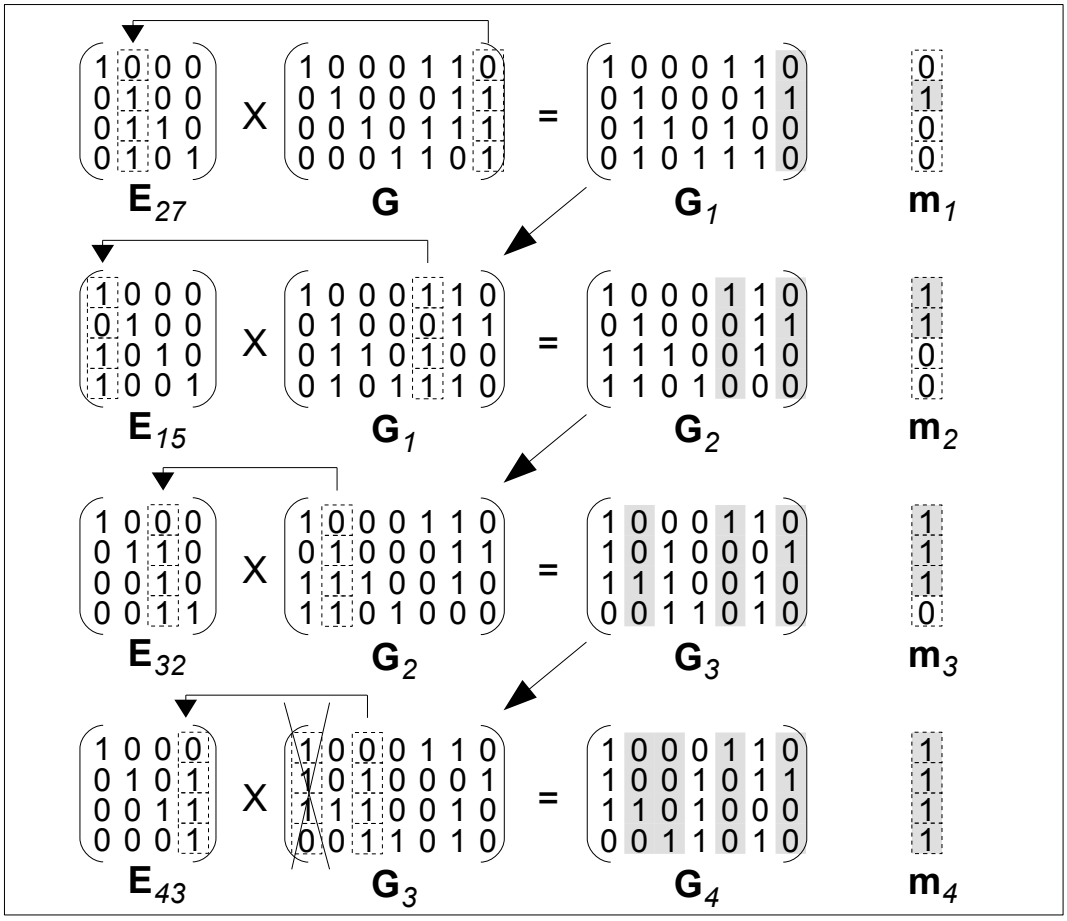

Figura 14 – Determinação do CI mais confiável a partir do vetor **S** = [7 5 2 1 3 4 6]

Como mostra a figura, as colunas de **G** vão sendo reduzidas a colunas unitárias na sequência ditada pelo vetor de índices mais confiáveis **S**. Inicialmente a coluna 7 é examinada, um pivô é encontrado em sua linha 2 e a redução é efetuada por meio da multiplicação à esquerda de **G** por $\mathbf{E}_{27}$, produzindo $\mathbf{G}_1$ e a máscara $\mathbf{m}_1$, que serão usadas no passo seguinte. A matriz $\mathbf{E}_{27}$ é a matriz elementar do tipo III, obtida substituindo-se a segunda coluna de uma matriz identidade pela sétima coluna de **G**. Nesse primeiro passo a máscara de busca de pivôs estava inicialmente toda zerada, indicando que não há restrições quanto á escolha de uma posição não nula para pivô. Nos passos subsequentes as máscaras $\mathbf{m}_1$, $\mathbf{m}_2$, etc. indicam quais posições ainda são elegíveis para pivô. Como se vê, no último passo a coluna a

examinar seria a coluna 1, porém a máscara $\mathbf{m}_3$ indica que apenas a posição da linha 4 é válida para a busca de pivôs e essa posição é nula. Portanto a busca do pivô na coluna 1 falha. O algorítimo porém continua com a próxima coluna mais confiável, que é a coluna 3, e, nessa coluna, encontra um pivô na linha 4. O algorítimo só encerra quando a máscara contém *k* posições um.

No exemplo, o algorítimo determinou, em uma única passagem, que as *k* colunas mais confiáveis são as colunas 7,5,2 e 3. Naturalmente, o conjunto ideal teria sido o das colunas 7,5,2 e 1, porém esse conjunto não forma um CI, logo o algorítimo retornou o próximo conjunto mais confiável. Assim o vetor **S** das confiabilidades = [7,5,2,1] precisou ser alterado para um vetor que será chamado de **SI** = [7,5,2,3], ou seja, o vetor da sequência de componentes mais confiáveis **e** linearmente independentes. No caso particular das *k* componentes mais confiáveis do vetor recebido serem também linearmente independentes valerá **SI** = **S**. Adicionalmente, o algorítimo fornece a nova matriz, $\mathbf{G}_4$ no exemplo, que servirá na próxima etapa para recodificar os símbolos menos confiáveis a partir dos mais confiáveis. Essa matriz será genericamente chamada $\mathbf{G}_n$, onde o índice *n* denota “nova”.

O processo, como mostrado acima, é muito semelhante à determinação de uma matriz inversa. De fato, se fosse possível conhecer a priori o fato de que o conjunto de informação mais confiável no caso seria aquele formado pelas colunas 7,5,2 e 3 da matriz **G**, então teria sido possível buscar a inversa da matriz formada por essas colunas e essa inversa, quando multiplicada pela **G**, produziria a $\mathbf{G}_4$, embora com as linhas trocadas de maneira que as colunas 7,5,2 e 3 constituíssem a matriz identidade. A implementação do processo de inversão em *hardware* pode ser realizado de maneira semelhante, e com eficiência comparável, como mostraram Jasinski, Pedroni, Gortan e Godoy em (JASINSKI, 2010). Como a informação de quais colunas da matriz **G** constituem um CI não está disponível a priori, a técnica de inversão de matriz só pode ser aplicada através de um processo de tentativa e erro, o que reduziria a eficiência total. No exemplo em questão, a primeira tentativa, de inverter a matriz constituída pelas colunas 7,5,2 e 1 de **G** falharia, obrigando a uma nova tentativa com outro conjunto.

3.1.2 Quantidade de colunas a examinar para a obtenção de $\mathbf{G}_n$

Na subseção anterior, para encontrar 4 colunas LI da matriz G foi neces-

sário examinar um total de 5 colunas. Dado um código de bloco linear genérico $C(n,k,d)$, surgem portanto duas questões fundamentais na análise do algorítimo:

– até quantas colunas será preciso examinar no pior caso antes de obter k colunas LI, ou seja, até obter um CI?
– qual a probabilidade de se encontrar k colunas LI, ou seja, um CI, examinando apenas k colunas, apenas $k + 1$ colunas, etc.?

Adicionalmente, é interessante verificar, para efeito de análise de desempenho, a partir de quantas colunas examinadas será possível encontrar colunas LD, ou seja, poderá ocorrer não encontrar um pivô na próxima coluna. Isso pode ser colocado na forma da seguinte pergunta:

– qual a quantidade máxima de colunas que, selecionadas arbitrariamente, serão sempre LI?

Essas questões serão respondidas nas subseções que seguem.

3.1.2.1 Quantidade máxima de colunas que, selecionadas arbitrariamente, serão sempre LI

Para que um conjunto qualquer de colunas da matriz geradora **G** de um código C resulte LD é necessário que seja possível formar um vetor **v** contendo valores um nas posições correspondentes às colunas em análise e zero nas demais posições, tal que $\mathbf{G} \times \mathbf{v}^T = \mathbf{0}$. O vetor **v** assim formado irá necessariamente pertencer ao código dual de C e suas posições não nulas definirão o conjunto de colunas de **G** que serão LD. O peso mínimo de um vetor **v** pertencente ao código dual de C irá portanto definir a cardinalidade mínima de um conjunto de colunas de **G** que poderá ser LD. Qualquer conjunto de colunas de cardinalidade inferior ao peso mínimo de **v** será portanto necessariamente LI. Como observado na seção 2.1.8, o peso mínimo de um vetor pertencente a um código qualquer será igual à distância mínima desse código. Portanto a cardinalidade mínima de um conjunto qualquer de colunas de **G** LD será igual à distância mínima do código dual de C, que será aqui denotado por $d^{\perp}$. Logo qualquer conjunto de $d^{\perp} - 1$ colunas de **G** será necessariamente LI.

Este resultado responde à terceira pergunta feita em 3.1.2 acima e será realçado a seguir para servir de referência posteriormente:

Dado um código de blocos linear $C(n,k,d)$ e conhecida a distância mínima

$d^{\perp}$ de seu código dual, qualquer conjunto de $d^{\perp} - 1$ colunas de sua matriz geradora será necessariamente LI.

3.1.2.2 Quantidade máxima de colunas a examinar, para garantidamente encontrar *k* colunas LI

Como foi visto na seção 2.2.1, para que um determinado conjunto de de colunas de uma matriz geradora seja LD é necessário que exista uma palavra-código contendo zeros nas posições correspondentes às colunas do conjunto. Caso essa palavra-código não exista então o conjunto em questão será LI. Para um determinado código $C(n,k,d)$, a maior quantidade de zeros que uma palavra-código não nula poderá conter será $n - d$, visto que d, como já observado na seção 2.1.8, sendo a distância mínima de Hamming do código, será também o peso mínimo de qualquer palavra-código não nula. Assim o maior conjunto de colunas LD possível, para um determinado código, será $n - d$. Consequentemente, qualquer conjunto de $n - d + 1$ colunas da matriz geradora será necessariamente LI e conterá portanto um CI. Esse resultado, que responde à primeira pergunta feita anteriormente em 3.1.2, é conhecido e foi apresentado na forma de teorema em (HUFFMAN, PLESS, 2003, p. 13), sendo reproduzido para referência no ANEXO III. O resultado foi ressaltado abaixo para facilitar referências posteriores:

Dado um código de bloco linear $C(n,k,d)$, qualquer conjunto de $n - d + 1$ colunas de sua matriz geradora irá conter um conjunto de informação.

3.1.2.3 Probabilidades de ser necessário examinar *k*, *k*+1, etc.. colunas da matriz geradora até obter um CI

Sabe-se, a partir dos resultados das seções anteriores, que dado um arranjo qualquer de colunas da matriz geradora de um código, para se obter um conjunto de informação é necessário examinar, no melhor caso k dessas colunas e, no pior, $n - d + 1$ colunas. Surge então a questão de avaliar com que frequência o melhor caso será obtido, com que frequência ocorrerá o pior caso e qual a probabilidade dos casos intermediários.

Tomando novamente o código $C(15,7,5)$ como exemplo, sabe-se, a partir dos resultados da seção 2.2.1 e equações (2.23) e (2.24), que a probabilidade p_7 de se obter um CI examinando as primeiras k colunas (no caso $k = 7$) do conjunto será:

$$p_7 = \frac{qtde\ conj\ 7\ col\ LI}{qtde\ total\ conj\ 7\ col} = \frac{3240}{6495} \times 100 = 50{,}35\ \% \qquad (3.2)$$

Sabe-se, igualmente, que a probabilidade p_{n-d+1} (p_{11} no exemplo em pauta) de se obter um CI examinando as primeiras $n - d + 1$ colunas será 100 %. Deve-se então examinar os restantes casos intermediários, p_8, p_9 e p_{10}. Em cada caso, a probabilidade de sucesso p_i será igual a $1 - pf_i$, onde pf_i é a probabilidade de falha em se obter um CI examinando um conjunto qualquer de i colunas da matriz geradora. Essa probabilidade de falha será dada por:

$$pf_i = \frac{quantidade\ de\ conjuntos\ de\ i\ colunas\ LD}{quantidade\ total\ de\ conjuntos\ de\ i\ colunas} \qquad (3.3)$$

Na seção 2.2.1, foi visto que os conjuntos de posições de uma palavra-código, ou conjuntos de colunas da matriz geradora, que não constituem um CI podem ser determinados a partir do conhecimento da distribuição de pesos das palavras do código, enumerando as palavras-código cujas posições correspondentes são todas nulas. Esse conhecimento será então utilizado para determinar p_8, p_9 e p_{10} para o código $C(15,7,5)$ do exemplo.

A determinação de p_{10} será feita contando-se todas as palavras-código com a possibilidade de 10 posições nulas. No caso essas serão as 18 palavras-código de peso 5 do código:

$$\begin{aligned} p_{10} &= (1 - f_{10}) \times 100\ \% \\ &= \left[1 - \frac{18 \times \binom{10}{10}}{\binom{15}{10}} \right] \times 100\ \% \\ &= 99{,}4\ \% \end{aligned} \qquad (3.4)$$

A determinação de p_9 será feita contando-se todos os conjuntos de 9 posições nulas que podem ser formados a partir das 18 palavras de peso 5, mais todos os conjuntos de 9 posições nulas que podem ser formados a partir das 30 palavras de peso 6, e observando-se que não há duplicidade de contagem:

$$
\begin{aligned}
p_9 &= (1-f_9)\times 100\ \% \\
&= \left[\ 1-\frac{18\times\binom{10}{9}+30\times\binom{9}{9}}{\binom{15}{9}}\right]\times 100\ \% \\
&= 95{,}8\ \%
\end{aligned}
\tag{3.5}
$$

De forma semelhante, a determinação de p8 será feita contando-se todos os conjuntos de 8 posições nulas que podem ser obtidos a partir das palavras de peso 5, 6 e 7 e verificando-se não ser possível a contagem em duplicidade em cada caso:

$$
\begin{aligned}
p_8 &= (1-f_8)\times 100\ \% \\
&= \left[\ 1-\frac{18\times\binom{10}{8}+30\times\binom{9}{8}+15\times\binom{8}{8}}{\binom{15}{8}}\right]\times 100\ \% \\
&= 82{,}98\ \%
\end{aligned}
\tag{3.6}
$$

As expressões utilizadas até o momento podem ser generalizadas, introduzindo-se a notação:

A_j = quantidade de palavras-código de peso j

D_{ij} = conjuntos de i posições nulas das palavras-código de peso j contadas em duplicidade.

Utilizando essas definições pode-se escrever de forma generalizada:

$$
\begin{aligned}
p_i &= (1-f_i)\times 100\ \% \\
&= \left[\ 1-\frac{\sum_{j=1}^{n-i}\left[A_j\times\binom{n-j}{i}-D_{ij}\right]}{\binom{n}{i}}\right]\times 100\ \%
\end{aligned}
\tag{3.7}
$$

A determinação das probabilidades a partir da expressão (3.7) porém é bastante trabalhosa devido à necessidade do cálculo de todos os valores de D_{ij}, os quais, para cada valor de i, devem ser determinados a partir da matriz de possíveis combinações, como já exemplificado para ao exemplo de i = 7 para o código C(15,7,5) na seção 2.2.1. Por outro lado, muitos dos valores D_{ij} são nulos, como

ocorreu no exemplo anterior, ou, quando não são nulos, são relativamente pequenos em comparação ao termo $A_j \times \binom{n-j}{i}$, de forma que, quando desprezados, não alteram substancialmente o resultado, o que fornece uma excelente forma de obtenção de um limitante inferior eficiente para as probabilidades p_i.

Uma outra forma de determinação das probabilidades p_i pode ser implementada realizando-se simulações com um número razoavelmente grande de iterações, através, por exemplo, de um *script* para Matlab como implementado no Script 6 no ANEXO I. A quantidade de iterações a utilizar em cada simulação pode ser determinada como detalhado no ANEXO IV.

Tabela 3.1: Porcentagem de sucesso na obtenção de um CI – *C*(15,7,5)

Colunas examinadas	% Simulada	% Teórica
7	50,12	50,35
8	82,96	82,98
9	95,85	95,80
10	99,39	99,40
11	100,00	100,00

A Tabela 3.1 mostra que os valores simulados através do Script 6 são praticamente os mesmos que os obtidos pela análise teórica nesta seção.

Outros valores, levantados com o mesmo *script* já citado do Matlab, estão apresentados em uma forma mais visual nos gráficos da Figura 15 para os códigos *C*(15,7,5) (para comparação), *C*(24,12,8) e *C*(48,24,13). É interessante notar que, para todos os códigos apresentados, a probabilidade de se atingir o limite superior de $n - d + 1$, deduzido na seção 3.1.2.2, é muito pequena, inferior a um ponto porcentual. De fato, de posse da distribuição de probabilidades é simples determinar o valor médio para cada caso obtendo:

- 7,72 colunas em média para o código *C*(15,7,5).
- 12,71 colunas em média para o código *C*(24,12,8).
- 24,32 colunas em média para o código *C*(48,24,12).

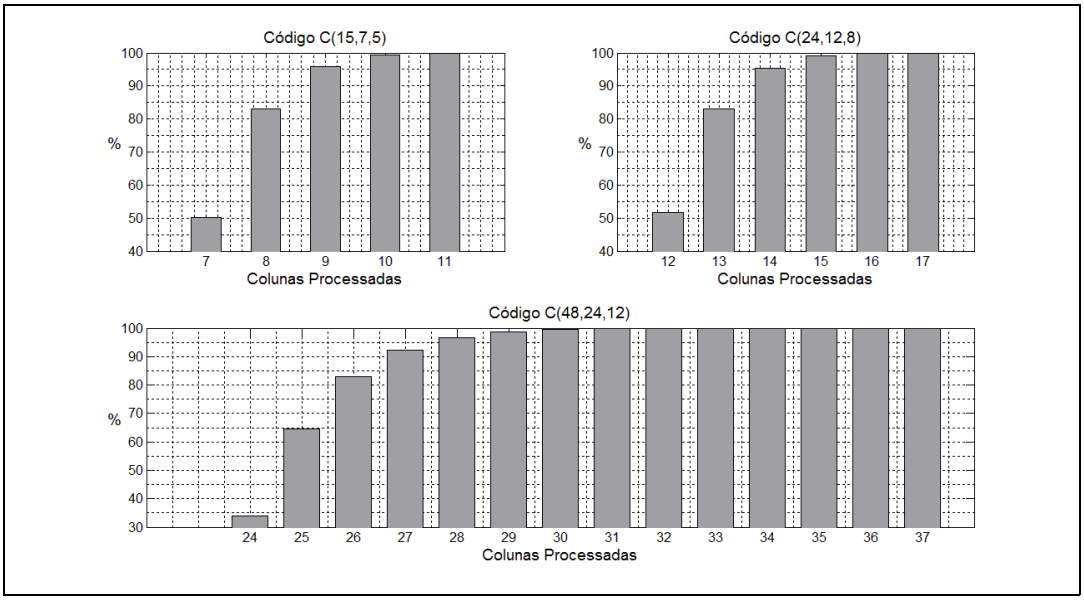

Figura 15 – Probababilidades de obter um CI examinando *k* ou mais colunas

Na Figura 15, os valores apresentados para o código $C(48,24,12)$ para 24 colunas ainda contém uma pequena imprecisão, pois a quantidade de iterações utilizada (10^6) com o Matlab não é suficiente para se alcançar uma precisão de ±1 x 10^{-7}. Nesse caso, de acordo com a equação (IV.3) no ANEXO IV, para se estimar $p \cong 0{,}99999975$ com 95% de confiança com uma precisão de ± 1 x 10^{-7} são necessárias $9{,}6 \times 10^7 \cong 10^8$ iterações. Para superar essa limitação foram desenvolvidos os programas Dup4824 (listagens 1, 2, 3 e 4 no ANEXO II), Tent4824 (listagens 12, 13, e 14 no ANEXO II) e Gauss4824 (listagem 15 no ANEXO II), em linguagem C, que permitem executar o número necessário de iterações e permitem também realizar uma análise exaustiva dos conjuntos de zeros duplicados fornecendo valores mais precisos para esse código, como mostrando na Tabela 3.2. Todos os três programas se apoiam em rotinas desenvolvidas em linguagem C para realizar a multiplicação de vetor pela matriz geradora, eliminação Gaussiana modificada, geração de permutação aleatória de índices e obtenção da listagem de todas as 2^{24} palavras-código agrupadas por peso, as quais foram congregadas na biblioteca estática C482412 (listagens 5 a 11 no ANEXO II).

Tabela 3.2: Porcentagem de sucesso na obtenção de um CI – *C*(48,24,12)

Colunas examinadas	% Simulada	% Teórica
24	33,907545	34,000767
25	64,458765	64,113501
26	82,884232	82,809630
27	92,335037	92,324438
28	96,776209	96,771499
29	98,728366	98,727450
30	99,536068	99,535569
31	99,846423	99,845972
32	99,955085	99,954792
33	99,988714	99,988704
34	99,997741	99,997741
35	99,999668	99,999677
36	99,999983	99,999975

A coluna de porcentagem simulada na Tabela 3.2 foi obtida por meio do programa Gauss4824, com 10^8 iterações. A cada iteração foi gerada uma permutação aleatória dos índices das colunas da matriz, para simular uma ordenação de confiabilidades de símbolos, e, a partir dessa ordenação, foi executada a eliminação de Gauss modificada, e anotada a quantidade de colunas percorridas até a obtenção de todos os 24 pivôs.

A coluna de porcentagem teórica na Tabela 3.2 foi obtida aplicando-se a equação (3.7). Para utilizar essa equação é necessário determinar, além da distribuição de peso de código, também os termos D_{ij} das quantidades de palavras de peso j com i zeros em posições comuns. Isso foi feito com o auxílio do programa Dups4824 (listagens 1, 2, 3 e 4 no ANEXO II). Os detalhes dessas deduções estão descritos no ANEXO V.

3.1.3 Recodificação dos símbolos mais confiáveis com a matriz G_n

Os k símbolos mais confiáveis da palavra-código recebida que formam

um conjunto de informação são selecionados de acordo com os critérios discutidos nas seções anteriores. A decodificação por decisão abrupta dos símbolos irá então formar uma nova mensagem, a qual será recodificada com a matriz $\mathbf{G}_n$. A ordem desses símbolos na nova mensagem deve ser tal que preserve sua posição original na nova palavra-código resultante da multiplicação. Tomando ainda como exemplo o caso ilustrado na Figura 14, e chamando aos símbolos da palavra-código recebida e decodificada por decisão abrupta de s_1 a s_7, vê-se que a ordem a ser utilizada nesse caso deve ser $\mathbf{m}_e$ = [s_7 s_5 s_2 s_3], onde foi utilizado o índice "*e*" para denotar o fato que a ordem dos símbolos da mensagem foi "*embaralhada*" De fato, multiplicando $\mathbf{m}_e$ pela matriz $\mathbf{G}_4$ do exemplo vê-se que o resultado será:

$$[\, p_1 s_2 s_3 p_4 s_5 p_6 s_7 \,] \;=\; \mathbf{m}_e \times \mathbf{G}_4 \tag{3.8}$$

ou seja, os símbolos mais confiáveis, s_2,s_3,s_5 e s_7, foram preservados em suas posições originais e os demais p_1,p_4 e p_6, determinados em função destes. Isso foi possível graças ao pré embaralhamento da ordem dos símbolos em $\mathbf{m}_e$. No exemplo recém analisado, a mensagem $\mathbf{m}_e$ foi obtida por inspeção da matriz $\mathbf{G}_4$. Existe porém uma maneira sistemática de se obter a mensagem $\mathbf{m}_e$: basta obter a matriz $\mathbf{G}_{r0}$ a partir da $\mathbf{G}_n$ ($\mathbf{G}_4$ no exemplo) zerando todas as colunas não unitárias de $\mathbf{G}_n$. Chamando a palavra-código recebida decodificada por decisão abrupta de **pcra** = [s_1 s_2 … s_n], a mensagem $\mathbf{m}_e$ é obtida a partir de:

$$\mathbf{m}_e \;=\; \mathbf{pcra} \times \mathbf{G}_{r0}^{\mathrm{T}} \tag{3.9}$$

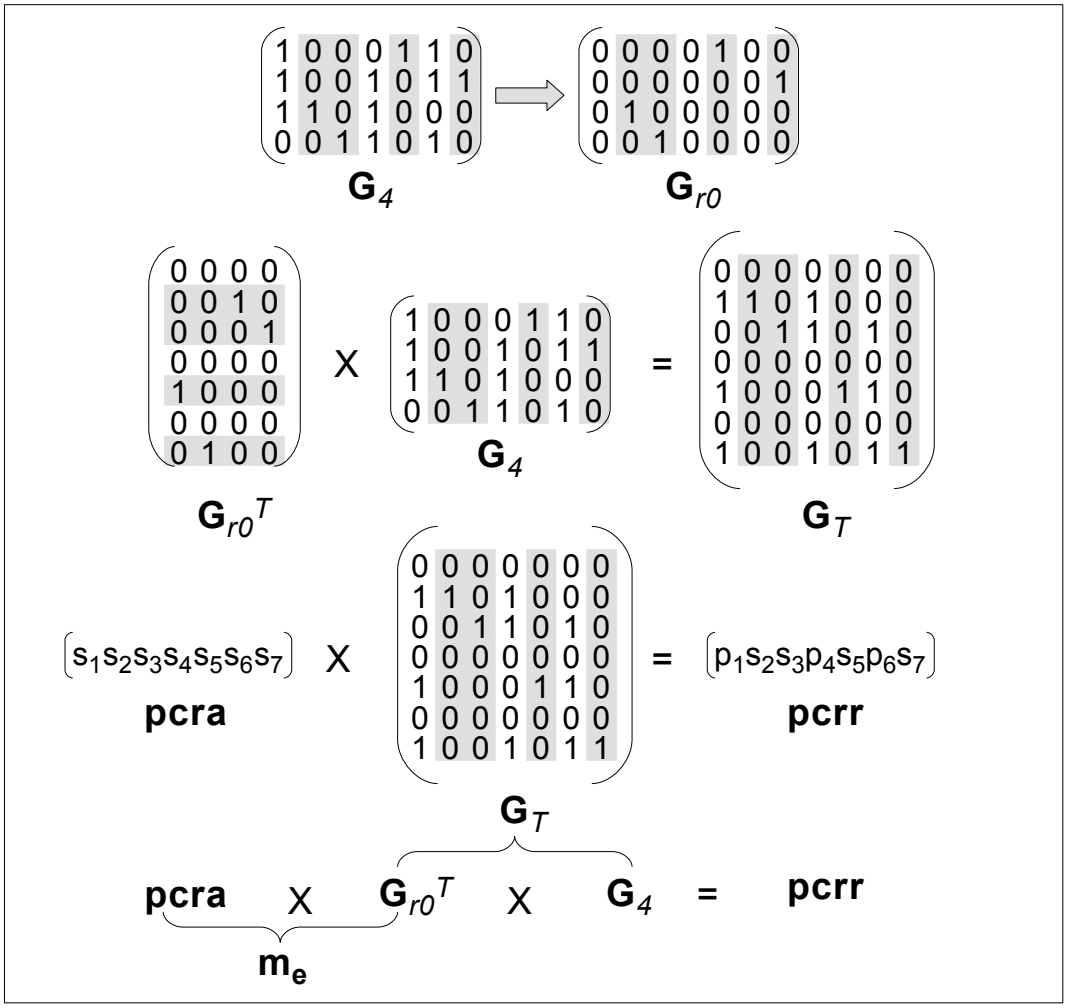

Figura 16 – Recodificação da palavra-código recebida

A justificativa para essa expressão é relativamente simples: após a RGJM, as CMC's da $\mathbf{G}_n$ constituem uma matriz ortogonal, similar às matrizes $\mathbf{E}_{nm}$ apresentadas na seção 2.2.3. Ao criar a $\mathbf{G}_{r0}$ a partir dessas colunas e zerando as demais se está na verdade extraindo a matriz ortogonal de $\mathbf{G}_n$ e ao efetuar sua transposição se está obtendo sua inversa. Se a $\mathbf{G}_{r0}^T$ for multiplicada pela $\mathbf{G}_n$ o resultado será uma matriz de dimensão $n \times n$, cujas posições correspondentes às posições das CMC's serão colunas de uma matriz identidade e portanto preservarão a posição dos símbolos mais confiáveis. A Figura 16 ilustra esse fato para o exemplo anterior tratado na Figura 14. Como se pode notar na figura, devido à propriedade associativa da multiplicação de matrizes, pode-se obter a palavra-código recebida recodificada **pcrr** de duas formas equivalentes, conforme abaixo:

$$\begin{aligned} \mathbf{pcrr} &= \mathbf{m}_e \times \mathbf{G}_n, \quad \mathbf{m}_e = \mathbf{pcra} \times \mathbf{G}_{r0}{}^{\mathrm{T}} \\ \mathbf{pcrr} &= \mathbf{pcra} \times \mathbf{G}_T, \quad \mathbf{G}_T = \mathbf{G}_{r0}{}^{\mathrm{T}} \times \mathbf{G}_n \end{aligned} \tag{3.10}$$

embora ambas as formas na equação (3.10) sejam equivalentes, a primeira tem a vantagem de fornecer um acesso direto à mensagem embaralhada $\mathbf{m}_e$, da qual serão modificados um ou mais símbolos considerados mais prováveis de estar em erro, como foi discutido na subseção 3.1.4.4. Além disso a primeira expressão da equação (3.10) não necessita armazenar a matriz $\mathbf{G}_T$ de dimensões n x n. Neste exemplo, caso os símbolos mais confiáveis recebidos, que são s_2, s_3, s_5 e s_7, estejam isentos de erros então se terá restaurado e recuperado corretamente os demais símbolos que terão o valor p_1, p_4 e p_6, todos dependentes apenas dos mais confiáveis. De fato, da multiplicação matricial ilustrada na Figura 16, vê-se que:

$$\begin{aligned} p_1 &= s_2 + s_5 + s_7 \\ p_4 &= s_2 + s_3 + s_7 \\ p_6 &= s_3 + s_5 + s_7 \end{aligned} \tag{3.11}$$

ou seja, os símbolos p_1, p_4 e p_6 foram restaurados exclusivamente a partir dos símbolos mais confiáveis e linearmente independentes s_2, s_3, s_5 e s_7.

O exemplo acima ilustra ainda o fato de que a solução encontrada, embora seja a melhor possível no caso, não é ideal. O vetor de índices das posições mais confiáveis nesse exemplo valia **S** = [7 5 2 1 3 4 6]. Logo, a solução ideal teria sido restaurar os símbolos das posições 3, 4 e 6 a partir da decodificação abrupta dos símbolos 7,5,2 e 1. Isso porém não foi possível devido ao fato das posições 7,5,2 e 1 não serem LI. Portanto, vê-se, nesse exemplo, que está sendo restaurado um símbolo de uma posição mais confiável, p_1, a partir de outro de uma posição menos confiável, s_3. Essa inversão de confiabilidade ocorre de forma ainda mais drástica para códigos mais longos. Conclui-se portanto pela necessidade de utilizar mensagens alternativas, na tentativa de corrigir possíveis erros das posições mais confiáveis e LI selecionadas. Os critérios para a seleção dessas mensagens alternativas serão discutidos na subseção 3.1.4.4.

3.1.4 Geração do subconjunto de palavras-código candidatas

Uma vez determinado o conjunto de símbolos mais confiáveis e que constituem um CI, pode-se recodificá-los por meio da matriz $\mathbf{G}_n$, obtendo os demais sím-

bolos. Porém, os símbolos mais confiáveis podem, por sua vez, também estar incorretos. Nesse caso, a palavra-código obtida por meio da recodificação estará também incorreta.

Cabe portanto avaliar as probabilidades de erro de cada símbolo ou conjunto de símbolos dentre os mais confiáveis e, em função dessa avaliação, gerar palavras alternativas modificando os símbolos mais confiáveis e recodificando-os, obtendo assim um conjunto de candidatas à correta decodificação. Dentre o conjunto de candidatas será então selecionada a que tiver a menor distância euclidiana para a palavra-código recebida.

Note-se porém que se forem feitas todas as possíveis alterações nos k símbolos mais confiáveis haverá um total de 2^k alternativas e será necessário executar o mesmo número de comparações da decodificação MLD. É portanto fundamental poder estimar as probabilidades de erros dos vários símbolos ou conjuntos de símbolos dentre os mais confiáveis de maneira a selecionar apenas aquelas alternativas com maior probabilidade de erro, delimitando assim o número de comparações necessárias.

3.1.4.1 Probabilidade de erro de um símbolo não ordenado

Este estudo irá supor a palavra codificada composta de n símbolos binários, os quais são modulados segundo um processo BPSK (do inglês *Binary Phase Shift Keying*), onde cada símbolo assume o valor +1 ou – 1 e são transmitidos através de um canal, onde sofrem a interferência de ruído aditivo gaussiano branco. Sempre que o nível de ruído adicionado é de tal monta que leva um símbolo de valor +1 a assumir um valor negativo, ou um símbolo de valor – 1 a assumir um valor positivo, estão satisfeitas as condições para a ocorrência de um erro. Esse erro ocorrerá em um decodificador que atribua um valor + 1 a todos os símbolos com valores positivos e – 1 a todos os símbolos com valores negativos.

Se a relação sinal ruído for conhecida, a probabilidade de ocorrência de um erro em um símbolo qualquer será facilmente determinável, uma vez que sua distribuição é normal.

A relação sinal ruído *SNR* (da sigla em inglês "*Signal* to *Noise Ratio*") pode ser expressa como (BLAHUT, 2003, cap. 12)

$$SNR = E_b / N_o \tag{3.12}$$

onde E_b é a energia do sinal de interesse por símbolo e N_o é a densidade espectral de energia do ruído em Watt / Hz.

A energia por símbolo é dada por:

$$E_b = A^2 T \tag{3.13}$$

onde A é a amplitude do sinal (+ 1 ou – 1 em nosso caso) e T a duração do sinal. Por outro, lado a variância do ruído é definida (BLAHUT, 2003, cap. 12) como:

$$\sigma^2 = N_o / 2T \tag{3.14}$$

A partir das equações acima vê-se que o desvio padrão do valor do ruído sobreposto ao sinal é dado por:

$$\sigma = \frac{A}{\sqrt{(2 E_b / N_o)}} = \frac{A}{\sqrt{(2 SNR)}} \tag{3.15}$$

Geralmente, porém, a relação sinal ruído é expressa em decibéis, sendo que então vale:

$$E_b / N_{o\ dB} = 10 \log_{10}(SNR) \tag{3.16}$$

Portanto, conhecida a relação sinal ruído em decibéis, o valor do desvio padrão do ruído é facilmente obtido das equações acima e vale:

$$\sigma = \frac{A}{\sqrt{(2 \cdot 10^{\frac{E_b / N_{o\ dB}}{10}})}} \tag{3.17}$$

Finalmente, a equação (3.14) considera que toda a energia disponível para o envio da mensagem é igualmente distribuída por todos os símbolos da mensagem. Em teoria da codificação, entretanto, costuma-se comparar o desempenho de uma mensagem codificada com seu equivalente sem codificação, e, neste caso, considera-se que a mesma energia originalmente distribuída pelos k símbolos da mensagem sem codificação será distribuída pelos n símbolos da mensagem codificada. Portanto, a energia por símbolo codificado passa a ficar multiplicada por um fator $R = k / n$, o que faz com que a equação (3.12), no caso da mensagem codificada, seja reescri-

ta como:

$$SNR_c = E_b \cdot R / N_o = \frac{E_b \cdot (k/n)}{N_o} \tag{3.18}$$

e a correspondente equação (3.14) fique:

$$\sigma = \frac{A}{\sqrt{(2 \cdot R \cdot 10^{\frac{E_b/N_o}{10}})}} \tag{3.19}$$

A expressão para a densidade de probabilidade do ruído superposto a um sinal de amplitude *A* = +1 será portanto:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{-\frac{1}{2}\frac{(x-1)^2}{\sigma^2}} \tag{3.20}$$

com σ dada pela equação (3.19). O valor de *f*(*x*) pode ser rapidamente obtido no Matlab por meio da função `normpdf`(*x*,μ,σ).

A título de exemplo, será considerada a probabilidade de erro de um símbolo oriundo de uma palavra codificada, com amplitude A = +1, relação sinal ruído $E_b/N_{o\ db}$ = 1 dB, comprimento da mensagem *k* = 12 símbolos e comprimento da palavra codificada *n* = 24 símbolos. Neste caso, a equação (3.19) fornece σ = 0.8913. Resultará, portanto, uma distribuição de valores segundo uma normal, com parâmetros μ = +1 e σ = 0.8913. Na Figura 17, a área hachurada representa a probabilidade do valor do símbolo assumir valores negativos e, portanto, estar sujeito a uma interpretação errônea pelo decodificador.

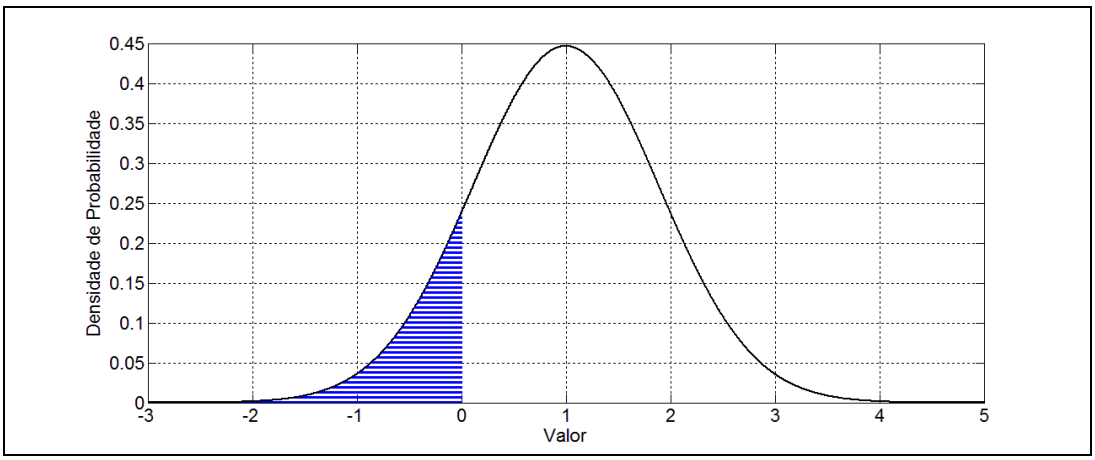

Figura 17 – Probabilidade de erro de um único símbolo, não ordenado

O valor da probabilidade de erro neste caso pode ser obtida de uma tabela, ou, por exemplo, com a função do Matlab `normcdf`(0,μ,σ). Neste exemplo específico, o valor resultará igual a 0.1309, ou seja, 13,1 % de probabilidade de erro.

3.1.4.2 Probabilidade de erro de um símbolo dentro de uma ordenação

A análise da subseção anterior é válida para qualquer dos símbolos transmitidos em uma palavra-código, quando esses não são ordenados por confiabilidade. Entretanto, o que se pretende determinar é a probabilidade de erro em cada um dos *n* símbolos quando estes são ordenados em ordem decrescente do módulo de seu valor. A idéia é que quanto maior o módulo do valor do símbolo recebido – quanto mais distante de zero este valor estiver – maior será sua "confiabilidade". Como será visto, neste caso a distribuição de probabilidade de cada símbolo deixa de ser normal.

Supondo agora que tenha sido recebida uma palavra-código, composta de *n* símbolos ordenados segundo a ordem decrescente do módulo de seus valores, ou seja, por ordem decrescente de sua confiabilidade. Antes da ordenação, todos os símbolos recebidos tinham distribuição de probabilidades normal e taxa de erros como mostrado na Figura 17. Deseja-se agora conhecer a distribuição e taxa de erros para cada um dos símbolos após a ordenação.

Inicialmente será analisado o caso mais simples, onde $n = 2$. Neste caso, tem-se o primeiro símbolo da ordenação como o mais confiável e o segundo como o menos confiável. A densidade de probabilidade p_{mais}, de que o símbolo, tendo assumido um valor qualquer x_p, seja o mais confiável, será dada pela probabilidade de que o símbolo tenha assumido o valor x_p, condicionada à probabilidade de que o outro símbolo tenha assumido um valor em módulo inferior ao módulo de x_p. Chamando o valor assumido pelo símbolo mais confiável de *v1*, o assumido pelo menos confiável de *v2*, pode-se escrever:

$$p_{mais} = P(v1 = x_p \mid |v2| < |x_p|) \tag{3.21}$$

Como v1 e v2 são estatisticamente independentes, tem-se

$$p_{mais} = P(v1 = x_p) \times P(|v2| < |x_p|) \tag{3.22}$$

com:

$$\begin{aligned} P(v1 = x_p) &= f(x_p)dx \\ P(|v2| < |x_p|) &= \int_{x_p}^{-x_p} f(u)\,du \end{aligned} \tag{3.23}$$

logo, usando a (3.22) vem:

$$p_{mais} = \underbrace{f(x_p) \times \int_{x_p}^{-x_p} f(ud)\,du}_{f_{mais}(x_p)} \times dx \tag{3.24}$$

onde identifica-se facilmente os dois primeiros termos do produto no segundo membro da (3.24) como sendo a função densidade de probabilidade para o símbolo mais confiável, como mostrado pela chave sob esses termos.

Em termos da expressão (3.20) e de sua integral, a função densidade de probabilidade para a probabilidade expressa na equação (3.21) pode ser colocada como:

$$f_{mais}(x_p) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\frac{(x_p-1)^2}{\sigma^2}} \times \frac{1}{\sqrt{2\pi}\sigma} \int_{x_p}^{-x_p} e^{-\frac{1}{2}\frac{(x-1)^2}{\sigma^2}} dx \tag{3.25}$$

A Figura 18 indica as duas parcelas do produto da equação (3.21) (e igualmente da equação (3.25)), para o caso de x_p = – 0,5. O traço com círculo marca o valor da densidade de probabilidade do símbolo assumir o valor x_p = – 0,5, enquanto que a área hachurada indica a probabilidade do outro símbolo assumir um valor em módulo inferior a 0,5, ou seja, estar contido no intervalo [– 0,5 +0,5].

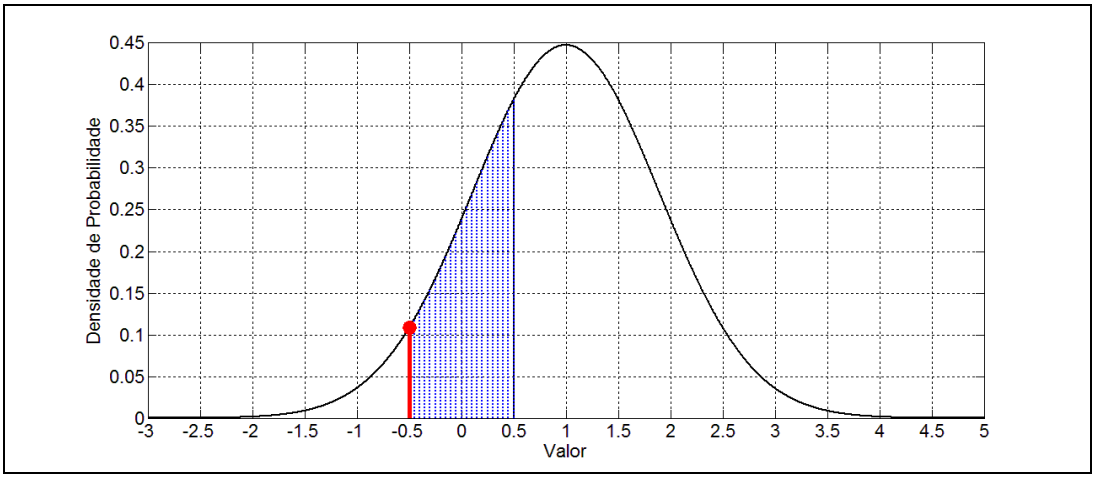

Figura 18 – Probabilidade do módulo do símbolo menos confiável ser inferior a 0,5

Para o símbolo menos confiável, a distribuição de probabilidades é obtida de forma semelhante, com sua densidade de probabilidade dada por:

$$p_{menos} = P(v1 = x_p \mid |v2| > |x_p|) \tag{3.26}$$

ou seja, a área a ser considerada para o segundo termo do produto agora é a área externa, ou complementar, à área mostrada na Figura 18.

Nesse caso, de forma análoga ao que foi feito para p_{mais}, a expressão para p_{menos} fica:

$$p_{menos} = \underbrace{f(x_p) \times \left(1 - \int_{x_p}^{-x_p} f(ud)\,du\right)}_{f_{menos}(x_p)} \times dx \tag{3.27}$$

de onde é possível, usando novamente a equação (3.20), obter a densidade de probabilidade no ponto x_p para o símbolo menos confiável:

$$f_{menos}(x_p) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\frac{(x_p-1)^2}{\sigma^2}} \times \left(1 - \frac{1}{\sqrt{2\pi}\sigma} \int_{x_p}^{-x_p} e^{-\frac{1}{2}\frac{(x-1)^2}{\sigma^2}} dx\right) \tag{3.28}$$

A determinação da distribuição completa para p_{mais} e p_{menos} precisa ser realizada numericamente, uma vez que depende da área sob a curva normal, que só pode ser obtida por meio de tabelas ou numericamente. No Matlab isso pode ser facilmente obtido fazendo uso das funções `normpdf`(x,μ,σ) e `normcdf`(x,μ,σ). O resultado está mostrado na Figura 19.

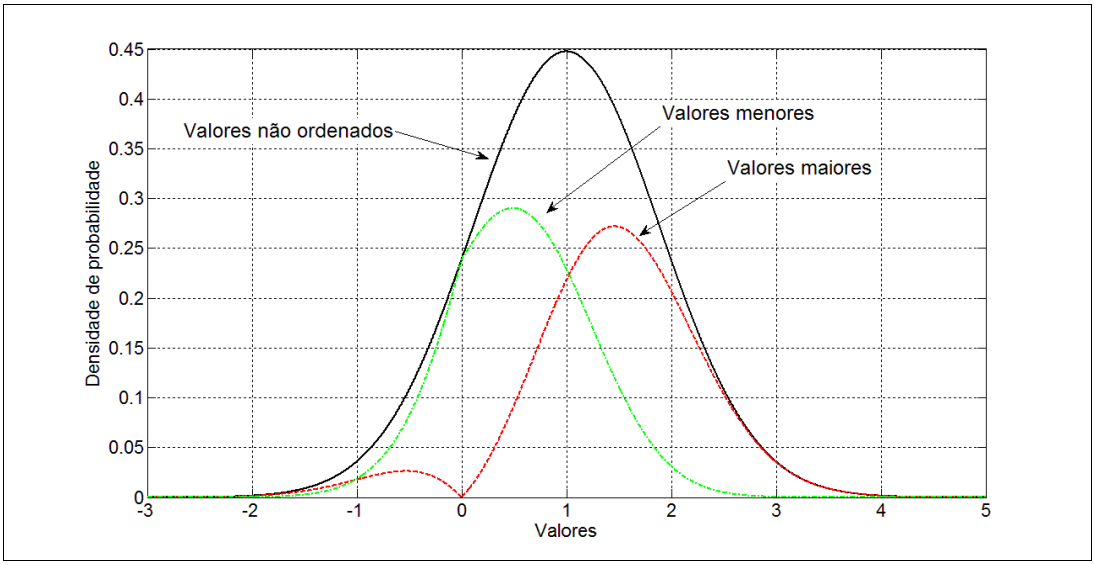

Figura 19 – Distribuição de probabilidades para dois símbolos ordenados

Nessa figura percebe-se que a soma das curvas parciais resulta – em cada ponto – igual à curva normal. Cada uma das áreas das curvas parciais portanto vale 0,5, indicando que ainda não foram normalizadas, e não servem para se extrair o valor da probabilidade de um símbolo assumir um valor negativo e portanto possibilitar ocorrência de um erro.

A Figura 20 mostra as curvas parciais normalizadas e também as áreas correspondentes às respectivas probabilidades de erro de símbolo, áreas essas obtidas integrando-se as equações (3.25) e (3.28). Essas áreas, quando computadas para os valores do exemplo, resultam nos valores 0,0563 para o símbolo mais confiável, e 0,2058 para o menos confiável. Ou seja, os 13,1 % de erros da distribuição original agora se dividem de forma polarizada: 5,6 % aparecem no símbolo mais confiável, enquanto que 20,6 % ficam com o menos confiável. No total tem-se em média os 13,1 % originais, que é igual à média dos dois valores calculados.

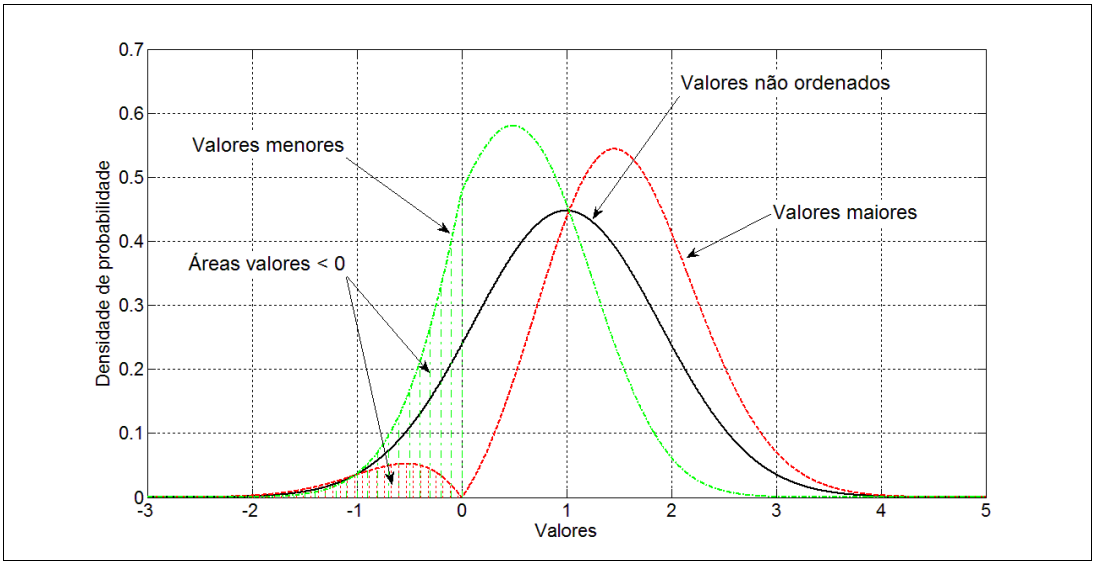

Figura 20 – Distribuição de probabilidades normalizada para dois símbolos ordenados

Para validar os resultados obtidos através da teoria e do cálculo numérico, foi feita a seguinte simulação: foram geradas 5 x 10^5 palavras código compostas de dois símbolos, ambos valendo +1 – portanto já modulados em BPSK (do inglês *Binary Phase Shift Keying*) – e sobre cada um foi superposto um valor aleatório distribuído segundo uma distribuição normal de $\mu = 0$ e desvio padrão $\sigma = 0{,}8913$, idêntico ao utilizado nos cálculos mostrados nas figuras anteriores. O valores foram então ordenados por ordem decrescente de módulo e a seguir lançados em um histograma, composto por 200 intervalos, utilizando as funções do Matlab `ecdf`() ("**e**mpirical **c**umulative probability **d**istribution **f**unction") e `ecdfhist`() ("**hist**ogram from **e**mpirical **c**umulative **f**unction"). Os histogramas gerados podem ser visto na Figura 21 onde pode-se notar uma excelente conformidade com as curvas teóricas, validando os resultados teóricos.

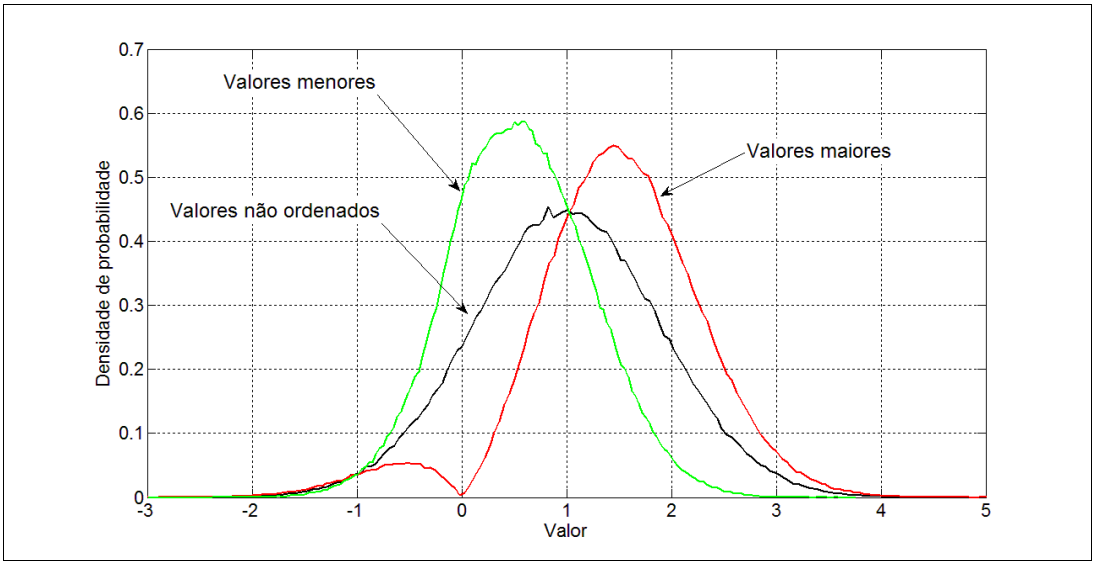

Figura 21 – Distribuição de probabilidades para dois símbolos ordenados (simulação)

Para obter uma generalização dos resultados conseguidos até este ponto para palavras com uma quantidade *n* qualquer de símbolos é ainda interessante analisar o caso particular para 3 símbolos. Para o símbolo mais confiável é necessário impor que para cada valor x_p que esse símbolo assuma, os outros dois estarão dentro de um intervalo [– x_p, x_p]. Chamando de *v1*, *v2* e *v3* os valores assumidos respectivamente pelos símbolos mais confiável, de confiabilidade intermediária e menos confiável, tem-se, para o símbolo mais confiável, a probabilidade dada por:

$$p_{mais} = P(v1 = x_p \mid |v2| < |x_p| \mid |v3| < |x_p|) \tag{3.29}$$

portanto, neste caso tem-se o valor da densidade de probabilidade no ponto x_p, multiplicada pelo quadrado da área contida sob a curva no intervalo [– x_p, + x_p], e pode-se escrever:

$$p_{mais} = P(v = x_p) \times P(|v| < |x_p|)^2 \tag{3.30}$$

A função densidade de probabilidade para p_{mais} neste caso fica:

$$f_{mais}(x_p) = \frac{1}{\sqrt{2\pi}\,\sigma} e^{-\frac{1}{2}\frac{(x_p-1)^2}{\sigma^2}} \times \left(\frac{1}{\sqrt{2\pi}\,\sigma} \int_{x_p}^{-x_p} e^{-\frac{1}{2}\frac{(x-1)^2}{\sigma^2}} dx \right)^2 \tag{3.31}$$

Já para o símbolo de confiabilidade intermediária, é necessário considerar duas possibilidades:

$$p_{inter} = P(v2=x_p \mid |v1|>|x_p| \mid |v3|<|x_p|) + P(v2=x_p \mid |v1|<|x_p| \mid |v3|>|x_p|) \tag{3.32}$$

como as distribuições originais para *v1*, *v2* e *v3* são idênticas, pode-se escrever:

$$p_{inter}=2\times P(v=x_p) \times P(|v|>|x_p|) \times P(|v|<|x_p|) \tag{3.33}$$

o que fornece uma densidade de probabilidade dada por:

$$f_{inter}(x_p)=\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{1}{2}\frac{(x_p-1)^2}{\sigma^2}}\times\left(1-\frac{1}{\sqrt{2\pi}\sigma}\int_{x_p}^{-x_p}e^{-\frac{1}{2}\frac{(x-1)^2}{\sigma^2}}dx\right)\times\frac{1}{\sqrt{2\pi}\sigma}\int_{x_p}^{-x_p}e^{-\frac{1}{2}\frac{(x-1)^2}{\sigma^2}}dx \tag{3.34}$$

finalmente, para o símbolo de menor confiabilidade, tem-se:

$$p_{menos}=P(v1=x_p \mid |v2|>|x_p| \mid |v3|>|x_p|) \tag{3.35}$$

que também pode ser reescrita como:

$$p_{menos}=P(v=x_p) \times P(|v|>|x_p|)^2 \tag{3.36}$$

sendo que a correspondente densidade de probabilidade fica:

$$f_{menos}(x_p)=\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{1}{2}\frac{(x_p-1)^2}{\sigma^2}}\times\left(1-\frac{1}{\sqrt{2\pi}\sigma}\int_{x_p}^{-x_p}e^{-\frac{1}{2}\frac{(x-1)^2}{\sigma^2}}dx\right)^2 \tag{3.37}$$

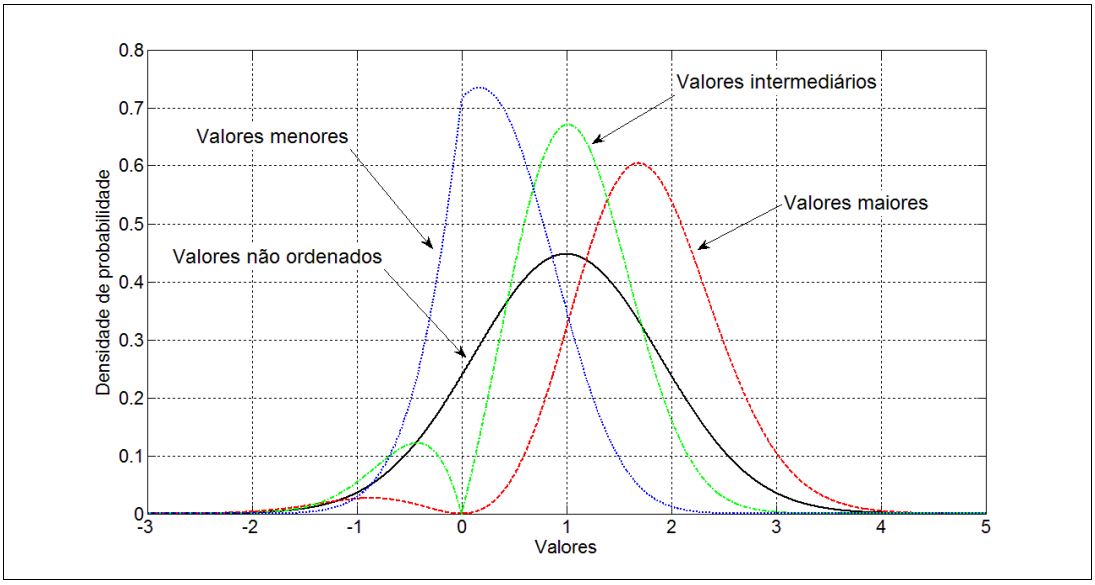

Figura 22 – Distribuição de probabilidades para três símbolos ordenados

A distribuição de probabilidades para este caso, obtida por meios numéri-

cos, está mostrada na Figura 22, sendo que os correspondentes valores empíricos obtidos de forma análoga ao caso para n = 2 estão mostrados na Figura 23.

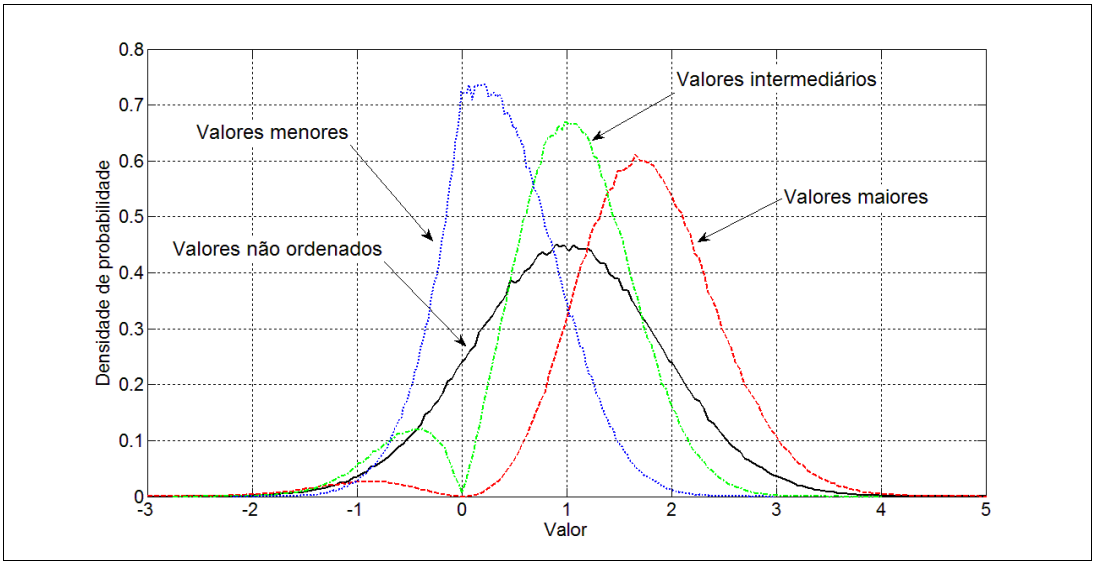

Figura 23 – Distribuição de probabilidades para três símbolos ordenados, (simulação)

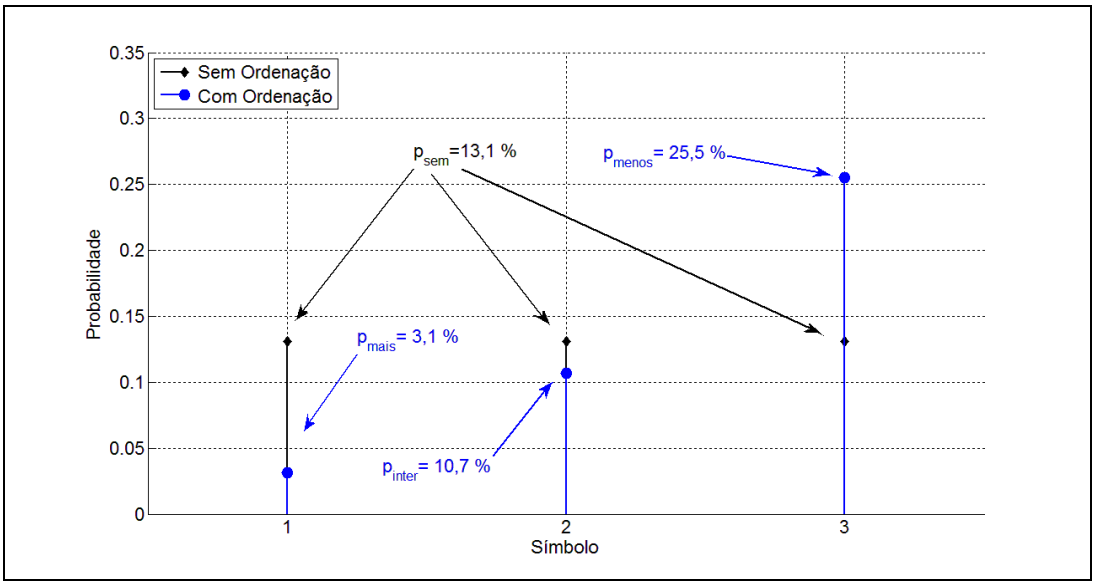

Figura 24 – Probabilidade de erro para 3 símbolos com e sem ordenação

A Figura 24 mostra o comportamento das probabilidades dos valores dos símbolos incorrerem em erro por assumir – neste caso – valores negativos. Novamente, pode-se observar que a média das probabilidades parciais corresponde à probabilidade sem ordenação. Como era de se esperar, a ordenação torna alguns símbolos mais confiáveis em detrimento de outros. O que esta análise permite, entretanto, é a determinação quantitativa desses valores.

Até este ponto, foram analisados os casos particulares de 2 e 3 símbolos, com o objetivo de facilitar a compreensão do problema e também de mostrar a tendência geral, para permitir encontrar uma solução geral para as distribuições de probabilidades, o que será abordado a seguir.

Para facilitar a notação, os n símbolos serão numerados de 0 a $n-1$, chamado-os de s_0, s_1, .. s_{n-1}, e às suas respectivas probabilidades de ps_0, ps_1, .. ps_{n-1}, com densidades de probabilidade fs_0, fs_1, .. fs_{n-1}, onde s_0 corresponde ao símbolo mais confiável e s_{n-1} ao menos confiável.

Como foi visto, a probabilidade do símbolo mais confiável de assumir um determinado valor x_p depende da probabilidade desse símbolo assumir o valor x_p, condicionada à probabilidade de todos os demais de manterem o módulo de seus valores inferior ao módulo de x_p. Simbolicamente, pode-se expressar essa condição, para n símbolos, como:

$$ps_0 = P(v = x_p) \times P(|v| < |x_p|)^{n-1} \tag{3.38}$$

Já para o símbolo menos confiável sabe-se que sua densidade de probabilidade de assumir um determinado valor x_p depende da probabilidade desse símbolo assumir o valor x_p, condicionada à probabilidade de todos os demais de manterem o módulo de seus valores superior ao módulo de x_p. Isso resulta em:

$$ps_{n-1} = P(v = x_p) \times P(|v| > |x_p|)^{n-1} \tag{3.39}$$

Para os demais símbolos intermediários, a densidade de probabilidade será obtida adicionando-se as parcelas resultantes de todas as possíveis combinações dos que estarão com a confiabilidade maior e menor que o símbolo em questão. Isso permite deduzir a seguinte expressão geral, a qual engloba inclusive os dois casos das equações (3.38) e (3.39):

$$ps_i = \binom{n-1}{i} \times P(v = x_p) \times P(|v| < |x_p|)^{n-1-i} \times P(|v| > |x_p|)^i \qquad i = 0 \ldots n-1 \tag{3.40}$$

e a correspondente função densidade de probabilidade fica:

$$fs_i(x_p) = \binom{n-1}{i} \cdot \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\frac{(x_p-1)^2}{\sigma^2}} \times \left(\frac{1}{\sqrt{2\pi}\sigma} \int_{x_p}^{-x_p} e^{-\frac{1}{2}\frac{(x-1)^2}{\sigma^2}} dx \right)^{n-1-i} \times \left(1 - \frac{1}{\sqrt{2\pi}\sigma} \int_{x_p}^{-x_p} e^{-\frac{1}{2}\frac{(x-1)^2}{\sigma^2}} dx \right)^{i} \quad (3.41)$$

A partir da equação (3.40) pode-se obter as distribuições de probabilidade, assim como as probabilidades de erro para palavras com quaisquer quantidades de símbolos. A título de exemplo, a Figura 25 mostra a distribuição para *n* = 24:

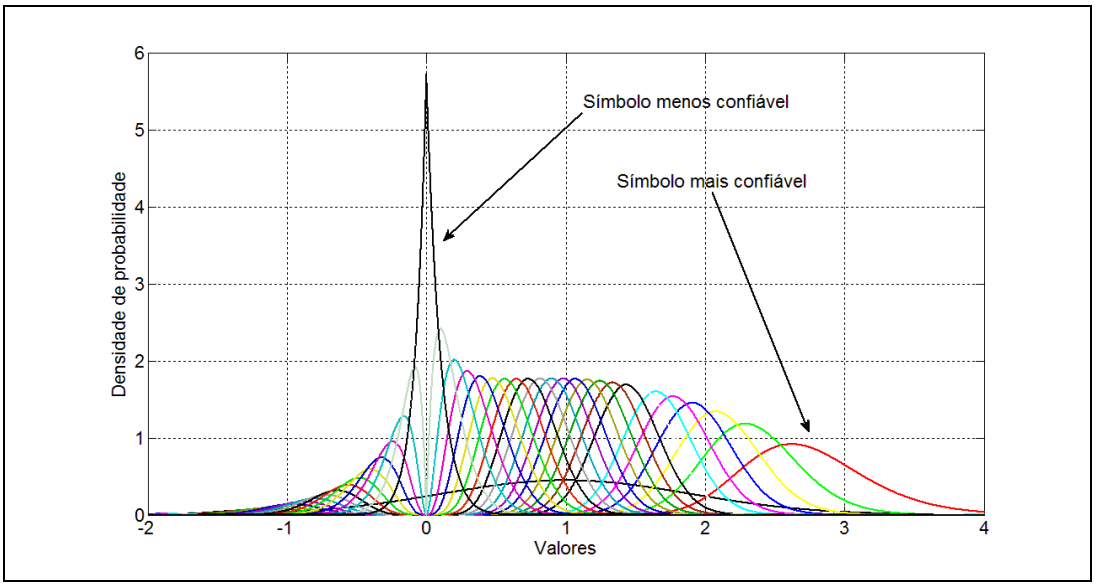

Figura 25 – Distribuição de probabilidades para 24 símbolos ordenados

As correspondentes probabilidades de erro para cada símbolo estão na Figura 26:

As probabilidades ilustradas na Figura 26 foram obtidas a partir da equação (3.5) por:

$$ps_i = \int_{-\infty}^{0} fs_i(x_p)\, dx_p \quad (3.42)$$

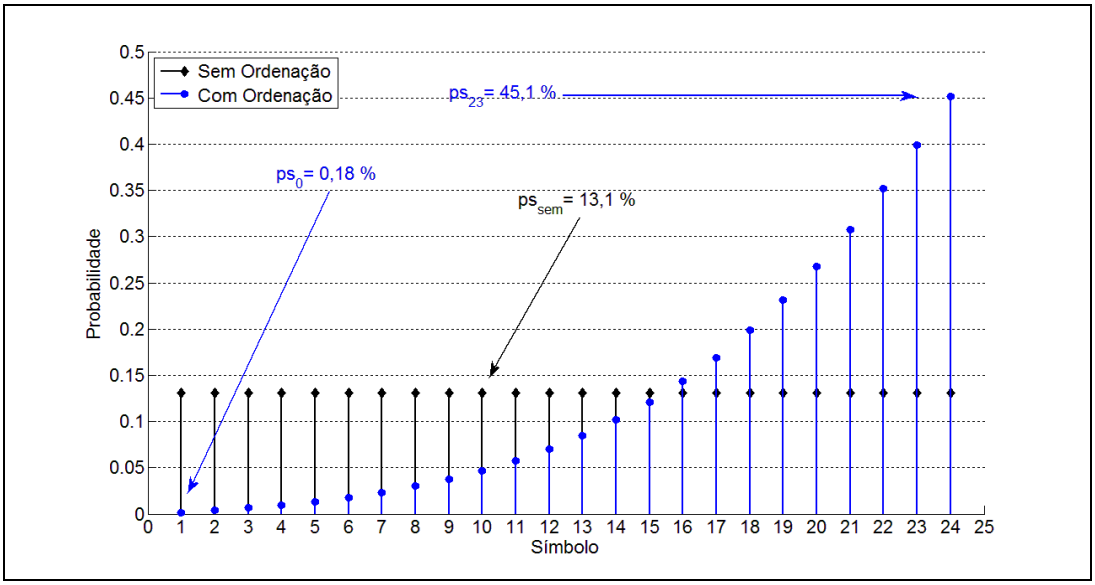

Figura 26 – Probabilidade de erro para 24 símbolos, com e sem ordenação

Também para este caso, com a finalidade de validar os resultados teóricos, foi feita uma simulação com 10^5 palavras código geradas aleatoriamente, constituídas de 24 símbolos cada e os resultados estão mostrados na Figura 27, onde pode-se ver que as simulações validam perfeitamente os resultados teóricos obtidos numericamente. Além disso, as áreas sob as curvas da Figura 27, calculadas no intervalo $[-\infty, 0]$ (equação (3.42)) são, para todos os efeitos práticos, idênticas aos valores mostrados na Figura 26 e por isso não foram repetidas aqui.

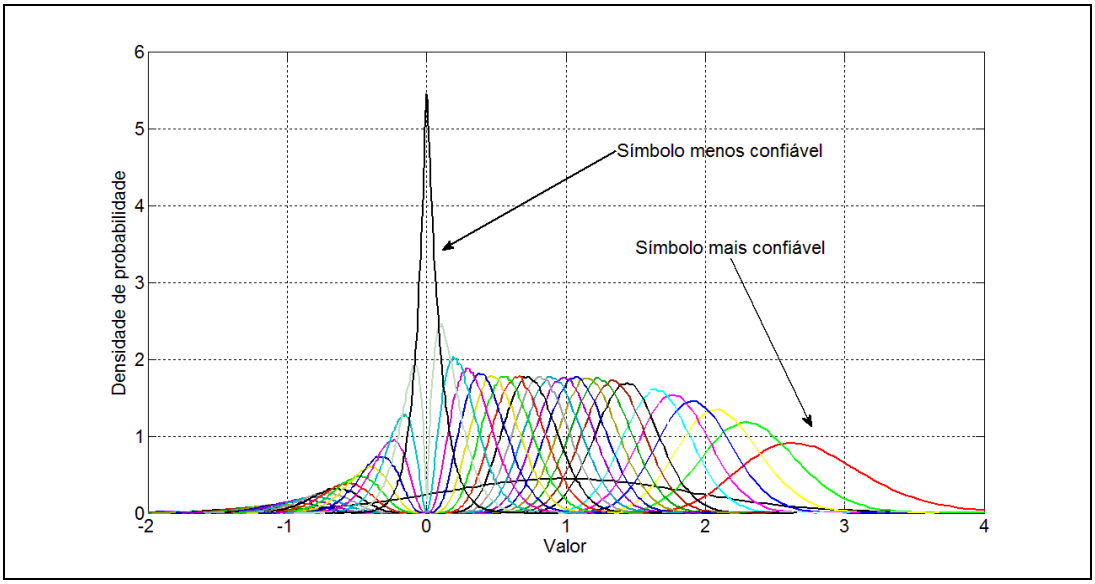

Figura 27 – Distribuição de probabilidades para 24 símbolos ordenados (simulação)

3.1.4.3 Probabilidade conjunta de erro de mais de um símbolo, dentro de uma ordenação

As considerações e deduções da seção anterior permitem determinar a probabilidade de um determinado símbolo, dentro de um conjunto ordenado de símbolos de uma palavra-código recebida, estar em erro.

Infelizmente, essa informação não é suficiente para se selecionar alternativas à palavra código mais confiável, como sugerido na subseção 3.1.4.

Conhecida a palavra-código candidata mais confiável, o que se deseja saber, não é apenas a probabilidade de, por exemplo, seu símbolo menos confiável estar em erro. O que é necessário determinar é a probabilidade desse símbolo estar em erro condicionada à probabilidade dos demais estarem corretos. Ou seja, seria desejável determinar a probabilidade de se ter, por exemplo, um único erro no símbolo menos confiável, ou então, de estarem apenas os dois símbolos menos confiáveis em erro, e assim por diante. Trata-se portanto de determinar probabilidades conjuntas.

Uma primeira tentativa de responder a essa pergunta poderia ser realizada utilizando-se o conhecimento das probabilidades ps_0, ps_1, etc.. determinadas na subseção anterior. Por exemplo, ainda utilizando a notação adotada na nessa subseção, seria possível tentar determinar a probabilidade PU_{n-1} de somente o símbolo menos confiável s_{n-1} estar em erro como:

$$PU_{n-1} = p_{n-1} \times \prod_{s=0}^{n-2} (1 - p_s) \tag{3.43}$$

Ou ainda, genericamente, a probabilidade de que dentre os n símbolos de uma palavra-código apenas os símbolos i, j, k, etc.. estejam em erro seria dada por:

$$PU_{i,k,j,\ldots} = \prod_{s=i,k,j,\ldots} p_s \times \prod_{s \neq i,k,j,\ldots} (1 - p_s) \tag{3.44}$$

Infelizmente, essas equações só estariam corretas caso as probabilidades p_s se referissem a eventos estatisticamente independentes, o que, devido à ordenação prévia dos símbolos, não é o caso.

Assim as equações (3.43) e (3.44) fornecem apenas valores aproximadamente corretos. Um cálculo mais exato envolve determinação de integrais múltiplas, e é o objeto do estudo da disciplina de ordenação estatística, como abordada por

exemplo em (DAVID 2003) e (BALAKRISHNAM 1998). Como a complexidade desses cálculos supera a capacidade computacional disponível atualmente, recorreu-se mais uma vez a simulações, fazendo porém uso dos conhecimentos sobre a formação de conjuntos de informação como descritos na seção 2.2, o que será detalhado a seguir.

3.1.4.4 Obtenção do conjunto de candidatas mais prováveis por meio de simulações

Como foi visto no início da seção 3.1.4, uma vez obtida a mensagem mais confiável, constituída pelas k posições mais confiáveis e linearmente independentes da palavra-código recebida, é ainda interessante determinar quais conjuntos de símbolos dessa mensagem têm a maior probabilidade de estar em erro e, comutando o valor desse símbolos, criar mensagens alternativas.

A solução quantitativa desse problema esbarra em duas dificuldades: em primeiro lugar, como foi visto nas subseções anteriores 3.1.4.1 a 3.1.4.3, a determinação da probabilidade de erro de determinados conjuntos de símbolos dentro de uma ordenação é computacionalmente muito complexa. Em segundo lugar, ainda que o problema anterior tivesse uma solução simples, é preciso levar em conta o fato de que os símbolos da mensagem mais confiável derivam de posições variáveis dentro da palavra-código recebida. Para melhor esclarecer essa afirmação, será dado um exemplo para o código $C(15,7,5)$. Dada uma palavra-código recebida e uma reordenação qualquer de seus 15 símbolos, sabe-se, a partir do resultado apresentado na subseção 3.1.2.1, que apenas seus $d_{hmin}{}^{\perp} - 1 = 3$ símbolos serão sempre LI. Sabe-se também, de acordo com os resultados da subseção 3.1.2.2, que todo e qualquer conjunto de $n - d_{hmin} + 1 = 11$ de seus símbolos será sempre LD. Portanto, dada uma palavra-código recebida para o código $C(15,7,5)$, ao se formar a mensagem constituída por seus $k = 7$ símbolos mais confiáveis e LI, resultará que os três primeiros símbolos dessa mensagem serão sempre obtidos a partir dos três símbolos mais confiáveis da palavra-código recebida, porém os demais 4 símbolos poderão ocupar posições variáveis entre a 4ª e a 11ª posição dentro da palavra-código recebida, uma vez que no pior caso, 11 colunas da matriz **G** rearranjada deverão ser examinadas para se encontrar garantidamente 7 colunas LI. Percebe-se portanto que, nesse caso, os 4 símbolos menos confiáveis da mensagem estarão

distribuídos de alguma maneira entre as 8 posições 4 a 11 da palavra-código recebida. Existem, teoricamente, $\binom{8}{4}=70$ formas diferentes de distribuir os 4 símbolos menos confiáveis entre as 8 possíveis posições, embora não necessariamente todas as possibilidades possam ocorrer.

A Figura 28 ilustra um possível exemplo de formação da mensagem mais confiável.

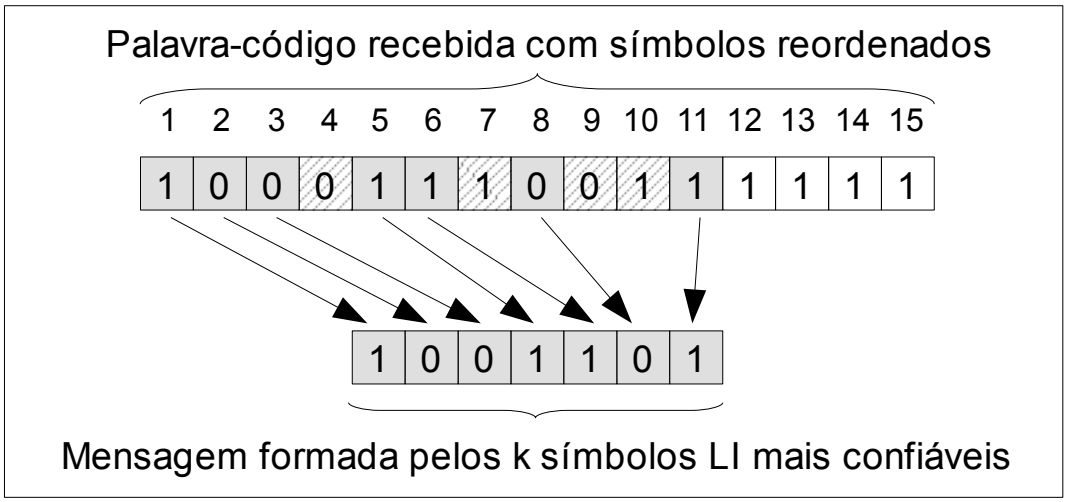

Figura 28 – Formação da mensagem

Nesse exemplo, o quarto símbolo mais confiável da mensagem corresponde ao quinto na palavra-código recebida, o quinto corresponde ao sexto, o sexto ao oitavo e, finalmente, o sétimo ao décimo primeiro mais confiável da palavra-código recebida. Desejando-se portanto, nesse exemplo, determinar a probabilidade dos últimos dois símbolos da mensagem formada estarem simultaneamente em erro, vê-se que isso corresponderia, nesse caso específico, a determinar a probabilidade de o décimo primeiro e do oitavo símbolos da palavra-código recebida estarem simultaneamente em erro. Para uma outra palavra-código recebida entretanto, essa mesma determinação envolveria outras posições de símbolos na palavra-código, dependendo de quais das posições entre a 4ª e a 11ª foram ocupadas por símbolos LI. Logo, além do conhecimento da probabilidade de determinados conjuntos de símbolos da palavra-código estarem em erro, é também necessário conhecer a probabilidade de ocorrência de cada um dos padrões de distribuição dos *k* símbolos mais confiáveis na palavra código. No exemplo considerado para o código $C(15,7,5)$ é necessário conhecer a probabilidade de ocorrência de cada uma das 70 possíveis distribuições dos 4 símbolos menos confiáveis da mensagem na palavra-código re-

cebida.

Em face das dificuldades discutidas até aqui, o problema da determinação quantitativa dos conjuntos de símbolos mais prováveis de estarem em erro na mensagem formada pelos *k* símbolos LI mais confiáveis da palavra-código recebida foi solucionado em duas etapas.

Na primeira foram levantadas, por meio de simulações, as probabilidades de incidência dos diversos padrões de distribuição de símbolos LI entre as posições $d_{hmin}^{\perp}$ e $n - d_{hmin} + 1$, para os diversos códigos. Isso foi feito utilizando-se o *script* Matlab `GeraProbPadr` (Script 7), o qual gera 10^5 permutações das colunas da matriz geradora do código e levanta todos os padrões resultantes para as *k* primeiras colunas LI, assim como suas probabilidades de incidência. Para tanto, para cada permutação gerada o *script* procede à eliminação modificada de Gauss, utilizando o *script* `GeraPadrLi` (Script 6), que nada mais é que uma versão adaptada do *script* `GeraGneGr0` (Script 5), para retornar o padrão de distribuição de colunas LI, em vez das matrizes $\mathbf{G}_n$ e $\mathbf{G}_{r0}$.

A quantidade teórica de padrões de distribuição de símbolos LI entre as posições $d_{hmin}^{\perp}$ e $n - d_{hmin} + 1$ pode ser determinada por:

$$nr_{comb\,LI} = \begin{pmatrix} n - d_{hmin}^{\perp} - d_{hmin} + 2 \\ k - d_{hmin}^{\perp} + 1 \end{pmatrix} \tag{3.45}$$

Ao se examinar os resultados dessa primeira etapa, é interessante observar que, para todos os códigos analisados, apenas um pequeno número de padrões ocorre em 99,99 % dos casos. Os demais padrões ou nunca ocorrem ou o fazem com uma taxa de incidência extremamente pequena, inferior a 1×10^{-4}. A Tabela 3.3 a seguir mostra os resultados obtidos para os principais códigos analisados, comparando a quantidade teórica de padrões com a efetivamente encontrada para taxas de incidência iguais ou superiores a 10^{-4}.

Tabela 3.3: Quantidade de padrões de colunas LI teórica e obtida por simulação

Código examinado	Quantidade Simulada – taxa ≥ 10^{-4}	Quantidade Teórica
$C(7,4,3)$	2	2
$C(15,7,5)$	25	70
$C(24,12,8)$	29	252
$C(48,24,12)$	132	10.400.600

Na segunda etapa da determinação das probabilidades de erros de conjuntos de símbolos da mensagem mais confiável pode-se então atingir o objetivo final, que é montar a matriz dos apagamentos que será aplicada à mensagem para obtenção das candidatas alternativas mais prováveis. Entende-se aqui como apagamento a comutação de um ou mais símbolos da mensagem mais confiável. Assim a matriz dos apagamentos será constituída de várias linhas de k colunas, cada coluna contendo zero na posição em que o símbolo da mensagem mais confiável é preservado e um na posição em que esse símbolo é alterado.

Inicialmente a matriz dos apagamentos é montada contendo uma linha toda nula (nenhuma inversão de símbolo, ou seja, corresponde à mensagem mais confiável original), depois k linhas, cada uma com apenas uma das colunas diferente de zero, depois $\binom{k}{2}$ linhas, cada uma com duas colunas diferentes de zero, a seguir $\binom{k}{3}$ linhas, cada uma com três colunas diferentes de zero e assim por diante.

O objetivo final da segunda etapa não é apenas montar a matriz dos apagamentos como descrito acima, o que é trivial, mas sim montar a matriz e ordenar suas linhas em ordem decrescente de probabilidade de incidência de seu padrão de erros. Além disso, é adicionalmente importante poder obter a probabilidade de ocorrência de cada um dos padrões de erro na matriz, de maneira a ter um instrumento quantitativo que permita truncar a mesma a partir de uma certa quantidade de linhas, em função da taxa de erros de decodificação pretendida.

Para atingir o objetivo dessa segunda etapa, o *script* `SimulPrb` (Script 9)

do Matlab monta inicialmente uma matriz composta de tantas linhas quantos forem os padrões de colunas LI encontrados na primeira etapa, e tantas colunas quantos forem os padrões de erro a analisar. Por exemplo, no caso do código $C(15,7,5)$, embora existam teoricamente 70 possíveis padrões de colunas LI, apenas 26 tem uma taxa de incidência significativa, assim a matriz terá 26 linhas. Para esse código, levando a análise até um máximo de três símbolos simultaneamente errados, o número de colunas dessa matriz será dado por $\binom{7}{0} + \binom{7}{1} + \binom{7}{2} + \binom{7}{3} = 64$ colunas. Portanto, para esse código será montada uma matriz de 26 linhas por 64 colunas. Cada coluna dessa matriz irá conter a taxa de erro de bit encontrada por simulação, para o padrão de colunas LI correspondente à sua linha. Por exemplo, para o código $C(15,7,5)$, os símbolos 6 e 7 da mensagem mais confiável, que são os menos confiáveis dentre todos, poderão incorrer em erro quando oriundos das posições 6 e 7 da palavra-código recebida. Mas poderão também incorrer em erro, embora com uma taxa de erro diferente, quando oriundos das posições 8 e 11 da palavra-código recebida, como mostrado no exemplo da Figura 28, na página 77. O *script* `SimulPrb` (Script 9) do Matlab levanta então, por meio de simulação, as taxas de erro parciais para cada uma das posições da matriz. Esse *script* utiliza dois *scripts* auxiliares, `MontaApagamentos` (Script 10) e `GeraPadrErros` (Script 11), para realizar suas funções. Uma vez feito isso, a taxa total para cada padrão de erro é obtida fazendo-se a ponderação das taxas parciais obtidas pela taxa de incidência de cada linha, a qual havia sido obtida na primeira etapa. De posse então de todas as taxas de erro globais para cada padrão de apagamentos, é feita uma ordenação das mesmas e das correspondentes linhas da matriz inicial de apagamentos, atingido-se o objetivo final dessa segunda etapa.

A Figura 29 mostra as primeiras 14 linhas da matriz de apagamentos ordenada, obtida para o código $C(15,7,5)$ assim como suas probabilidades de erro, tanto para cada padrão quanto acumuladas. A matriz está incompleta. Como indicado anteriormente, 64 padrões possíveis de erro foram levantados pelo *script*. Entretanto, como pode ser notado, os 14 padrões mais prováveis cobrem 99,26 % das possibilidades.

Código C(15,7,5)
Padrões mais prováveis de erros

Padrão de erros							Taxa	Acumulado
0	0	0	0	0	0	0	0,7928	0,7928
0	0	0	0	0	0	1	0,0720	0,8648
0	0	0	0	0	1	0	0,0429	0,9077
0	0	0	0	1	0	0	0,0285	0,9362
0	0	0	1	0	0	0	0,0188	0,9550
0	0	1	0	0	0	0	0,0121	0,9671
0	1	0	0	0	0	0	0,0068	0,9739
0	0	0	0	0	1	1	0,0053	0,9792
0	0	0	0	1	0	1	0,0033	0,9826
1	0	0	0	0	0	0	0,0029	0,9855
0	0	0	0	1	1	0	0,0022	0,9877
0	0	0	1	0	0	1	0,0022	0,9898
0	0	0	1	0	1	0	0,0014	0,9912
0	0	1	0	0	0	1	0,0014	0,9926

Figura 29 – Padrões de erros mais prováveis para código $C(15,7,5)$

Outro ponto interessante de ser notado na Figura 29 é que alguns padrões de erro de dois símbolos simultaneamente são mais prováveis que outros de um único símbolo. Esse fato não é uma particularidade do código específico deste exemplo e foi igualmente observado nos resultados obtidos com códigos mais longos como o $C(24,12,8)$ e $C(48,24,12)$.

3.2 Dimensionamento do número de candidatas

De acordo com Blahut em (BLAHUT, 2003) cap. 12, pode-se definir o ganho de codificação de um decodificador como a redução em dB's, da relação E_b/N_0,

necessária para se obter, utilizando o decodificador, a mesma taxa de erros de bit que seria obtida sem o mesmo. A Figura 30 ilustra esse conceito. Na figura a linha tracejada indica a taxa de erros obtida pelo canal sem utilização de código e a linha simples mostra o desempenho quando o código é aplicado.

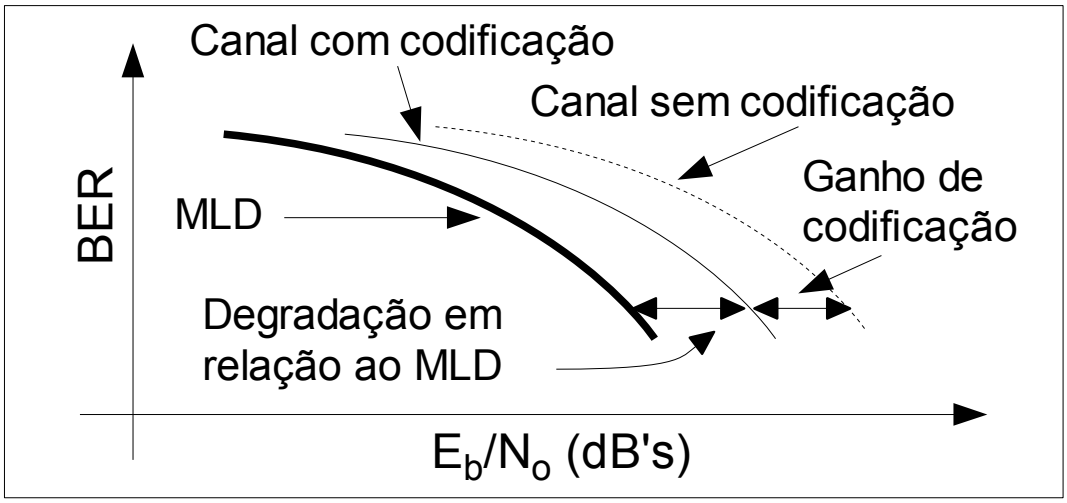

Figura 30 – Ganho de codificação e degradação em relação ao MLD

Na Figura 30 está indicado também o desempenho de um decodificador por decisão suave por máxima verossimilhança (MLD), na linha mais grossa. Como no caso de decodificação por decisão suave o decodificador por MLD é o de melhor desempenho possível, será proposto, neste trabalho, avaliar o desempenho desejado, não em relação ao canal sem codificação, mas sim em relação ao MLD, como mostrado na figura. Fala-se portanto não em ganho de codificação mas em redução, ou degradação, de ganho de codificação em relação ao ganho que pode ser atingido por meio da decodificação MLD. Assim, o objetivo desta seção é fornecer ao projetista de *hardware*, critérios para dimensionar a quantidade de candidatas a utilizar na decodificação por conjuntos de informação, de maneira a permitir apenas uma determinada degradação máxima desejada do ganho de codificação em relação ao MLD.

3.2.1 Obtenção das diferenças de taxas de erros de bit em relação ao MLD

O primeiro passo para se determinar a degradação do ganho de codificação em relação ao ganho do decodificador MLD é a determinação das diferenças de taxas de erros em relação ao MLD como função da quantidade de candidatas utilizadas. Para isso foram realizadas simulações por meio do Matlab, com um número

suficientemente grande de candidatas – para cada código contemplado – de maneira que a taxa de erros de bit obtida fosse da mesma ordem de grandeza da taxa MLD (do inglês "*Maximum Likelihood Decoding*"). O subconjunto de candidatas utilizado foi obtido conforme detalhado na subseção 3.1.4, e ordenado por ordem decrescente de confiabilidade. A seguir foram levantadas as diversas taxas de erro obtidas com vários conjuntos parciais desse conjunto, e calculadas as diferenças em cada caso em relação ao MLD. Por exemplo, para o código $C(24,12,8)$, em simulações realizadas com 10^6 palavras-código geradas aleatoriamente e submetidas à interferência de ruído aditivo gaussiano branco, determinou-se as taxas, mostradas na Tabela 3.4, para decodificação com decisão suave por MLD (4096 candidatas) e por conjuntos de informação utilizando as 80 candidatas mais confiáveis.

Tabela 3.4: Taxa de erros de bit em simulações com 10^6 candidatas – $C(24,12,8)$

Eb / No (dB)	Decodificação por MLD (%)	Decodificação por C. I. com 80 candidatas Diferença em relação ao MLD (%)
1	4,36	$2{,}58 \times 10^{-4}$
2	1,62	$< 10^{-6}$
3	0,41	5×10^{-5}
4	0,06	$< 10^{-6}$

A Tabela 3.4 mostra que para se obter taxas de erro de bit com o algoritmo BP praticamente iguais às obtidas pela decodificação por MLD para o código $C(24,12,8)$ é suficiente utilizar 80 candidatas. Para a implementação em *hardware* porém interessa saber quais os compromissos possíveis em cada caso. Se em vez de 80 candidatas for desejável, ou mesmo necessário, em função dos recursos disponíveis, utilizar apenas 40, ou 20, ou uma quantidade ainda menor de candidatas, qual será a degradação de ganho de codificação incorrida em relação ao MLD? Para responder a essa pergunta é necessário conhecer inicialmente as diferenças de taxas de erros de bit para cada quantidade diferente de candidatas. O Script 12 para Matlab, no ANEXO I, permite particionar o conjunto máximo de candidatas em um número arbitrário de sub-conjuntos e então determinar as taxas para cada caso.

Utilizando ainda o código $C(24,12,8)$ como exemplo, o particionamento

das 80 candidatas em 5 grupos de 16 e a aplicação do Script 12 resultou nos valores listados na Tabela 3.5.

Tabela 3.5: Redução da taxa de erros de bit em relação ao MLD – *C*(24,12,8)

Eb / No (dB)	16 candidatas	32 candidatas	48 candidatas	64 candidatas	80 candidatas
1	$8{,}58 \times 10^{-2}$	$1{,}34 \times 10^{-2}$	$2{,}54 \times 10^{-3}$	$7{,}42 \times 10^{-4}$	$2{,}58 \times 10^{-4}$
2	$4{,}93 \times 10^{-2}$	$8{,}07 \times 10^{-3}$	$1{,}16 \times 10^{-3}$	$8{,}34 \times 10^{-6}$	$< 10^{-6}$
3	$1{,}77 \times 10^{-2}$	$2{,}04 \times 10^{-3}$	$6{,}58 \times 10^{-4}$	$4{,}25 \times 10^{-4}$	5×10^{-5}
4	$4{,}50 \times 10^{-3}$	$5{,}67 \times 10^{-4}$	1.67×10^{-5}	$< 10^{-6}$	$< 10^{-6}$

A Tabela 3.5 é apenas um exemplo ilustrativo e o Script 12 pode ser aplicado a qualquer código, com qualquer particionamento desejável. Por exemplo, no caso do código $C(24,12,8)$ já abordado, poderia ser feito um particionamento em 80 pontos, obtendo-se a redução da taxa de erros de bit para uma, duas, etc.. candidatas. A única ressalva é que, para códigos mais longos, a obtenção da taxa MLD através do Script 16, utilizado pelo Script 12, deixa de ser viável, em função da quantidade excessivamente grande de candidatas. Nesse caso – por exemplo para os códigos $C(48,23,12)$ e $C(66,33,12)$ – a taxa foi estimada utilizando-se o valor obtido para uma quantidade de candidatas tal que a sua duplicação não alterava mais o valor da taxa nos três algarismos mais significativos da mesma. Por esse motivo o Script 12 permite ajustar o booleano "*do_mld*" para determinar efetivamente a taxa MLD através da comparação exaustiva com todas as palavras-código ou então utilizar apenas a estimativa especificada acima.

Os resultados do Script 12 porém, ainda que interessantes do ponto de vista teórico, não fornecem critérios para uma decisão prática sobre a quantidade de candidatas a utilizar. Por esse motivo na próxima seção será visto como transformá-los em informações úteis à implementação em *hardware*, na forma de degradação de ganho de codificação em dB's.

3.2.2 Degradação do ganho de codificação.

Para determinar a degradação do ganho de codificação a partir das matrizes de diferenças de taxas de erros de bits foi utilizada uma estratégia de avaliação por faixa de nível de ruído. Assim foram estimados os pontos médios das curvas de

taxas de erros de bit entre dois níveis de relação sinal / ruído.

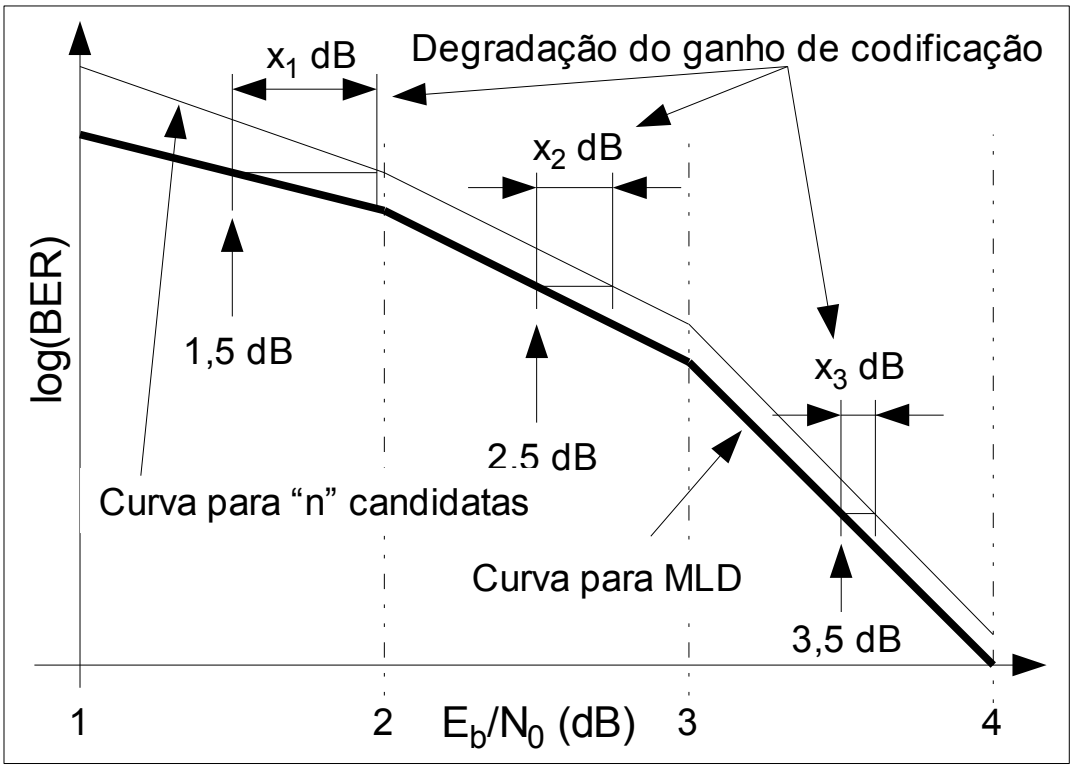

Figura 31 – Degradação do ganho de codificação em relação ao MLD

A idéia pode ser melhor compreendida examinando-se a Figura 31. Na figura os valores de x_1, x_2 e x_3 representam a quantidade de dB's que uma determinada curva para *n* candidatas precisa ser deslocada para a esquerda em média e para cada faixa de ruído, para se sobrepor à curva da decodificação por MLD. Os valores de x_i dB's, para cada faixa de E_b/N_o de *i* a *i* +1 dB's podem ser facilmente obtidos utilizando-se relações de proporcionalidade como indicado na Figura 32.

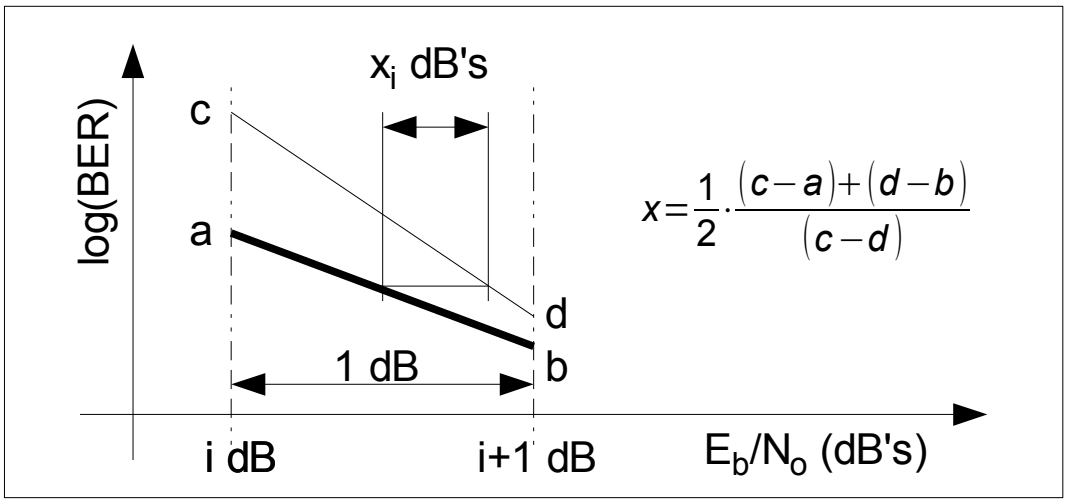

Figura 32 – Determinação dos valores de degradação do ganho de codificação

Os valores de *a*, *b*, *c* e *d* indicados na Figura 32 representam o logaritmo

da taxa erros de bit nos extremos da faixa.

Os conceitos apresentados nas figuras 31 e 32 foram utilizados para criar os *scripts* Matlab 17 e 18, os quais transformam as reduções de taxas de erros de bit nas respectivas degradações, em dB's, dos ganhos de codificação em relação ao MLD. Ainda atendo-se ao exemplo do código $C(24,12,8)$ abordado anteriormente, alguns valores da degradação são apresentados na Tabela 3.6.

Tabela 3.6: Degradação do ganho em relação ao MLD – $C(24,12,8)$

(valores em dB)

Eb / No (dB)	4 candidatas	16 candidatas	28 candidatas	40 candidatas	48 candidatas
1 a 2	0,64	$2{,}52 \times 10^{-2}$	$6{,}25 \times 10^{-3}$	$1{,}84 \times 10^{-3}$	$< 10^{-3}$
2 a 3	0,82	$2{,}65 \times 10^{-2}$	$6{,}10 \times 10^{-3}$	$1{,}76 \times 10^{-3}$	$< 10^{-3}$
3 a 4	1,08	$3{,}05 \times 10^{-2}$	$6{,}34 \times 10^{-3}$	$1{,}51 \times 10^{-3}$	$< 10^{-3}$

Para esse código em particular percebe-se então que a regra sugerida em alguns trabalhos como (GODOY, 2010a), (GORTAN ,2010a) e (BRANTE, 2011) de se utilizar $k + 1$ candidatas ($k + 1$ no caso = 13) é razoavelmente válida, uma vez que, para essa quantidade de candidatas os valores de degradação de ganho em relação ao MLD (do inglês “Maximum Likelihood Decoding”) ficam em torno de centésimos de decibéis. Todavia essa não é uma regra válida para qualquer código. Particularmente, para códigos mais longos a tendência é um aumento exponencial do número de candidatas necessárias para se atingir o mesmo nível de degradação em relação ao MLD. A Tabela 3.7 ilustra esse fato apresentando alguns valores para o código $C(48,24,12)$.

Tabela 3.7: Degradação do ganho em relação ao MLD – *C*(48,24,12)

(valores em dB)

Eb / No (dB)	20 candidatas	80 candidatas	140 candidatas	200 candidatas	260 candidatas
1 a 2	0,36	0,07	$2{,}51 \times 10^{-2}$	$1{,}11 \times 10^{-2}$	$6{,}26 \times 10^{-3}$
2 a 3	0,49	0,10	$3{,}11 \times 10^{-2}$	$1{,}30 \times 10^{-2}$	$6{,}69 \times 10^{-3}$
3 a 4	0,72	0,15	$4{,}24 \times 10^{-2}$	$1{,}98 \times 10^{-2}$	$9{,}76 \times 10^{-3}$

Uma visão geral das relações envolvidas pode ser obtida inspecionando-se a Figura 33, que procura apresentar de maneira comparativa os valores obtidos através dos *scripts* Matlab 17 e 18. Essa figura foi gerada com o auxílio do Script 19 para Matlab.

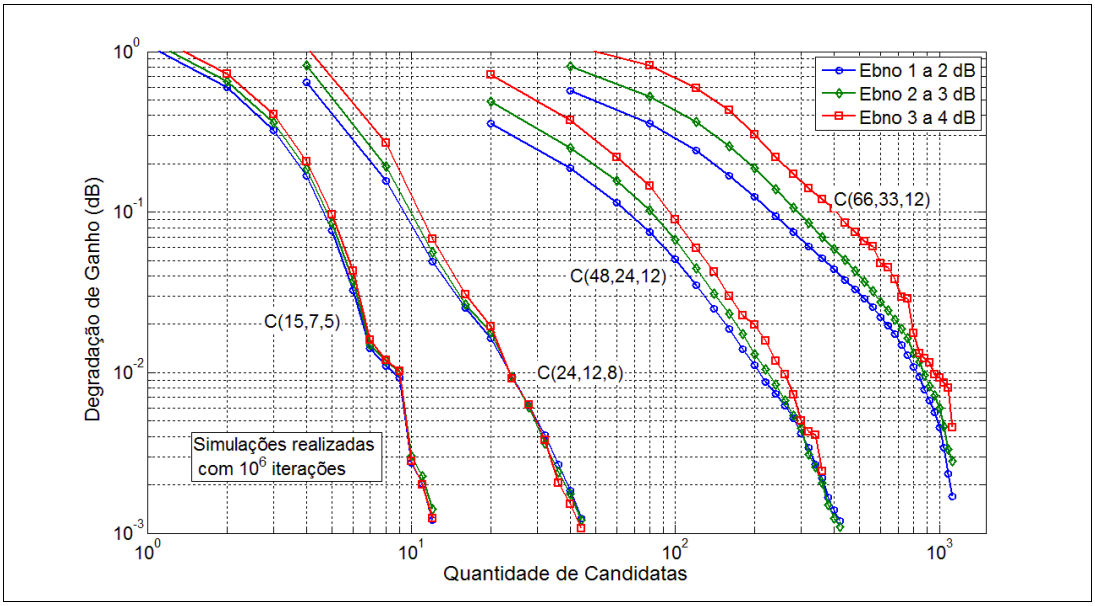

Figura 33 – Degradação de ganho de codificação – códigos de vários comprimentos

Na figura pode-se ler no eixo horizontal a quantidade de candidatas a utilizar de forma a se obter uma determinada degradação do ganho de codificação em relação ao MLD – lida no eixo vertical – para diversos códigos de forma comparativa.

3.3 Critérios de parada

A decodificação suave por máxima verossimilhança consiste em comparar a distância euclidiana entre a palavra-código recebida e todas as possíveis palavras-código pertencentes ao código em questão. Foi visto que a decodificação por decisão suave por conjuntos de informação permite reduzir a quantidade de comparações para um sub-conjunto seleto de palavras-código, o qual tem uma probabilidade extremamente alta de conter a palavra-código mais próxima – em termos de distância euclidiana – da palavra-código recebida. As sub-seções anteriores descreveram os meios para se obter o sub-conjunto em questão, e também as justificativas para tanto.

Conquanto a redução do universo de comparações represente um grande passo no sentido de aumentar a eficiência da decodificação, um aumento ainda maior dessa eficiência pode ser alcançado com a utilização de critérios de parada.

Entende-se por critério de parada, um teste, que, quando realizado em relação à palavra-código recebida e a uma possível candidata à decodificação, permita interromper a sequência de comparações, declarando aquela candidata como a correta decodificação para a palavra-código recebida.

Um exemplo trivial para um critério de parada seria o de verificar se a distância euclidiana entre a palavra-código recebida e a candidata em análise é inferior à distância euclidiana mínima do código, como definida na seção 2.1.13, equação (2.8). Se isso for verdade a sequência de comparações pode ser interrompida, prescindindo-se das demais comparações. Embora esse critério seja útil para fins ilustrativos do conceito de critério de parada, e ainda que seja também de fácil aplicação, sua eficiência é muito baixa, não fornecendo, em média, uma redução do número de operações que justifique sua implementação. A justificativa para essa baixa eficiência é que, de uma forma geral, as distâncias euclidianas entre as palavras de um código não são todas iguais à mínima distância euclidiana do código. Há portanto sempre a possibilidade da distância entre a palavra código recebida e a candidata em análise ser a menor possível e ainda assim superar o valor da distância euclidiana mínima.

Existem, porém diversos outros critérios, com graus variados de eficiência e de dificuldade de implementação, que serão analisados nas próximas sub-seções. Os principais são: o critério GMD, ou “Generalized Minimum Distance”, de Forney (FORNEY, 1966), o critério do cone, também de Forney (FORNEY, 1966), o critério

BGW (iniciais de Barros, Godoy e Wille) (GODOY, 1998), o critério de Taipale e Pursley (1991) e, finalmente, o critério BGWG (iniciais de Barros, Godoy, Wille e Gortan), proposto neste trabalho, como uma modificação do critério BGW com fins a viabilizar sua aplicação em *hardware*.

3.3.1 Critério de parada GMD

O critério de parada GMD, apresentado em (FORNEY, 1966) e discutido em (BLAHUT, 1983), afirma que, para um determinado código de comprimento *n* e distância mínima de Hamming d_{Hmin}, dada uma palavra-código recebida **v** e uma palavra-código candidata à sua decodificação **c**, ambas moduladas em BPSK (do inglês *Binary Phase Shift Keying*), se for satisfeita a condição:

$$\langle \mathbf{v}, \mathbf{c} \rangle > n - d_{Hmin} \tag{3.46}$$

então **c** será única, o que equivale a dizer que: $\langle \mathbf{v},\mathbf{c} \rangle = \langle \mathbf{v},\mathbf{c} \rangle_{max}$.

Ora, maximizar o produto interno entre **v** e **c** como acima, equivale a minimizar a distância euclidiana entre essas duas palavras-código e, consequentemente, pode-se afirmar que **v** pertence à região de Voronoi de **c** e portanto **c** é a melhor decodificação possível para **v**. O ANEXO VI contém a demonstração da validade do critério. A equação (3.46) só é valida se as componentes do vetor **v** estiverem contidas no intervalo {–1,+1}, ou seja, se **v** tiver sido normalizado. Como será mostrado na sub-seção abordando os resultados de simulações para a comparação dos critérios, a eficiência do critério GMD não é muito boa.

3.3.2 Critério de parada do Cone

O critério de parada do limiar do cone (FORNEY, 1966) baseia-se no princípio de que se o ângulo compreendido pelo vetor recebido e a palavra candidata for menor que a metade do menor ângulo entre duas palavras código então o vetor recebido estará contido na região de Voronoi da palavra candidata.

O critério será válido para todos os caso em que os módulos de todas as palavras-código forem iguais, como é o caso para modulação BPSK (do inglês *Binary Phase Shift Keying*).

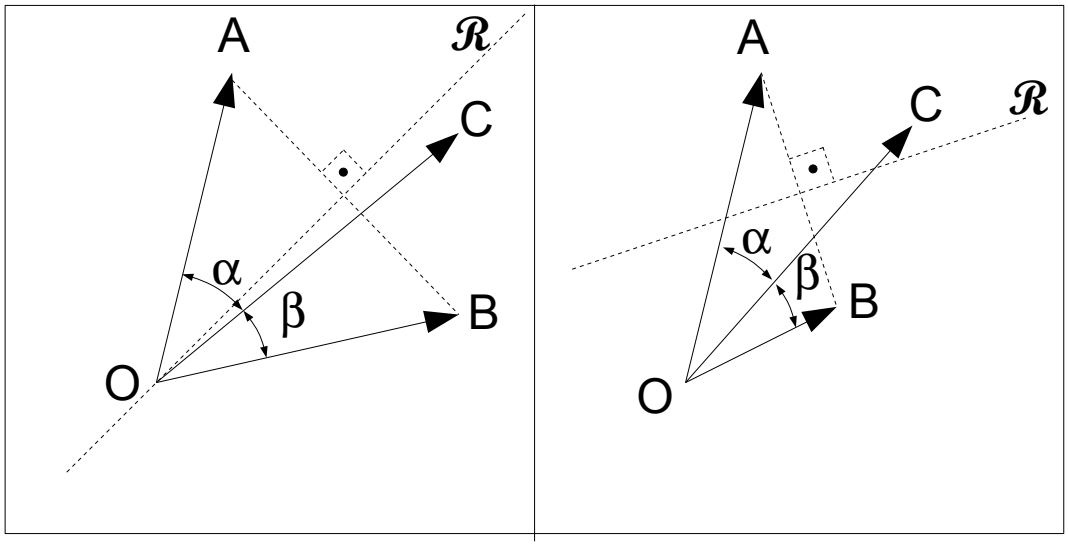

Figura 34 – Interpretação geométrica para validade do critério de parada do cone

A Figura 34 procura dar uma interpretação geométrica bidimensional ao critério, fazendo um exame no plano. Na figura vê-se duas situações comparadas lado a lado. Os vetores $\overline{OA}$ e $\overline{OB}$ representam duas palavras-código candidatas e o vetor $\overline{OC}$ uma palavra-código recebida, à qual foi acrescido um possível erro durante sua transmissão.

Deseja-se então determinar a qual dos dois vetores, $\overline{OA}$ ou $\overline{OB}$, o vetor $\overline{OC}$ mais se aproxima. Em ambas as figuras, a reta pontilhada $\mathcal{R}$ representa o lugar geométrico dos pontos equidistantes a A e B, obtida como a mediatriz do segmento $\overline{AB}$.

Portanto se o erro acrescido ao vetor $\overline{OC}$ for tal que o ponto C caia no semi-plano superior à reta $\mathcal{R}$ isso significará que a distância $\overline{CA}$ é inferior à distância $\overline{CB}$ e $\overline{OA}$ será a melhor decodificação para $\overline{OC}$, se não resultará o contrário e $\overline{OB}$ será a melhor decodificação para $\overline{OC}$.

Como se vê no lado esquerdo da Figura 34, o critério de pertinência ao semi-plano inferior ou superior pode ser substituído pelo critério do menor ângulo. Nesse caso, como o ângulo β é inferior ao ângulo α, pode-se afirmar que o ponto C pertence ao semi-plano inferior à reta $\mathcal{R}$. Isso só é possível devido à igualdade dos módulos de $\overline{OA}$ e $\overline{OB}$. Já no lado direito dessa figura, o critério do menor ângulo não é sempre válido. No exemplo, embora β seja inferior a α, o ponto C encontra-se no semi-plano superior a $\mathcal{R}$ e a melhor decodificação para $\overline{OC}$ seria $\overline{OA}$ e não $\overline{OB}$.

Se o menor ângulo entre dois vetores quaisquer pertencentes ao código for δ, então se o ângulo entre um vetor recebido e a palavra-código candidata for in-

ferior à metade de δ será possível afirmar que o vetor recebido pertence à região de Voronoi da palavra-código candidata em questão.

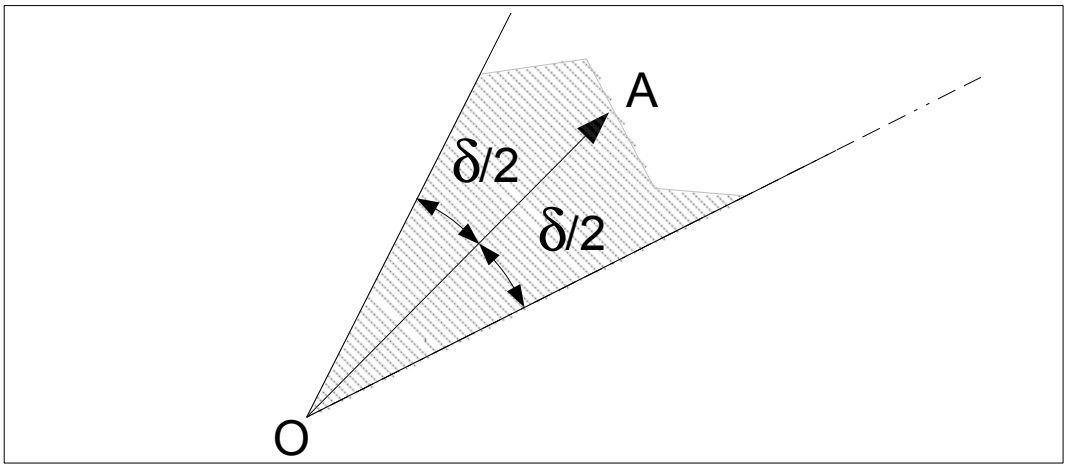

Figura 35 – Região de Voronoi para vetor OA

A Figura 35 ilustra essa situação para o caso bidimensional. A região hachurada, que teoricamente se estende até o infinito na sua parte superior, representa a região de Voronoi para o vetor $\overline{OA}$, supondo que δ seja o menor ângulo possível entre dois vetores pertencentes ao código. Se o exemplo for estendido para três dimensões, a região hachurada se transformará em um cone, de onde deriva o nome do critério. A rigor, para códigos com *n* dimensões tem-se um hiper-cone *n*-dimensional.

Analiticamente, dados um vetor **v** e uma palavra-código candidata **c**, ambos modulados em BPSK (do inglês *Binary Phase Shift Keying*), o critério do cone permite afirmar que **v** pertencerá à região de Voronoi de **c** se a seguinte desigualdade for satisfeita:

$$\langle \mathbf{c}, \mathbf{v} \rangle \quad > \quad \|\mathbf{v}\| \times \sqrt{n - d_{Hmin}} \tag{3.47}$$

O ANEXO VII contém a demonstração para a expressão (3.47). Nesse anexo são também apresentadas formas alternativas da expressão (3.47), que podem simplificar sua aplicação.

3.3.3 Comparação analítica entre os critérios GMD e do Cone

A semelhança entre as equações (3.46), para o critério de parada GMD, e (3.47), para o critério do Cone, sugere uma comparação para o desempenho de cada limitante. Como foi visto na seção anterior, o critério GMD só é válido para um

vetor **v** que tenha sido normalizado de forma a ter suas componentes contidas no intervalo {–1,+1}. Embora o critério do Cone não tenha essa limitação, para que a comparação entre os dois casos seja válida deve-se supor que também na equação (3.47) o vetor **v** foi normalizado no intervalo {–1,+1}. Na equação (3.46) o produto interno <**c**,**v**> é comparado com o fator ($n - d_{hmin}$), enquanto que na equação (3.47) esse mesmo produto interno é comparado com com o fator $||\mathbf{v}|| \times (n - d_{hmin})^{1/2}$. A relação entre os dois fatores de comparação vale portanto:

$$\frac{||\mathbf{v}|| \times \sqrt{n - d_{hmin}}}{(n - d_{hmin})} = \frac{||\mathbf{v}||}{\sqrt{n - d_{hmin}}} \tag{3.48}$$

Como o vetor **v** foi normalizado, seu módulo ||**v**|| poderá variar entre um valor máximo = $n^{1/2}$ – que deverá ocorrer para valores extremamente baixos de ruído – e um valor mínimo próximo de zero para valores altos de ruído superposto. Portanto sempre que o valor do ruído agregado ao vetor **v** for de tal monta que a normalização de suas componentes entre {–1,+1} leve o valor de seu módulo a satisfazer a relação:

$$||\mathbf{v}|| < \sqrt{n - d_{hmin}} \tag{3.49}$$

resultará uma situação em que o limitante do cone será mais facilmente satisfeito que o GMD. Conclui-se portanto que o limitante do Cone será mais eficiente em condições de baixa relação sinal/ruído, que são as situações em que um aumento de eficiência do código é mais importante, visto que haverá mais erros a corrigir.

3.3.4 O critério de parada BGW

O critério de parada BGW foi apresentado pela primeira vez em (BARROS, 1997). A sigla que lhe empresta o nome deriva das iniciais dos autores do artigo citado.

Esse critério pode ser enunciado da seguinte forma: dados um código *C* de comprimento *n* e distância mínima de Hamming d_{Hmin}, e dados um vetor **v** recebido e uma palavra código **c**, candidata à sua decodificação, ambos modulados em BPSK (do inglês *Binary Phase Shift Keying*), pode-se afirmar que **c** será a melhor decodificação possível para **v** se a soma das d_{Hmin} posições menos negativas da soma híbrida entre **v** e **c** resultar em um valor negativo. O critério aqui exposto pode ser enunciado na forma do seguinte teorema:

Se a soma das d_{Hmin} posições menos negativas da soma híbrida entre **v** e **c** for negativa então **y** $\in$ V(**c**). Esse teorema e sua demonstração formal estão apresentados no ANEXO IX. Uma demonstração mais descritiva da validade do critério, fazendo uso de um exemplo será feita a seguir.

O critério faz uso do fato, discutido na subseção 2.1.18, de que se a soma híbrida **y' = y** [+] **c** $\in$ $V(\mathbf{c}_0)$ então **y** $\in$ V(**c**). Para aplicar o critério é necessário determinar **y' = y** [+] **c**, ordenar suas componentes em ordem decrescente de valor e então somar os primeiros d_{Hmin} valores da ordenação. Se o valor da soma resultante for negativo então o critério terá sido satisfeito e **c** deverá ser declarada como a correta decodificação para **v**.

Em 2.1.18 foi visto que minimizar a distância euclidiana entre **y** e uma palavra-código candidata **c** equivale a minimizar a soma das componentes de **y' = y** [+] **c**

Chamando de $\mathbb{S}$ ao conjunto dos índices das d_{Hmin} posições menos negativas de **y'**, e de $\overline{\mathbb{S}}$ seu complemento, ou seja ao conjunto dos índices das restantes $n - d_{Hmin}$ posições de **y'**, a soma das componentes de **y'** pode ser escrita como:

$$\sum_i y'_i = \sum_{i \in \boldsymbol{S}} y'_i + \sum_{i \in \overline{\boldsymbol{S}}} y'_i = S_+ + S_- \tag{3.50}$$

onde é utilizada a notação S_+ para denotar a soma das componentes de **y'** com índices em $\mathbb{S}$ e S_- para denotar a soma das componentes de **y'** com índices em $\overline{\mathbb{S}}$.

Da definição de $\mathbb{S}$ fica claro que valerá sempre:

$$S_+ = \sum_{i \in \boldsymbol{S}} y'_i \; > \; \sum_{i \in \overline{\boldsymbol{S}}} y'_i = S_- \tag{3.51}$$

Também da definição de $\mathbb{S}$ pode-se concluir que:

$$\begin{gathered} S_+ = \sum_{i \in \boldsymbol{S}} y'_i < 0 \;\Rightarrow\; S_- = \sum_{i \in \overline{\boldsymbol{S}}} y'_i < 0 \\ e \\ S_+ = \sum_{i \in \boldsymbol{S}} y'_i < 0 \;\Rightarrow\; y'_i < 0 \quad \forall i \in \overline{\boldsymbol{S}} \end{gathered} \tag{3.52}$$

A equação (3.52) mostra que quando $S_+ < 0$ todas as componentes de y'_i com i pertencente a $\overline{\mathbb{S}}$ são negativas, ou seja, todas as componentes de y_i com i pertencente a $\overline{\mathbb{S}}$ têm o mesmo sinal que as correspondentes componentes c_i. Já as

componentes y'_i com i pertencente a $\mathbb{S}$ podem ser ou todas negativas ou algumas positivas e outras negativas, desde que sua soma S_+ resulte negativa como pressuposto, não podendo porém ser todas positivas, pois isso negaria o pressuposto. Chamando de *m* à quantidade de componentes positivas de **y'**, tem-se que:

$$0 \leq m < d_{Hmin} \tag{3.53}$$

A validade do critério BGW pode então ser analisada da seguinte maneira: dado um vetor **y** recebido, suponha-se que foi encontrada uma palavra-código candidata **c**, tal que **y'** = **y** [+] **c** seja tal que S_+, como definido na (3.50) resulte negativo. Deve-se então procurar encontrar uma outra palavra-código **c'**, mais próxima de **y** do que **c**, ou seja uma palavra-código **c'** tal que produza **y''** = **y** [+] **c'** tal que se obtenha $\sum_i y''_i < \sum_i y'_i$ Se o fato de S+ < 0 fizer concluir que é impossível encontrar uma **c'** mais próxima de **y** então a validade do critério terá sido demonstrada.

Para melhor acompanhar o raciocínio sugere-se fazer referencia à Figura 36, onde foi considerado um código $C(15,7,5)$ e onde as 15 componentes y'_i do vetor **y'** foram ordenadas em ordem decrescente de valor. Na figura apenas duas das componentes de **y'** resultaram positivas e portanto *m* = 2.

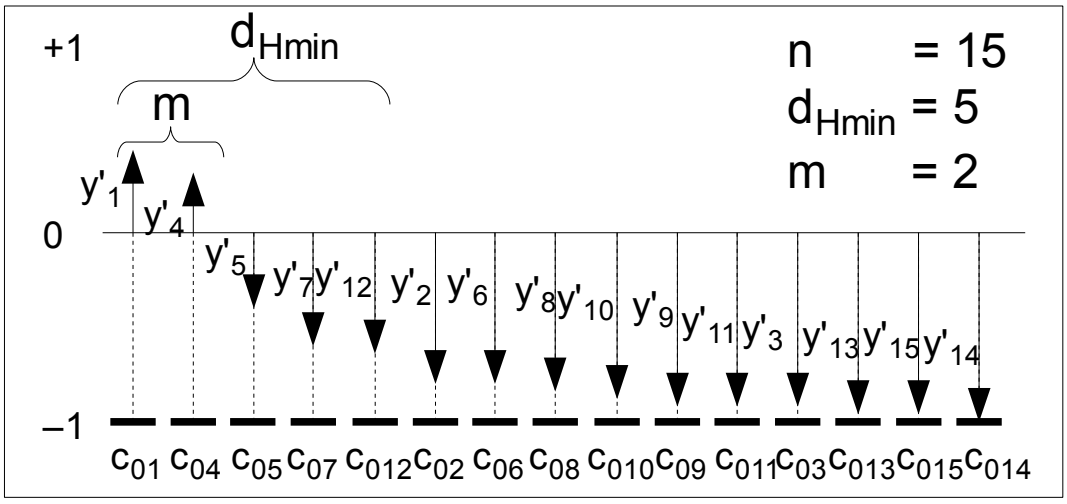

Figura 36 – Exemplo de vetor **y'** com componentes ordenadas – Código $C(15,7,5)$

Cada inversão de um componente de **c** para transformá-la na **c'** procurada irá causar a inversão na correspondente componente de **y'**. Como deseja-se diminuir a soma das componentes **y'**, percebe-se, examinando a figura, que as com-

ponentes de **c** a serem alteradas devem ser aquelas em que as correspondentes componentes de **y'** são positivas. Essas componentes serão em número de *m*, com $m < d_{Hmin}$. No exemplo da Figura 36, com $m = 2$ e essas componentes são as de índice 1 e 4.

Infelizmente, alterando apenas m componentes de **c** não é possível obter uma nova palavra-código candidata **c'** válida pois a nova **c'** deverá diferir de **c** em no mínimo d_{Hmin} posições e $m < d_{Hmin}$.

Torna-se necessário portanto realizar pelo menos outras $m - d_{Hmin}$ alterações em componentes de **c** para obter uma nova **c'** válida. Porém as *m* alterações já realizadas eram as únicas que podiam contribuir para reduzir a soma das componentes de **y'** , logo as $m - d_{Hmin}$ alterações adicionais irão necessariamente contribuir com aumentos de **y'**.

Mais uma vez examinando a Figura 36, percebe-se que, para minimizar o aumento devido às $m - d_{Hmin}$ alterações adicionais, a escolha dessas $m - d_{Hmin}$ posições adicionais deverá recair nas componentes mais positivas de **y'** dentre as restantes, pois essas posições, ao serem invertidas, fornecerão a menor contribuição possível ao aumento de **y'**. Na figura do exemplo, essas componentes são as de índices $i = 5, 7$ e 12.

Conclui-se portanto que para poder reduzir o valor de **y'** através de alterações de **c** que conduzam a uma **c'** mais próxima de **y** e pertencente ao código, a condição necessária, porém não suficiente, é que a soma das *m* componentes positivas de **y'** supere a soma das $m - d_{Hmin}$ componentes adicionais, selecionadas dentre as menos negativas restantes. A condição não é suficiente pois não é garantido que alterando-se exatamente essas d_{Hmin} posições de **c** a **c'** obtida pertencerá ao código.

Ora a condição enunciada acima equivale a impor a condição de que a soma das d_{Hmin} componentes mais positivas de **y'**, ou S_+, seja positiva. Assim, se essa soma resultar negativa não será possível encontrar uma palavra-código **c'** mais próxima de **y** do que **c** e a busca pode ser interrompida, declarando-se **c** como a melhor decodificação para **y**.

As considerações acima justificam a validade do critério. No exemplo dado $m > 0$, entretanto se m = 0 resultará igualmente $S_+ < 0$ e o critério também será satisfeito. Já $S_+ = 0$ representa a situação em que **y** será equidistante a **c** e à **c'** pro-

posta. Como não há garantias que a **c'** pertença ao código, deve-se optar por **c**. Com isso o critério pode ser estendido para $S_+ \leq 0$.

Taipale e Pursley (1991) desenvolveram um critério de parada que, embora descrito diferentemente, é essencialmente equivalente ao BGW.

Gortan (2002), mostrou a equivalência desses dois critérios, desenvolvidos de forma independente por pesquisadores diferentes.

Os critérios BGW e o de Taipale e Pursley apresentam uma eficiência superior ao do critério do Cone, descrito anteriormente, como mostram os resultados das simulações, apresentados mais adiante nas tabelas 3.8 a 3.10 na página 101. Em termos porém de implementação em *hardware* ambos apresentam o inconveniente de necessitar uma ordenação de seus símbolos para a aplicação do critério. Devido a esse motivo, uma variante sub-ótima do critério BGW, porém ainda com uma eficiência superior à do critério do Cone, que procura levantar essa restrição, será apresentada na próxima subseção.

3.3.5 O critério de parada BGWG

O critério BGWG é uma variante sub-ótima do critério BGW. A motivação para seu desenvolvimento foi poder implementar de forma eficiente o critério BGW em *hardware* por meio de FPGAs (do inglês *Field Programmable Gate Arrays*).

Ao se tentar implementar o critério BGW em *hardware* programável notou-se que a ordenação das n componentes do vetor recebido por ordem decrescente de valor consumia n clocks para cada palavra-código candidata analisada. Ora como cada análise de palavra-código candidata é realizada em um único clock em *hardware*, o consumo de n clocks adicionais para se decidir se a sequência de comparações pode ser interrompida faz com que, em média, a quantidade total de clocks utilizada seja maior aplicando-se o critério do que sem ele.

Durante a implementação do algorítimo BP em *hardware*, notou-se que uma ordenação das componentes do vetor recebido **y** já era feita inicialmente, com a finalidade de estimar a confiabilidade relativa de cada uma das componentes. Desenvolveu-se então uma maneira de utilizar o resultado dessa ordenação para implementar, ainda que de uma forma sub-ótima, o algorítimo BGW.

A dificuldade, nesse caso, é que a ordenação das confiabilidades é feita com base no valor absoluto, ou módulo, das componentes do vetor recebido **y**, en-

quanto que a aplicação do algorítimo BGW necessita de uma ordenação de valores – com sinal – das componentes do vetor soma híbrida **y'** = **y** [+] **c**, o qual será diferente para cada palavra-código candidata **c** a analisar.

A dificuldade citada pode ser superada, ainda que em parte, levando-se em consideração as relações existentes entre as componentes de **y**, **y'** e **c**. Como foi visto no item 3.1 e sub-seções, o algorítimo BP consiste em se reduzir o universo de candidatas à decodificação de **y** de 2^k candidatas – para a decodificação por MLD – para uma quantidade substancialmente menor, que será chamada aqui de *q*, obtendo uma série de candidatas $\mathbf{c_0}$, $\mathbf{c_1}$, … $\mathbf{c_{q-1}}$, ordenadas por ordem decrescente de probabilidade de serem a correta decodificação para **y**.

A soma híbrida de **y** com cada uma das *q* candidatas $\mathbf{c_0}$, $\mathbf{c_1}$, … $\mathbf{c_{q-1}}$ irá produzir a sequência $\mathbf{y'_0}$, $\mathbf{y'_1}$, … $\mathbf{y'_{q-1}}$. Suponha-se a seguir, apenas para efeito de análise e argumentação, que as componentes de $\mathbf{y'_0}$, $\mathbf{y'_1}$, … $\mathbf{y'_{q-1}}$ tenham sido reordenadas de acordo com o vetor de confiabilidades **SI**, descrito em 3.1.1, e que os vetores assim reordenados foram chamados de $\mathbf{y''_0}$, $\mathbf{y''_1}$, … $\mathbf{y''_{q-1}}$.

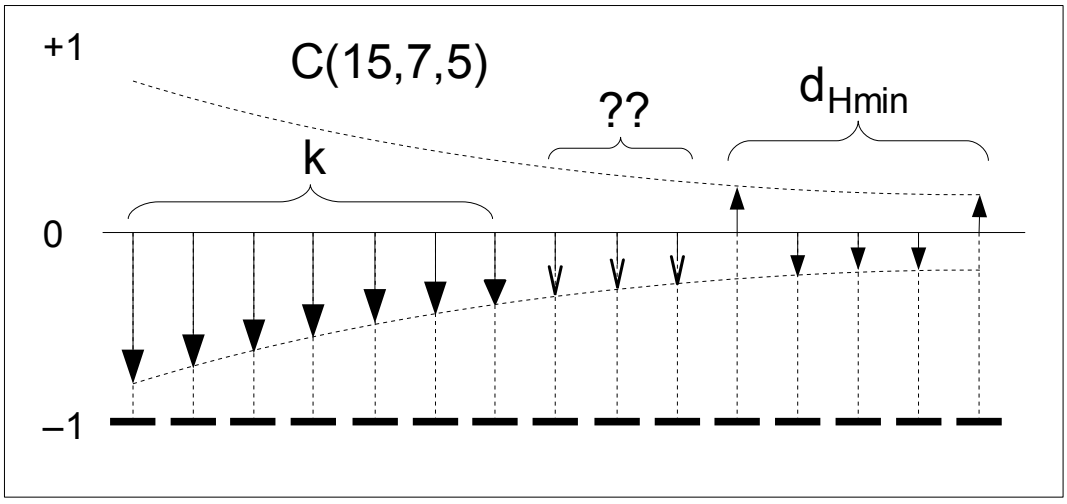

Figura 37 – Distribuição das componentes para $\mathbf{y''_0}$

A Figura 37 mostra um gráfico da sequência $\mathbf{y''_0}$, tomando como exemplo um código *C*(15,7,5).

Na figura, as duas curvas tracejadas indicam os possíveis valores assumidos pelas várias componentes de $\mathbf{y''_0}$ dada a sua ordenação por valor absoluto de acordo com o vetor **SI**. Porém, no caso pode-se afirmar que as primeiras *k* componentes de $\mathbf{y''_0}$ serão necessariamente negativas por construção. Lembrando que $\mathbf{y''_0}$ foi obtida a partir de $\mathbf{y'_0}$, a qual por sua vez resultou da soma híbrida de **y** com $\mathbf{c_0}$,

sendo que as *k* componentes mais confiáveis de $\mathbf{c_0}$ foram obtidas por decodificação abrupta das *k* componentes mais confiáveis de **y** e, portanto, têm o mesmo sinal, obrigando as correspondentes componentes da soma híbrida a serem negativas.

O critério BGW pode então ser aplicado realizando-se a soma das últimas d_{Hmin} componentes como indicado na figura e verificando se essa soma é inferior ou igual a zero. Porém o resultado dessa verificação só terá validade se as demais $n - k - d_{Hmin}$ componentes, na figura indicadas com dois pontos de interrogação, forem realmente negativas. Se apenas uma dessas componentes for positiva, isso invalidará o resultado da aplicação do critério BGW. O critério BGWG portanto para esse caso pode ser enunciado como: se a soma das últimas d_{Hmin} componentes de $\mathbf{y''_0}$ resultar negativa ou nula e se nenhuma das componentes de $\mathbf{y''_0}$ indicadas com sinal de interrogação for positiva, então a análise da sequência pode ser interrompida.

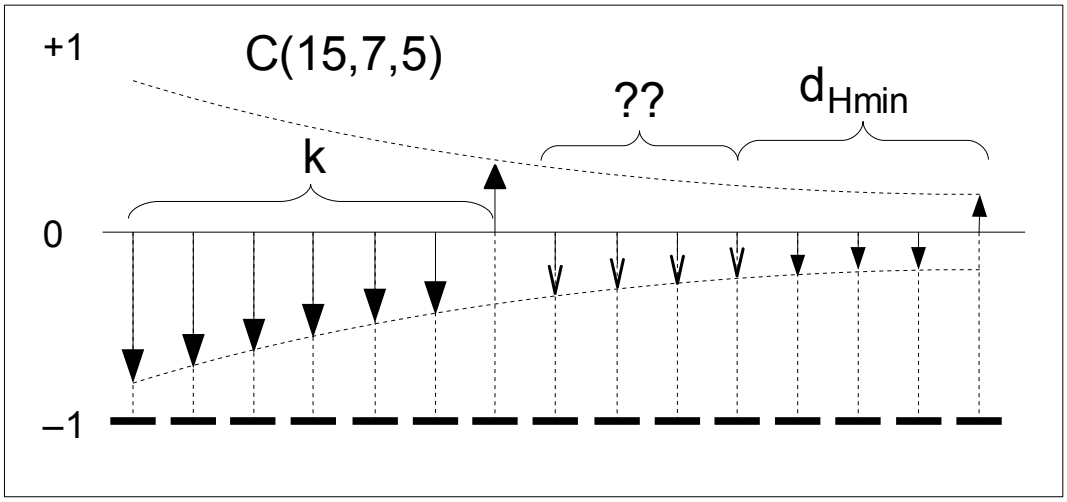

Figura 38 – Distribuição das componentes para $\mathbf{y''_1}$

Para as demais sequências, $\mathbf{y''_1}$,$\mathbf{y''_2}$, etc.., é necessário levar em conta as posições, dentre as primeiras *k*, que foram invertidas.

A Figura 38 exemplifica esses outros casos utilizando como exemplo a sequência $\mathbf{y''_1}$, na qual a componente menos confiável, dentre as *k* mais confiáveis, foi invertida. Nesse caso, a componente invertida será necessariamente positiva e deverá portanto fazer parte da soma das d_{Hmin} componentes mais positivas. No caso ela irá substituir a componente de maior módulo dentre as últimas d_{Hmin} componentes. Isso só será correto se a componente substituída não for positiva. Essa componente portanto deverá passar a integrar o conjunto de componentes que não poderão ser positivas sob pena de invalidar o teste. No exemplo da figura, a componente de número 7 passa a substituir a de número 11 na soma das d_{Hmin} mais positivas e

essa componente passa a integrar o conjunto marcado com dois pontos de interrogação, composto pelas componentes 8, 9, 10 e 11.

Para aplicar o critério BGWG portanto deve-se dispor de duas máscaras, $\mathbf{M1_i}$ e $\mathbf{M2_i}$, de comprimento de *n* bits. $\mathbf{M1_i}$ servirá para selecionar as posições de $\mathbf{y''_i}$ a serem somadas e $\mathbf{M2_i}$ para selecionar as posições que, sendo positivas, irão invalidar o teste. No exemplo da Figura 37, para *i* = 0, resultaria:

$$\begin{aligned} \mathbf{M1_0} &= [\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,1\,1\,1\,1\,] \quad e \\ \mathbf{M2_0} &= [\,0\,0\,0\,0\,0\,0\,0\,1\,1\,1\,0\,0\,0\,0\,0\,] \end{aligned} \tag{3.54}$$

já no exemplo da Figura 38, para *i* = 1, resultaria:

$$\begin{aligned} \mathbf{M1_1} &= [\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,1\,1\,1\,] \quad e \\ \mathbf{M2_1} &= [\,0\,0\,0\,0\,0\,0\,0\,1\,1\,1\,1\,0\,0\,0\,0\,] \end{aligned} \tag{3.55}$$

Independentemente do valor de *i*, conhecida a máscara $\mathbf{M1_i}$, a máscara $\mathbf{M2_i}$ pode ser obtida a partir de $\mathbf{M1_i}$ invertendo-se todos os bits de $\mathbf{M1_i}$ e zerando suas primeiras *k* posições.

As máscaras $\mathbf{M1_i}$ por sua vez podem ser obtidas diretamente da matriz de apagamentos, como definida no item 3.1.4.4. A matriz de apagamentos é formada por *k* colunas e por uma quantidade de linhas *q* igual à quantidade de candidatas.

Para obter a matriz **T1**, cujas linhas são compostas por todas as *q* máscaras $\mathbf{M1_i}$, basta estender a matriz de apagamentos com mais *n* – *k* colunas e preencher as *n* – *k* posições adicionais de cada linha com tantos 1's quantos forem necessários para que cada linha de **T1** tenha exatamente d_{Hmin} 1's, iniciando o preenchimento pelas últimas colunas de **T1**.

$$\mathbf{T1} = \begin{bmatrix} \overbrace{0\,0\,0\,0\,0\,0\,0}^{k}\;\; 0\,0\,0\,1\,1\,1\,1 \\ 0\,0\,0\,0\,0\,0\,1\;\; 0\,0\,0\,0\,1\,1\,1 \\ 0\,0\,0\,0\,0\,1\,0\;\; 0\,0\,0\,0\,1\,1\,1 \\ \cdots \\ 1\,0\,0\,0\,0\,0\,0\;\; 0\,0\,0\,0\,1\,1\,1 \\ 0\,0\,0\,0\,0\,1\,1\;\; 0\,0\,0\,0\,0\,1\,1 \\ 0\,0\,0\,0\,1\,1\,0\;\; 0\,0\,0\,0\,0\,1\,1 \\ \cdots \\ \underbrace{0\,0\,0\,0\,1\,1\,1\;\; 0\,0\,0\,0\,0\,0\,1\,1}_{n} \end{bmatrix} \tag{3.56}$$

A expressão (3.56) mostra um exemplo de obtenção da matriz **T1**, gerada a partir da matriz de apagamentos para o código $C(15,7,5)$ mostrada na Figura 29, na página 81. Cada linha da matriz **T1** mostrada contém uma das máscaras **M1** necessárias para aplicação do critério BGWG e contém exatamente d_{Hmin} = 5 bits 1. As primeiras k colunas de **T1** formam a matriz de apagamentos para o código $C(15,7,5)$.

Tanto a matriz de apagamentos como a matriz **T1** são fixas para cada código e determinadas a priori, não envolvendo qualquer processamento em tempo real exceto a leitura de seus valores.

Até este ponto foi suposto, apenas em teoria e para facilitar a compreensão do critério, que as componentes da soma híbrida entre o vetor recebido **y** e a palavra-código candidata **c** tivessem sido reordenadas de acordo com o vetor de confiabilidades **SI**. Isso porém não é feito na prática, utilizando-se em vez disso o vetor **SI** para indexar as várias colunas acessadas, o que é equivalente. Em termos das máscaras **M1** – e, consequentemente, também **M2** – portanto é necessário acrescentar que não é suficiente apenas tomar a linha correspondente da matriz **T1**. É também necessário indexar coerentemente as posições de **M1** com os índices contidos no vetor **SI**. Em termos de implementação em *hardware* trata-se apenas de uma operação executada por lógica combinacional, não acarretando aumento do tempo de processamento. As máscaras **M1** e **M2** reordenadas de acordo com o vetor de confiabilidades **SI** serão doravante denotadas por $\mathbf{M1_r}$ e $\mathbf{M2_r}$ para diferenciar entre os dois casos.

O critério de parada BGWG se resume portanto em satisfazer duas condições, que serão chamadas de condição $\mathbf{cd_1}$ e condição $\mathbf{cd_2}$:

Condição $\mathbf{cd_1}$: A soma das componentes da soma híbrida entre o vetor recebido **y** e a palavra-código candidata **c**, filtradas pela máscara $\mathbf{M1_r}$, deverá ser negativa ou nula.

Condição $\mathbf{cd_2}$: Nenhuma das componentes da soma híbrida entre o vetor recebido **y** e a palavra-código candidata **c**, filtradas pela máscara $\mathbf{M2_r}$, poderá ser positiva.

Percebe-se então por que a eficiência do critério BGWG é inferior à do BGW: toda vez que a condição $\mathbf{cd_2}$ não é satisfeita significa que dentre as posições

filtradas pela máscara $\mathbf{M2}_r$ há uma ou mais positivas, as quais a rigor deveriam ser incluídas dentre as d_{Hmin} mais positivas para então se aplicar o critério BGW. Entretanto como isso implicaria em tempo de processamento adicional, isso não é feito.

As simulações realizadas mostraram porém que a incidência da condição $\mathbf{cd_2}$ não satisfeita é relativamente baixa, de maneira que a eficiência do critério BGWG, ainda que inferior à do critério BGW, é superior aos demais critérios analisados, justificando sua implementação.

3.3.6 Comparação da eficiência dos critérios de parada

As tabelas 3.8, 3.9 e 3.10 resumem os resultados de simulações realizadas por meio dos *scripts* 20 a 23 ANEXO I, com a finalidade de comparar a eficiência dos critérios de parada examinados até aqui, tanto dos critérios entre si, para um mesmo código, como para um mesmo critério quando aplicado a códigos de comprimento e complexidade diferentes.

Tabela 3.8: Redução porcentual de palavras-código examinadas – *C*(15,7,5)

Eb / No (dB)	Critério GMD (%)	Critério Cone (%)	Critério BGWG (%)	Critério BGW (%)
1	0,06	23,25	55,52	55,71
2	0,15	35,45	67,22	69,18
3	0,36	51,24	78,72	81,32
4	0,83	67,89	87,65	89,89
5	1,77	82,23	93,08	94,44

Tabela 3.9: Redução porcentual de palavras-código examinadas – C(24,12,8)

Eb / No (dB)	Critério GMD (%)	Critério Cone (%)	Critério BGWG (%)	Critério BGW (%)
1	0,00	19,41	55,46	60,71
2	0,00	35,21	69,58	77,80
3	0,02	56,10	82,61	90,33
4	0,05	76,98	91,86	96,78
5	0,02	91,47	96,81	98,80

Tabela 3.10: Redução porcentual de palavras-código examinadas – C(48,24,12)

Eb / No (dB)	Critério GMD (%)	Critério Cone (%)	Critério BGWG (%)	Critério BGW (%)
1	0,00	0,44	16,03	27,22
2	0,00	2,38	31,52	52,67
3	0,00	9,96	53,16	79,08
4	0,00	30,19	75,03	94,70
5	0,00	62,79	90,33	99,26

Como pode ser visto nas tabelas, todos os critérios, com exceção do GMD, têm bom desempenho para alta relação sinal ruído. Os critérios BGW e BGWG mantêm seu desempenho mesmo para códigos mais longos quando a relação sinal ruído é alta. Para baixas relações sinal ruído e códigos longos a eficiência cai bastante. Porém o critério BGW, assim como sua versão sub-ótima BGWG, ainda oferecem economias de processamento em geral acima de um quarto das palavras a analisar. Nas simulações para o código C(15,7,5) foram considerados conjuntos de 30 candidatas, para o C(24,12,8) 120 candidatas e para o C(48,24,12), 500 candidatas. A quantidade de candidatas em cada caso foi selecionada de forma a se obter um desempenho para a taxa de erros de bit na situação de pior condição de relação sinal ruído tal que coincidisse com a taxa para a decodificação por MLD nos dois algarismos mais significativos.

3.4 Influência da normalização e da quantização

Um estudo de algoritmos de decodificação de códigos de bloco com vistas a viabilizar sua implementação em *hardware* programável não pode deixar de examinar o efeito da discretização de valores para permitir seu processamento de forma digital.

Na verdade, ao se realizar simulações em computadores, por exemplo com um *software* como o Matlab, o qual utiliza de uma forma geral em seus cálculos o formato “double” da norma IEEE–457, implementado por praticamente a totalidade dos processadores atuais, já implica em uma discretização de valores. Porém, como no caso do “double” padrão IEEE–457 o número de intervalos de discretização é 2^{52}, esse fato em geral pode ser desconsiderado sem maiores consequências. Outra porém é a situação no caso de implementação em *hardware* programável, onde cada bit adicional na representação de valores impacta diretamente na quantidade de recursos e portanto no custo envolvidos.

Para examinar o efeito da discretização dos valores de nível de sinal sobre o desempenho do decodificador é necessário conhecer ou pelo menos modelar de maneira coerente a natureza do sinal.

Na seção 3.1.4 e subseções foram analisadas as diversas condições e probabilidades de erro com base em um modelo de ruído branco gaussiano aditivo (AWGN). No caso de decodificação por decisão suave foi visto porém que o demodulador submete a palavra-código recebida a um processo de normalização. Esse processo não afeta a probabilidade de erro de bit, uma vez que tanto o valor médio quanto o desvio padrão ficam divididos por uma constante e a variável aleatória padronizada permanece inalterada. Porém o fato do desvio padrão do sinal ser reduzido pelo processo de normalização irá afetar a sensibilidade do sinal à discretização de seus valores. Nas subseções seguintes serão portanto investigados primeiro o efeito da normalização na redução do desvio padrão do sinal e a seguir a influência da quantidade de intervalos de quantização no desempenho do decodificador.

3.4.1 Influência da normalização no desvio padrão do sinal recebido

Dado um código de bloco de comprimento *n* e comprimento da mensa-

gem *k*, com componentes moduladas em BPSK (do inglês *Binary Phase Shift Keying*), de amplitude A, submetido a um canal com relação de sinal ruído E_b/N_0 = *x* dB, foi visto, na seção 3.1.4.1, equação (3.19), na página 63, que o desvio padrão resultante para o nível do sinal será:

$$\sigma_r = \frac{A}{\sqrt{\left(2 \cdot \frac{k}{n} \cdot 10^{\frac{x}{10}}\right)}} \tag{3.57}$$

onde o subscrito *r* foi utilizado para indicar que se trata do desvio real do ruído superposto ao sinal, em contraposição àquele resultante após a normalização.

O demodulador porém, para preservar a confiabilidade relativa dos símbolos recebidos realiza uma normalização a cada bloco de *n* símbolos, dividindo todos os níveis pelo de maior valor, normalizando o sinal entre um valor máximo = + 1 e um valor mínimo = – 1. Neste estudo será desconsiderada, por simplicidade, a possibilidade de truncamento de valores acima de um certo máximo, ainda que comum na prática.

Deseja-se então determinar qual o desvio padrão equivalente, ou normalizado, para um sinal compreendido entre os valores limites ± 1. Neste caso pode-se mais uma vez fazer uso da teoria da ordenação estatística, a qual entre outras coisas permite determinar a distribuição de valores extremos de vários conjuntos de amostras. Conhecida a função distribuição de probabilidade dos elementos de amostras de um conjunto, a correspondente função distribuição cumulativa de probabilidade dos valores máximos de cada amostra de *n* elementos pode ser dada (ARNOLD, 2008, pg. 12, eq. 2.2.12) por:

$$F_{n:n}(x) = F(x)^n \tag{3.58}$$

onde o índice *n*:*n* indica o *n*-ésimo elemento de uma amostra de *n* elementos ordenados em ordem crescente de valor, ou seja o índice indica o valor máximo de cada amostra.

A correspondente função densidade de probabilidade pode então ser obtida derivando-se a equação (3.58), obtendo:

$$f_{n:n}(x) = n \cdot f(x) \cdot F(x)^{n-1} \tag{3.59}$$

No caso específico em estudo tem-se blocos de *n* símbolos, com valores médios iguais a ± 1 e desvio padrão σ_r dado pela equação (3.57). Considerando-se por exemplo apenas o valor médio + 1, a equação (3.59) fica:

$$f_{n:n}(x)=\frac{n}{\sqrt{2\pi}\,\sigma}\,\mathrm{e}^{-\frac{1}{2}\frac{(x-1)^2}{\sigma_x^2}}\times\left(\frac{1}{\sqrt{2\pi}\,\sigma_r}\int_{-\infty}^{x}\mathrm{e}^{-\frac{1}{2}\frac{(u-1)^2}{\sigma_r^2}}\,du\right)^{n-1} \tag{3.60}$$

Conhecendo-se $f_{n:n}(x)$ pode-se então obter seu valor médio, que será o valor pelo qual, em média, serão normalizados todos os símbolos recebidos. Dessa forma também o desvio padrão dos símbolos ficará dividido pelo valor médio. obtém-se então:

$$\mu_{n:n}=E[f_{n:n}(x)]=\int_{-\infty}^{+\infty}x\cdot f_{n:n}(x)\cdot dx \tag{3.61}$$

e o desvio padrão dos sinais normalizados valerá portanto:

$$\sigma_n=\frac{\sigma_r}{\mu_{n:n}} \tag{3.62}$$

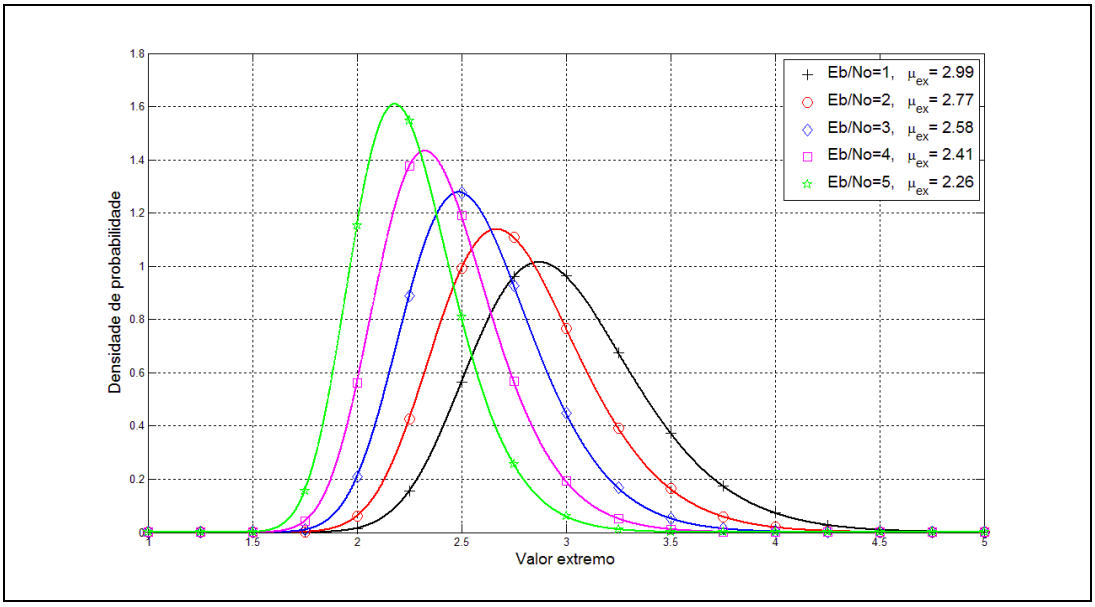

Figura 39 – Distribuição teórica de valores extremos para código *C*(48,24,12)

A Figura 39 mostra a distribuição obtida a partir da equação (3.60) para o código *C*(48,24,12), supondo vários níveis de ruído superposto. Os valores e o gráfi-

co foram obtidos através do Script 25 listado no ANEXO I. Para validar os resultados acima foram realizadas simulações com 10^5 palavras-código e gerados os respectivos histogramas para os valores extremos. A Figura 40 mostra as envoltórias dos histogramas para o código $C(48,24,12)$. Os valores e o gráfico foram obtidos através do Script 26 listado no ANEXO I.

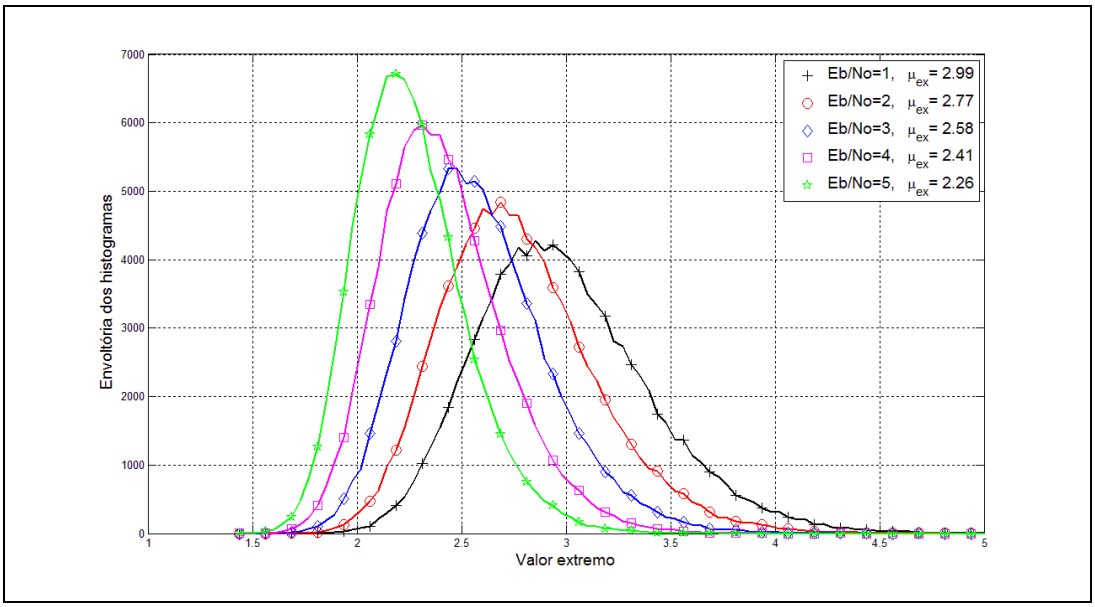

Figura 40 – Envoltórias de histogramas de valores extremos para código $C(48,24,12)$

Nas legendas das figuras estão também listados os valores médios obtidos para os valores máximos para cada relação de sinal ruído considerada.

As tabelas 3.11, 3.12 e 3.13 resumem os resultados para 3 códigos diferentes. A última coluna de cada tabela contém o desvio padrão do sinal normalizado obtido por meio de simulações com 10^5 palavras-código, através de seu estimador. Como pode ser visto nas tabelas, os valores do desvio padrão teórico o obtido por simulações difererem em menos de uma parte em 100.

Tabela 3.11: Desvio padrão dos valores normalizados – *C* (15,7,5)

Eb / No (dB)	σ_r	σ_{ext}	μ_{ext}	$\sigma_n = \sigma_r / \mu_{ext}$	σ_n *simulado*
1	0,9225	0,5063	2,6014	0,3546	0,3575
2	0,8222	0,4512	2,4273	0,3387	0,3412
3	0,7328	0,4022	2,2720	0,3225	0,3245
4	0,6531	0,3585	2,1337	0,3061	0,3077
5	0,5821	0,3195	2,0104	0,2895	0,2910

Tabela 3.12: Desvio padrão dos valores normalizados – *C* (24,12,8)

Eb / No (dB)	σ_r	σ_{ext}	μ_{ext}	$\sigma_n = \sigma_r / \mu_{ext}$	σ_n *simulado*
1	0,8913	0,4558	2,7359	0,3258	0,3306
2	0,7943	0,4062	2,5471	0,3119	0,3160
3	0,7079	0,3620	2,3788	0,2976	0,3016
4	0,6310	0,3227	2,2289	0,2831	0,2866
5	0,5623	0,2876	2,0953	0,2684	0,2722

Tabela 3.13: Desvio padrão dos valores normalizados – *C* (48,24,12)

Eb / No (dB)	σ_r	σ_{ext}	μ_{ext}	$\sigma_n = \sigma_r / \mu_{ext}$	σ_n *simulado*
1	0,8913	0,4160	2,9903	0,2981	0,3036
2	0,7943	0,3708	2,7738	0,2864	0,2915
3	0,7079	0,3304	2,5809	0,2743	0,2790
4	0,6310	0,2945	2,4090	0,2619	0,2668
5	0,5623	0,2625	2,2558	0,2493	0,2539

3.4.2 Influência da quantidade de intervalos de quantização

A discretização de um sinal em *nq* níveis de quantização, com cada intervalo com uma amplitude *q* implica na introdução de um erro aleatório com valor má-

ximo igual a $\pm q$ / 2. Por esse motivo muitos modelos de estudo equiparam a quantização em nq intervalos de amplitude q à adição de um ruído uniformemente distribuído entre $+q/2$ e $-q/2$, com média zero e variância $q^2/12$. Entretanto Widrow em (WIDROW, 2008, cap. 4) mostrou que esse modelo, a que ele denomina de PQN, sigla para "*Pseudo Quantization Noise*" é apenas uma aproximação, tendo validade apenas dentro de determinadas condições. O objetivo do estudo de Widrow é investigar as condições em que a distribuição de probabilidades e outras estatísticas do sinal original podem ser recuperadas sem erros a partir do sinal quantizado. Seus resultados entretanto servirão também para justificar a aplicabilidade do modelo PQN a nosso estudo.

De acordo com Widrow, o modelo PQN pode ser aplicado para intervalos de quantização de até aproximadamente um desvio padrão, com precisão melhor que 1 parte em 10^7 para estimativas do desvio padrão no caso de sinais com distribuição Gaussiana. Já em 1974 Dorsch, em (DORSCH,1974) sugeriu a utilização de intervalos da ordem de $0{,}35\sigma$ para as suas simulações por computador. No estudo em pauta, serão avaliadas as consequências de se discretizar o sinal recebido e normalizado com de 3 a 8 bits de quantização, ou seja, de $2^3 = 8$ até $2^8 = 256$ intervalos de quantização. Para um sinal normalizado entre ± 1 isso irá produzir intervalos com amplitudes entre $q = 2/2^3 = 0{,}25$ até $q = 2/2^8 = 0{,}0078125$, de maneira que, comparando com os valores de σ_n da última coluna das tabelas 3.11 a 3.13, no pior caso resultará sempre $q < \sigma_n$ e o modelo PQN se aplica sem maiores restrições.

Aplicando portando o modelo PQN tem-se que o sinal quantizado pode ser representado como uma variável aleatória resultante da soma de duas variáveis aleatórias independentes: uma delas é o sinal normalizado com distribuição Gaussiana com média μ_{ext} e σ_{ext} conforme os valores das tabelas 3.11 a 3.13. A outra é um sinal uniformemente distribuído entre $\pm q/2$, com média $\mu_u = 0$ e $\sigma^2_u = q^2/12$.

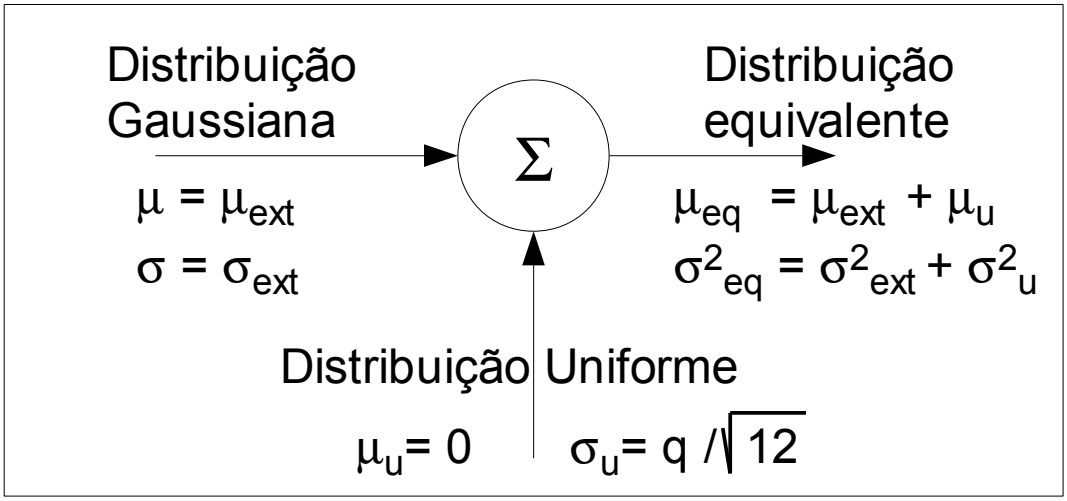

Figura 41 – Modelo PQN para avaliação do efeito do ruído de quantização

A Figura 41 ilustra a aplicação do modelo. Como as variáveis aleatórias envolvidas são estatisticamente independentes a média resultante será a soma das médias e a variância resultante será a soma das variâncias. Definindo o intervalo de quantização como uma fração α da variância normalizada σ_n tem-se:

$$\alpha = \frac{q}{\sigma_n} \quad \Rightarrow \quad q = \alpha \sigma_n \tag{3.63}$$

e o desvio padrão equivalente pode ser colocado como:

$$\sigma_{eq} = \sqrt{\sigma_n^2 + \frac{q^2}{12}} = \sigma_n \sqrt{1 + \frac{\alpha^2}{12}} \tag{3.64}$$

ou seja, tudo se passa como se o desvio padrão do sinal normalizado fosse incrementado de um fator $(1 + \alpha^2/12)^{½}$ além disso, como o modelo é linear, pode-se estimar o desvio padrão real do ruído Gaussiano que causaria o mesmo efeito da quantização, multiplicando ambos os membros da equação (3.64) por μ_{ext}:

$$\sigma_{req} = \mu_{ext} \times \sigma_{eq} = \mu_{ext} \times \sigma_n \sqrt{1 + \frac{\alpha^2}{12}} = \sigma_r \times \sqrt{1 + \frac{\alpha^2}{12}} \tag{3.65}$$

A partir da equação (3.65) pode-se então determinar a degradação da relação sinal / ruído decorrente da quantização. Reescrevendo a equação (3.19) isolando a relação sinal / ruído em decibéis obtém-se:

$$(E_b/N_0)_r \;=\; 10\log_{10}\left(\frac{A^2}{2R\sigma_r^2}\right) \tag{3.66}$$

E em termos da relação sinal / ruído equivalente:

$$(E_b/N_0)_{req} \;=\; 10\log_{10}\left(\frac{A^2}{2R\sigma_{req}^2}\right) \;=\; 10\log_{10}\left(\frac{A^2}{2R\sigma_r^2(1+\frac{\alpha^2}{12})}\right) \tag{3.67}$$

E a redução em decibéis da relação sinal / ruído pode ser obtida subtraindo-se a equação (3.67) da equação (3.66):

$$\Delta(E_b/N_0) \;=\; (E_b/N_0)_r - (E_b/N_0)_{req} \;=\; 10\log_{10}\left(1+\frac{\alpha^2}{12}\right), \qquad \alpha \;=\; \frac{q}{\sigma_n} \tag{3.68}$$

As tabelas 3.14 a 3.16 e as figuras 42 a 44 resumem a degradação das relações sinal / ruído resultantes para códigos com taxas e comprimentos diferentes, obtidos a partir da equação (3.68), utilizando o Script 27 do matlab, contido no ANEXO I.

Tabela 3.14: Degradação da relação S/R devida à quantização – *C* (15,7,5)

qtde de bits	E_b/N_o = 1 dB	E_b/N_o = 2 dB	E_b/N_o = 3 dB	E_b/N_o = 4 dB	E_b/N_o = 5 dB
3	0,1763	0,1928	0,2122	0,2349	0,2618
4	0,0447	0,0490	0,0540	0,0599	0,0670
5	0,0112	0,0123	0,0136	0,0151	0,0168
6	0,0028	0,0031	0,0034	0,0038	0,0042
7	0,0007	0,0008	0,0008	0,0009	0,0011
8	0,0002	0,0002	0,0002	0,0002	0,0003

Tabela 3.15: Degradação da relação S/R devida à quantização – *C* (24,12,8)

qtde de bits	E_b/N_o = 1 dB	E_b/N_o = 2 dB	E_b/N_o = 3 dB	E_b/N_o = 4 dB	E_b/N_o = 5 dB
3	0,2080	0,2265	0,2482	0,2734	0,3032
4	0,0530	0,0577	0,0634	0,0700	0,0778
5	0,0133	0,0145	0,0159	0,0176	0,0196
6	0,0033	0,0036	0,0040	0,0044	0,0049
7	0,0008	0,0009	0,0010	0,0011	0,0012
8	0,0002	0,0002	0,0002	0,0003	0,0003

Tabela 3.16: Degradação da relação S/R devida à quantização – *C* (48,24,12)

qtde de bits	E_b/N_o = 1 dB	E_b/N_o = 2 dB	E_b/N_o = 3 dB	E_b/N_o = 4 dB	E_b/N_o = 5 dB
3	0,2474	0,2674	0,2907	0,3179	0,3495
4	0,0632	0,0684	0,0745	0,0817	0,0900
5	0,0159	0,0172	0,0187	0,0206	0,0227
6	0,0040	0,0043	0,0047	0,0051	0,0057
7	0,0010	0,0011	0,0012	0,0013	0,0014
8	0,0002	0,0003	0,0003	0,0003	0,0004

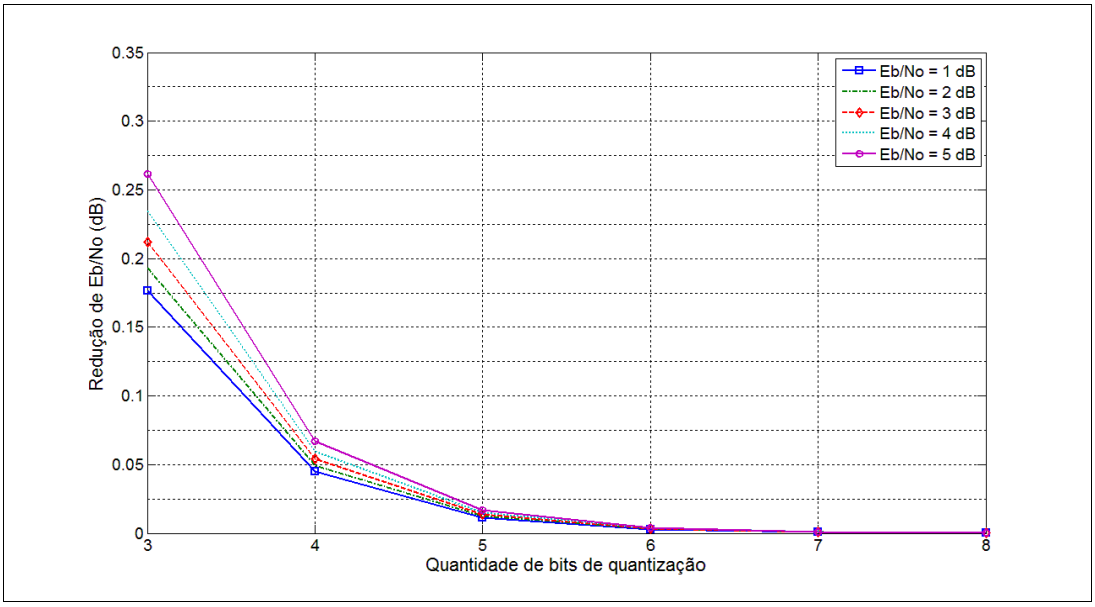

Figura 42 – Degradação da relação S/R devida à quantização – código $C(15,7,5)$

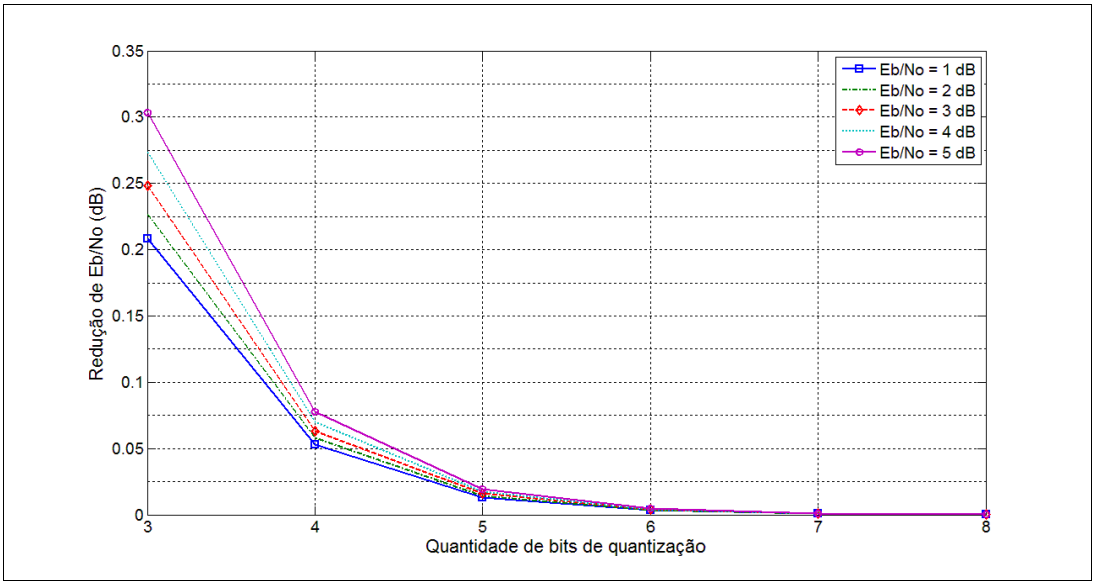

Figura 43 – Degradação da relação S/R devida à quantização – código $C(24,12,8)$

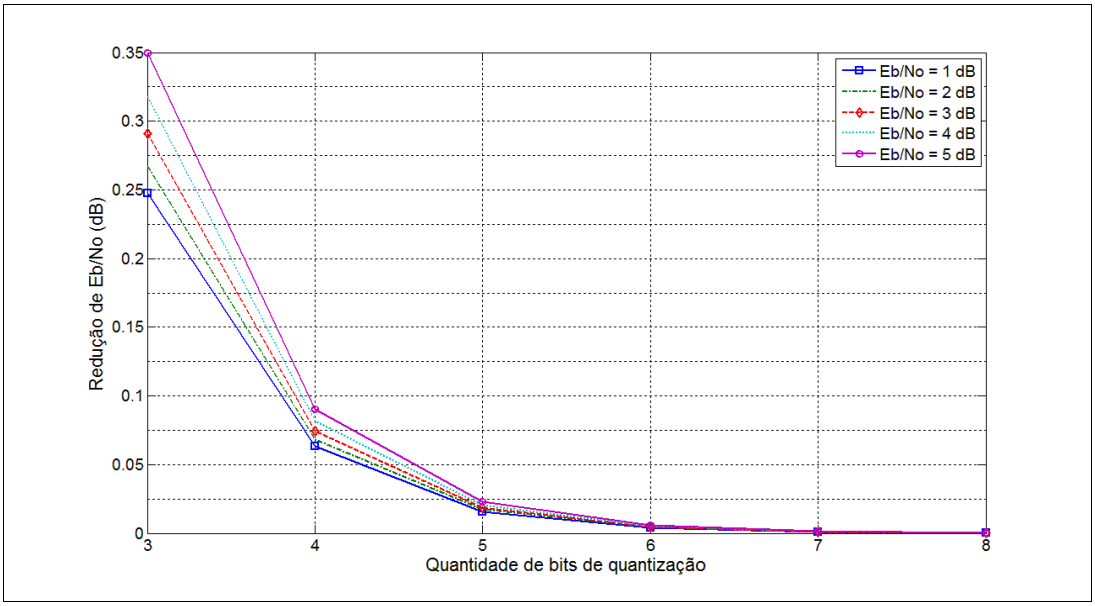

Figura 44 – Degradação da relação S/R devida à quantização – código $C(48,24,12)$

3.4.3 Densidade de probabilidade da distribuição do modelo PQN

Na seção anterior foi visto como determinar o desvio padrão equivalente do sinal quantizado e a correspondente relação sinal ruído E_b/N_o. Como o modelo de estudo até este ponto foi o de um sinal submetido à interferência de ruído Gaussiano aditivo branco – AWGN – resta ainda inquirir até que ponto o efeito da quantização do sinal recebido se afasta desse modelo.

O modelo PQN (do inglês *Pseudo Quantization Noise*) adotado consiste na soma de duas variáveis aleatórias com distribuições diferentes de probabilidades. O resultado da soma de duas variáveis aleatórias será também uma variável aleatória e sua distribuição é obtida através da convolução das distribuições originais. O Script 28 matlab, no ANEXO I, realiza essa convolução para uma variável normal padronizada e uma distribuição uniforme entre ± q/2, com $q = \alpha\sigma$. O resultado está mostrado nas figuras 45 e 46. Na Figura 45 foi implementada a relação $\alpha = q / \sigma = 4$, apenas com o intuito de evidenciar a diferença entre o resultado da convolução e uma distribuição normal com o mesmo desvio padrão. Mesmo nessa situação exagerada a diferença entre as duas curvas difere em menos de 2,5 partes em 100. Nos casos abordados neste estudo tem-se sempre $\alpha < 1$, e a diferença entre as curvas fica inferior 4 partes em 1000. A Figura 46 ilustra a situação para $\alpha = 1$. Nessa figura a curva normal e o resultado da convolução se sobrepõem de forma a ficarem indistinguíveis. A figura foi truncada no eixo vertical para melhor realçar as curvas

sendo comparadas.

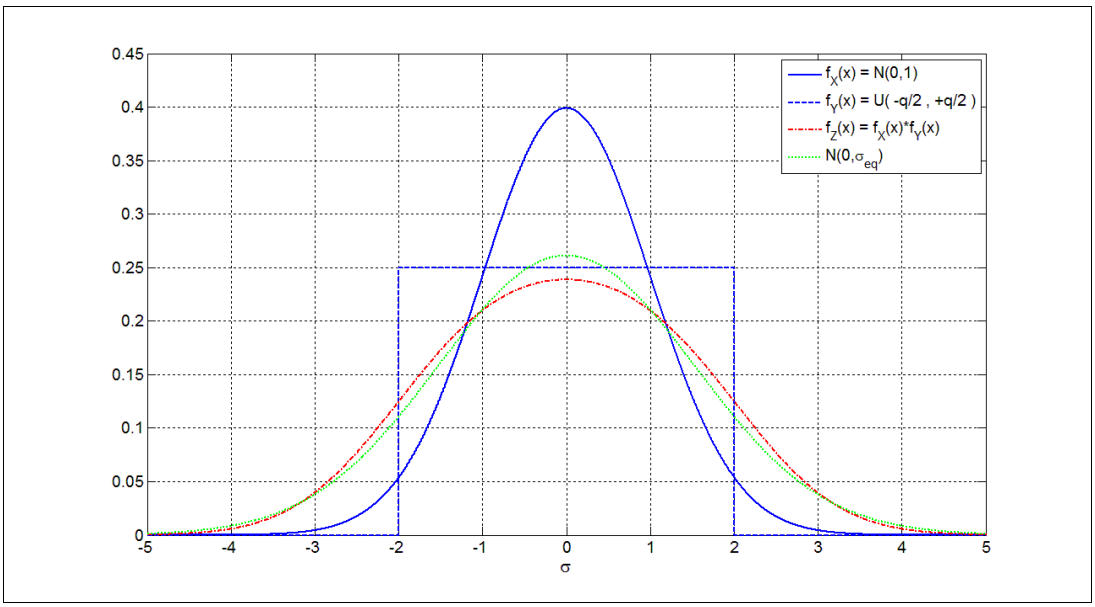

Figura 45 – Distribuição resultante para $q = 4\ \sigma$

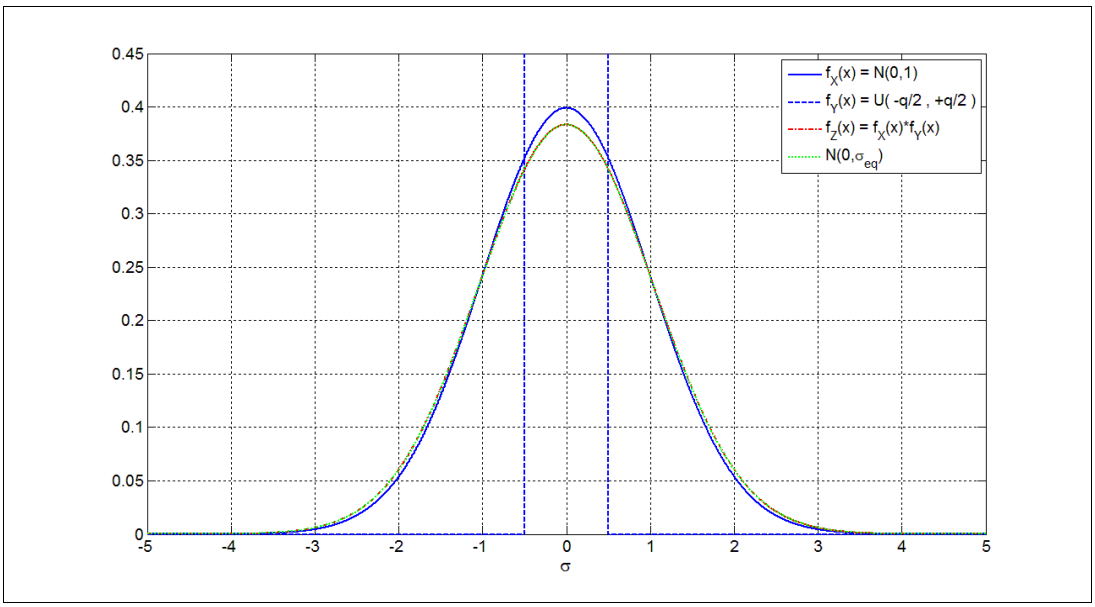

Figura 46 – Distribuição resultante para $q = \sigma$

3.4.4 Validação dos resultados através de simulações

As seções anteriores concluíram, através da utilização do modelo PQN (do inglês *Pseudo Quantization Noise*), que a discretização do sinal recebido norma-

lizado equivale a uma degradação da relação sinal ruído e forneceram meios de estimar quantitativamente essa degradação. Porém, a capacidade de estimar uma relação sinal ruído equivalente não é garantia de que o algorítimo de decodificação terá, com o sinal quantizado, o mesmo desempenho que teria com o sinal não quantizado porém submetido a uma relação sinal ruído correspondentemente reduzida. O motivo desse questionamento é que a verdadeira distribuição de probabilidade do sinal quantizado não é gaussiana e sim, de acordo com Widrow (WIDROW 2008, cap. 4), é constituída por uma série de impulsos de Dirac, cuja área corresponde à área do intervalo de quantização sob a curva normal, como mostrado na Figura 47.

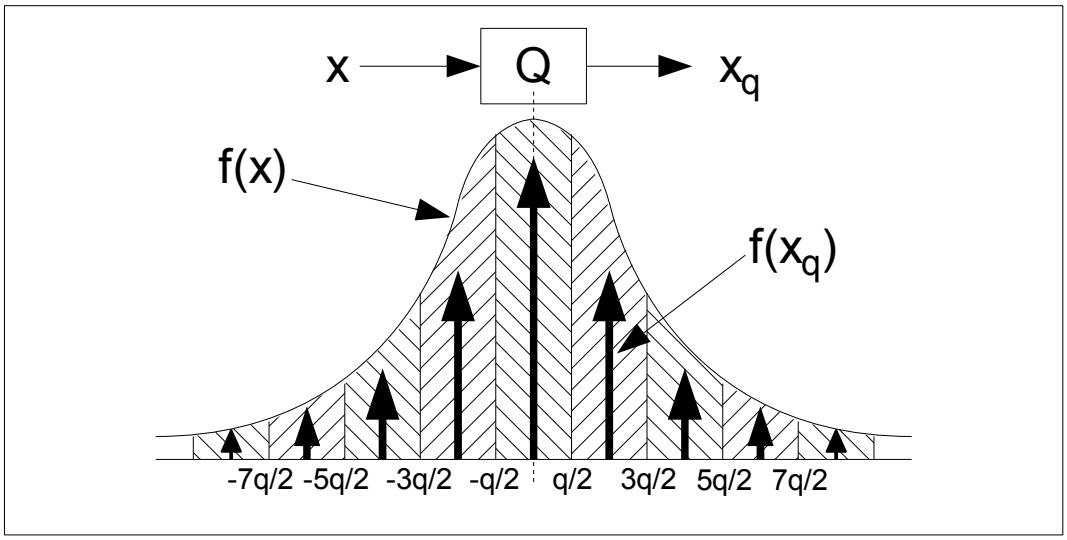

Figura 47 – Distribuição do sinal antes e após a quantização

A figura faz sentido se se considerar que o processo de quantização aproxima todos os valores contidos dentro de um intervalo de quantização para o valor central a esse intervalo, ou seja, toda a área de um intervalo sob a curva de distribuição fica concentrada em um único ponto que é o centro do intervalo. Widrow, em (WIDROW 2008, cap. 4), mostrou que as áreas dos impulsos de Dirac correspondem ao valor da amostra, no centro do intervalo, da convolução do sinal não quantizado com um pulso retangular correspondente à distribuição uniforme do ruído do modelo PQN.

Dada, portanto, a verdadeira natureza da distribuição de probabilidades do sinal discretizado, cabe inquirir se o desempenho do algorítimo de decodificação tem o mesmo comportamento para esse sinal que teria para um sinal não discretizado mas submetido à uma relação sinal ruído reduzida de acordo com os valores deduzidos nas seções anteriores.

Essa verificação foi feita através de simulações realizadas com o Script 29 do matlab, listado no ANEXO I, na página 164, para os códigos *C*(15,7,5), *C*(24,12,8) e *C*(48,24,12). A estratégia utilizada pelo *script* consiste em primeiro normalizar o sinal recebido entre ± 1 e a seguir, aplicando a função "`QuantizaPcr`", listada no Script 30, no ANEXO I, na página 171, associar a todos os valores pertencentes a cada faixa um único valor correspondente ao centro da faixa. A Figura 48 ilustra esse processo para o caso particular de 8 níveis.

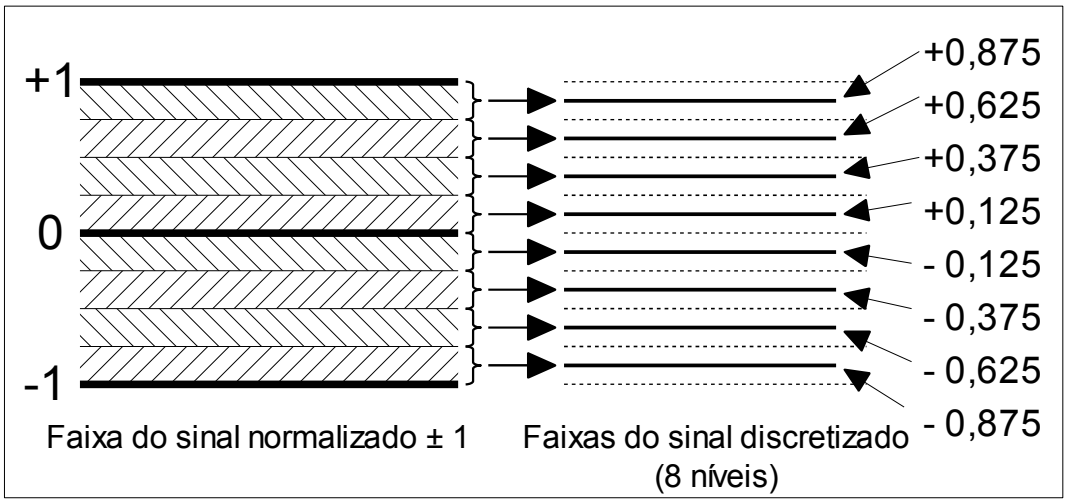

Figura 48 – Atribuição de valores discretos às faixas de quantização

As curvas das figuras 49 a 51 mostram os resultados obtidos para o caso de quantização com 3 bits, ou seja, 8 níveis.

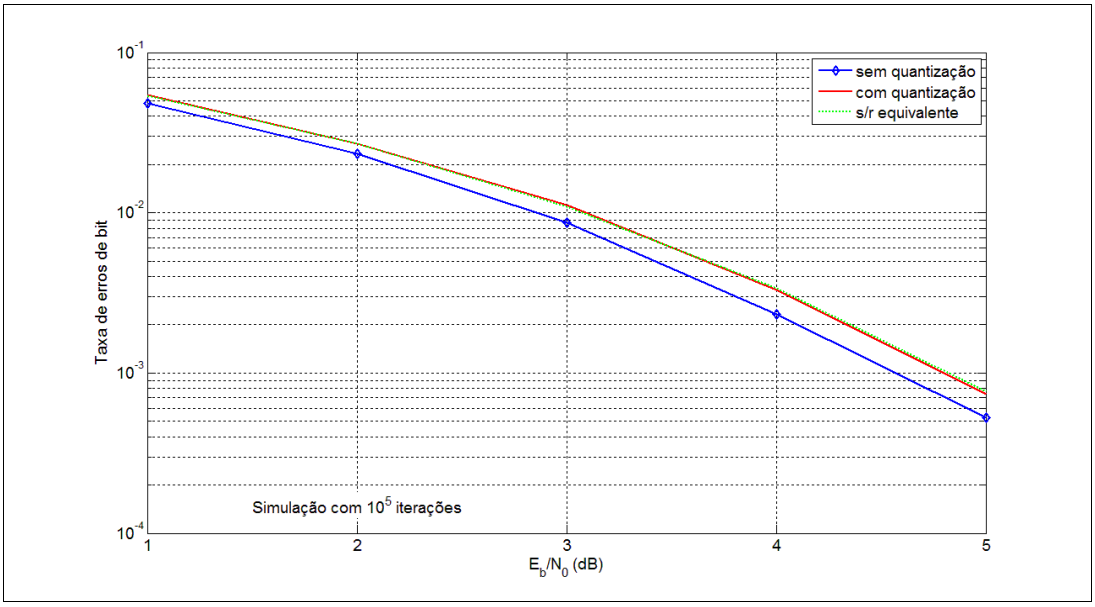

Figura 49 – Validação dos efeitos da quantização (8 níveis) – código *C*(15,7,5)

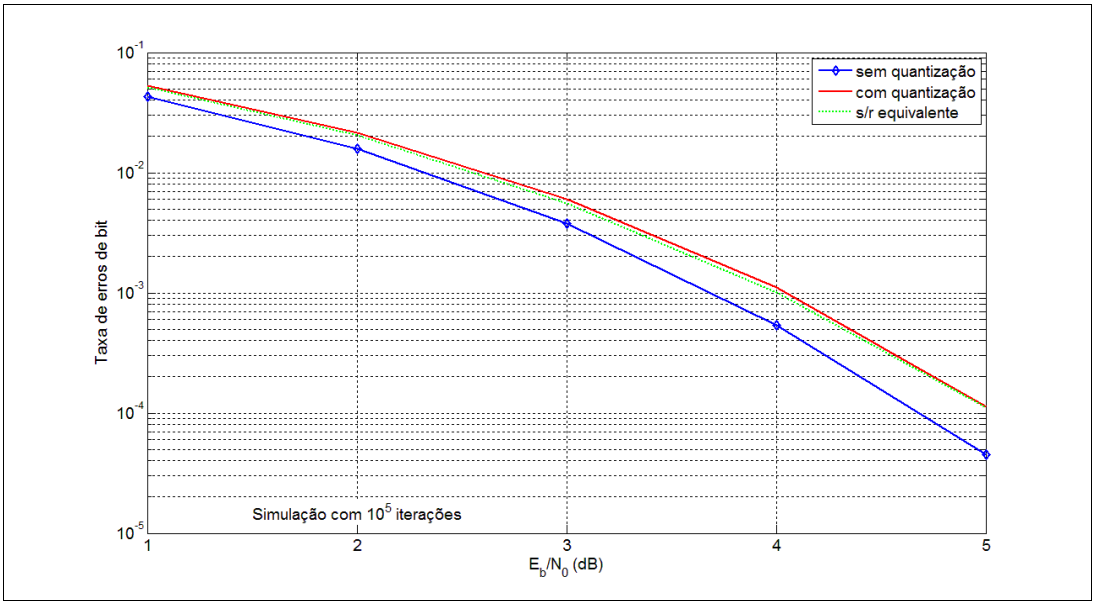

Figura 50 – Validação dos efeitos da quantização (8 níveis) – código $C(24,12,8)$

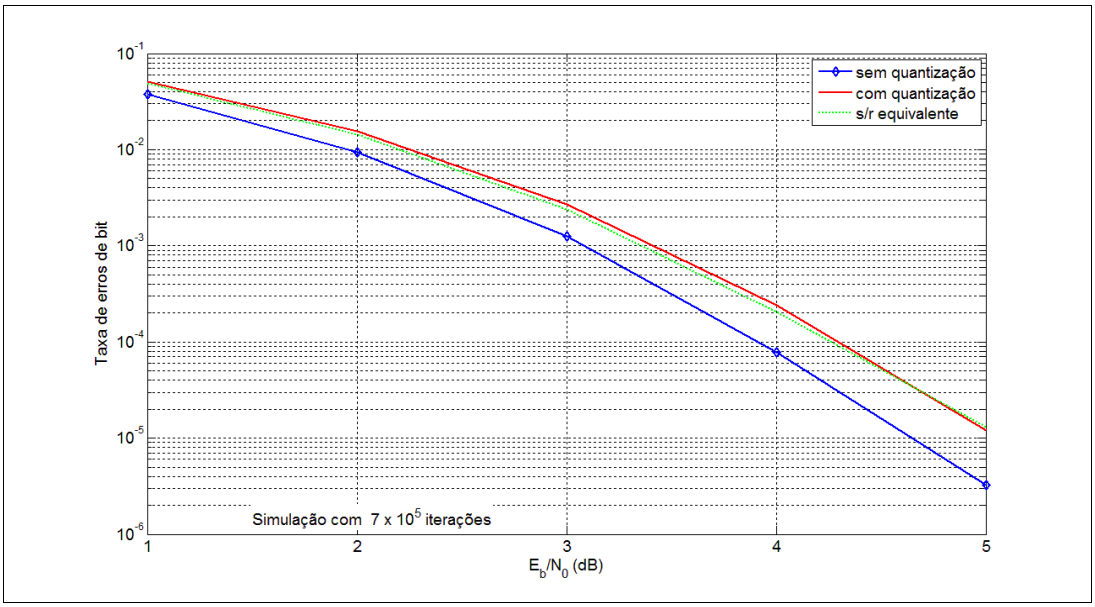

Figura 51 – Validação dos efeitos da quantização (8 níveis) – código $C(48,24,12)$

Como se vê nas figuras, os resultados validam perfeitamente as estimativas de degradação de sinal ruído determinadas nas subseções anteriores. As curvas para desempenho do algorítimo com valores discretizados e aquelas para o mesmo algorítimo porém com a redução equivalente da relação sinal ruído estão

praticamente superpostas. Os *scripts* permitem levantar os valores e as curvas para até 256 níveis, porém a partir de 32 níveis as três curvas ficam praticamente superpostas, motivo pelo qual não são apresentadas aqui.

Conclui-se portanto que as curvas e tabelas apresentadas na subseção 3.4.2, assim como as equações e meios apresentados para obtê-las, representam uma ferramenta válida para capacitar o projetista de *hardware* a decidir por um compromisso entre recursos de *hardware* a utilizar e degradação da relação sinal ruído introduzida pela discretização dos sinais recebidos.

4 IMPLEMENTAÇÃO PRÁTICA E RESULTADOS OBTIDOS

As análises realizadas e os métodos desenvolvidos neste trabalho tiveram como objetivo fornecer a base teórica e responder questões referentes à implementação de decodificadores de códigos de bloco em *hardware*. A implementação propriamente dita foi realizada por pesquisadores do LME – Laboratório de Microeletrônica da UTFPR (Universidade Tecnológica Federal do Paraná). A seguir será apresentado um breve resumo dos resultados obtidos nas várias implementações.

A primeira implementação (GORTAN, 2010a) foi realizada com uma FPGA Altera da família Stratix III e seu principal objetivo foi validar a implementação básica do algoritmo BP, usando as várias operações matriciais desenvolvidas e relatadas no item 3.1 desta dissertação. Adicionalmente, os primeiros estudos sobre a utilização da redução de Gauss modificada levaram também ao desenvolvimento de uma implementação em FPGA de um algoritmo para inversão de matrizes binárias não singulares de dimensão N x N com complexidade O(N), como relatado em (JASINSKI, 2010).

Tabela 4.1: Resultados da síntese – Altera Stratix III – EP3SL70F780C2

Código	Registradores	LUTs	f_{max} (MHz)	Latência (ciclos)	ttd_{max} (ciclos)	*througput* (Mbps)
C(7,4,3)	291	443	159,1	19	5	127,3
C(15,7,5)4	838	1214	111,1	36	11	70,7
C(24,12,8)	1954	3272	84,2	56	17	59,4
C(48,24,12)	6808	10295	55,5	112	37	36,0
C(66,33,12)	12387	17933	50,1	157	55	30,0
C(78,39,14)	16972	26639	41,4	185	66	24,8

Na primeira implementação do decodificador o número de candidatas utilizado foi simplesmente k + 1, para todos os códigos implementados. A Tabela 4.1 mostra os resultados alcançados tanto em termos de utilização de recursos como registradores e tabelas de consulta (LUTs, da sigla em inglês para "*Look-up Tables*"), quanto em parâmetros de desempenho como frequência máxima de operação, retardos e quantidade de bits processados por segundo.

Uma segunda implementação (GORTAN, 2010b) foi então realizada com uma FPGA Altera da família Stratix IV, na qual foram introduzidas facilidades para utilização de um número variável de candidatas, de acordo com os resultados desenvolvidos no item 3.2 desta dissertação.

Tabela 4.2: Resultados da síntese – Altera Stratix IV – EP4SGX70DF292C2X

Código	Registradores	LUTs	f_{max} (MHz)	Latência (ciclos)	ttd_{max} (ciclos)	*througput* (Mbps)
C(7,4,3)	253	390	161,5	19	5	129,2
C(15,7,5)4	579	837	121,8	40	12	71,1
C(24,12,8)	1159	2006	103,5	84	41	30,3
C(48,24,12)	3436	5679	82,0	667	580	3,4

A Tabela 4.2 resume a utilização de recursos e o desempenho obtidos nessa segunda implementação. Neste caso é importante ressaltar que o número de ciclos necessários para a decodificação de cada palavra passa a variar com a quantidade de candidatas utilizadas. Os valores referidos na Tabela 4.2 foram obtidos utilizando-se em cada caso uma quantidade de candidatas suficiente para se atingir 99% do desempenho de um decodificador MLD (do inglês "*Maximum Likelihood Decoding"*).

Finalmente, essa segunda implementação foi modificada (GORTAN, 2012 - submetido) para acrescentar o critério de parada BGWG (iniciais de Barros, Godoy, Wille e Gortan) ou HO-BGW (do inglês "*Hardware Oriented* BGW"), de acordo com o desenvolvido e apresentado no item 3.3 desta dissertação, para reduzir o número de candidatas examinadas.

Tabela 4.3: Uso de HW – critério de parada – Altera Stratix IV – EP4SGX70DF292C2X

	C(7,4,3)	C(15,7,5)	C(24,12,8)	C(48,24,12)	aumento médio (%)
LUTs	894	1688,0	3095	9197	129,2
Registradores	860	1729,0	3011	7556	71,1
f_{max} (MHz)	145,3	127,6	107,9	92,7	30,3

A Tabela 4.3 mostra como essa implementação exigiu um aumento substancial da quantidade de recursos, basicamente devidos à introdução do critério de parada. Por outro lado o número de candidatas examinadas pôde ser reduzido em até 96,8%, mantendo-se a mesma frequência de operação, a qual em alguns casos chegou mesmo a aumentar. Com isso a quantidade de bits processados por segundo ou "throughput" pôde ser aumentada para até 30 vezes o valor original, justificando plenamente a maior utilização de recursos. Não obstante o aumento de recursos, o circuito utilizado apresenta uma alta eficiência em termos da área utilizada. No caso do código mais longo implementado, o $C(48,24,12)$, apenas 20% da área da FPGA da família Stratix IV foi utilizada.

5 CONCLUSÕES E PROPOSTAS PARA TRABALHOS FUTUROS

Este trabalho procurou enfocar a decodificação de códigos de bloco lineares por meio de conjuntos de informação sob uma perspectiva que os tornasse passíveis de implementação em *hardware* programável, como por exemplo em FPGAs (do inglês *Field Programmable Gate Array*).

Dentro dessa ótica buscou-se sempre realizar uma abordagem quantitativa, que permitisse avaliar até que ponto a utilização de mais recursos de *hardware* seria traduzida por aumentos de eficiência tanto em termos de taxas de erros de bit menores ou ganhos de codificação maiores como em velocidade de operação ou latência de decodificação.

Assim, não apenas foi proposto um método sistemático para obtenção de conjuntos de informação, mas foram fornecidos meios de se avaliar as quantidades mínima e máxima de operações para esse fim, assim como suas relativas probabilidades de ocorrência.

Qualitativamente, a decodificação de códigos de bloco por decisão suave por meio de conjuntos de informação consiste em se conseguir isolar, dentre todas as palavras do código, um conjunto de palavras-código candidatas altamente provável de conter a decodificação correta e, dentre essas, selecionar a de menor distância euclidiana para a palavra recebida. Neste trabalho, foram propostos meios de se avaliar as probabilidades relativas entre as várias candidatas possíveis, permitindo sua ordenação por nível de confiabilidade. Além disso, foram desenvolvidos métodos para se dimensionar a degradação do ganho de codificação quando comparado ao ganho de codificação do MLD, em função da quantidade de candidatas a se empregar. Consegue-se dessa forma fundamentar o emprego de maiores ou menores recursos em uma implementação em *hardware* com base no desempenho quantitativo desejado.

No caso do emprego de técnicas de conjuntos de informação implementadas em *hardware*, dois fatores influenciam a latência do decodificador ou o tempo médio de decodificação: por um lado, a quantidade de operações necessárias para se encontrar o conjunto de informação constituído pelas posições mais confiáveis da palavra código recebida, por outro, a quantidade de palavras-código candidatas al-

ternativas a se utilizar. Como já mencionado, ambos os fatores foram analisados de maneira a possibilitar uma abordagem quantitativa de seus efeitos. Entretanto, no que diz respeito ao número de candidatas a serem examinadas a cada ciclo de decodificação, uma otimização adicional pode ser introduzida na forma de regras ou critérios de parada. Um critério de parada, quando satisfeito, permite encerrar a sequência de análise das candidatas antecipadamente, reduzindo a latência de decodificação. Neste trabalho, diversos critérios de parada existentes foram avaliados e o mais eficiente foi adaptado de forma a viabilizar sua utilização em *hardware* programável. A adaptação foi necessária para evitar que a implementação do critério introduzisse uma latência superior à redução à qual o mesmo se propunha. Foram preparados *scripts* para a simulação do emprego do critério proposto com vários códigos, fornecendo, uma vez mais, dados quantitativos úteis para a tomada de decisão quanto à sua implementação em *hardware* programável.

Finalmente, em algoritmos de decodificação por decisão suave, a comparação entre valores analógicos como distâncias euclidianas são uma parte central do processo de decodificação. A discretização desses valores para processamento digital em *hardware* é portanto inevitável e atenção deve ser dada à quantidade ideal de níveis de quantização a utilizar, de forma a nem sobrecarregar o *hardware* devido a processamento com palavras desnecessariamente longas e nem reduzir o ganho de codificação devido a processamento como palavras muito curtas. A parte final deste trabalho analisou a influência da quantidade de níveis de quantização no desempenho do decodificador, fornecendo, ali também, dados quantitativos para dimensionamento em *hardware* programável.

Como trabalhos futuros, sugere-se a análise da viabilidade das técnicas aqui apresentadas para sistemas com modulação multipolar, uma vez que todos os estudos realizados aqui foram baseados em modulação simples bipolar. Igualmente, seria interessante verificar se os conhecimentos obtidos até este ponto podem ser aproveitados ou adaptados ao uso de outras técnicas de codificação, particularmente de códigos LDPC – “Low Density Parity Check Codes”, os quais têm recebido renovada atenção dos pesquisadores ultimamente. Outro campo promissor de se beneficiar com a aplicação de decodificadores de códigos de bloco implementados em *hardware* é o da inserção e extração de marcas d'água em documentos digitais, como sugerido por Gortan e Charles W. Fung em (FUNG, 2011).

ANEXO I
Scripts e Funções para Octave / Matlab

```
function dp = AchaPesos(G)
% Determina a distribuição de pesos de um código
% entradas:
%   G - matriz geradora do código de dimensões k x n
% saídas:
%   dp - vetor distribuição de pesos de dimensões 1 x n

    [k n]=size(G);              % encontra as dimensões da matriz
    dp=zeros(1,n);              % pre-aloca vetor dp

    %-------- loop para codificar todas as palavras: -------------------

    for i=1:2^k-1               % para todas as mensagens exceto a nula:
        m=bitget(i,k:-1:1);     % gera a mensagem em binário
        c=mod(m*G,2);           % codifica-a com a matriz
        p=sum(c);               % encontra o peso da palavra-código
        dp(p)=dp(p)+1;          % atualiza o vetor de pesos
    end
end
```

Script 1: Obtenção da distribuição de pesos de um código.

```
function ma = AgrupaPesos(G,dp)
% Agrupa as palavras código ordenadas por peso
% entradas:
%   G  - matriz geradora do código de dimensões k x n.
%   dp - vetor com a distribuição de pesos de dimensões 1 x n.
%   limitação: k max suportado = 12
% saídas:
%   ma - matriz das palavras código agrupadas por pesos em ordem
%        crescente - dimensões 2^k-2 x n.  Não inclui a palavra de
%        peso 0 nem a de peso n.
    [k n]=size(G);              % obtem as dimensões da matriz

    %----------- Verificação da coerência das entradas: ----------

    if size(dp,2) ~= n          % verifica coerência das dimensões
        fprintf('Dimensões de G e pd incoerentes!\n');
        fprintf('encerrando...\n');
        return;
    end

    %----------- Verificação do limite suportado: ----------------
    if k > 12                   % máximo k suportado é 24
        fprintf('k max suportado = 24 - encerrando...\n');
        return;
    end

    % ---------- Preparação da matriz de saída e seus índices: ---

    ma = zeros(2^k-2,n);        % prealocamos a matriz agrupada
    dp(n)=[];                   % eliminamos vetor tudo 1
    idx = find(dp);             % obtemos índices não nulos de dp
    vp = dp(idx);               % montamos vetor com pesos não nulos
    sidx = size(idx,2);         % descobrimos quantos são
    vidx=zeros(1,sidx);         % prealocamos vetor offset índices ma
    vidx(1)=1;                  % primeiro offset é 1
    for i=2:sidx                % demais de acordo com distr. pesos
        vidx(i)=vidx(i-1)+vp(i-1);
    end                         % vidx contém offsets para cada faixa

    %-------- loop para codificar todas as palavras: -------------------

    for i=1:2^k-2               % todas as mensagens exceto peso 0 e n
        m=bitget(i,k:-1:1);     % gera a mensagem em binário
        c=mod(m*G,2);           % codifica-a com a matriz
        p=sum(c);               % obtem seu peso
        ixp=find(idx==p);       % obtem posição no vetor de índices
        ma(vidx(ixp),:)=c;      % insere no offset correspondente ao peso
        vidx(ixp)=vidx(ixp)+1;  % incrementa offset dessa faixa de peso
    end
end
```

Script 2: Agrupamento de palavras código por peso.

```
function dupP5 = AchaDupP5(ma)
% Conta todos os conjuntos de 7 zeros comuns a duas palavras de
% peso 5 para o código C(15,7,5).
% entradas:
%  ma    - matriz das palavras código agrupadas por pesos em ordem
%          crescente - dimensões 2^k-2 x n.  Não inclui a palavra de
%          peso 0 nem a de peso n. Neste caso é a matriz específica
%          para o código C(15,7,5). Obtida com o script AgrupaPesos()
% saídas:
%  dupP5 - Quantidade de duplicatas encontradas
    p5=ma(1:18,:);                  % extrai as 18 palavras de peso 5
    dupP5=0;                        % inicializa a contagem em zero
    for i=1:18                      % varre todas as 18
        for j=i:18                  % contra as que ainda não varreu
            b=xor(p5(i,:),p5(j,:)); % realiza a soma
            if sum(b)==6            % se a soma tiver peso 6 então
                dupP5=dupP5+1;      % incrementa a contatagem pois há
            end                     % 7 zeros em comum nas duas palavras
        end                         % de peso 5
    end
end
```

Script 3: Determinação do número de palavras com 7 zeros em comum

```
function li = TestaLI(G,cols)
% Testa se o conjunto cols de colunas de G é LI
% entradas:
%   G    - matriz geradora de código de dimensões k x n
%   cols - especifica conjunto de colunas - dimensões 1 x k
% saídas:
%   li   - booleano indicando se o conjunto é ou não LI.

    [k n] = size(G);                % obtém dimensões de G
    li = false;                     % em princípio não é LI
    if ~isequal(size(cols),[1 k])   % verifica consistência dos dados
        fprintf('Dimensões de G e cols incompatíveis. Encerrando..\n');
        return
    end
    G1=G(:,cols);                   % extrai colunas de G segundo cols
    colmsk = zeros(k,1);            % máscara para pivôs já encontrados
    % ------------ loop para processar as k colunas -----------------
    for i=1:k                       % percorre as k colunas
        col=and(G1(:,i),~colmsk);   % mascara a coluna da vez
        if ~any(col)                % teste para ver se existe pelo
            return;                 % menos uma posição não nula
        end                         % para gerar próximo pivô
        j = find(col,1,'first');    % obtemos índice do próximo pivô
        e=eye(k);                   % monta a matriz elementar tipo III
        e(:,j)=G1(:,i);             % a partir da matriz identidade
        G1=mod(e*G1,2);             % multiplica para gerar pivô
        colmsk=or(colmsk,G1(:,i));  % atualiza a máscara com novo pivô
    end
    li=true;                        % sucesso! Conjunto é LI!
end
```

Script 4: Função para testar se um conjunto de colunas é LI

```
function [Gn Gr0]=GeraGneGr0(G,S)
% Obtém a Gnova e a matriz de permutações
% entradas:
%   G   - matriz geradora de código de dimensões k x n
%   S   - vetor dos índices das posições mais confiáveis - dim. 1 x k
% saídas:
%   Gn  - Matriz Gnova recodifica posições menos confiáveis
%   Gr0 - Matriz de permutações - extrai símbolos mais confiáveis

    [k n]=size(G);                          % obtém dimensões da matriz
    if ~isequal(size(S),[1 n])              % verifica consistência dos dados
        fprintf('Dimensões de G e S incompatíveis. Encerrando..\n');
        return
    end
    Gn=G;                                   % inicializa Gn = G
    Gr0=zeros(k,n);                         % inicializa Gr0 = tudo zero
    colmsk=zeros(k,1);                      % inicializa máscara vazia
    p=zeros(k);                             % inicializa matriz de permutações
    for i=1:n                               % percorre as colunas
        idx=S(i);                           % obtém índice da vez
        colx=Gn(:,idx);                     % obtém coluna da vez
        col=and(colx,~colmsk);              % mascara a coluna da vez
        if ~any(col)                        % testa se haverá pivô
            continue;                       % se não houver vai para a
        end                                 % próxima coluna mais confiável
        j = find(col,1,'first');            % obtém índice do próximo pivô
        e=eye(k);                           % prepara matriz elementar III
        e(:,j)=colx;                        % monta matriz elementar III
        Gn=mod(e*Gn,2);                     % executa operação elem. III
        Gr0(:,idx)=Gn(:,idx);               % atualiza Gr0
        colmsk=or(colmsk,Gn(:,idx));        % atualiza a máscara
        s1=sum(colmsk);                     % calcula quantos pivôs já obteve
        p(:,s1)=Gn(:,idx);                  % atualiza matriz permutações
        if s1==k                            % testa se já obteve k pivôs
            break;                          % se já obteve sai do loop
        end                                 % para encerrar.
    end
    Gr0=p'*Gr0;                             % permuta as linhas de Gr0
    Gn=p'*Gn;                               % e também as de Gn
end
```

Script 5: Determinação das matrizes Gn e Gr0.

```
function [p_col p_acum ic_max]=ProbTent(G)
% Obtém a frequencia relativa de exame de colunas da matriz G
% para obter um CI a partir de um arranjo qquer de colunas
% entradas:
%   G   - matriz geradora de código de dimensões k x n
% saídas:
%   p_col  - probabilidade de se obter um CI com i colunas
%   p_acum - probabilidade acumulada de se obter um CI
%   ic_max - quantidade máxima de colunas processadas (pior caso)

    [k n]=size(G);                          % obtém dimensões da matriz
    p_col = zeros(1,n);                     % inicializa probab. colunas
    p_acum = zeros(1,n);                    % inicializa probab. acumul.
    ic_max = 0;                             % inicializa max col process.
    for cnt = 1:10^5                        % repete muitas vezes
        S = randperm(n);                    % obtem uma permutação aleatória
        Gn=G;                               % inicializa Gn = G
        colmsk=zeros(k,1);                  % inicializa máscara vazia
        for ic = 1:n                        % percorre até todas as colunas
            idx=S(ic);                      % obtém índice da vez
            colx=Gn(:,idx);                 % obtém coluna da vez
            col=and(colx,~colmsk);          % mascara a coluna da vez
            if ~any(col)                    % testa se haverá pivô
                ic=ic+1;                    % se não houver vai para a
                continue;                   % próxima coluna indexada
            end
            j = find(col,1,'first');        % obtém índice do próximo pivô
            e=eye(k);                       % prepara matriz elementar III
            e(:,j)=colx;                    % monta matriz elementar III
            Gn=mod(e*Gn,2);                 % executa operação elem. III
            colmsk=or(colmsk,Gn(:,idx));% atualiza a máscara
            if sum(colmsk)==k               % testa se já obteve k pivôs
                p_col(ic)=p_col(ic)+1;      % acumula nr. de obtenções
                if ic > ic_max              % armazena maior índice de
                    ic_max=ic;              % coluna percorrido até
                end                         % este ponto
                break;                      % com ic colunas e encerra.
            end
        end
    end                                     % fim do loop de repetição
    for i=k:n                               % acumula os valores
        p_acum(i)=p_col(i)+p_acum(i-1); % encontrados para colunas
    end
    p_acum=100*p_acum/cnt;                  % faz ajuste para expressar
    p_col=100*p_col/cnt;                    % valores em porcentagem
end
```

Script 6: Probabilidade de obtenção de um CI em k ou mais colunas

```
function padr=GeraPadrLi(G,S)
% Obtém o padrão das primeiras k colunas LI na matriz G reordenada por S
% entradas:
%   G     - matriz geradora de código de dimensões k x n
%   S     - vetor dos índices das posições mais confiáveis - dim. 1 x k
% saídas:
%   padr  - vetor de padrões com as k primeiras colunas LI encontradas

    [k n]=size(G);                          % obtém dimensões da matriz
    if ~isequal(size(S),[1 n])              % verifica consistência dos dados
        fprintf('Dimensões de G e S incompatíveis. Encerrando..\n');
        return
    end
    Gn=G;                                   % inicializa Gn = G
    padr = zeros(1,n);                      % inicializa padrão = tudo zero
    colmsk=zeros(k,1);                      % inicializa máscara vazia
    for i=1:n                               % percorre as colunas
        idx=S(i);                           % obtém índice da vez
        colx=Gn(:,idx);                     % obtém coluna da vez
        col=and(colx,~colmsk);              % mascara a coluna da vez
        if ~any(col)                        % testa se haverá pivô
            i=i+1;                          % se não houver vai para a
            continue;                       % próxima coluna mais confiável
        end
        j = find(col,1,'first');            % obtém índice do próximo pivô
        e=eye(k);                           % prepara matriz elementar III
        e(:,j)=colx;                        % monta matriz elementar III
        Gn=mod(e*Gn,2);                     % executa operação elem. III
        padr(1,i)=1;                        % atualiza o padrão encontrado
        colmsk=or(colmsk,Gn(:,idx));        % atualiza a máscara
        if sum(colmsk)==k                   % testa se já obteve k pivôs
            break;                          % se já obteve sai do loop
        end                                 % para encerrar.
    end
end
```

Script 7: Obtenção de padrões de colunas LI.

```
function [vm_s v_prob] = GeraProbPadr(G,dhmin,dhminT)
% Obtém os padrões de colunas LI e suas taxas de incidência
% entradas:
%   G      - matriz geradora de código de dimensões k x n
%   dhmin  - distância mínima de Hamming do código
%   dhminT - distância mínima de Hamming do código dual
% saídas:
%   vm_s   - matriz de padrões de colunas LI entre as posições dhminT e
%            n - dhmin +1 encontrados, ordenados por ordem decrescente de
%            probabilidade de incidência
%   v_prob - vetor com as taxas de incidência de cada um dos padrões
%            contidos em vm_s.

    [k n]=size(G);                          % obtém dimensões da matriz
    count = 10^5;                           % especifica quant. de iterações
    col_min=dhminT;                         % posição LD mínima
    col_max=n-dhmin+1;                      % posição LD máxima
    nr_col_max=col_max-col_min+1;           % qtde de bits dos padrões
    rand('state',0);                        % inicializa estado do gerador
    randn('state',0);                       % pseudo-aleatório sempre igual
    v_posic=zeros(1,count);                 % conterá assinaturas de cada
                                            % padrão de bits encontrado
    v=(2.^(0:nr_col_max-1))';               % pesos binários para assinaturas
    for i=1:count                           % processa count permutações
        S=randperm(n);                      % gera uma permutação aleatória
        posic = GeraPadrLi(G,S);            % obtém o padrão de colunas LI
        v_posic(i)=...                      % obtém a assinatura do padrão
            posic(col_min:col_max)*v;       % de colunas LI encontrado
        if mod(i,10000)==0                  % emite uma mensagem a cada
            disp(i);                        % 10.000 iterações para permitir
        end                                 % um acompanhamento do processo
    end
    v_posic_unique=...                      % obtém lista com todas as
        uint64(unique(v_posic));            % assinaturas diferentes achadas
    pos_max=size(v_posic_unique,2);         % conta quantas são ao todo
    vm=zeros(pos_max,nr_col_max);           % inicializa matriz para todos os
                                            % padrões de bits encontrados
    v_cnt=zeros(pos_max,1);                 % vetor para contagem de qtde de
                                            % ocorrências de cada padrão
    for i=1:pos_max                         % para cada assinatura encontrada:
        vm(i,:)=...                         % armazena o correspondente padrão
            bitget(v_posic_unique(i),...% de bits
                   1:nr_col_max);
        v_cnt(i)=sum(...                    % e também a sua quantidade de
            v_posic==v_posic_unique(i));% ocorrências
    end
    [v_cnt_s idx]=sort(v_cnt,'descend');% ordena por número de ocorrências
    vm_s=vm(idx,:);                         % reordena igualmente os padrões
    v_prob=v_cnt_s./count;                  % calcula a taxa de incidência
end
```

Script 8: Obtenção dos possíveis padrões de colunas LI e suas taxas de incidência.

```
function [mte_s prob_mte_s prob_mte_cdf]=...
                                SimulPrb(k,n,dhmin,dhminT,vm_s,v_prob)
% Encontra os padrões mais prováveis de erros e suas probabilidades
% entradas:
%   k      - quantidade de símbolos da mensagem
%   n      - quantidade de símbolos da palavra código
%   dhmin  - distância mínima de Hamming do código
%   dhminT - distância mínima de Hamming do código dual
%   vm_s   - matriz com padrões de distribuição de colunas LI
%   v_prob - vetor com as probabilidades de incidência dos padrões de vm_s
% saídas:
%   mte_s        - matriz com possíveis padrões de erros, ordenados por
%                  ordem decrescente de probabilidade de ocorrência
%   prob_mte_s   - vetor com as probabilidades de ocorrência de cada linha
%                  da matriz mte_s
%   prob_mte_cdf - vetor com as probabilidades acumuladas das linhas da
%                  matriz mte_s

    nr_comb = size(v_prob,1);            % obtém a qtde de combinações
    vms_e=[ones(nr_comb,dhminT-1) vm_s];% estende vm_s a n-dhmin+1 colunas
    vms_e_idx=zeros(nr_comb,k);          % gera matriz com índices de vms_e
    for i=1:nr_comb                      % obtem os índices das posições
        vms_e_idx(i,:)=find(vms_e(i,:));% não nulas de cada linha da ma-
    end                                  % triz vms_e (= nr_comb x k)

    ebno=1;                              % determinação do desvio padrão a
    R=k/n;                               % utilizar na geração do ruído
    sr = 10.^(ebno./10);                 % gaussiano a superpor em cada
    sigma = sqrt(1./(2*R.*sr));          % símbolo

    randn('state',37);                   % inicializa gerador pseudo-aleat.
    mte=MontaApagamentos(k,3);           % obtem a matriz de apagamentos

    v=2.^((0:k-1)');                     % obtem vetor de assinaturas das
    va= mte*v;                           % linhas da mte para comparações
    prob_mte=zeros(nr_comb,size(va,1)); % inicializa matriz probabilidades
                                         % 1 linha por padrão de colunas LI
                                         % 1 coluna por padrão erros da mte
    count = 10000;                       % executa lotes de 10000 por vez
    n_rod = 100;                         % faz 100 lotes ao todo
                                         % informa qtas rodadas e tamanho
                                         % de cada lote por rodada:
    fprintf('Iniciando %3d rodadas de %8d palavras cada\n',n_rod,count);
    for r=1:n_rod                        % loop com número de lotes
        tic;                             % avalia tempo de execução do lote
        padr=GeraPadrErros(count,...     % chama a função que irá gerar o
                           n,...         % lote de count palavras-código
                           sigma,...     % e retornar os padrões de erro
                           dhmin);       % detectados
        max_unique=0;                    % inicializa qtde padrões encontr.
                                         % continua na próxima página ...
```

Script 9: Obtenção dos padrões de erro mais prováveis e suas probabilidades.

```
        % continuação da página anterior....
        % -------------- loop para todas combinações de colunas LI: -----
        for j=1:nr_comb                          % para cada combinação
            padr_e=padr(:,vms_e_idx(j,:))*v;     % encontra as assinaturas
            [padr_e_u,ia,ib]=...                 % e faz a
                intersect(padr_e,va);            % intersecção com va
            size_padr_e_u=size(padr_e_u,1);      % descobre quantas são:
            for i=1:size_padr_e_u                % para cada uma
                prob_mte(j,ib(i))=...            % acumula as
                    prob_mte(j,ib(i))+...        % ocorrências em
                    sum(padr_e==padr_e_u(i));    % comum
            end
            if size_padr_e_u>max_unique          % armazena a maior qtde
                max_unique=size_padr_e_u;        % encontrada até aqui
            end
        end                                      % fim do loop
        % ------- fim do loop para todas as combinações de colunas LI
        fprintf(...                          % mostra resultados parciais
        'rodada %3d completa em %8.2f segundos - max_unique = %5d\n',...
        r,toc,max_unique);                   % e desempenhi da rodada
    end                                      % fim da rodada
    prob_mte=prob_mte./(n_rod*count);        % obtém a frequencia relativa
    prob_mte_pond=prob_mte'*v_prob;          % podera pela taxa de incidência
                                             % dos padrões de colunas LI
    [prob_mte_s idx]=...                     % ordena as probabilidades em
        sort(prob_mte_pond,'descend');       % ordem decrescente
    mte_s=mte(idx(:,1),:);                   % e também a matriz mte pelo mesmo
                                             % critério
    prob_mte_cdf=zeros(1,size(va,1));        % determina as probabilidades acu-
    prob_mte_cdf(1)=prob_mte_s(1);           % muladas - inicializa 1. valor e
    for i=2:size(va,1)                       % acumula os demais
        prob_mte_cdf(i)=...                  % atual = soma de todos até ele
            prob_mte_cdf(i-1)+...
            prob_mte_s(i);
    end
end
```

Script 9: – Continuação.

```
function mte=MontaApagamentos(k, nivel)
% Monta a matriz com os padrões de apagamentos de 0 a nivel apagamentos
% entradas:
%   k     - número de símbolos da mensagem
%   nível - até quantos erros simultâneos a considerar por linhas
% saídas:
%   mte  - matriz com possíveis padrões de erros, desde zero erros até
%          nível erros por linha

    linhas=1;                           % no mínimo uma linha (zero erros)
    for i=1:nivel                       % loop para determinar quantas
        linhas=linhas+nchoosek(k,i);    % linhas serão ao todo para
    end                                 % até nível erros por linha
    mte=zeros(linhas,k);                % inicializa a matriz toda zerada
    offset=1;                           % inicializa offset de faixa nivel
    for i=1:nivel                       % loop para cada nível
        vc=nchoosek(1:k,i);             % gera as combinações do nivel
        for j=1:size(vc,1)              % coloca cada uma em uma linha da
            mte(j+offset,vc(j,:))=1;    % da matriz, a partir de cada
        end                             % offset de faixa de nivel
        offset=offset+j;                % atualiza offset para próxima
    end                                 % faixa de nível
end
```

Script 10: Montagem da matriz de padrões de erros (não ordenada).

```
function m_padr=GeraPadrErros(count,n,sigma,dhmin)
% Obtém 'count' padrões de erros nas n-dhmin+1 posições das palavras-
código
% entradas:
%   count  - quantidade de padrões a gerar
%   n      - coprimento original das palavras-código
%   sigma  - desvio padrão do ruído gaussiano superposto
%   dhmin  - distância mínima de Hamming do código
% saídas:
%   m_padr - matriz count x n-dhmin+1 contendo os padrões de erros
%            contidos nos primeiros n-dhmin+1 símbolos de count palavras.
%            se o símbolo estiver em erro a coluna conterá 1, caso contrá-
%            rio conterá zero.
% obs: Como no caso interessa apenas a probabilidade de erro, todos os
%      símbolos são gerados com valor +1 e um erro é anotado se o ruído
%      superposto tornar o símbolo negativo.
%      Após a superposição do ruído gaussiano os símbolos de todas as
%      palavras são reordenados por ordem descrescente de confiabilidade.

    recebx = normrnd(1,sigma,count,n);  % gera matriz count x n símbolos
    recebx_abs=abs(recebx);             % obtem valor absoluto dos símb.
    [recebx_abs_s idx]=...              % e ordena em ordem descendente
        sort(recebx_abs,2,'descend');   % ao longo das linhas
    clear recebx_abs;                   % só os índices da ordenação são
    clear recebx_abs_s;                 % necessários - libera a memória
                                        % artifício p/ evitar loop for():
    m_offset=(0:n:n*count-n)'*ones(1,n);% ajusta offsets dos índices para
    idx2=idx+m_offset;                  % aplicá-los à matriz original
    clear idx;                          % libera memória da idx original
    clear m_offset;                     % e tambem dos offsets
    recebx_t = recebx';                 % usa matriz original transposta
    clear recebx;                       % pois matlab opera sobre colunas
    recebx_s=recebx_t(idx2);            % aplica índices já com offsets
    clear idx2;                         % libera memória da matriz origi-
    clear recebx_t;                     % nal e dos índices de ordenação
                                        % somente as primeiras n-dhmin+1
    recebx_s(:,n-dhmin+2:end)=[];       % posições serão retornadas
    m_padr=recebx_s<=0;                 % valores <=0 -> erro (=1)
end
```

Script 11: Gera aleatoriamente um lote de 'count' padrões de erro para análise.

```
% Script para simular a diferença entre a taxa de erros obtida por
% decodificação por decisão suave por meio de conjuntos de informação e
% decodificação por decisão suave por máxima verossimilhança (MLD) como
% função da quantidade de candidatas utilizadas.
%=========================================================================
% Estratégia:
%   Simula-se a passagem de counter_max palavras código moduladas em bpsk
%   por size_ebno canais submetidos cada um a uma taxa de ruído AWGN
%   diferente.  Cada palavra código recebida em cada canal é então
%   decodificada por meio de conjuntos de informação de ptos maneiras
%   diferentes, cada maneira utilizando uma quantidade diferente de
%   palavras código candidatas, selecionadas dentre as mais prováveis.  Em
%   paralelo é feita também a decodificação por MLD e todas as taxas de
%   erros anotadas.  As diferenças entre as taxas obtidas com os vários
%   conjuntos de candidatas e o caso do MLD são então determinadas.
%   O script permite assim estimar o número ideal de candidatas a se
%   utilizar em função da diferença, ou degradação de desempenho, tolera-
%   dos em relação ao desempenho da decodificação por MLD.
% Obs:
%   Para códigos mais longos, a partir do C(48,24,12), a determinação da
%   taxa de erros por decodificação por MLD não é viável com este script.
%   Nesse caso é feita uma estimativa, aproximando-se a taxa para o valor
%   obtido na decodificação por conjuntos de informação com uma quantidade
%   suficientemente grande de candidatas tal que a diferença obtida
%   adicionando-se mais candidatas fica abaixo de um determinado limiar.
%=========================================================================
% Utiliza as seguintes funções:
%   Gera_Pct_MPcr()    - Gera a palavra-código transmitida e a matriz de
%                        palavras-código recebidas em cada canal
%   decode_cji_m()     - Realiza a decodificação por decisão suave por
%                        conjuntos de informação das palavras código rece-
%                        bidas em cada canal para ptos cnjts de candidatas
%                        Retorna os erros constatados nas posições dos
%                        símbolos da mensagem.
%   decode_cji_m_mld() - Realiza a decodificação por MLD das palavras
%                        código recebidas em cada canal.  Retorna os erros
%                        constatados nas posições dos símbolos da mensagem
%   GeraCW()           - Gera todas as 2^k palavras do código, já modula-
%                        das em bpsk
%-------------------------------------------------------------------------
%limpar variáveis e command window
close all;
clear all;
clc;
%-------------------------------------------------------------------------
```

Script 12: Degradação do ganho de codificação em relação ao MLD – 1a Parte

```
%% Declaração e inicialização de variáveis globais e parâmetros:
global sigma;                           % desvio padrão - função de Eb/No
global G;                               % matriz geradora do código
global k;                               % comprimento da mensagem
global n;                               % comprimento do código
global m_apag;                          % matriz de apagamentos
global ptos;                            % quantos conjuntos de candidatas
global Ones_1xptos;                     % vetor tudo um de dim. 1 x ptos
global msk;                             % máscara filtro de cnj candidat.
global ebno_max;                        % até qual Eb/No processar
global Ones_sebnox1;                    % vetor tudo um size_ebno x 1
global ones_m_apag;                     % vetor tudo um
global Ones_1xn;                        % vetor tudo um 1 x n
global LI_MAX;                          % = n - dhmin + 1
global I_mask_mld;                      % máscara filtro mensagem para mld
global I_mask_cji;                      % máscara filtro mesagem para cji
global CW;                              % matriz das 2^k palavras em bpsk
%-------------------------------------------------------------------------
% Selecionar um dos códigos abaixo:
% em cada caso selecionar nr. de linhas de m_apag e quantidade de ptos
% a amostrar.

%    load G74.mat -ascii;
%    G=G74;
%    clear G74;
%    dhmin = 3;
%    load Prob_7_4_3_106.mat mte_s
%    m_apag = mte_s(1:15,:);
%    do_mld = true;
%    ptos =  15;

%    load G157.mat;
%    dhmin = 5;
%    load Prob_15_7_5_106.mat mte_s
%    m_apag = mte_s(1:30,:);
%    clear mte_s;
%    do_mld = true;
%    ptos =  10;


   load G24.mat -ascii;
   G=G24;
   clear G24;
   dhmin = 8;
   load Prob_24_12_8_106.mat mte_s
   m_apag = mte_s(1:80,:);
   do_mld = true;
   ptos = 20;
```

Script 12: Degradação do ganho de codificação em relação ao MLD – 2a Parte

```
%       load -ascii G48.mat;
%       G=G48;
%       clear G48;
%       dhmin = 12;
%       load Prob_48_24_12_106.mat mte_s
%       m_apag = mte_s(1:500,:);
%       clear mte_s;
%       do_mld = false;
%       ptos = 25;

%       load -ascii G663312.mat;
%       G=G663312;
%       clear G663312;
%       dhmin = 12;
%       load Prob_66_33_12_106.mat mte_s
%       m_apag = mte_s(1:1200,:);
%       clear mte_s;
%       do_mld = false;
%       ptos = 30;

[k n] = size(G);                          % dimensões do código
I_mask = ~(sum(G,1)-1);                   % de G (sistemática direita/esq)
I_mask_cji=ones(ptos,1)*I_mask;
LI_MAX=n-dhmin+1;
Ones_1xn=ones(1,n);
Ones_1xptos=ones(1,ptos);


Lt = size(m_apag,1);      % anotamos qtas linhas temos agora ao todo
ones_m_apag=ones(Lt,1);
delta = Lt/ptos;
vvidx=delta:delta:Lt;

%vamos criar a máscara que o CodingLoss_decoder utilizará para achar os
%máximos da matriz para cada delta:
% inicialmente criamos uma matriz de LtxLt um's, uppertriangular:
m = triu(ones(Lt));

% agora criamos um vetor de índices de matriz para extrair só as colunas
% de m que nos interessam, de delta em delta:
msk=m(:,vvidx);
% não precisamos mais da matriz quadrada m
clear m;




if do_mld
    tic;
    CW=GeraCW(G);
    fprintf('GeraCW() executou  em %8.2f segundos\n\n',toc);
end
%--------------------------------------------------------------------------
```

Script 12: Degradação do ganho de codificação em relação ao MLD – 3a Parte

```
%% Especificação dos níveis de ruído e quantidade de iterações:

ebno=1:4;                               % níveis de ruído
ebno_max=max(ebno);                     % até qual nível processar
Ones_sebnox1=ones(size(ebno,2),1);
I_mask_mld=Ones_sebnox1*I_mask;         % máscara para filtrar pos. mensagem
counter_max = 10^5;                     % Quantidade de iterações
erros_cji = zeros(ebno_max,ptos);       % Contadores de erros cnj inf.
erros_mld = zeros(1,ebno_max);          % Contadores de erros mld

%-------------------------------------------------------------------------
%% Determinação dos desvios para todos os ebnos:
% O desvio padrão é sempre o mesmo para cada ebno e para cada bit. Depois,
% Gera_Pct_MPcr, criaremos um vetor aleatório 1 x n, com média zero e
% desvio padrão = 1, e multiplicaremos pela nossa matriz de desvios, para
% obter os desvios específicos de cada bit para cada Eb/No.

R=k/n;
sr = 10.^(ebno./10);
% A matriz de desvios sigma tem dimensão size_of_ebno x n.  Todas suas n
% colunas são idênticas:
sigma = (sqrt(1./(2*R*sr)))'*Ones_1xn;
%-------------------------------------------------------------------------
%% Processamento para counter_max iterações:

rand('state',0);                        % inicializa o estado do gerador
randn('state',0);                       % pseudo-aleatório sempre igual

tic;                                    % anota início para avaliar tempo
for counter=1:counter_max;              % loop para counter_max iterações

    [pct m_pcr] = Gera_Pct_MPcr();  % m_pcr contém uma palavra recebida
                                        % para cada canal Eb/No

    if do_mld                           % Se a determinação mld for viável:
        erros_mld = erros_mld+decode_cji_m_mld(m_pcr,pct);
    end

    erros_cji = erros_cji+decode_cji_m(m_pcr,pct);
    if mod(counter,1000)==0
        fprintf('%9d de %d iterações completadas em %6.1f segundos\n',...
                counter,counter_max,toc);
    end
end %end for counter
fprintf('Processamos em %8.1f segundos para %8d iterações\n',toc,counter);
%-------------------------------------------------------------------------
```

Script 12: Degradação do ganho de codificação em relação ao MLD – 4a Parte

```
%% Determinação das taxas de erros
tx_cji     = erros_cji/(counter*k); % taxa para cnj informação

if do_mld
    tx_mld = erros_mld/(counter*k); % taxa real de para mld
else
    tx_mld = tx_cji(:,ptos)';       % taxa estimada para mld
end
dif_mld=tx_cji-tx_mld'*Ones_1xptos; % diferenças cnj inf x mld
%-------------------------------------------------------------------------
%% Salvamento dos resultados em disco:
% Criação de nome único para salvar os dados desta simulação:
nome =  sprintf('C(%02d,%02d,%02d)-10%d-%s.mat',...
        n,k,dhmin,log10(counter_max),datestr(now,'dd_mm_yy-HH_MM_SS'));
% e agora salvamos os dados:
save (nome,'tx_mld','tx_cji','dif_mld','vvidx');
%-------------------------------------------------------------------------
%% Apresentação dos dados na tela:
% Identificação do código e quantidade de iterações usadas:
fprintf('\nSimulação para código C(%d,%d,%d):\n\n',n,k,dhmin);
fprintf('iterações        = %8d\n',counter_max);
%-------------------------------------------------------------------------
% Se feita a decodificação por MLD mostra os resultados:
if do_mld
    fprintf('\nerros mld        =');
    for i=1:ebno_max
        fprintf(' % 8d',erros_mld(i));
    end
    fprintf('\n');
    fprintf('taxa mld         =');
    for i=1:ebno_max
        fprintf(' %1.6f',tx_mld(i)*100);
    end
    fprintf(' %%\n');
end
fprintf('\n');
%-------------------------------------------------------------------------
```

Script 12: Degradação do ganho de codificação em relação ao MLD – 5a Parte

```
% Apresentação de resultados da decodificação por conjuntos de informação:
fprintf('Qtd de candidatas:');
for i=1:ptos
    fprintf(' % 9d',vvidx(i));
end
fprintf('\n\n');

for i=1:ebno_max
    fprintf('erro cji(ebno=%d) =',i);
    for j=1:ptos
        fprintf(' %9d',erros_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('taxa cji(ebno=%d) =',i);
    for j=1:ptos
        fprintf(' % 1.6f',tx_cji(i,j)*100);
    end
    fprintf(' %%\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('dif mld(ebno=%d)  =',i);
    for j=1:ptos
        fprintf(' % 1.6f',dif_mld(i,j)*100);
    end
    fprintf(' %%\n');
end
fprintf('\n');
```

Script 12: Degradação do ganho de codificação em relação ao MLD – 6a Parte

```
function [pct m_pcr] = Gera_Pct_MPcr()
% Simula a palavra-código trasmitida e a recebida em size_ebno canais
% entradas:
%   utiliza as variáveis globais para aumentar a eficiência:
%   sigma - desvios padrão função da relação sinal ruido Eb/No
%   G     - matriz geradora do código
%   k     - comprimento da mensagem
%   n     - comprimento do código
% saídas:
%   pct   - Palavra códig transmitida
%   m_pcr - matriz com size_ebno palavras código recebidas moduladas em
%           bpsk com ruído superposto.

global sigma;                          % vetor com size_ebno desvios, um para
                                       % cada valor de ebno.
global G;                              % matriz geradora do código
global k;                              % comprimento da mensagem
global n;                              % comprimento do código
global Ones_sebnox1;                   % vetor size_ebno x 1 tudo um
global Ones_1xn;                       % vetor 1 x n tudo um

msg = round(rand(1,k));                % Gera mensagem aleatória de comp. k
pct = mod(msg*G,2);                    % Codifica palavra-código transmitida
transx =  Ones_sebnox1*(2*pct-1);      % Modula pct em BPSK e gera size_ebno
                                       % cópias idênticas
gauss=Ones_sebnox1*randn(1,n);         % size_ebno cópias com desvio = 1
gc=gauss.*sigma;                       % ajusta os desvios de cada cópia
m_pcr = transx + gc;                   % adiciona os desvios a cada pct
maxc=max(abs(m_pcr),[],2)*Ones_1xn;    % encontra máximo de cada linha e cria
                                       % size_ebno linhas de máximos
m_pcr = (1-eps)*m_pcr./maxc;           % normaliza todas as size_ebno linhas

end
```

Script 13: Simulação da passagem da palavra código por vários canais.

```
function erros_cji = decode_cji_m(m_pcr,pct)
% Decodifica as palavras código recebidas - determina os erros ocorridos
% entradas:
%   m_pcr   - matriz com size_ebno p. código recebidas com ruído gauss
%   pct     - palavra código transmitida (para avaliar erros)
%
%   utiliza ainda as variáveis globais para aumentar a eficiência:
%   G           - matriz geradora do código
%   n           - comprimento do código
%   m_apag      - matriz com os padrões de apagamentos
%   ones_m_apag - vetor coluna tudo um de dimensões ptos x 1
%   ebno_max    - quantos valores de Eb/No processar a partir de 1 dB
%   I_mask_cji  - máscara das colunas unitárias da matriz geradora G
%   LI_MAX      - = n-dhmin+1 = qtde máxima de colunas p/ encontrar um CI.
%   msk         - máscara para selecionar os conjuntos de candidatas
%   ptos        - qtde de pontos a examinar na sequência de candidatas
%   Ones_1xptos - vetor linha tudo um de dimensões 1 x ptos.
% saídas:
%   erros_cji   - matriz com size_ebno linhas e ptos colunas com os erros
%                 encontrados para os vários conjuntos de candidatas, para
%                 cada valor de Eb/No.
global G            n       m_apag  ones_m_apag     ebno_max;
global I_mask_cji   LI_MAX  msk     ptos            Ones_1xptos;
Sant = zeros(1,n);                  % vetor S anterior, inicialmente nulo
m_pcra = ceil(m_pcr);               % decodificação abrupta das m_pcr
m_pcr_abs=abs(m_pcr);               % obtemos os módulos das m_pcr
[Y S]=sort(m_pcr_abs,2,'descend');  % ordenamos símbolos por módulo
erros_cji=zeros(ebno_max,ptos);     % inicializamos matriz de resultados
for i=1:ebno_max                    % processamos cada ebno no loop for()
    if ~isequal(S(i,1:LI_MAX),Sant(1:LI_MAX)) % se a sequência ordenada
        [Gn Gr0]=GeraGneGr0(G,S(i,:));        % até LI_MAX colunas for a
        Sant = S(i,:);                        % mesma anterior usa as ma-
    end                                       % trizes Gn/Gr0 anteriores
    me = m_pcra(i,:)*Gr0';          % me é 1 x k (mensagem embaralhada)
    Me = ones_m_apag*me;            % Lt cópias idênticas de me
    ME=xor(Me,m_apag);              % Cria as candidatas alternativas
    mat_cand=mod(ME*Gn,2);          % Multiplica cada cand pela Gn
    mat_cand_psk=2*mat_cand-1;      % Modula as candidatas em bpsk
    ct=mat_cand_psk*m_pcr(i,:)';    % ct= produtos internos das candidatas
                                    % com a palavra recebida no canal i
    mct = (ct*Ones_1xptos).*msk;    % cada coluna de mct tem os prod. int.
                                    % p/ as primeiras x candidatas
    [ignore, Imax]=max(mct);        % Imax contém os máximos de cada colu-
                                    % na de mct.  Indexa melhor candidata
                                    % de cada conjunto (coluna)
    mat_infr = mat_cand(Imax,:);    % mat_infr contém a melhores candida-
                                    % tas de cada conjunto
    mat_vdif = and(xor(mat_infr,... % encontra as diferenças entre a
              Ones_1xptos'*pct),... % pct e as candidatas escolhidas entre
              I_mask_cji);          % os váios conjuntos.
    erros_cji(i,:)=sum(mat_vdif,2)';% acumula as diferenças e retorna
end                                 % fim do loop for
end
```

Script 14: Decodificação das palavras-código recebidas em cada canal.

```
function erros_mld = decode_cji_m_mld(m_pcr,pct)
% Decodifica as palavras código recebidas por mld - retorna erros encontr.
% entradas:
%   m_pcr   - matriz com size_ebno palavras código recebidas com ruído
%             gaussiano superposto
%   pct     - palavra código transmitida (para avaliar erros)
%
%   utiliza ainda as variáveis globais para aumentar a eficiência:
%
%   CW           - matriz com todas as 2^k palavras do código em bpsk
%   Ones_sebnox1 - vetor coluna tudo um de dimensões ptos x 1
%   I_mask_mld   - máscara das colunas unitárias da matriz geradora G
% saídas:
%   erros_cji    - matriz com size_ebno linhas e uma coluna, contendo a
%                  quantidade de erros de bit cometidos por cada canal nas
%                  posições referentes à mensagem.
%
% Faz uso da matriz CW contendo todas as 2^k possíveis palavras código já
% moduladas em bpsk, para comparar as m_pcr com cada uma e determinar a
% menor distância euclidiana.
%
% Apenas os erros nas posições referentes aos bits da mensagem são
% contabilizados.  Para isso faz uso da matriz I_mask_mld, de size_of_ebno
% linhas idênticas de n colunas, cada coluna contendo 1 (se a posição
% pertence à mensagem e deve ser contabilizada) ou 0 (caso contrário).

global I_mask_mld;
global CW;
global Ones_sebnox1;

    % Determinação de todos os 2^k produtos internos de cada m_pcr pelas
    % 2^k palavras do código (ct_mld é 2^k x size_of_ebno):
    ct_mld = CW*m_pcr';

    % determina os índices dos produtos internos máximos de cada coluna
    % de mct:
    [ignore Imax_mld] = max(ct_mld);

    % seleciona as candidatas vencedoras e as demodula para binário:
    infr_mld = ceil((1-eps)*CW(Imax_mld,:));

    % encontra as diferenças de cada candidata selecionada em relação à
    % pct, mas só nas posições da mensagem (filtradas com I_mask_mld):
    v_dif_mld = and(xor(infr_mld,Ones_sebnox1*pct),I_mask_mld);

    % acumula as diferenças encontradas e retorna:
    erros_mld=sum(v_dif_mld,2)';
end
```

Script 15: Decodificação por MLD das palavras-código recebidas nos canais

```
function cw = GeraCW(G)
% Gera todas as 2^k palavras do código moduladas em bpsk
% entradas:
%   G - matriz geradora do código
% saídas:
%   cw - matriz de 2^k x n com todas as 2^k possíveis palavras código
%        já moduladas em bpsk

    k =size(G,1);                    % obtem o comprimento k da mensagem
    m=zeros(2^k,k);                  % prepara matriz para 2^k mensagens
    for i=0:2^k-1                    % gera todas as mensagens e as
        m(i+1,:)=bitget(i,k:-1:1);   % coloca na matriz das mensagens
    end
    cw=mod(m*G,2);                   % multiplica todas as mensagens pela
    cw = 2*cw-1;                     % matriz geradora e as modula em bpsk
end
```

Script 16: Geração da matriz com todas as palavras de um código.

```
function ddB = delta_dB(a,b,c,d)
% Determina a diferença média da relação sinal ruído
% entradas:
%   a - log da taxa de erro de bit para Eb/No = i dB's na curva para MLD
%   b - log da taxa de erro de bit para Eb/No = i+1 dB's na curva para MLD
%   c - log da taxa de erro de bit para Eb/No = i dB's na curva cnj inf.
%   d - log da taxa de erro de bit para Eb/No = i+1 dB's na curva cnj inf.
% saídas:
%   ddB   - Delta médio em dB's na faixa entre i e i+1 dB's
ddB=0.5*(((log10(c)-log10(a))+(log10(d)-log10(b)))/(log10(c)-log10(d)));
end
```

Script 17: Determinação da degradação do ganho de codificação.

```
function mat_deltas = Monta_deltas_dB(tx_mld,tx_cji)
% Monta a matriz dos deltas médios em dB entre o MLD e várias curvas cji.
% entradas:
%   tx_mld - vetor linha das taxas de erro de bit para size_ebno relações
%            de sinal ruído - tx_mld é 1 x size_ebno.
%   tx_cji - matriz das taxas de erro de bit para Lt diferentes conjuntos
%            de palavras candidatas - tx_cji é Lt x size_ebno.
% saídas:
%   mat_deltas - Delta médio em dB's entre a curva para MLD e as curvas
%                para os diversos conjuntos de palavras candidatas.
%                mat_deltas é (size_ebno - 1) x Lt
% chama:       - delta_dB() para os cálculos individuais
    linhas = size(tx_mld,2)-1;          % obtem size_ebno - 1
    colunas = size(tx_cji,2);           % obtem qtde de conjuntos de cand.
    mat_deltas=zeros(linhas,colunas);   % inicializa mat_deltas
    for col=1:colunas                   % faz para cada conj. de candidat.
        for lin = 1:linhas              % e para cada faixa de Eb/No
            mat_deltas(lin,col)=delta_dB(tx_mld(lin),...
                                         tx_mld(lin+1),...
                                         tx_cji(lin,col),...
                                         tx_cji(lin+1,col));
        end
    end
end
```

Script 18: Monta matriz dos deltas em dB's em relação ao MLD.

```
% Script para apresentar graficamente a degradação do ganho de codificação
% em relação ao MLD para códigos C(15,7,5), C(24,12,8), C(48,24,12) e
% C(66,33,12).  Utiliza as respectivas matrizes de delts de cada código
% geradas pelo script Monta_deltas_dB.m.  Utiliza também os respectivos
% vetores vvidx com as quantidades de candidatas de cada conjunto.
%=========================================================================
tit=sprintf('Degradação do Ganho de Codificação em relação ao MLD');
scrz=get(0,'ScreenSize');
fh=figure(11);
lw=1.5;
set(fh,'OuterPosition',scrz);
ll1=loglog(vvidx_66_33_12,mat_deltas_66_33_12,'LineWidth',lw);
set(ll1(1),'Marker','o','DisplayName','Ebno 1 a 2 dB');
set(ll1(2),'Marker','diamond','DisplayName','Ebno 2 a 3 dB');
set(ll1(3),'Marker','square','DisplayName','Ebno 3 a 4 dB');
legend(gca,'show');
grid;
hold on;
ll1=loglog(vvidx_48_24_12,mat_deltas_48_24_12,'LineWidth',lw);
set(ll1(1),'Marker','o','DisplayName','Ebno 1 a 2 dB');
set(ll1(2),'Marker','diamond','DisplayName','Ebno 2 a 3 dB');
set(ll1(3),'Marker','square','DisplayName','Ebno 3 a 4 dB');

ll1=loglog(vvidx_24_12_8,mat_deltas_24_12_8,'LineWidth',lw);
set(ll1(1),'Marker','o','DisplayName','Ebno 1 a 2 dB');
set(ll1(2),'Marker','diamond','DisplayName','Ebno 2 a 3 dB');
set(ll1(3),'Marker','square','DisplayName','Ebno 3 a 4 dB');

ll1=loglog(vvidx_15_7_5,mat_deltas_15_7_5,'LineWidth',lw);
set(ll1(1),'Marker','o','DisplayName','Ebno 1 a 2 dB');
set(ll1(2),'Marker','diamond','DisplayName','Ebno 2 a 3 dB');
set(ll1(3),'Marker','square','DisplayName','Ebno 3 a 4 dB');

title(tit,'FontSize',16);
xlabel('Quantidade de Candidatas','FontSize',16);
ylabel('Degradação de Ganho em dB''s','FontSize',16);
set(gca,'FontSize',16);
xlim([1 1500]);
ylim([0.001 1]);

str_iter='Simulações realizadas \newline com 10^6 iterações';
text(1.5,0.003,str_iter,'FontSize',16,'EdgeColor','k','BackGroundColor','w
');

text(3,0.02,'C(15,7,5','FontSize',16,'BackGroundColor','w');
text(30,0.01,'C(24,12,8','FontSize',16,'BackGroundColor','w');
text(40,0.04,'C(48,24,12','FontSize',16,'BackGroundColor','w');
text(400,0.12,'C(66,33,12','FontSize',16,'BackGroundColor','w');
```

Script 19: Gera gráfico com degradação de ganhos para vários códigos.

```
% Script para simular a redução de palavras examinadas devida a diversos
% critérios de parada: GMD, Cone, BGW e BGWG
%=========================================================================
% Estratégia - dados:
%       - counter = qtde de iterações ( = palavras recebidas a decodificar)
%       - q       = qtde de candidatas a examinar de cada vez
% temos:  - qtde total de candidatas a examinar sem nenhum critério de
%           parada:
%         - q_tot = q x counter
% obtendo:
%         - n_crit = quantidade de iterações em que o critério atuou e
%         - n_busc = quantidade total de candidatas examinadas quando o
%                    critério atuou
% pode-se determinar o total efetivo de candidatas examinadas através de:
%         - tot_efet = n_busc + (counter - n_crit) x q
% a redução porcentual será então:
%         - red_perc = (q_tot - tot_efet) / q_tot
%=========================================================================
% Utiliza as seguintes funções:
% Gera_Pct_Pcr()   - simula as palavras código transmitidas e recebidas
% decode_cji()     - realiza a decodificação e verificação dos critérios
% Gera_Cand()      - chamada por decode_cji para obter as candidatas
% Gera_Gn_Gr0_SI() - chamada por Gera_Cand() para codificar as candidatas
%=========================================================================
%limpar variáveis e command window
close all;
clear all;
clc;

%%
global sigma;                           % desvio padrão - função de Eb/No
global G;                               % matriz geradora do código
global k;                               % comprimento da mensagem
global n;                               % comprimento do código
global m_apag;                          % matriz de apagamentos
global ptos;                            % quantos conjuntos de candidatas

% ----------- Selecionar uma das 4 alternativas de códigos a seguir:
%    load G74.mat;
%    G=G74;
%    clear G74;
%    dhmin = 3;
%    load Prob_7_4_3_106.mat mte_s
%    m_apag = mte_s(1:15,:);
%    do_mld = true;
%    ptos =  15;
```

Script 20: Parte 1 de 7 – Comparação do desempenho de critérios de parada.

```
%     load G157.mat;
%     dhmin = 5;
%     load Prob_15_7_5_106.mat mte_s
%     m_apag = mte_s(1:30,:);
%     ptos =  10;

%     load -ascii G24.mat;
%     G=G24;
%     clear G24;
%     dhmin = 8;
%     load Prob_24_12_8_106.mat mte_s
%     m_apag = mte_s(1:120,:);
%     clear mte_s;
%     ptos = 10;


      load -ascii G48.mat;
      G=G48;
      clear G48;
      dhmin = 12;
      load Prob_48_24_12_106.mat mte_s
      m_apag = mte_s(1:500,:);
      clear mte_s;
      do_mld = false;
      ptos = 10;

Lt = size(m_apag,1);                        % qtas linhas ao todo em m_apag
global stat_vv;                             % estatística qtde candidatas

[k n] = size(G);                            % dimensões do código
global I_mask;                              % máscara para colunas unitárias
I_mask = ~(sum(G,1)-1);                     % de G (sistemática direita/esq)
global m_T1;                                % matriz das máscaras M1 (BGWG)
global m_T2;                                % matriz das máscaras M2 (BGWG)
m_T1 = [m_apag zeros(Lt,n-k)];              % inicialmente preenche com zeros
m_T2 = zeros(Lt,n);

for i=1:Lt                                  % processa uma por linha de m_apag
    resto=dhmin-sum(m_apag(i,:));
    m_T1(i,k+1:n)=[zeros(1,n-k-resto) ones(1,resto)];
    m_T2(i,k+1:n-resto)=ones(1,n-k-resto);
end

%%
global delta;                               % pontos de delta em delta
delta = Lt/ptos;                            % a cada quantas candidatas
vvidx=delta:delta:Lt;                       % índice de número de candidatas
stat_vv=zeros(ptos,Lt);                     % estatística por cj candidatas
ebno_max = 5;                               % enbo_max (quantos loops faremos)
counter_max =[10^2 10^2 10^2 10^2 10^2];% quantas iterações por cada Eb/No
```

Script 20: Parte 2 de 7 – Comparação do desempenho de critérios de parada.

```
%contadores de erro:
erro_cji            = zeros(ebno_max,ptos);

% Quantidade total de iterações em que o limitante atuou:
bgw_cji             = zeros(ebno_max,ptos);
bgwg_cji            = zeros(ebno_max,ptos);
gmd_cji             = zeros(ebno_max,ptos);
con_cji             = zeros(ebno_max,ptos);

% Quantidade total de palavras candidatas examinadas nos casos em que
% o limitante atuou.
busca_cji           = zeros(ebno_max,ptos);
buscag_cji          = zeros(ebno_max,ptos);
busca_gmd_cji       = zeros(ebno_max,ptos);
busca_con_cji       = zeros(ebno_max,ptos);

% vetores com as taxas para cada ebno:

tx_cji              = zeros(ebno_max,ptos);  % taxa de erros de bit

tx_bgw_cji          = zeros(ebno_max,ptos);  % taxa de redução % bgw
tx_bgwg_cji         = zeros(ebno_max,ptos);  % taxa de redução % bgwg
tx_gmd_cji          = zeros(ebno_max,ptos);  % taxa de redução % gmd
tx_con_cji          = zeros(ebno_max,ptos);  % taxa de redução % cone
```

Script 20: Parte 3 de 7 – Comparação do desempenho de critérios de parada.

```
%%
%---------------- Loop para cada valor de Eb/No: ------------------------
for ebno = 1:ebno_max,

    sr = 10^(ebno/10);                  % determinação do desvio padrão
    sigma = (sqrt(1/(2*k*sr/n)));       % muda a cada Eb/No
    rand('state',0);                    % inicialização do gerador (pseudo)
    randn('state',0);                   % aleatório
    tic;                                % avaliação do tempo de processam.
    %--------------Loop para cada palavra código processada: -------------
    for counter=1:counter_max(ebno)     % qtde de iterações por Eb/No
        % ------ Geração da palavra código transmitida e recebida: -------
        [pct pcr] = Gera_Pct_Pcr();
        % ------- Decodificação e análise da atuação dos critérios: ------
        [buscas_gmd buscasg buscas  buscas_con ...
         gmd bgwg bgw con ...
         erros] = decode_cji(pcr,pct,dhmin);
        % ------- Acumulação dos resultados para cada Eb/No: -------------
        erro_cji(ebno,:)      = erro_cji(ebno,:)      + erros;

        bgw_cji(ebno,:)       = bgw_cji(ebno,:)       + bgw;
        bgwg_cji(ebno,:)      = bgwg_cji(ebno,:)      + bgwg;
        gmd_cji(ebno,:)       = gmd_cji(ebno,:)       + gmd;
        con_cji(ebno,:)       = con_cji(ebno,:)       + con;

        busca_cji(ebno,:)     = busca_cji(ebno,:)     + buscas;
        buscag_cji(ebno,:)    = buscag_cji(ebno,:)    + buscasg;
        busca_gmd_cji(ebno,:) = busca_gmd_cji(ebno,:) + buscas_gmd;
        busca_con_cji(ebno,:) = busca_con_cji(ebno,:) + buscas_con;

    end    % ---------- Fim do loop para cada palavra código.
    % ---------- Mostra tempo de processamento na tela -------------------
    fprintf(...
        'ebno = %d processou em %8.1f segundos para %8d iterações\n',...
         ebno,toc,counter);
    % --------- Cálculo das taxas de erros e de atuação dos critérios: ---
    tx_cji(ebno,:)      = erro_cji(ebno,:)  / (counter*k);

    pal_tot = counter.*vvidx;

    economia_bgw = busca_cji(ebno,:)+(counter-bgw_cji(ebno,:)).*vvidx;
    tx_bgw_cji(ebno,:)  = 1 - economia_bgw./pal_tot;

    economia_bgwg = buscag_cji(ebno,:)+(counter-bgwg_cji(ebno,:)).*vvidx;
    tx_bgwg_cji(ebno,:) = 1 - economia_bgwg./pal_tot;

    economia_gmd = busca_gmd_cji(ebno,:)+(counter-gmd_cji(ebno,:)).*vvidx;
    tx_gmd_cji(ebno,:)  = 1 - economia_gmd./pal_tot;

    economia_con = busca_con_cji(ebno,:)+(counter-con_cji(ebno,:)).*vvidx;
    tx_con_cji(ebno,:)  = 1 - economia_con./pal_tot;
end
%-------------------Fim do loop para cada valor de Eb/No ------------------
```

Script 20: Parte 4 de 7 – Comparação do desempenho de critérios de parada.

```
%% Apresentação dos resultados na janela de comandos:
fprintf('\nSimulação para código C(%d,%d,%d):\n\n',n,k,dhmin);
fprintf('iterações        =');
for i=1:ebno_max
    fprintf(' %8d',counter_max(i));
end
fprintf('\n');

fprintf('\n');

fprintf('Qtd de candidatas:');
for i=1:ptos
    fprintf(' %8d',vvidx(i));
end
fprintf('\n\n');

for i=1:ebno_max
    fprintf('erro cji(ebno=%d) =',i);
    for j=1:ptos
        fprintf(' %8d',erro_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('taxa cji(ebno=%d) =',i);
    for j=1:ptos
        fprintf(' %1.6f',tx_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('busc cji(ebno=%d) =',i);
    for j=1:ptos
        fprintf(' %8d',busca_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('buscg cji(ebno=%d)=',i);
    for j=1:ptos
        fprintf(' %8d',buscag_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');
```

Script 20: Parte 5 de 7 – Comparação do desempenho de critérios de parada.

```
for i=1:ebno_max
    fprintf('bsgmd cji(ebno=%d)=',i);
    for j=1:ptos
        fprintf(' %8d',busca_gmd_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('bgw cji(ebno=%d)  =',i);
    for j=1:ptos
        fprintf(' %8d',bgw_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('bgwg cji(ebno=%d) =',i);
    for j=1:ptos
        fprintf(' %8d',bgwg_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('gmd cji(ebno=%d)  =',i);
    for j=1:ptos
        fprintf(' %8d',gmd_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');
for i=1:ebno_max
    fprintf('con cji(ebno=%d)  =',i);
    for j=1:ptos
        fprintf(' %8d',con_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');
```

Script 20: Parte 6 de 7 – Comparação do desempenho de critérios de parada.

```
for i=1:ebno_max
    fprintf('red. bgw(ebno=%d) =',i);
    for j=1:ptos
        fprintf(' %8.2f',tx_bgw_cji(i,j)*100);
    end
    fprintf(' %%\n');
end
fprintf('\n');
for i=1:ebno_max
    fprintf('red. bgwg(ebno=%d)=',i);
    for j=1:ptos
        fprintf(' %8.2f',tx_bgwg_cji(i,j)*100);
    end
    fprintf(' %%\n');
end
fprintf('\n');
for i=1:ebno_max
    fprintf('red. gmd(ebno=%d) =',i);
    for j=1:ptos
        fprintf(' %8.2f',tx_gmd_cji(i,j)*100);
    end
    fprintf(' %%\n');
end
fprintf('\n');
for i=1:ebno_max
    fprintf('red. con(ebno=%d) =',i);
    for j=1:ptos
        fprintf(' %8.2f',tx_con_cji(i,j)*100);
    end
    fprintf(' %%\n');
end
fprintf('\n');
fprintf('\nNenhum erro de palavra a partir da candidata
%d\n',1+find(stat_vv(ptos,:),1,'last'));
```

Script 20: Parte 7 de 7 – Comparação do desempenho de critérios de parada.

```
function [pct pcr] = Gera_Pct_Pcr()
% Simula a palavra-código trasmitida e a recebida
% entradas:
%   utiliza as variáveis globais para aumentar a eficiência:
%   sigma - desvio padrão função da relação sinal ruido Eb/No
%   G     - matriz geradora do código
%   k     - comprimento da mensagem
% saídas:
%   pct - Palavra códig transmitida
%   pcr - Palavra código recebida com ruído superposto

global sigma;
global G;
global k;

msg = round(rand(1,k));              % Gera mensagem aleatória de comp. k
pct = mod(msg*G,2);                  % Codifica palavra-código transmitida
transx = 2*pct-1;                    % Modula pct em BPSK
pcr =  normrnd(transx,sigma);        % Superpõe ruído gaussiano
pcr = (1-eps)*pcr./max(abs(pcr));    % Normaliza entre -1/+1

return
```

Script 21: Simula as palavras-código transmitida e recebida.

```
function [buscas_gmd buscasg buscas buscas_con   ...
          gmd bgwg bgw con ...
          erros] = decode_cji(pcr,pct,dhmin)
% Decodifica a palavra código recebida - determina erros e taxas de parada
% entradas:
%   pcr     - palavra código recebida com ruído gaussiano superposto
%   pct     - palavra código transmitida (para avaliar erros)
%   dhmin   - distância mínima de Hamming do código
%
%   utiliza ainda as variáveis globais para aumentar a eficiência:
%
%   I_mask  - máscara das coluns unitárias da matriz geradora G
%   ptos    - qtde de pontos a examinar na sequência de candidatas
%   delta   - conjuntos de candidatas de delta em delta num total de ptos
%   m_T1    - matriz das máscara M1 para BGWG
%   m_T2    - matriz das máscaras M2 para BGWG
%   stat_vv - estatística a partir de qtas candidatas não há mais erros
%   n       - comprimento do código
% saídas:
%   buscas_xx - quantidade de candidatas examinadas quando o critério xx
%               atuou
%   xx        - = 1 se o critério xx atuou (xx = gmd, cone, bgw e bgwg)
%   erros     - quantidade de bits encontrados em erro na decodificaçao.

global I_mask;
global ptos;
global delta;
global m_T1;
global m_T2;
global stat_vv;
global n;

    [SI mat_cand] = Gera_Cand(pcr);      % obtem as cadidatas e SI
    mat_cand_psk = 2*mat_cand-1;         % modula as candidatas em BPSK
    erros = zeros(1,ptos);               % inicializa erros por ptos

    bgw     = zeros(1,ptos);             % flag critério bgw por ptos
    bgwg    = zeros(1,ptos);             % flag critério bgwg por ptos
    gmd     = zeros(1,ptos);             % flag critério gmd por ptos
    con     = zeros(1,ptos);             % flag critério cone por ptos

    buscas  = zeros(1,ptos);             % candidatas examinadas com bgw
    buscasg = zeros(1,ptos);             % cadidatas examinadas com bgwg
    buscas_gmd = zeros(1,ptos);          % candidatas examinadas com gmd
    buscas_con = zeros(1,ptos);          % cadidatas examinadas com cone

    ct=mat_cand_psk*pcr';                % produtos internos a maximizar
```

Script 22: Parte 1 de 2 – Realiza a decodificação por conjuntos de informação.

```
    for i=1:ptos                          % análise por pontos:
        [ignore Imax] = ...               % encontra o máximo produto interno
            max(ct(1:i*delta));           % = mínima distância euclidiana
        infr=mat_cand(Imax,:);            % obtem a candidata correspondente
        v_dif = ...                       % encontra as diferenças de bits
            and(xor(infr,pct),I_mask);    % dentre os da mensagem (I_mask)
        erros(i)=sum(v_dif);              % conta quantos erros de bit

        if erros(i)>0                     % se houve erros para essa
            stat_vv(i,Imax) =...          % quantidade de candidatas (ptos)
                stat_vv(i,Imax)+1;        % incrementa stat_vv para estatíst.
        end
        % Critérios:  --------------------conhecendo a melhor candidata,
        infr_bpsk = 2*infr-1;             % vejamos quais critérios teriam
                                          % parado nela:
        sh = -infr_bpsk.*pcr;             % determinação da soma híbrida
        % ---------- critério BGWG: -----------------------------------
        sh1=sh(1,SI);                     % bgwg: soma híbrida permutada
        if ~any(and(ceil(sh1),m_T2(Imax,:))) % Aplica máscara M2
            if sum(sh1.*m_T1(Imax,:))<0 % se passar aplica máscara M1
                bgwg(i) = 1;              % se passar seta flag bgwg
                buscasg(i) = Imax;        % anota quantas candidatas teria
            end                           % examinado
        end
        % ---------- critério Cone: -----------------------------------
        if sum(sh)<-sqrt(pcr*pcr'*(n-dhmin))
            con(i) = 1;                   % se passar seta flag Cone
            buscas_con(i) = Imax;         % anota quantas candidatas teria
        end                               % examinado
        % ---------- critério BGW:  -----------------------------------
        sh = sort(sh,2,'descend');        % necessita ordenação
        if sum(sh(1:dhmin))<0             % se passar
            bgw(i) = 1;                   % seta flag bgw
            buscas(i) = Imax;             % anota quantas candidatas teria
        end                               % examinado
        % ---------- critério GMD:  -----------------------------------
        if sum(sh)<(dhmin-n)              % se passar
            gmd(i) = 1;                   % seta flag gmd
            buscas_gmd(i) = Imax;         % anota quantas candidatas teria
        end                               % examinado
    end
end
```

Script 22: Parte 2 de 2 – Realiza a decodificação por conjuntos de informação.

```
function [SI mat_cand]= Gera_Cand(pcr)
% Gera a matriz de candidatas à decodificação
% entradas:
%   pcr      - palavra código recebida - modulada em BPSK
%   G        - variável global contendo a matriz geradora do código
%   m_apag   - matriz de apagamentos para o código
% saídas:
%   SI       - Vetor ordenação das confiabilidades com as primeiras k
%              posições LI - obtido através de chamada a Gera_Gn_Gr0_Si()
%   mat_cand - matriz com as cadidatas mais prováveis à decodificação de
%              pcr.  Obtida via implementação dos apagamentos de m_apag.

global G;
global m_apag;

pcra = ceil(pcr);                    % decodificação abrupta de pcr
pcr_abs = abs(pcr);                  % confiabilidades = módulo componentes
[Y S] = sort(pcr_abs,2,'descend');   % vetor S - ordenação de confiabilid.
[SI Gn Gr0] = Gera_Gn_Gr0_SI(G,S);   % Gera Gn, Gr0 e SI
me = pcra*Gr0';                      % obtem mensagem pré embaralhada me
Me = ones(size(m_apag,1),1)*me;      % gera matriz Me com Lt cópias de me
ME = xor(Me,m_apag);                 % aplica apagamentos cfe. m_apag
mat_cand = mod(ME*Gn,2);             % cada candidata = me x Gn

return
```

Script 23: Gera o conjunto de candidatas mais prováveis.

```
function [SI Gn Gr0]=Gera_Gn_Gr0_SI(G,S)
% Obtém a Gnova e a matriz de permutações
% entradas:
%   G   - matriz geradora de código de dimensões k x n
%   S   - vetor dos índices das posições mais confiáveis - dim. 1 x k
% saídas:
%   SI  - Vetor ordenação das confiabilidades com as primeiras k posições
%         LI
%   Gn  - Matriz Gnova recodifica posições menos confiáveis
%   Gr0 - Matriz de permutações - extrai símbolos mais confiáveis

    [k n]=size(G);                          % obtém dimensões da matriz
    if ~isequal(size(S),[1 n])              % verifica consistência dos dados
        fprintf('Dimensões de G e S incompatíveis. Encerrando..\n');
        return
    end
    Gn=G;                                   % inicializa Gn = G
    Gr0=zeros(k,n);                         % inicializa Gr0 = tudo zero
    colmsk=zeros(k,1);                      % inicializa máscara vazia
    SI = zeros(1,n);                        % inicializa SI vazio
    n_ld = 0;                               % qtde colunas LD inicialmente zero
    p=zeros(k);                             % inicializa matriz de permutações
    for i=1:n                               % percorre as colunas
        idx=S(i);                           % obtém índice da vez
        colx=Gn(:,idx);                     % obtém coluna da vez
        col=and(colx,~colmsk);              % mascara a coluna da vez
        if ~any(col)                        % testa se haverá pivô
            n_ld = n_ld + 1;                % se não houver incrementa n_ld,
            SI(k+n_ld)=idx;                 % muda o índice para após as k LD
            continue;                       % e vai para a
        end                                 % próxima coluna mais confiável
        j = find(col,1,'first');            % obtém índice do próximo pivô
        e=eye(k);                           % prepara matriz elementar III
        e(:,j)=colx;                        % monta matriz elementar III
        Gn=mod(e*Gn,2);                     % executa operação elem. III
        Gr0(:,idx)=Gn(:,idx);               % atualiza Gr0
        colmsk=or(colmsk,Gn(:,idx));        % atualiza a máscara
        scm=sum(colmsk);                    % conta quantos pivô já fez
        SI(scm)=idx;                        % guarda o índice em sua posição.
        p(:,scm)=Gn(:,idx);                 % atualiza matriz permutações
        if scm==k                           % testa se já obteve k pivôs
            SI(1,k+n_ld+1:end) =...         % se já obteve completa os índices
                    S(1,k+n_ld+1:end);      % do vetor SI iguais aos de S e
            break;                          % sai do loop
        end                                 % para encerrar.
    end
    Gr0=p'*Gr0;                             % permuta as linhas de Gr0
    Gn=p'*Gn;                               % e também as de Gn
end
```

Script 24: Gera as matrizes Gn, Gr0 e o vetor de ordenação de confiabilidades SI.

```
function [sigr sigex muex sign mun]=DistrMaximos(n,k)
% Determina a distribuição de valores extremos de ruído sobreposto a
partir
% das equações teóricas
% entradas:
%   n     - número de símbolos da palavra-código
%   k     - número de símbolos da mensagem
% saídas:
%   sigr  - vetor com desvios padrão aplicados para Eb/No de 1 a 5 dB
%   sigex - vetor com desvios padrão dos valores extremos
%   muex  - vetor com as médias dos valores extremos
%   sign  - vetor com os desvios padrão resultantes da normalização
%   muen  - vetor com os valores médios resultantes da normalização
    dx=0.001;                              % Intervalo de integração
    x=1:dx:5;                              % Domínio de análise / plotagem
    x1=1:0.25:5;                           % Domínio auxiliar para marcadores
    mrk=['+','o','d','s','p'];             % Lista de marcadores
    clr=['k','r','b','m','g'];             % Lista de cores
    h=zeros(1,5);                          % armazena handles para legendas
    ebno=1:5;                              % processa Eb/No de 1 a 5 dB
    sr=10.^(ebno/10);                      % relação sinal ruído
    sigr=sqrt(1./(2*k.*sr/n));             % desvio padrão real resultante
    sigex=zeros(1,5);                      % armazena desvio dos extremos
    muex=zeros(1,5);                       % armazena média dos extremos
    sign=zeros(1,5);                       % armazena desvio normalizado
    mun=zeros(1,5);                        % armazena sinal médio normaliz.
    for i=1:5                              % loop para cada valor de Eb/No
        dp=@(x)n*normpdf(x,1,sigr(i))...% função densidade de probabilid.
            .*normcdf(x,1,sigr(i))...      % dos valores máximos
            .^(n-1);
        dpm=@(x)x.*dp(x);                  % função para integrar a média
        dps=@(x)x.^2.*dp(x);               % função para integrar o desvio p.
        muex(i)=quad(dpm,1,6);             % média dos valores máximos
        sigex(i)=sqrt(quad(dps,1,6)...     % desvio padrão dos valores máx.
                          -muex(i)^2);
        mun(i)=1/muex(i);                  % média dos valores normalizados
        sign(i)=sigr(i)/muex(i);           % desvio padrão dos valores norm.
        plot(x,dp(x),'Color',clr(i),... % gera gráfico da densidade de
            'LineWidth',1.5);              % probabilidade resultante
        hold on;                           % continua no mesmo gráfico
        h(i)=plot(x1,dp(x1),mrk(i),...     % plota só os marcadores e salva
                'MarkerEdgeColor',...      % o handle de cada gráfico para
                clr(i),'MarkerSize',10);% implementar a legenda abaixo
    end                                    % fim do loop Eb/No
    grid;                                  % coloca grid no gráfico
    lg=@(x)sprintf(...                     % cria uma função que gerará as
     'Eb/No=%d,   \\mu_{ex}= %4.2f',... % legendas no gráfico contendo
      x,muex(x));                          % os valores médios extremos
    lh=legend(h,lg(1),lg(2),lg(3),...      % coloca legenda com marcadores
                           lg(4),lg(5));% e médias extremas no gráfico
    set(lh,'FontSize',14);                 % ajusta o tamanho da legenda
   sTit=sprintf('Distribuição de valores extremos para código C(%d,%d)'...
                 ,n,k);                    % gera o título do gráfico
    title(sTit,'fontsize',16);             % coloca o título no gráfico
    xlabel('Valor extremo','fontsize',14);             % legenda do eixo x
    ylabel('Densidade de probabilidade','fontsize',14);% legenda do eixo y
end
```

Script 25: Obtém a distribuição de valores máximos em amostras de n elementos.

```
function [sigr sigex muex sign mun]=HistMaximos(n,k)
% Determina a distribuição de valores extremos de ruído sobreposto a
partir
% de simulações
% entradas:
%   n     - número de símbolos da palavra-código
%   k     - número de símbolos da mensagem
% saídas:
%   sigr  - vetor com desvios padrão resultantes para Eb/No de 1 a 5 dB
%   sigex - vetor com desvios padrão dos valores extremos
%   muex  - vetor com as médias dos valores extremos
%   sign  - vetor com os desvios padrão resultantes da normalização
%   muen  - vetor com os valores médios resultantes da normalização
    ebno=1:5;                           % processa Eb/No de 1 a 5 dB
    sr=10.^(ebno/10);                   % relação sinal ruído
    sig=sqrt(1./(2*k.*sr/n));           % desvio padrão aplicado
    sigr=zeros(1,5);                    % armazena desvio resultante
    sigex=zeros(1,5);                   % armazena desvio dos extremos
    muex=zeros(1,5);                    % armazena média dos extremos
    sign=zeros(1,5);                    % armazena desvio normalizado
    mun=zeros(1,5);                     % armazena sinal médio normaliz.
    cnt=10^5;                           % quantas palavras-código simular
    maxs=zeros(5,cnt);                  % valores máximos de cada palavra
    for i=1:5                           % loop para cada valor de Eb/No
        pcs=zeros(cnt,n);               % matriz com as palavras código
        pcsn=zeros(cnt,n);              % matriz das palavras normalizadas
        for lin=1:cnt                   % para cada linha da matriz
            pcs(lin,:) =...             % gera a palavra aleatória com
                normrnd(1,sig(i),1,n);  % valores média 1 e desvio sig
            maxs(i,lin)=max(pcs(lin,:));% armazena o máximo desta palavra
            pcsn(lin,:) = pcs(lin,:)/...% normaliza a palavra pelo max abs
                max(abs(pcs(lin,:)));
        end                             % fim da geração de cnt palavras
        v=reshape(pcs,1,[]);            % enfileira todas as pcs em 1 lin.
        sigr(i) = std(v);               % acha desvio resultante p/ Eb/No
        v=reshape(pcsn,1,[]);           % agora todas as pcsn em 1 linha
        sign(i) = std(v);               % acha desvio das palavras normal.
        mun(i) = mean(v);               % acha média das pal. normalizadas
        sigex(i) = std(maxs(i,:));      % acha desvio dos extremos
        muex(i) = mean(maxs(i,:));      % acha média dos extremos
    end                                 % fim das simulações Eb/No 1 a 5
    [hs x] = hist(maxs',100);           % obtém todos os 5 histogramas
    n1=1:3:size(x,1);
    mrk=['+','o','d','s','p'];          % Lista de marcadores
    clr=['k','r','b','m','g'];          % Lista de cores
    h=zeros(1,5);                       % armazena handles para legendas
```

Script 26: Obtenção de histogramas de valores extremos – 1a Parte

```
    for i=1:5                               % agora os 5 gráficos:
        plot(x,hs(:,i),'Color',...          % gera gráfico da envoltória dos
            clr(i),'LineWidth',1.5);        % histogramas resultantes
        hold on;                            % continua no mesmo gráfico
        h(i)=plot(x(n1,:),hs(n1,i),...      % plota marcadores
            mrk(i),'MarkerEdgeColor',...
            clr(i),'MarkerSize',10);
    end
    grid;                                   % coloca grid no gráfico
    xlim([1 5]);                            % trunca gráfico em x = 5
    lg=@(x)sprintf(...                      % cria uma função que gerará as
     'Eb/No=%d,   \\mu_{ex}= %4.2f',...     % legendas no gráfico contendo
      x,muex(x));                           % os valores médios extremos
    lh=legend(h,lg(1),lg(2),lg(3),...       % coloca legenda com marcadores
                            lg(4),lg(5));% e médias extremas no gráfico
    set(lh,'FontSize',14);                  % ajusta o tamanho da legenda
    sTit=sprintf('Histograma de valores extremos para código C(%d,%d)'...
                  ,n,k);                    % gera o título do gráfico
    title(sTit,'fontsize',16);              % coloca o título no gráfico
    xlabel('Valor extremo',...              % coloca legenda do eixo x
           'fontsize',14);
    ylabel('Envoltória dos histogramas',...
           'fontsize',14);                  % coloca legenda do eixo y
end
```

Script 26: Obtenção de histogramas de valores extremos – 2a Parte

```
% Script para determinar a redução da relação sinal ruído Eb/No em  dB's
% causada pela discretização do sinal recebido, quantizado em n bits
%=========================================================================
clc;                                                % limpeza da janela cmd
close all;                                          % fechamento figuras
clear all;                                          % limpeza das variáveis
%=========================================================================
%% Determinação dos dados de entrada
% selecionar uma das opções de código:
% code='C(15,7,5)';
% sign =[0.3546 0.3387 0.3225 0.3061 0.2895 ];    % para código C(15,7,5)
code='C(24,12,8)';
sign =[0.3258 0.3119 0.2976 0.2831 0.2684 ];    % para código C(24,12,8)
% code='C(48,24,12)';
% sign =[0.2981 0.2864 0.2743 0.2619 0.2493 ];    % para código
C(48,24,12)
nbits=3:8;                                          % 3 a 8 bits de quantiz.
q=2./2.^nbits;                                      % intervalo de quantização
%=========================================================================
%% Cálculo da redução.
mq=q'*ones(1,5);                                    % 5 col. idênticas (=q')
msn=ones(6,1)*sign;                                 % 6 lin. idênticas(=sign)
alfa=mq./msn;                                       % cálculo fatores q/sign
fat=1+alfa.^2/12;                                   % fator multiplicativo
red_db=10*log10(fat);                               % cálculo da redução em dB
%=========================================================================
%% Apresentaçao dos resultados em forma gráfica
close all;
h=plot(nbits,red_db,'LineWidth',2.0);
set(h,{'Marker'},{'s','none','none','none','none'}');
set(h,{'LineStyle'},{'-','-.','--',':','-'}');
tit=sprintf(...
 'Redução da relação sinal ruído em função da quantização - código %s',...
   code);
title(tit,'fontsize',16);
legend('Eb/No = 1 dB','Eb/No = 2 dB','Eb/No = 3 dB','Eb/No = 4 dB',...
       'Eb/No = 5 dB');
grid;
set(gca,'YMinorGrid','on');
set(gca,'xtick',nbits);
set(gca,'fontsize',14);
xlabel('Quantidade de bits de quantização','fontsize',16);
ylabel('Redução de Eb/No (dB''s)','fontsize',16);
%=========================================================================
%% Apresentação dos dados em forma de tabela:
clc;
fprintf('Redução da relação Sinal / Ruído para Código %s\n\n',code);
fprintf('Eb/No =              ');
fprintf('1        2        3        4        5  dB''s\n\n');
for i=1:6
    fprintf('n bits = %d  => ',nbits(i));
    fprintf('%0.4f   %0.4f   %0.4f   %0.4f   %0.4f  dB''s\n',...
            red_db(i,1),red_db(i,2),red_db(i,3),...
            red_db(i,4),red_db(i,5));
end
```

Script 27: Redução da relação sinal / ruído devida à quantização.

```
function TesteNormal(alfa)
% Verifica a qualidade da aproximação para a norma da convolução de uma
% distribuição normal padronizada com uma distribuição uniforme de méda
% zero.
% entrada:
%   alfa  - relação entre a amplitude q da distribuição uniforme e o
%           desvio padrão da normal (=1 para normal padronizada).
% saídas:
%   nenhuma
    sigma=1;                              % desvio padrão da normal
    q=alfa*sigma;                         % amplitude da distr. uniforme
    dx=sigma/1000;                        % intervalo para cálculo
    x=-5*sigma:dx:5*sigma;                % análise entre -5 e + 5 sigma
    y=normpdf(x,0,sigma);                 % y é a distribuição normal
    A=-q/2;                               % extremos da distribuição
    B=+q/2;                               % uniforme
    y2=unifpdf(x,A,B);                    % distribuição uniforme
    y3=dx*conv(y,y2);                     % convolução normal * uniforme
    x3=-10*sigma:dx:10*sigma;             % convolução tem n+m-1 pontos
    seq=sqrt(sum(x3.^2.*y3)/sum(y3));     % cálculo sigma equiv. pratico
    seqt=sqrt(sigma^2+q^2/12);            % cálculo sigma equiv. teórico
    y4=normpdf(x3,0,seqt);                % normal com sigma equiv. teórico
    dd=abs(y3-y4);                        % diferença absoluta
%=========================================================================
% Apresentação dos resultados em forma gráfica:
    close all;                            % fecha figuras anteriores
    plot(x,y,'LineWidth',2.0);            % plota a normal
    grid
    hold on
    plot(x,y2,'--','LineWidth',2.0);      % plota a uniforme
    plot(x3,y3,'-.r','LineWidth',2.0);    % plota a convolução das duas
    plot(x3,y4,':g','LineWidth',2.0);     % plota a normal equivalente
    ylim([0 0.45]);
    xlim([-5 +5]);
    legend('f_X(x) = N(0,1)','f_Y(x) = U( -q/2 , +q/2 )',...
           'f_Z(x) = f_X(x)*f_Y(x)','N(0,\sigma_{eq})');
    xlabel('\sigma','FontSize',16);
    set(gca,'FontSize',14);
%=========================================================================
% Apresentação dos resultados na janela de comandos:
    clc;
    fprintf('seq = %f - seqt = %f - diferença = %f\n',seq,seqt,seq-seqt);
    fprintf('Máxima diferença absoluta = %f\n',max(dd));
    fprintf('Mínima diferença absoluta = %f\n',min(dd));
end
```

Script 28: Teste de normalidade da soma do sinal com ruído de quantização.

```
% Script para comparar o efeito da quantização da palavra código recebido
% com uma redução da relação de sinal ruído
%=========================================================================
% Utiliza as seguintes funções:
% Gera_Pct_Pcr()    - simula as palavras código transmitidas e recebidas
%                     retorna a palavra código recebida com e sem
%                     quantização.
% QuantizaPcr()     - gera uma versão quantizada da palavra código recebida
% decode_cji_eq()   - realiza a decodificação da pcr_eq
% decode_cjiq()     - realiza a decodificação da pcr e da pcr quantizada
% Gera_Cand()       - chamada por decode_cji/q para obter as candidatas
% Gera_Gn_Gr0_SI()  - chamada por Gera_Cand() para codificar as candidatas
%=========================================================================
%limpar variáveis e command window
close all;
clear all;
clc;
%---- Variáveis definidas globalmente para acelerar o processamento: -----
%%
global sigma;                           % desvio padrão - função de Eb/No
global G;                               % matriz geradora do código
global k;                               % comprimento da mensagem
global n;                               % comprimento do código
global m_apag;                          % matriz de apagamentos
global ptos;                            % quantos conjuntos de candidatas
global I_mask;                          % máscara para colunas unitárias
global nqb;                             % número de bits de quantização
global mxq;                             % amplitude máxima quantizada
global hqn;                             % metade da faixa 2^nqb-1
global delta;                           % pontos de delta em delta
```

Script 29: Validação dos valores de degradação da relação sinal ruído – 1a Parte.

```
% ----------- Selecionar uma das 4 alternativas de códigos a seguir:
%    load G74.mat;
%    G=G74;
%    clear G74;
%    dhmin = 3;
%    load Prob_7_4_3_106.mat mte_s
%    m_apag = mte_s(1:15,:);
%    clear mte_s;
%    load Redux_C(15,7,5).mat
%    ptos =  15;

   load G157.mat;
   dhmin = 5;
   load Prob_15_7_5_106.mat mte_s
   m_apag = mte_s(1:30,:);
   clear mte_s;
   load Redux_C(15,7,5).mat
   ptos =  10;

%    load -ascii G24.mat;
%    G=G24;
%    clear G24;
%    dhmin = 8;
%    load Prob_24_12_8_106.mat mte_s
%    m_apag = mte_s(1:120,:);
%    clear mte_s;
%    load Redux_C(24,12,8).mat
%    ptos = 10;

%    load -ascii G48.mat;
%    G=G48;
%    clear G48;
%    dhmin = 12;
%    load Prob_48_24_12_106.mat mte_s
%    m_apag = mte_s(1:500,:);
%    clear mte_s;
%    do_mld = false;
%    load Redux_C(48,24,12).mat
%    ptos = 10;
```

Script 29: Validação dos valores de degradação da relação sinal ruído – 2a. Parte.

```
%-------------------------------------------------------------------------
%% Preparação e inicialização dos dados e variáveis globais:
[k n] = size(G);                          % dimensões do código
I_mask = ~(sum(G,1)-1);                   % G (sistemática direita/esq)

nqb = 3;                                  % ajustar a qtde de bits desejada
mxq=1-2^-nqb;                             % valores válidos entre 3 e 8 bits
idr=nqb-2;                                % indexa vetor de reduções em dB
hqn=(2^nqb-1)/2;                          % inicializa metade faixa quantiz.

Lt = size(m_apag,1);                      % qtas linhas ao todo em m_apag

delta = Lt/ptos;                          % a cada quantas candidatas
vvidx=delta:delta:Lt;                     % índice de número de candidatas
ebno_max = 5;                             % enbo_max (quantos loops faremos)
x=4;                                      % expoente qtde iterações
z=1;                                      % mantissa qtde iterações
counter_max =...                          % quantas iterações para
   [z*10^x z*10^x z*10^x z*10^x z*10^x];% cada Eb/No

                                          %contadores de erro:
erro_cji        = zeros(ebno_max,ptos); % erros sem quantização
erroq_cji       = zeros(ebno_max,ptos); % erros com quantização
erro_eq_cji     = zeros(ebno_max,ptos); % erros equivalentes

erroqa_cji      = zeros(1,ebno_max);    % erros dec abrupta com quantização
erro_eqa_cji    = zeros(1,ebno_max);    % erros dec abrupta equivalentes

                                          % taxas de erros para cada ebno:
tx_cji          = zeros(ebno_max,ptos); % tx erros de bit s/ quantiz.
txq_cji         = zeros(ebno_max,ptos); % tx erros de bit c/ quantiz.
tx_eq_cji       = zeros(ebno_max,ptos); % tx erros de bit equivalentes
```

Script 29: Validação dos valores de degradação da relação sinal ruído – 3a. Parte.

```
%-------------------------------------------------------------------------
%% Simulação com e sem quantização:
%---------------- Loop para cada valor de Eb/No: -------------------------
for ebno = 1:ebno_max,

    sr = 10^(ebno/10);                  % determinação do desvio padrão
    sigma = (sqrt(1/(2*k*sr/n)));       % muda a cada Eb/No

    rand('state',0);                    % inicialização do gerador (pseudo)
    randn('state',0);                   % aleatório

    tic;                                % avaliação do tempo de processam.
    %--------------Loop para cada palavra código processada: -------------
    for counter=1:counter_max(ebno)     % qtde de iterações por Eb/No

        % ------ Geração da palavra código transmitida e recebida: -------
        [pct pcr] = Gera_Pct_Pcr();
        pcrq = QuantizaPcr(pcr);

        [erros errosq errosqa] = decode_cjiq(pcr,pcrq,pct);
        % ------- Acumulação dos resultados para cada Eb/No: -------------
        erro_cji(ebno,:)      = erro_cji(ebno,:)      + erros;
        erroq_cji(ebno,:)     = erroq_cji(ebno,:)     + errosq;
        erroqa_cji(ebno)      = erroqa_cji(ebno)      + errosqa;
    end
    % ---------- Fim do loop para cada palavra código.

    % ---------- Mostra tempo de processamento na tela ------------------
    fprintf(...
        'ebno = %d processou em %8.1f segundos para %8d iterações\n',...
         ebno,toc,counter);
    % --------- Cálculo das taxas de erros: ------------------------------
    tx_cji(ebno,:)  = erro_cji(ebno,:)  / (counter*k);
    txq_cji(ebno,:) = erroq_cji(ebno,:) / (counter*k);
end
fprintf('-----------------------------------------------------------\n');
%------------------Fim do loop para cada valor de Eb/No ------------------
```

Script 29: Validação dos valores de degradação da relação sinal ruído – 4a. Parte

```
%% Simulação com relação sinal ruído equivalente:
%---------------- Loop para cada valor de Eb/No: ------------------------
for ebno = 1:ebno_max,

    sr = 10^((ebno-red_db(idr,ebno))/10);
    sigma = (sqrt(1/(2*k*sr/n)));       % muda a cada Eb/No

    rand('state',0);                    % inicialização do gerador (pseudo)
    randn('state',0);                   % aleatório

    tic;                                % avaliação do tempo de processam.
    %--------------Loop para cada palavra código processada: -------------
    for counter=1:counter_max(ebno)     % qtde de iterações por Eb/No

        % ------ Geração da palavra código transmitida e recebida: -------
        [pct pcr_eq] = Gera_Pct_Pcr();

        [erros_eq erros_eqa] = decode_cji_eq(pcr_eq,pct);
        % ------- Acumulação dos resultados para cada Eb/No: -------------
        erro_eq_cji(ebno,:) = erro_eq_cji(ebno,:) + erros_eq;
        erro_eqa_cji(ebno)  = erro_eqa_cji(ebno)  + erros_eqa;
    end
    % ---------- Fim do loop para cada palavra código.
    % ---------- Mostra tempo de processamento na tela -------------------
    fprintf(...
        'ebno = %d processou em %8.1f segundos para %8d iterações\n',...
         ebno,toc,counter);
    % --------- Cálculo das taxas de erros e de atuação dos critérios: ---
    tx_eq_cji(ebno,:)   = erro_eq_cji(ebno,:)  / (counter*k);
end
%-------------------Fim do loop para cada valor de Eb/No ------------------
```

Script 29: Validação dos valores de degradação da relação sinal ruído – 5a. Parte.

```
%-------------------------------------------------------------------------
%% Apresentação dos resultados em forma de gráfico:
close all;
semilogy(tx_cji(:,10),'-bd','LineWidt',2.0);
grid;
hold on;
semilogy(txq_cji(:,10),'-r','LineWidt',2.0);
semilogy(tx_eq_cji(:,10),':g','LineWidt',2.0);
tit=sprintf('Código C(%d,%d,%d) - efeito da quantização com % d bits',...
             n,k,dhmin,nqb);
title(tit,'FontSize',16);
legend('sem quantização','com quantização','s/r equivalente');
set(gca,'xtick',1:5);
set(gca,'FontSize',14);
xlabel('E_b/N_0 (dB)','FontSize',14);
ylabel('Taxa de erros de bit','FontSize',14);
s=sprintf('Simulação com %d x 10^%d iterações',z,x);
yl=ylim;
text(1.5,1.5*yl(1),s,'FontSize',14,'BackgroundColor',[1 1 1]);

%-------------------------------------------------------------------------
%% Apresentação dos resultados na janela de comandos:
fprintf('\nSimulação para código C(%d,%d,%d) e ',n,k,dhmin);
fprintf('quantização com %d bits: \n\n',nqb);
fprintf('A)Decodificação abrupta: \n\n');
fprintf('Eb/No                =');
for i=1:ebno_max
    fprintf(' %8d',i);
end
fprintf('\n');

fprintf('iterações            =');
for i=1:ebno_max
    fprintf(' %8d',counter_max(i));
end
fprintf('\n');

fprintf('errosq dec. abrupta  =');
for i=1:ebno_max
    fprintf(' %8d',erroqa_cji(i));
end
fprintf('\n');

fprintf('erros_eq dec. abrupta =');
for i=1:ebno_max
    fprintf(' %8d',erro_eqa_cji(i));
end
fprintf('\n');

fprintf('\n');
```

Script 29: Validação dos valores de degradação da relação sinal ruído – 6a. Parte

```
fprintf('B)Decodificação por conjuntos de informação: \n\n');

fprintf('Qtd de candidatas:   ');
for i=1:ptos
    fprintf(' %8d',vvidx(i));
end
fprintf('\n\n');

fprintf('B1)Sem quantização: \n\n');

for i=1:ebno_max
    fprintf('erro cji(Eb/No=%d)    =',i);
    for j=1:ptos
        fprintf(' %8d',erro_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('taxa cji(Eb/No=%d)    =',i);
    for j=1:ptos
        fprintf(' %1.6f',tx_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

fprintf('B2)Com quantização com %d bits: \n\n',nqb);

for i=1:ebno_max
    fprintf('erroq cji(Eb/No=%d)   =',i);
    for j=1:ptos
        fprintf(' %8d',erroq_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('taxaq cji(Eb/No=%d)   =',i);
    for j=1:ptos
        fprintf(' %1.6f',txq_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');
```

Script 29: Validação dos valores de degradação da relação sinal ruído – 7a. Parte

```
fprintf('B3)Com redução equivalente de Eb/No: \n\n');

for i=1:ebno_max
    fprintf('erro_eq cji(Eb/No=%d) =',i);
    for j=1:ptos
        fprintf(' %8d',erro_eq_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

for i=1:ebno_max
    fprintf('taxa_eq cji(Eb/No=%d) =',i);
    for j=1:ptos
        fprintf(' %1.6f',tx_eq_cji(i,j));
    end
    fprintf('\n');
end
fprintf('\n');
```

Script 29: Validação dos valores de degradação da relação sinal ruído – Final

```
function pcrq = QuantizaPcr(pcr)
% Simula a discretização do sinal para quantização com nqb bits.
% entradas:
%   pcr      - palavra código recebida - modulada em BPSK
% saídas:
%   pcrq     - palavra código recebida quantizada.

    global hqn;                     % hqn = (2^nqb - 1)/2
    global mxq;                     % mxq = 1 - 2^(-nqb)

    pcrq = hqn*pcr/max(abs(pcr));   % normaliza entre -hqn e +hqn
    pcrq = pcrq + hqn;              % desloca tudo de hqn
    pcrq = double(round(pcrq));     % arredonda para valores inteiros
    pcrq = mxq*(pcrq/hqn-1);        % normaliza entre -mxq e +mxq
end
```

Script 30: Função para discretizar a palavra código recebida.

```
function [erros errosq errosqa] = decode_cjiq(pcr,pcrq,pct)
% Decodifica a palavra código recebida - determina erros e taxas de parada
% entradas:
%   pcr     - palavra código recebida com ruído gaussiano superposto
%   pcrq    - palavra código recebida com ruído superposto e quantizada
%   pct     - palavra código transmitida (para avaliar erros)
%
%   utiliza ainda as variáveis globais para aumentar a eficiência:
%
%   I_mask  - máscara das coluns unitárias da matriz geradora G
%   ptos    - qtde de pontos a examinar na sequência de candidatas
%   delta   - conjuntos de candidatas de delta em delta num total de ptos
% saídas:
%   erros     - quantidade de bits em erro na decodificaçao de pcr.
%   errosq    - quantidade de bits em erro na decodificaçao de pcrq.

global I_mask;                          % mascara posições da mensagem
global ptos;                            % nr. de pontos a avaliar
global delta;                           % delta entre pontos (candidatas)

    pcrqa = ceil(pcrq);                 % decodificação abrupta de pcrq
    errosqa = sum(and(xor(pcrqa,pct),...% soma erros por decodificação
                  I_mask));             % abrupta

    [SI mat_cand] = Gera_Cand(pcr);     % obtem as cadidatas e SI
    mat_cand_psk = 2*mat_cand-1;        % modula as candidatas em BPSK

    erros    = zeros(1,ptos);           % inicializa erros por ptos
    errosq   = zeros(1,ptos);           % inicializa erros por ptos

    ct    = mat_cand_psk*pcr';          % produtos internos a maximizar
    ctq   = mat_cand_psk*pcrq';         % produtos internos a maximizar

    for i=1:ptos                        % análise por pontos:
        [ignore Imax] = ...             % encontra o máximo produto interno
            max(ct(1:i*delta));         % = mínima distância euclidiana
        infr=mat_cand(Imax,:);          % obtem a candidata correspondente
        v_dif = ...                     % encontra as diferenças de bits
            and(xor(infr,pct),I_mask);  % dentre os da mensagem (I_mask)
        erros(i)=sum(v_dif);            % conta quantos erros de bit

        [ignore Imax] = ...             % encontra o máximo produto interno
            max(ctq(1:i*delta));        % = mínima distância euclidiana
        infr=mat_cand(Imax,:);          % obtem a candidata correspondente
        v_dif = ...                     % encontra as diferenças de bits
            and(xor(infr,pct),I_mask);  % dentre os da mensagem (I_mask)
        errosq(i)=sum(v_dif);           % conta quantos erros de bit
    end
end
```

Script 31: Função para decodificar a palavra código com e sem quantização.

```
function [erros_eq erros_eqa] = decode_cji_eq(pcr_eq,pct)
% Decodifica a palavra código recebida - determina erros e taxas de parada
% entradas:
%   pcr     - palavra código recebida com ruído gaussiano superposto
%   pct     - palavra código transmitida (para avaliar erros)
%
%   utiliza ainda as variáveis globais para aumentar a eficiência:
%
%   I_mask  - máscara das coluns unitárias da matriz geradora G
%   ptos    - qtde de pontos a examinar na sequência de candidatas
%   delta   - conjuntos de candidatas de delta em delta num total de ptos
% saídas:
%   erros_eq  - quantidade de bits encontrados em erro na decodificaçao.

global I_mask;                          % mascara posições da mensagem
global ptos;                            % nr. de pontos a avaliar
global delta;                           % delta entre pontos (candidatas)

    pcr_eqa = ceil(pcr_eq);             % decodificação abrupta de pcr_eq
    erros_eqa = ...                     % soma de erros por decodificação
      sum(and(xor(pcr_eqa,pct),I_mask));% abrupta

    [SI2 mat_cand_eq] = Gera_Cand(pcr_eq); % obtem as cadidatas e SI
    mat_cand_eq_psk = 2*mat_cand_eq-1;  % modula as candidatas em BPSK
    erros_eq = zeros(1,ptos);           % inicializa erros por ptos
    ct_eq = mat_cand_eq_psk*pcr_eq';    % produtos internos a maximizar

    for i=1:ptos                        % análise por pontos:
        [ignore Imax] = ...             % encontra o máximo produto interno
            max(ct_eq(1:i*delta));      % = mínima distância euclidiana
        infr=mat_cand_eq(Imax,:);       % obtem a candidata correspondente
        v_dif = ...                     % encontra as diferenças de bits
            and(xor(infr,pct),I_mask);  % dentre os da mensagem (I_mask)
        erros_eq(i)=sum(v_dif);         % conta quantos erros de bit
    end
end
```

Script 32: Função para decodificação com relação sinal/ruído equivalente.

ANEXO II
Programas e Bibliotecas em C++

```
//*****************************************************************************************
// Programa:    Dups4824
// Tipo:        Win32 Console Application
// Autor:       Gortan
// Propósito:   Determinar conjuntos de zeros comuns às palavras do código C(48,24,12)
// Utiliza:     Biblioteca C4824.lib
// Módulo:      Dups4824.cpp - Módulo principal (contém o ponto de entrada main())
//
// Utilização:  Dups4824 x y - onde x e y são os pesos das palavras a cruzar
//              se x e/ou y não forem especificados o programa assume o valor 12
//
// Criado com:  Visual Studio 2008
// Histórico:   - criado em 08/05/2011
//*****************************************************************************************

//*****************************************************************************************
#include <windows.h>                                    // acesso ao subsistema Win32
#include <stdio.h>                                      // CRT (printf e outras)
#include <conio.h>                                      // console i/o (_getch())
#include <locale.h>                                     // ajuste caracteres português
#include "C482412.h"                                    // protótipos e ctes C4824.lib
#include "Dups4824.h"                                   // protótipos deste módulo
//*****************************************************************************************

//*****************************************************************************************
// Função:      GetIniFim()
// Propósito:   Obter o peso das palavras especificado na linha de comando e os offsets
//              de início e fim na matriz das palavras agrupadas por peso (ma)
//              Utilizada por GetParam() para interpretar as strings da linha de comando
//
// Entradas:    String da linha de comando com o peso das palavras a cruzar
// Saídas:      Preenche os offsets e o peso com seus valores (passados como referência)
//*****************************************************************************************
void GetIniFim(DWORD& ini,                              // offset de início em ma
               DWORD& fim,                              // offset de fim em ma
               DWORD& p,                                // peso especificado
               char *par)                               // string da linha de comando
{                                                       //
    DWORD _ini;                                         // variável local para obter
    sscanf(par,"%d",&_ini);                             // o peso via sscanf()
    p=_ini;                                             // preenche o peso
    switch(_ini)                                        // seleciona os offsets
    {                                                   //
    default:                                            // se o peso especificado não
        ini=A_12_FIRST;                                 // for válido assume 12
        fim=A_12_LAST;                                  //
        p=12;                                           //
        break;                                          //
    case 16:                                            // peso 16
        ini=A_16_FIRST;                                 //
        fim=A_16_LAST;                                  //
        break;                                          //
    case 20:                                            // peso 20
        ini=A_20_FIRST;                                 //
        fim=A_20_LAST;                                  //
        break;                                          //
    case 24:                                            // peso 24
        ini=A_24_FIRST;                                 //
        fim=A_24_LAST;                                  //
        break;                                          //
    }                                                   //
}                                                       //
//*****************************************************************************************
```

Listagem 1: Includes e função GetIniFim() do módulo Dups4824.c

```
//*****************************************************************************************
// Função:      MostraRes()
// Propósito:   Colocar na tela os resultados intermediários ou finais obtidos
// Entradas:    Vetor com quantidade de zeros em comum e pesos das palavras envolvidas
// Saídas:      Nenhuma - as saídas vão para a tela
//*****************************************************************************************
void MostraRes(PDWORDLONG dups,                             // vetor com zeros em comum
               DWORD p1,                                    // peso das palavras x
               DWORD p2)                                    // peso das palavras y
{                                                           //
    printf("Palavras de peso %d e %d com:\n\n", p1, p2);    // cabeçalho do resultado
                                                            //
    for(DWORD i=0;i<13;++i)                                 // mostra todas as 13
    {                                                       // possibilidades, desde
        printf("\t%d zeros em comum: %15I64u\n",            // 12 zeros em comum até
               (i+6)*2,dups[i]);                            // 30 zeros em comum
    }                                                       //
}                                                           //
//*****************************************************************************************

//*****************************************************************************************
// Função:      GetParam()
// Propósito:   Obter e interpretar os parâmetros digitados na linha de comando
// Entradas:    Linha de comando e referências onde colocar os resultados
// Saídas:      Referências para offsets de início e fim e pesos envolvidos
//*****************************************************************************************
void GetParam(int argc, char *argv[],                       // linha de comando
              DWORD& ini_i, DWORD& fim_i,                   // offsets das palavras x
              DWORD& ini_j, DWORD& fim_j,                   // offsets das palavras y
              DWORD& p1,    DWORD& p2)                      // peso de x e peso de y
{                                                           //
    switch(argc)                                            // analisa quantos parâmetros
    {                                                       // o usuário digitou:
    case 2:                                                 // só um - considera só x
        GetIniFim(ini_i,fim_i,p1,argv[1]);                  // interpreta o param x
        ini_j=A_12_FIRST;                                   // já o y assume = 12
        fim_j=A_12_LAST;                                    //
        p2=12;                                              //
        break;                                              //
    case 3:                                                 // dois - considera x e y
        GetIniFim(ini_i,fim_i,p1,argv[1]);                  // interpreta o param x
        GetIniFim(ini_j,fim_j,p2,argv[2]);                  // e também o param y
        break;                                              //
    default:                                                // qualquer outra coisa
        ini_i=A_12_FIRST;                                   // desconsidera e assume
        fim_i=A_12_LAST;                                    // 12 x 12
        p1=12;                                              //
        ini_j=ini_i;                                        //
        fim_j=fim_i;                                        //
        p2=p1;                                              //
        break;                                              //
    }                                                       //
}                                                           //
//*****************************************************************************************
```

Listagem 2: Funções MostraRes() e GetParam() do módulo Dups4824.c

```
//*****************************************************************************************
// Função:      main()
// Propósito:   Ponto de entrada do programa
// Entradas:    Argumentos da linha de comando com peso das palavras a cruzar
// Saídas:      Nenhuma
//*****************************************************************************************
void main(int argc, char *argv[])                                 //
{                                                                 //
    DWORD ticks, i, j, ini_i, fim_i,                              // variáveis locais
          ini_j, fim_j, p1, p2;                                   // utilizadas
                                                                  //
    PDWORDLONG dups_z=(PDWORDLONG)                                // dups_z é um vetor para
                       HeapAlloc(  GetProcessHeap(),              // conter número de zeros em
                                   HEAP_ZERO_MEMORY,              // comum - 12,14,...,30 nas
                                   13*sizeof(DWORDLONG));         // posições 0 a 12
    if(!dups_z)                                                   // dups_z é alocado do
    {                                                             // processor heap - se não
        printf("Não há memória suficiente. Encerrando\n");        // houver memória suficiente
        return;                                                   // desistimos
    }                                                             //
                                                                  //
    setlocale(LC_ALL,"");                                         // ajusta os caracteres
                                                                  // default da máquina
    GetParam(argc, argv, ini_i, fim_i,                            // obtem os parâmetros da
             ini_j, fim_j, p1, p2);                               // linha de comando
                                                                  //
    printf("Encontrando zeros comuns entre "                      // informa ao usuário o
           "palavras de peso %d e %d \n\n",                       // que está sendo processando
           p1, p2);                                               //
                                                                  //
    printf("São %d palavras de peso %d x %d "                     // diz também quantas
           "palavras de peso %d\n\n",                             // palavras serão cruzadas
           fim_i-ini_i, p1, fim_j-ini_j, p2);                     // com quantas
                                                                  //
    PDWORDLONG ma = MontaMa(G);                                   // monta a matriz com as
                                                                  // palavras agrupadas por
    printf("--------------------------------"                     // peso (em C4824.lib)
           "-------------------------------\n"                    //
           "Agora iniciando a análise de "                        // anuncia início do
           "posições zero em comum:\n\n");                        // cruzamento
                                                                  //
    ticks=GetTickCount();                                         // anota o tempo de início
                                                                  //
    for(i = ini_i; i < fim_i; ++i)                                // faz para cada uma das
    {                                                             // palavras de peso x
        DWORD k = i - ini_i;                                      // ajusta offset em zero
                                                                  // para contar quantas já
        if(k && !(k%10000))                                       // foram processadas
        {                                                         // a cada 10.000 mostra
            printf("\n%4d palavras de peso %d "                   // o andamento
                   "processadas em %d ms\n\n",                    //
                   k, p1, GetTickCount()-ticks);                  // e o tempo transcorrido
            MostraRes(dups_z,p1,p2);                              // mostra resultado e
            ticks=GetTickCount();                                 // reinicia a contagem de
        }                                                         // tempo
        for(j = ini_j; j < fim_j; ++j)                            // varre todas as de peso
        {                                                         // y contra a de peso x
            DWORD peso =                                          // conta quantos zeros em
                    AchaPeso((~ma[i] & ~ma[j]) & C48_MASK);       // comum as duas têm
            if(peso) ++dups_z[peso/2-6];                          // elimina peso=0 (24x24)
        }                                                         // e armazena a contagem
    }                                                             // em dups_z
                                                                  //
    printf("\nTotal de %4d palavras de peso %d "                  // chega aqui após cruzar
           "processadas em %d ms\n\n",                            // todas - mostra o tempo
           i-ini_i,p1,GetTickCount()-ticks);                      // gasto na última batelada
                                                                  //
    MostraRes(dups_z,p1,p2);                                      // mostra resultado final
    printf("\nPressione qualquer tecla para encerrar...\n");      // dá uma chance ao usuário
    _getch();                                                     // de ver o que ocorreu
}                                                                 //
//*****************************************************************************************
```

Listagem 3: Função main() do módulo Dups4824.c

```
//*****************************************************************************************
// Programa:    Dups4824
// Tipo:        Win32 Console Application
// Autor:       Gortan
// Propósito:   Determinar conjuntos de zeros comuns às palavras do código C(48,24,12)
// Utiliza:     Biblioteca C4824.lib
// Módulo:      Dups4824.h - Congrega protótipos de funções definidas em Dups4824.cpp
//
// Criado com:  Visual Studio 2008
// Histórico:   - criado em 08/05/2011
//*****************************************************************************************
void GetIniFim(DWORD& ini,DWORD& fim, DWORD& p, char *par);

void MostraRes(PDWORDLONG dups, DWORD p1, DWORD p2);

void GetParam(int argc,     char *argv[],
              DWORD& ini_i, DWORD& fim_i,
              DWORD& ini_j, DWORD& fim_j,
              DWORD& p1,    DWORD& p2);
//*****************************************************************************************
```

Listagem 4: Arquivo de inclusão Dups4824.h

```
//*****************************************************************************************
// Programa:    Dups4824
// Tipo:        Win32 Application – biblioteca estática.
// Autor:       Gortan
// Propósito:   Determinar conjuntos de zeros comuns às palavras do código C(48,24,12)
// Utiliza:     Biblioteca C4824.lib
// Módulo:      C482412.cpp - Congrega rotinas da biblioteca estática C482412.lib
//
// Criado com:  Visual Studio 2008
// Histórico:   - criado em 08/05/2011
//*****************************************************************************************

//*****************************************************************************************
// Arquivos de inclusão utilizados:
//*****************************************************************************************
#include <windows.h>                                        // acesso ao subsistema Win32
#include <stdio.h>                                          // CRT (printf e outras)
#include "C482412.h"                                        // protótipos e ctes C4824.lib
//*****************************************************************************************

//*****************************************************************************************
// Função:      AchaPeso()
// Propósito:   Determinar quantos bits 1 existem na palavra __int64
//
// Entradas:    Palavra __int64 que deve ser "pesada"
// Saídas:      Peso da palavra
//
// obs:         O algorítimo utilizado aqui é o sugerido por Henry S. Warren em
//              "Hacker's Delight" (Addison Wesley, 2002) - Cap 5 "Counting Bits"
//*****************************************************************************************
DWORD AchaPeso(DWORDLONG x)                                 // recebe a palavra
{                                                           //
    x = (x & 0x5555555555555555) + ((x >> 1) & 0x5555555555555555); //
    x = (x & 0x3333333333333333) + ((x >> 2) & 0x3333333333333333); //
    x = (x & 0x0F0F0F0F0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F0F0F0F0F); //
    x = (x & 0x00FF00FF00FF00FF) + ((x >> 8) & 0x00FF00FF00FF00FF); //
    x = (x & 0x0000FFFF0000FFFF) + ((x >>16) & 0x0000FFFF0000FFFF); //
    x = (x & 0x00000000FFFFFFFF) + ((x >>32) & 0x00000000FFFFFFFF); //
                                                            //
    return (DWORD)x;                                        // retorna o peso
}                                                           //
//*****************************************************************************************
```

Listagem 5: Biblioteca C481224.lib – Includes e Função AchaPeso().

```
//*********************************************************************************************
// Função:      Vetor_X_G()
// Propósito:   Multiplicar uma palavra de 24 bits pela matriz G (em GF2) obtendo a
//              palavra código de 48 bits
//
// Entradas:        Palavra-mensagem de 24 bits (como DWORD)
// Saídas:      Palavra código de 48 bits   (como DWORDLONG)
//
// obs:         A multiplicação vetor x matriz em GF2 consiste em somar (em GF2 = xor)
//              as linhas da matriz G cujas correspondentes posições na mensagem contêm
//              bits 1
//*********************************************************************************************
DWORDLONG Vetor_X_G(DWORD p, const DWORDLONG* G)            // Multiplicando (p) e
{                                                           // Multiplicador (G)
    DWORD i, msk = 1<<23;                                   // inicializamos a máscara
    DWORDLONG pc = 0;                                       // e a palavra código pc
                                                            //
    for(i=0;i<24;++i)                                       // mascara cada bit de p
    {                                                       // e acumula o xor se o
        if((msk>>i) & p) pc^=G[i];                          // bit for 1
    }                                                       //
    return pc;                                              // entrega o resultado
}                                                           //
//*********************************************************************************************

//*********************************************************************************************
// Função:      CalcPeso()
// Propósito:   Obter a palavra código pc a partir da mensagem p e da matriz geradora G e
//              determinar seu peso.
//
// Entradas:    Palavra-mensagem p de 24 bits (como DWORD), referência para a palavra
//              código pc (obtida com Vetor_X_G()) e ponteiro para a matriz geradora G
// Saídas:      Peso da palavra código pc (obtido com AchaPeso())
//*********************************************************************************************
DWORD CalcPeso(DWORD p,                                     // mensagem a codificar
               DWORDLONG& pc,                               // palavra código (como ref)
               const DWORDLONG* G)                          // ptr para matriz geradora
{                                                           //
    pc = Vetor_X_G(p, G);                                   // codifica a mensagem
    return AchaPeso(pc);                                    // determina seu peso
}                                                           //
//*********************************************************************************************
```

Listagem 6: Biblioteca C482412.lib - Funções Vetor_X_G e CalcPeso().

```
        case 36:                                            // continuação da página anterior
            ma[idx_36++] = pc;                              //
            break;                                          //
        case 48:                                            //
            ma[idx_48++] = pc;                              //
            break;                                          //
        }                                                   //
    }                                                       //
    printf("Tempo transcorrido = %d ms\n\n",                // informa quanto tempo
           GetTickCount()-ticks);                           // gastou
    return ma;                                              // chamdor fica responsável
}                                                           // pela liberação da memória
//*********************************************************************************************
```

Listagem 7: Biblioteca C482412 - Função MontaMa().

```
//*****************************************************************************************
// Função:      MontaMa()
// Propósito:   Montar uma matriz com as 2^24 palavras do código C(48,24,12) agrupadas
//              por peso
// Entradas:    Ponteiro para a matriz geradora G do código
// Saídas:      Ponteiro para a matriz ma das palavras agrupadas por peso
//
// obs:         Tanto a matriz ma como a matriz G são implementadas como um vetor de
//              de inteiros tipo __int64, com cada bit do inteiro correspondendo a
//              um bit de uma linha da matriz.  Logo cada vetor terá tantos elementos
//              quantas forem as linhas da matriz que implementa.  A matriz G está
//              definida como constante em C482412.h
//
//              Como cada __int64 tem 64 bits, e o comprimento das palavras é de apenas
//              48 bits, o usuário deverá mascarar os bits não utilizados com C48_MASK
//              (definida em C482412.h) sempre que necessário.
//*****************************************************************************************
PDWORDLONG MontaMa(const DWORDLONG* G)                      // recebe ptr para G
{                                                           //
    DWORD i, ticks;                                         //
    PDWORDLONG ma;                                          // ptr para memória a alocar
    DWORD idx_0 = 0,                                        // índices das várias faixas
          idx_12 = A_12_FIRST,                              // de peso na matriz ma
          idx_16 = A_16_FIRST,                              //
          idx_20 = A_20_FIRST,                              //
          idx_24 = A_24_FIRST,                              //
          idx_28 = A_28_FIRST,                              //
          idx_32 = A_32_FIRST,                              //
          idx_36 = A_36_FIRST,                              //
          idx_48 = A_48_FIRST;                              //
                                                            //
    ma = (PDWORDLONG) HeapAlloc(GetProcessHeap(),           // aloca memória para a matriz
                                HEAP_ZERO_MEMORY,           // são 2^24 x __int64
                                16777216*sizeof(DWORDLONG));//
    if(!ma)                                                 // se não houver memória
    {                                                       // suficiente informa e
        printf("Não há memória suficiente. "                // desiste
               "Encerrando ... \n");                        //
        return 0;                                           //
    }                                                       //
                                                            //
    printf("Por favor aguarde, montando a matriz ma das "   // informa ao usuário que
           "palavras agrupadas por peso ...\n\n");          // pode demorar um pouco
    ticks=GetTickCount();                                   // Inicializa contagem de
    for(i=0;i<16777216;++i)                                 // tempo
    {                                                       // calcula o peso de cada
        DWORDLONG pc;                                       // uma das 2^24 palavras
                                                            // e a armazena na sua
        switch(CalcPeso(i,pc, G))                           // faixa de peso na matriz
        {                                                   // ma
        case 0:                                             //
            ma[idx_0++] = pc;                               //
            break;                                          //
        case 12:                                            //
            ma[idx_12++] = pc;                              //
            break;                                          //
        case 16:                                            //
            ma[idx_16++] = pc;                              //
            break;                                          //
        case 20:                                            //
            ma[idx_20++] = pc;                              //
            break;                                          //
        case 24:                                            //
            ma[idx_24++] = pc;                              //
            break;                                          //
        case 28:                                            //
            ma[idx_28++] = pc;                              //
            break;                                          //
        case 32:                                            //
            ma[idx_32++] = pc;                              //
            break;                                          // continua na próxima página...
```

```
//*********************************************************************************************
// Programa:    Dups4824
// Tipo:        Win32 Console Application
// Autor:       Gortan
// Propósito:   Determinar conjuntos de zeros comuns às palavras do código C(48,24,12)
// Utiliza:     Biblioteca C4824.lib
// Módulo:      C482412.h - Congrega protótipos de funções definidas em C482412.cpp e
//              também constantes globais utilizadas pela biblioteca.
// Criado com:  Visual Studio 2008
// Histórico:   - criado em 08/05/2011
//*********************************************************************************************

//*********************************************************************************************
// Distribuição de peso para o código C(48,24,12)
//*********************************************************************************************
const DWORD A_0  =        1;                                // qtde de palavras de peso  0
const DWORD A_12 =    17296;                                // qtde de palavras de peso 12
const DWORD A_16 =   535095;                                // qtde de palavras de peso 16
const DWORD A_20 =  3995376;                                // qtde de palavras de peso 20
const DWORD A_24 =  7681680;                                // qtde de palavras de peso 24
const DWORD A_28 =  3995376;                                // qtde de palavras de peso 28
const DWORD A_32 =   535095;                                // qtde de palavras de peso 32
const DWORD A_36 =    17296;                                // qtde de palavras de peso 36
const DWORD A_48 =        1;                                // qtde de palavras de peso 48
//*********************************************************************************************

//*********************************************************************************************
// Máscara para os 48 bits menos significativos de um __int64
//*********************************************************************************************
const DWORDLONG C48_MASK = 0x0000ffffffffffff;
//*********************************************************************************************

//*********************************************************************************************
// Protótipos das funções definidas nesta biblioteca:
//*********************************************************************************************
PDWORDLONG  MontaMa(const DWORDLONG* G);
DWORD       AchaPeso(DWORDLONG x);
DWORDLONG   Vetor_X_G(DWORD p, const DWORDLONG* G);
DWORD       CalcPeso(DWORD p, DWORDLONG& pc, const DWORDLONG* G);
//*********************************************************************************************
```

Listagem 8: Biblioteca C482412.lib - Arquivo de inclusão C482412.h (parcial)

```
//*********************************************************************************************
// Índices para início e fim das faixas de agrupamento de pesos na matriz ma
//*********************************************************************************************
const DWORD A_12_FIRST = A_0;                                 // índices para a primeira e
const DWORD A_12_LAST  = A_12_FIRST + A_12;                   // última palavras de peso 12
                                                              //
const DWORD A_16_FIRST = A_12_LAST;                           // índices para a primeira e
const DWORD A_16_LAST  = A_16_FIRST + A_16;                   // última palavras de peso 16
                                                              //
const DWORD A_20_FIRST = A_16_LAST;                           // índices para a primeira e
const DWORD A_20_LAST  = A_20_FIRST + A_20;                   // última palavras de peso 20
                                                              //
const DWORD A_24_FIRST = A_20_LAST;                           // índices para a primeira e
const DWORD A_24_LAST  = A_24_FIRST + A_24;                   // última palavras de peso 24
                                                              //
const DWORD A_28_FIRST = A_24_LAST;                           // índices para a primeira e
const DWORD A_28_LAST  = A_28_FIRST + A_28;                   // última palavras de peso 28
                                                              //
const DWORD A_32_FIRST = A_28_LAST;                           // índices para a primeira e
const DWORD A_32_LAST  = A_32_FIRST + A_32;                   // última palavras de peso 32
                                                              //
const DWORD A_36_FIRST = A_32_LAST;                           // índices para a primeira e
const DWORD A_36_LAST  = A_36_FIRST + A_36;                   // última palavras de peso 36
                                                              //
const DWORD A_48_FIRST = A_36_LAST;                           // índice para a primeira e
                                                              // única palavra de peso 48
//*********************************************************************************************

//*********************************************************************************************
// Máscara para os 48 bits menos significativos de um __int64
//*********************************************************************************************
const DWORDLONG C48_MASK = 0x0000ffffffffffff;
//*********************************************************************************************

//*********************************************************************************************
// Definição da matriz geradora para o código C(48,24,12) como vetor de 24 __int64
//*********************************************************************************************
const DWORDLONG G[24]=  //
{                       //
    0x0000800000ee69ad, // 1000 0000 0000 0000 0000 0000 1110 1110 0110 1001 1010 1101
    0x00004000007734d7, // 0100 0000 0000 0000 0000 0000 0111 0111 0011 0100 1101 0111
    0x0000200000bb9a6b, // 0010 0000 0000 0000 0000 0000 1011 1011 1001 1010 0110 1011
    0x0000100000ddcd35, // 0001 0000 0000 0000 0000 0000 1101 1101 1100 1101 0011 0101
    0x00000800006ee69b, // 0000 1000 0000 0000 0000 0000 0110 1110 1110 0110 1001 1011
    0x0000040000b7734d, // 0000 0100 0000 0000 0000 0000 1011 0111 0111 0011 0100 1101
    0x00000200005bb9a7, // 0000 0010 0000 0000 0000 0000 0101 1011 1011 1001 1010 0111
    0x0000010000addcd3, // 0000 0001 0000 0000 0000 0000 1010 1101 1101 1100 1101 0011
    0x0000008000d6ee69, // 0000 0000 1000 0000 0000 0000 1101 0110 1110 1110 0110 1001
    0x00000040006b7735, // 0000 0000 0100 0000 0000 0000 0110 1011 0111 0111 0011 0101
    0x000000200035bb9b, // 0000 0000 0010 0000 0000 0000 0011 0101 1011 1011 1001 1011
    0x00000010009addcd, // 0000 0000 0001 0000 0000 0000 1001 1010 1101 1101 1100 1101
    0x00000008004d6ee7, // 0000 0000 0000 1000 0000 0000 0100 1101 0110 1110 1110 0111
    0x0000000400a6b773, // 0000 0000 0000 0100 0000 0000 1010 0110 1011 0111 0111 0011
    0x0000000200d35bb9, // 0000 0000 0000 0010 0000 0000 1101 0011 0101 1011 1011 1001
    0x000000010069addd, // 0000 0000 0000 0001 0000 0000 0110 1001 1010 1101 1101 1101
    0x000000008034d6ef, // 0000 0000 0000 0000 1000 0000 0011 0100 1101 0110 1110 1111
    0x00000000409a6b77, // 0000 0000 0000 0000 0100 0000 1001 1010 0110 1011 0111 0111
    0x0000000020cd35bb, // 0000 0000 0000 0000 0010 0000 1100 1101 0011 0101 1011 1011
    0x0000000010e69add, // 0000 0000 0000 0000 0001 0000 1110 0110 1001 1010 1101 1101
    0x0000000008734d6f, // 0000 0000 0000 0000 0000 1000 0111 0011 0100 1101 0110 1111
    0x0000000004b9a6b7, // 0000 0000 0000 0000 0000 0100 1011 1001 1010 0110 1011 0111
    0x0000000002dcd35b, // 0000 0000 0000 0000 0000 0010 1101 1100 1101 0011 0101 1011
    0x0000000001fffffe  // 0000 0000 0000 0000 0000 0001 1111 1111 1111 1111 1111 1110
};                      //
//*********************************************************************************************
```

Listagem 9: Biblioteca C482412.lib - Arquivo C482412.h - continuação.

```
//*****************************************************************************************
// Função:      RandPerm48()
// Propósito:   Gerar uma permutção aleatória dos 48 elementos de um vetor dado
//
// Entradas:    Ponteiro para o vetor cujos elementos devem ser permutados
// Saídas:      Máscara DWORDLONG com as primeiras n posições permutadas em 1
//*****************************************************************************************
DWORDLONG RandPerm48(PDWORD pidx, DWORD n)                  // ptr para vetor
{                                                           //
    DWORDLONG msk_p=0,msk=1;                                //
                                                            //
    for(DWORD pos=0;pos<48;++pos)                           // percorre todas as posições
    {                                                       //
        DWORD u = (DWORD)                                   //
            ((double)rand()/(RAND_MAX + 1)*(48 - pos)+pos); // sorteia um número entre a
        DWORD temp=pidx[u];                                 // posição e 47
        pidx[u]=pidx[pos];                                  // troca a posição pela do
        pidx[pos]=temp;                                     // número sorteado
    }                                                       //
    for(DWORD k=0;k<n;++k)                                  // gera uma máscara com as
    {                                                       // primeiras n posições
        msk_p|=(msk<<(48-pidx[k]));                         // permutadas
    }                                                       //
    return msk_p;                                           // retorna a máscara
}                                                           //
//*****************************************************************************************
```

Listagem 10: Biblioteca C482412.lib - Função RandPerm48().

```
//*****************************************************************************************
// Função:      GaussElim()
// Propósito:   Executar a eliminação de Gauss na matriz geradora do código C(48,24,12)
//              na sequência ditada por um vetor de índices, retornado a quantidade de
//              colunas examinadas até conseguir obter todos os 24 pivôs.
//
// Entradas:    Ponteiro para o vetor com sequência de índices a utilizar e ptr para
//              matriz geradora
// Saídas:      Quantidade de colunas examinadas até obter 24 pivôs.
//
// obs.:        Não é feita troca de linhas para manter os pivôs na sequência.  Os
//              pivôs são mantidos na linha encontrada e as linhas já com pivô são
//              mascaradas para que não sejam processadas novamente.
//*****************************************************************************************
DWORD GaussElim(PDWORD pidx,                                    // ptr para vetor de índices
                PDWORDLONG g48)                                 // ptr para matriz a processar
{                                                               //
    DWORDLONG msk      = 1;                                     // máscara da pos. a process.
    DWORD     msk_l    = 0xffffff,                              // mascara de linhas process.
              pivo_cnt = 0,                                     // contador de pivôs obtidos
              col;                                              // qtde de colunas percorridas
                                                                //
    for(col=0;pivo_cnt<24;++col)                                // percorre as colunas
    {                                                           // até encontrar 24 pivôs
        msk=((DWORDLONG)1<<(48-pidx[col]));                     // posiciona a máscara onde
        for(DWORD lin=0;lin<24;++lin)                           // procurar o pivô
        {                                                       //
            if((1<<lin) & msk_l)                                // se ainda não processou esta
            {                                                   // linha vai processá-la
                if(g48[lin] & msk)                              // se esta linha tiver pivô
                {                                               // na coluna dada por msk
                    msk_l &= (~(1<<lin));                       // dá a linha como processada
                    ++pivo_cnt;                                 // contabiliza o pivô
                    for(DWORD k=0;k<24;++k)                     // Agora executa xor da linha
                    {                                           // com pivô com todas as
                        if((1<<k) & msk_l)                      // que ainda não processou e
                        {                                       // que têm 1 na coluna do
                            if(g48[k] & msk)                    // pivô encontrado (elimina
                                g48[k]^=g48[lin];               // os demais)
                        }                                       //
                    }                                           //
                    break;                                      // sai do loop de linhas
                }                                               // quando encontra o pivô
            }                                                   //
        }                                                       //
    }                                                           //
    return col;                                                 //
}                                                               //
//*****************************************************************************************
```

Listagem 11: Biblioteca C482412 - Função GaussElim()

```
//*****************************************************************************************
// Programa:        Tent4824
// Tipo:            Win32 Console Application
// Autor:       Gortan
// Propósito:   Determinar a probabilidade de se obter um CI examinando n colunas da
//              matriz geradora do código C(48,24,12) - n variando entre 24 e 36
// Método:      Geração de uma grande quantidade de permutações aleatórias das colunas
//              da matriz geradora e busca exaustiva na matriz das palavras código
//              agrupadas por peso, por um conjunto de zeros coincidente com as
//              primeiras n colunas da permutação.
//
// Entradas:    Quantidade de testes a realizar é especificável na linha de comando.
//
// Utiliza:     Biblioteca C4824.lib
// Módulo:      Tent4824.cpp - Módulo principal (contém o ponto de entrada main())
//
// Criado com:  Visual Studio 2008
// Histórico:   - criado em 08/05/2011
//*****************************************************************************************

//*****************************************************************************************
// Arquivos de inclusão utilizados:
//*****************************************************************************************
#include <windows.h>                                    // acesso ao subsistema Win32
#include <stdio.h>                                      // CRT (printf e outras)
#include <conio.h>                                      // console i/o (_getch())
#include <locale.h>                                     // ajuste caracteres português
#include "C482412.h"                                    // protótipos e ctes C4824.lib
//*****************************************************************************************

//*****************************************************************************************
// Inicialização da string de configuração:
//*****************************************************************************************
#ifdef _DEBUG
const char config[]="Debug";
#else
const char config[]="Release";
#endif
//*****************************************************************************************

//*****************************************************************************************
// Defines para quantidade máxima e default de iterações:
//*****************************************************************************************
#define QTDE_MAX 1000000
#define QTDE_DEFAULT 1000
//*****************************************************************************************

//*****************************************************************************************
// Tabela de último valor a examinar dependendo do número de colunas
//*****************************************************************************************
const DWORD LastW[]=                                    //
{                                                       //
    A_24_LAST,                                          // testa 24 colunas
    A_20_LAST, A_20_LAST, A_20_LAST, A_20_LAST,         // testa 25,26,27,28 colunas
    A_16_LAST, A_16_LAST, A_16_LAST, A_16_LAST,         // testa 29,30,31,32 colunas
    A_12_LAST, A_12_LAST, A_12_LAST, A_12_LAST          // testa 33,34,35,36 colunas
};                                                      //
//*****************************************************************************************

//*****************************************************************************************
// Protótipos de funções definidas e usadas neste módulo:
//*****************************************************************************************
DWORD ContaFalhas(PDWORDLONG ma, DWORD zeros,DWORD Qtde);
DWORD GetParamIter(int argc, char *argv[]);
//*****************************************************************************************
```

Listagem 12: Programa Tent4824.c - includes, defines e constantes

```
//*****************************************************************************************
// Função:      main()
// Propósito:   Ponto de entrada do programa
// Entradas:    Argumentos da linha de comando com peso das palavras a cruzar
// Saídas:      Nenhuma
//*****************************************************************************************
void main(int argc, char *argv[])                          //
{                                                          //
                                                           //
    setlocale(LC_ALL,"");                                  // ajusta os caracteres
                                                           // default da máquina
                                                           //
    DWORD qtde=GetParamIter(argc, argv);                   // quantidade de iterações
    DWORD falhas[13]={0,0,0,0,0,0,0,0,0,0,0,0,0};          // contador de falhas
    printf("Probabilidades de obtenção "                   // inicializa a tela
           "de um C.I. em n colunas "                      // informando o que está
           "- Código C(48,24,12)\n\n");                    // sendo executado
    printf("\t\tMétodo da Busca Exaustiva\n\n");           // e com qual método
    printf("Rodando na configuração %s "                   // informa condições de
           "para %d iterações\n\n",config,qtde);           // teste ao usuário
                                                           // inicialmente zerado
    PDWORDLONG ma = MontaMa(G);                            // monta a matriz com as
                                                           // palavras agrupadas por peso
    DWORD ticks=GetTickCount();                            // inicializa contador de
                                                           // tempo
    for(DWORD i=24;i<37;++i)                               // analiza de 24 a 36
        falhas[i-24] = ContaFalhas(ma,i,qtde);             // colunas
                                                           //
    printf("\n\t----- Resumo da Ópera: -----\n\n");        // ao final faz um resumo
    printf("\tColunas\t\t\tFalhas\t\t\tTaxa de Sucesso\n\n");// para o usuário
    for(DWORD i=0;i<13;++i)                                // mostra todas as falhas
        printf("\t   %d\t\t\t% 6d\t\t\t% 7.2f %%\n",       // e correspondentes
               i+24,                                       // taxas de sucesso
               falhas[i],                                  //
               (1.0f-(float)falhas[i]/qtde)*100.0f);       //
    printf("\nTempo total transcorrido: %d ms\n\n",        // e mostra o tempo total
           GetTickCount()-ticks);                          // transcorrido no teste
    printf("\nPressione qualquer tecla "                   // segura a saída na tela
           "para encerrar...\n\n");                        // para o usuário examinar
    _getch();                                              //
}                                                          //
//*****************************************************************************************


//*****************************************************************************************
// Função:      GetParamIter()
// Propósito:   Obter a quantidade de iterações desejadas, digitadas na linha de comando
// Entradas:    Linha de comando
// Saídas:      Quantidade de iterações desejadas ou o valor default
//*****************************************************************************************
DWORD GetParamIter(int argc, char *argv[])                 // linha de comando
{                                                          //
    DWORD qtde;                                            // receberá a quantidade
    switch(argc)                                           // analisa quantos parâmetros
    {                                                      // o usuário digitou:
    case 2:                                                // só um - considera-o a qtde
        sscanf(argv[1],"%d",&qtde);                        // lê a string da linha de cmd
        if((qtde > 1) && (qtde < QTDE_MAX))                // verifica a plausibilidade e
        {                                                  // se for plausível
            return qtde;                                   // assume o valor
        }                                                  // caso contrário fall through
    default:                                               // qualquer outra coisa
        return QTDE_DEFAULT;                               // retorna a qtde default
    }                                                      //
}                                                          //
//*****************************************************************************************
```

Listagem 13: Programa Tent4824.c - Funções main() e GetParamIter().

```
//*********************************************************************************************
// Função:      ContaFalhas()
// Propósito:   Ponto de entrada do programa
// Entradas:    Argumentos da linha de comando com peso das palavras a cruzar
// Saídas:      Nenhuma
//*********************************************************************************************
DWORD ContaFalhas(PDWORDLONG ma, DWORD zeros,DWORD Qtde)    //
{                                                           //
    DWORD falhas=0;                                         // contador de falhas zerado
    DWORD idx[]={ 1, 2, 3, 4, 5, 6, 7, 8, 9,10,            // índices de colunas que
                 11,12,13,14,15,16,17,18,19,20,            // serão permutados
                 21,22,23,24,25,26,27,28,29,30,            //
                 31,32,33,34,35,36,37,38,39,40,            //
                 41,42,43,44,45,46,47,48};                 //
                                                           //
    printf("-----------------------------------------------"// anunciamos início da
           "\nIniciando teste com %d "                      // rodada
           "palavras em %d colunas\n\n",Qtde,zeros);        //
    DWORD fim=LastW[zeros-24];                              // o fim depende do número
    DWORD ticks=GetTickCount();                             // de colunas a examinar
    for(DWORD i=0;i<Qtde;++i)                               //
    {                                                       //
        DWORDLONG msk_p=RandPerm48(idx,zeros);              //
        _asm{                                               // Otimização asm reduz
            mov ebx, dword ptr msk_p[0]                     // tempo de execução em 20 %
            mov edx, dword ptr msk_p[4]                     // máscara sempre em ebx-edx
            mov esi,ma[0]                                   // esi indexa início de ma
            mov edi,fim                                     // edi indexa último valor
                                                            // a comparar
            mov ecx,1                                       // ecx conta loops
loop1:      mov eax,ebx                                     // pega máscara low
            and eax,dword ptr [esi+ecx*8]                   // faz AND com ma low
            jnz end1                                        // próximo se não deu zero
                                                            // se deu falta comparar high
            mov eax,edx                                     // pega máscara high
            and eax,dword ptr[esi+ecx*8+4]                  // faz AND com ma high
            jnz end1                                        // próximo se não deu zero
                                                            // se deu zero incrementamos
            mov eax,falhas                                  // contador de falhas
            add eax,1                                       // e
            mov falhas,eax                                  //
            jmp end2                                        // interrompemos o loop
                                                            //
end1:       add ecx,1                                       // atualizamos contador de
            cmp ecx,edi                                     // loop e testamos se já
            jl loop1                                        // chegamos ao fim
end2:                                                       //
        }                                                   // fim do bloco _asm
    }                                                       // fim do loop externo
    printf("\ttempo transcorrido foi %d ms\n\n"             // mostramos consumo de
           "\tfalhas ocorridas: %d\n\n"                     // tempo e falhas ocorridas
           "\tTaxa de sucesso =  %6.2f %%\n",              // mais taxa de sucesso ao
           GetTickCount()-ticks,                            // usuário
           falhas,                                          //
           (1.0f-(float)falhas/Qtde)*100.0f);               //
    return falhas;                                          // retornamos a qtde de
}                                                           // falhas encontradas
//*********************************************************************************************
```

Listagem 14: Programa Tent4824.c - Função ContaFalhas().

```
//*********************************************************************************************
// Programa:    Gauss4824
//
// Tipo:        Win32 Console Application
//
// Autor:       Gortan
//
// Propósito:   Determinar a probabilidade de se obter um CI examinando n colunas da
//              matriz geradora do código C(48,24,12) - n variando entre 24 e 36
//
// Método:      Geração de uma grande quantidade de permutações aleatórias das colunas
//              da matriz geradora e aplicação da eliminação de Gauss modificada a cada
//              permutação, contando a quantidade de coluns percorridas em cada caso.
//
// Entradas:    Quantidade de testes a realizar é especificável na linha de comando.
//
// Utiliza:     Biblioteca C4824.lib
//
// Módulo:      Gauss4824.cpp - Módulo principal (contém o ponto de entrada main())
//
// Criado com:  Visual Studio 2008
//
// Histórico:   - criado em 08/05/2011
//*********************************************************************************************

//*********************************************************************************************
// Arquivos de inclusão utilizados:
//*********************************************************************************************
#include <windows.h>                                        // acesso ao subsistema Win32
#include <stdio.h>                                          // CRT (printf e outras)
#include <conio.h>                                          // console i/o (_getch())
#include <locale.h>                                         // ajuste caracteres português
#include "C482412.h"                                        // protótipos e ctes C4824.lib
//*********************************************************************************************

//*********************************************************************************************
// Inicialização da string de configuração:
//*********************************************************************************************
#ifdef _DEBUG
const char config[]="Debug";
#else
const char config[]="Release";
#endif
//*********************************************************************************************

//*********************************************************************************************
// Defines para quantidade máxima e default de iterações:
//*********************************************************************************************
#define QTDE_MAX 1000000000
#define QTDE_DEFAULT 100000
//*********************************************************************************************

//*********************************************************************************************
// Protótipos de funções definidas e utilizads neste módulo:
//*********************************************************************************************
DWORD GetParamIter(int argc, char *argv[]);
//*********************************************************************************************
```

Listagem 15: Programa Gauss4824.c - includes, defines e constantes.

```
//*********************************************************************************************
// Função:      main()
// Propósito:   Ponto de entrada do programa
// Entradas:    Argumentos da linha de comando com peso das palavras a cruzar
// Saídas:      Nenhuma
//*********************************************************************************************
void main(int argc, char *argv[])                          //
{                                                          //
    DWORD idx[]={ 1, 2, 3, 4, 5, 6, 7, 8, 9,10,            // índices de colunas que
                 11,12,13,14,15,16,17,18,19,20,            // serão permutados
                 21,22,23,24,25,26,27,28,29,30,            //
                 31,32,33,34,35,36,37,38,39,40,            //
                 41,42,43,44,45,46,47,48};                 //
                                                           //
    setlocale(LC_ALL,"");                                  // ajusta os caracteres
    PDWORDLONG g48                                         // aloca memória para fazer
       = (PDWORDLONG) HeapAlloc(GetProcessHeap(),          // uma cópia da matriz
                                HEAP_ZERO_MEMORY,          // geradora
                                24*sizeof(DWORDLONG));     //
    if(!g48)                                               // se não houver memória
    {                                                      // suficiente informa e
        printf("Não há memória suficiente. "               // desiste
               "Encerrando ... \n");                       //
        return;                                            //
    }                                                      //
    for(DWORD i=0;i<24;++i) g48[i]=G[i];                   // realiza a cópia
    DWORD suc_cnt[14]={0,0,0,0,0,0,0,0,0,0,0,0,0,0};       // inicializa cont sucessos
    DWORD qtde=GetParamIter(argc, argv),colunas;           // especifica quantos testes
    printf("Probabilidades de obtenção "                   // inicializa a tela
           "de um C.I. em n colunas "                      // informando o que está
           "- Código C(48,24,12)\n\n");                    // sendo executado
    printf("\t\tMétodo da Eliminação de Gauss\n\n");       // e com qual método
    printf("Rodando na configuração %s "                   // informa condições de
           "para %d iterações\n\n",config,qtde);           // teste ao usuário
                                                           //
    DWORD ticks = GetTickCount();                          // inicializa contagem tempo
    for(DWORD i=0;i<qtde;++i)                              // loop de testes
    {                                                      // para qtde iterações
        RandPerm48(idx,0);                                 // gera permutação de índices
        colunas=GaussElim(idx,g48);                        // conta quantas colunas
        suc_cnt[colunas-24]++;                             // examinou para obter o CI
    }                                                      //
    HeapFree(GetProcessHeap(),0,g48);                      // libera a memória
                                                           // com a cópia da matriz
    for(DWORD k=1;k<14;++k)                                // acumula as tentativas
        suc_cnt[k]+=suc_cnt[k-1];                          //
    printf("Tempo transcorrido foi de %d ms\n\n",          // informa tempo de execução
           GetTickCount()-ticks);                          //
    for(DWORD j=0;j<14;++j)                                // mostra os resultados
        printf("%d colunas -> % 10d sucessos\n",           // acumulados
        j+24,                                              //
        suc_cnt[j]);                                       //
    printf("\nPressione qualquer tecla "                   // segura a saída na tela
           "para encerrar...\n\n");                        // para o usuário examinar
    _getch();                                              //
}                                                          //
//*********************************************************************************************
```

Listagem 16: Programa Gauss4824.c - Função main().

ANEXO III
Quantidade máxima de colunas da matriz geradora a examinar

O teorema aqui reproduzido foi apresentado por Vera Pless e W. Cary Huffman em (HUFFMAN, 2003).

Teorema:

Seja C um código $[n, k, d]$. Então qualquer conjunto de $n - d + 1$ posições de suas palavras-código irá conter um conjunto de informação. Adicionalmente, d é o maior número que satisfaz essa propriedade.

Demonstração:

Seja **G** a matriz geradora de C e seja **X** qualquer conjunto de s de suas colunas.

Para simplificar a argumentação, suponha-se que **X** seja o conjunto de suas últimas s posições. De acordo com o conceito de códigos equivalentes, qualquer outro conjunto pode ser reduzido a este permutando-se colunas de **G**, o que torna a suposição genérica.

Suponha-se que **X** não contenha um conjunto de informação. Faça-se **G** = [**A** | **B**] com **A** de dimensão k x $(n - s)$ e **B** de dimensão k x s.

Então o posto coluna de **B**, que é igual a seu posto linha, será inferior a k.

Deverá então existir uma combinação não trivial das linhas de **B** a qual é igual a **0**, e portanto deverá existir uma palavra-código **c** com zeros em suas últimas s posições.

Como **G** contém k linhas linearmente independentes, $\mathbf{c} \neq \mathbf{0}$ e portanto $d \leq n - s$, ou, o que é equivalente, $s \leq n - d$.

A demonstração segue da última afirmação, ou seja, para que **X** não contenha um conjunto de informação é necessário que $s \leq n - d$, logo para qualquer conjunto de $n - d + 1$ colunas de **G**, **X** irá conter, necessariamente, um conjunto de informação.

ANEXO IV
Quantidade de iterações necessárias nas simulações

Na análise de códigos corretores de erros, principalmente códigos longos, em face do grande número de palavra-código envolvidas, é muitas vezes necessário recorrer a simulações com o fim de determinar diversas proporções: proporção de palavras-código recebidas com erro, proporção de combinações de colunas da matriz geradora que constituem conjuntos de informação, etc. Surge então a necessidade de determinar o número ideal de iterações para se obter uma determinada precisão nos resultados. Este anexo, baseado na apresentação do professor Pedro Luiz Costa Neto (COSTA NETO, 1977), visa esclarecer este ponto.

De acordo com (COSTA NETO, 1977), para se obter uma estimativa de uma proporção *p* de uma população, o melhor estimador a se utilizar é a proporção obtida a partir de uma amostra, chamada de frequência relativa amostral p'. Esse é um estimador **justo** (pois sua média coincide com a da população), **consistente** (pois sua variância tende a zero quando o tamanho da amostra tende a infinito), **eficiente** (pois para um mesmo tamanho da amostra sua variância é menor que a de qualquer outro estimador escolhido) e, finalmente, **suficiente**, (pois contém o maior conteúdo de informação possível com relação ao parâmetro estimado – nesse caso a proporção da população).

Para a o caso da estimativa de *p* da população, a distribuição amostral de p' será conforme a distribuição de Bernoulli, a qual, para amostras suficientemente grandes, pode ser aproximada por uma distribuição normal, sem perda de precisão.

Assim vê-se em (COSTA NETO,1997) que:

$$p' = p \pm e_0 \tag{IV.1}$$

com o semi-intervalo de erro e_0 dado por:

$$e_0 = z_{\alpha/2}\sqrt{\frac{p(1-p)}{n}} \tag{IV.2}$$

onde:

e_0 – semi-intervalo de erro de estimação.

$z_{a/2}$ – variável normal padronizada para para intervalo de confiança α.

p – proporção populacional.

n – tamanho da amostra.

A Figura 52 ilustra melhor essas relações. Na figura vê-se a distribuição amostral do estimador *p'* de *p*, que foi aproximada por uma distribuição normal, o que é correto mesmo quando a distribuição de *p* não seja normal. No caso α é a probabilidade do intervalo $p' \pm e_0$ não conter o verdadeiro parâmetro *p* sendo estimado.

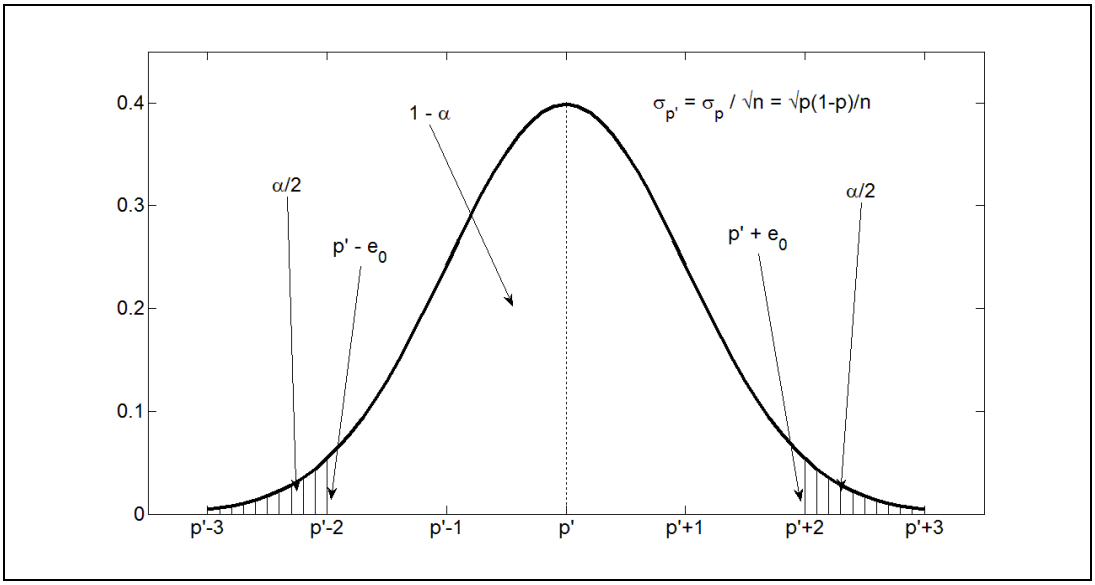

Figura 52 – Distribuição amostral de p'

A Figura 52 deve portanto ser interpretada da seguinte maneira: há uma probabilidade 1 – α do intervalo $p' \pm e_0$ conter o verdadeiro parâmetro estimado *p*[2].

A partir da equação (IV.2) pode-se então determinar o valor de *n* segundo:

$$n = \left(\frac{z_{\alpha/2}}{e_0} \right)^2 p(1-p) \qquad \text{(IV.3)}$$

A única dificuldade aqui é que a equação (IV.3) necessita do parâmetro *p*, justamente a grandeza a ser estimada.

Há duas possibilidades de se resolver esse dilema: a primeira é utilizar um limitante para a a grandeza *p*(1 – *p*) – como a proporção *p* pode variar entre 0 e 1, a grandeza *p*(1 – *p*) representa uma polinômio de grau 2 com valor máximo ocorrendo para *p* = 0,5, logo $p(1-p)\,|_{max} = 0{,}25$.

Uma segunda possibilidade, que permite chegar a valores menores para

2 Muitas vezes fala-se, erroneamente, da probabilidade do parâmetro *p cair no intervalo* $p' \pm e_0$. Isso é incorreto, uma vez que o valor de *p* é fixo, sendo que é *p'* e seu intervalo que variam, contendo ou não *p*.

n, é retirar-se inicialmente uma amostra piloto da população e obter uma primeira estimativa *p'* para a grandeza *p*, a qual será a seguir utilizada em lugar de *p* na equação (IV.3). Neste trabalho foi utilizada a segunda alternativa, procurando reduzir ao máximo o número *n* de iterações necessárias em cada caso.

Alternativamente, quando a quantidade *n* de iterações necessárias para uma determinada precisão se mostrou inviável do ponto de vista do tempo computacional envolvido, uma quantidade menor de iterações foi empregada e a equação (IV.2) foi utilizada para conhecer o erro incorrido nesse caso.

ANEXO V
Posições de zeros comuns às palavras do código *C*(48,24,12)

O código *C*(48,24,12) possui palavras-código de pesos 12, 16, 20, 24, 28, 32 e 36. Nesta análise interessa identificar palavras-código com 24 ou mais zeros em comum pois são os conjuntos de posições com 24 ou mais zeros que irão especificar conjuntos de posições que não constituem conjuntos de informação. Consequentemente, apenas as palavras de peso 12, 16, 20 e 24 estarão envolvidas nesta análise.

O maior conjunto de zeros possível em uma palavra-código do código *C*(48,24,12) será de 36 zeros, ocorrendo nas palavras de peso 12. Evidentemente, não é possível existirem duas palavras-código diferentes, ambas com peso 12 e com o mesmo conjunto de posições zero, pois isso implicaria que ambas as palavas são idênticas, pois a distância de Hamming entre as duas seria zero. Tampouco é possível encontrar duas palavras código diferentes, de peso 12, com 35 zeros em comum, pois isso implicaria em que a distância de Hamming entre ambas fosse 1, o que é impossível para esse código. Raciocínios análogos permitem determinar que, para esse código, apenas conjuntos de 30, 28, 26 e 24 zeros comuns podem ocorrer envolvendo palavras de peso de Hamming 12, 16, 20 e 24. A Tabela V.1 mostra as diversas possibilidades encontradas para o código *C*(48,24,12):

Tabela V.1: Posições zero em comum para palavras do código *C*(48,24,12).

Distância de Hamming entre palavras de peso P_n	Zeros em comum	Conjuntos encontrados
$D_H(P_{12},P_{12}) = 12$	30	924 x A_{12}
$D_H(P_{12},P_{12}) = 16$	28	7425 x A_{12}
$D_H(P_{16},P_{16}) = 12$	26	3584 x A_{16}
$D_H(P_{12},P_{12}) = 20$		8316 x A_{12}
$D_H(P_{16},P_{16}) = 16$	24	63960 x A_{16}
$D_H(P_{12},P_{16}) = 20$		1275 x A_{20}
$D_H(P_{12},P_{12}) = 24$		630 x A_{12}

A terceira coluna da tabela foi obtida por meio do programa Dup4824 (listagens 1, 2, 3 e 4 no ANEXO II), o qual montou inicialmente uma lista com todas as 2^{24} palavras-código agrupadas por peso e a seguir determinou todas as distâncias de Hamming envolvendo duas palavras de peso 12, duas de peso 16, etc., contando quantos conjuntos tinham a mesma distância de cada vez.

De posse dos dados da tabela é possível montar uma primeira aproximação para o cálculo das probabilidades de sucesso em encontrar um conjunto de informação examinando apenas *n* colunas, de acordo com a equação (3.7), da seção 3.1.2.3.

Do exposto na seção 3.1.2.2, sabe-se que com 37 colunas sempre será possível obter um CI Logo p_{37} = 100%. Aplicando a (3.7), vê-se que para 33 a 36 colunas apenas palavras de peso 12 estarão envolvidas e todos os D_{ij} serão nulos. Logo:

$$p_{36} = A_{12} \times \frac{\binom{36}{36}}{\binom{48}{36}} \times 100\% \;=\; 99{,}999975\% \qquad \text{(V.1)}$$

$$p_{35} = A_{12} \times \frac{\binom{36}{35}}{\binom{48}{35}} \times 100\% \;=\; 99{,}999677\% \qquad \text{(V.2)}$$

$$p_{34} = A_{12} \times \frac{\binom{36}{34}}{\binom{48}{34}} \times 100\% \;=\; 99{,}997741\% \qquad \text{(V.3)}$$

$$p_{33} = A_{12} \times \frac{\binom{36}{33}}{\binom{48}{33}} \times 100\% \;=\; 99{,}988704\% \qquad \text{(V.4)}$$

Entre 32 e 29 colunas, além das palavras de peso 12, também as de peso

16 estão envolvidas, sendo que para 30 e 29 colunas será necessário descontar os conjuntos de 30 zeros comuns devidos à possibilidade de existirem palavras de peso 12 com 30 zeros em comum:

$$p_{32} = \frac{A_{12} \times \binom{36}{32} + A_{16} \times \binom{32}{32}}{\binom{48}{32}} \times 100\% \;=\; 99{,}988704\% \qquad \text{(V.5)}$$

$$p_{31} = \frac{A_{12} \times \binom{36}{31} + A_{16} \times \binom{32}{31}}{\binom{48}{31}} \times 100\% \;=\; 99{,}845972\% \qquad \text{(V.6)}$$

$$p_{30} = \frac{A_{12} \times \left[\binom{36}{30} - \frac{924}{3}\binom{30}{30}\right] + A_{16} \times \binom{32}{30}}{\binom{48}{30}} \times 100\% \;=\; 99{,}535570\% \qquad \text{(V.7)}$$

$$p_{29} = \frac{A_{12} \times \left[\binom{36}{29} - \frac{924}{3}\binom{30}{29}\right] + A_{16} \times \binom{32}{29}}{\binom{48}{29}} \times 100\% \;=\; 98{,}727450\% \qquad \text{(V.8)}$$

Nas equações (V.7) e (V.8) foram descontados os 924 conjuntos de três palavras de peso 12 encontrados pelo programa Dup4824.

A partir de 28 colunas é preciso considerar também as contribuições das palavras de peso 20. Além disso é necessário descontar conjuntos de 28 zeros comuns a palavras de peso 12 e 16, resultantes de conjuntos de palavras de peso 12 com distância mínima de Hamming entre si de 16:

$$p_{28} = \frac{A_{12} \times \left[\binom{36}{28} - \frac{924}{3}\binom{30}{28} - \frac{7425}{2}\binom{28}{28} \right] + A_{16} \times \binom{32}{28} + A_{20} \times \binom{28}{28}}{\binom{48}{28}} \times 100\%$$

$$= 96{,}771499\% \qquad \text{(V.9)}$$

$$p_{27} = \frac{A_{12} \times \left[\binom{36}{27} - \frac{924}{3}\binom{30}{27} - \frac{7425}{2}\binom{28}{27} \right] + A_{16} \times \binom{32}{27} + A_{20} \times \binom{28}{27}}{\binom{48}{27}} \times 100\%$$

$$= 92{,}324439\% \qquad \text{(V.10)}$$

Nas equações (V.9) e (V.10) foram descontados, adicionalmente, os 7425 conjuntos de duas palavras de peso 12 com distância de Hamming 16 entre si encontrados pelo programa Dup4824.

A partir de 26 colunas há a necessidade de considerar conjuntos de 2 palavras de peso 12 com distância de Hamming entre si de 20 e conjuntos de 2 palavras de peso 16 com distância de Hamming entre si de 12:

$$p_{26} = \frac{A_{12} \times \left[\binom{36}{26} - \frac{924}{3}\binom{30}{26} - \frac{7425}{2}\binom{28}{26} - \frac{8316}{2}\binom{26}{26} \right]}{\binom{48}{26}} \times 100\%$$

$$+ \frac{A_{16} \times \left[\binom{32}{26} - \frac{3584}{2}\binom{26}{26} \times 5 \right]}{\binom{48}{26}} \times 100\%$$

$$+ \frac{A_{20} \times \binom{28}{26}}{\binom{48}{26}} \times 100\%$$

$$= 82{,}809631\% \qquad \text{(V.11)}$$

$$p_{25} = \frac{A_{12}\times\left[\binom{36}{25}-\frac{924}{3}\binom{30}{25}-\frac{7425}{2}\binom{28}{25}-\frac{8316}{2}\binom{26}{25}\right]}{\binom{48}{25}}\times 100\,\% + \frac{A_{16}\times\left[\binom{32}{25}-\frac{3584}{2}\binom{26}{25}\times 5\right]}{\binom{48}{25}}\times 100\,\% + \frac{A_{20}\times\binom{28}{25}}{\binom{48}{25}}\times 100\,\%$$

$$= \quad 64{,}113501\,\% \qquad \text{(V.12)}$$

Nas equações (V.11) e (V.12) foram descontados, adicionalmente, os 8316 conjuntos de duas palavras de peso 12 com distância de Hamming 20 entre si e os 3584 conjuntos de duas palavras de peso 16 com distância de Hamming 12 entre si encontrados pelo programa Dup4824. Particularmente, os conjuntos de 26 zeros comuns a duas palavras de peso 16 com distância de Hamming 12 precisam igualmente ser descontados quando contados entre os conjuntos de 28 zeros. Como cada conjunto de 26 zeros pode formar 4 outros de 28 zeros, esses conjuntos foram descontados, adicionalmente, 4 vezes, num total de 5, donde o fator multiplicativo 5 na equação. Note-se porém, que esse é um efeito de segunda ordem , apenas parcialmente verificado pelo programa Dup4824, o qual precisaria ser mais elaborado para também permitir avaliar com precisão esse tipo de superposição.

Finalmente, para o caso de 24 colunas, também as palavras de peso 24 devem ser consideradas, assim como os conjuntos de palavras de peso 16 com distância de Hamming 16 entre si e os conjuntos de 2 palavras de peso 12 com distância de Hamming 24 entre si:

$$p_{24} = \frac{A_{12}\times\left[\binom{36}{24} - \frac{924}{3}\binom{30}{24} - \frac{7425}{2}\binom{28}{24} - \frac{8316}{2}\binom{26}{24} - \frac{630}{2}\binom{24}{24}\right]}{\binom{48}{24}}\times 100\,\% + \frac{A_{16}\times\left[\binom{32}{24} - \frac{3584}{2}\binom{26}{24}\times 5 - \frac{63960}{3}\binom{24}{24}\right]}{\binom{48}{24}}\times 100\,\% + \frac{A_{20}\times\left[\binom{28}{24} - 1275\binom{24}{24}\right]}{\binom{48}{24}}\times 100\,\% + \frac{A_{24}\times\binom{24}{24}}{\binom{48}{24}}\times 100\,\%$$

$$= \quad 34{,}000373\,\% \qquad \text{(V.13)}$$

Na equação (V.13) foram descontados, adicionalmente, os 630 conjuntos de duas palavras de peso 12 com distância de Hamming 24 entre si, os 63960 conjuntos de três palavras de peso 16 com distância de Hamming 16 entre si e os 1725 conjuntos de palavras de peso 12, 16 e 20 com 24 zeros em comum encontrados pelo programa Dup4824, e foram igualmente adicionadas as contribuições das palavras de peso 24.

É ainda importante observar que, numericamente, as contribuições das palavras de peso 24 têm uma influência de ± 1 x 10^{-6} nos resultados e que as de peso 20 têm uma influência de ± 1 x 10^{-3} nos resultados, sendo que as diferenças devidas a efeitos de segunda ordem observadas devem ser buscadas preponderantemente nas sobreposições de conjuntos derivados das palavras de peso 16 e peso 12.

ANEXO VI
Demonstração da validade do critério de parada GMD

A demonstração a seguir está baseada no Teorema 15.3 em (BLAHUT, 1983).

Suponha-se que, dado um vetor recebido analógico **v**, com componentes compreendidas no intervalo [-1,+1], consiga-se encontrar uma candidata **c**, pertencente ao código, que satisfaça a relação:

$$\langle \mathbf{v}, \mathbf{c} \rangle > n - d_{Hmin} \tag{VI.1}$$

A demonstração consiste em se mostrar que para qualquer outra palavra **c'**, também pertencente ao código, valerá sempre:

$$\langle \mathbf{v}, \mathbf{c}' \rangle \leq n - d_{Hmin} \tag{VI.2}$$

Demonstração:

Seja *S* o conjunto de índices das componentes dos vetores, definido por:

$$S \equiv \{ i \mid c_i \neq c_i' \Rightarrow c_i = -c_i' \} \tag{VI.3}$$

E os produtos internos de **v** por **c** e **c'** podem ser colocados na forma:

$$\begin{aligned} \langle \mathbf{v}, \mathbf{c} \rangle &= \sum_{i \notin S} v_i c_i + \sum_{i \in S} v_i c_i = A_1 + A_2 \quad \text{(a)} \\ \langle \mathbf{v}, \mathbf{c}' \rangle &= \sum_{i \notin S} v_i c_i' + \sum_{i \in S} v_i c_i' = A_1 - A_2 \quad \text{(b)} \end{aligned} \tag{VI.4}$$

Mas, como **c** e **c'** diferem em pelo menos d_{Hmin} posições, pode-se escrever:

$$A_1 = \sum_{i \notin S} v_i, c_i \leq n - d_{Hmin} \tag{VI.5}$$

A justificativa para a equação (VI.5) consiste em se observar que o máximo valor de A_1 ocorrerá quando os sinais de v_i e c_i coincidirem. Nessa situação A_1 pode ser escrito como:

$$A_1 = \sum_{i \in \bar{S}} |v_i| \leq n - d_{Hmin} \tag{VI.6}$$

onde $\bar{S}$ representa o conjunto complemento de *S* e contém no máximo $n - d_{Hmin}$ ele-

mentos. Como por hipótese $|v_i| \leq 1$, a equação(VI.5) fica justificada.

Finalmente, como pela hipótese (VI.1) $<\mathbf{v},\mathbf{c}> > n - d_{Hmin}$, resultará necessariamente, pela (VI.4) (a), $A_2 > 0$, o que obriga $A_1 - A_2 \leq n - d_{Hmin}$, o que completa a demonstração.

ANEXO VII
Demonstração da validade do critério de parada do Cone

Inicialmente será mostrado que, dado um código linear de comprimento *n* e distância mínima de Hamming d_{Hmin}, o menor ângulo entre dois vetores quaisquer pertencentes ao código valerá:

$$\delta_{min} = \arccos\left(1 - \frac{2\,d_{Hmin}}{n}\right) \qquad \text{(VII.1)}$$

A equação (VII.2) permite determinar o ângulo entre dois vetores quaisquer $\mathbf{c}_1$ e $\mathbf{c}_2$:

$$\cos(\delta) = \frac{\langle \mathbf{c}_1, \mathbf{c}_2 \rangle}{\|\mathbf{c}_1\| \cdot \|\mathbf{c}_2\|} \quad \Rightarrow \quad \delta = \arccos\left(\frac{\langle \mathbf{c}_1, \mathbf{c}_2 \rangle}{\|\mathbf{c}_1\| \cdot \|\mathbf{c}_2\|}\right) \qquad \text{(VII.2)}$$

onde $\langle \mathbf{c}_1, \mathbf{c}_2 \rangle$ representa o produto interno entre os vetores $\mathbf{c_1}$ e $\mathbf{c_2}$, de comprimento *n*, e é obtido segundo:

$$\langle \mathbf{c}_1, \mathbf{c}_2 \rangle = \sum_{i=1}^{n} \mathbf{c}_{1i} \times \mathbf{c}_{2i} \qquad \text{(VII.3)}$$

e $\|\mathbf{c}\|$ representa o módulo ou norma de **c**, dado por:

$$\|\mathbf{c}\| = \sqrt{\langle \mathbf{c}, \mathbf{c} \rangle} \qquad \text{(VII.4)}$$

e portanto, para o caso de **c** ser uma palavra código de comprimento *n* modulada em BPSK (do inglês *Binary Phase Shift Keying*), como suas *n* componentes podendo assumir apenas os valores +1 ou – 1, resultará sempre:

$$\|\mathbf{c}\| = \sqrt{n}, \qquad \{\forall\, \mathbf{c} \ \textit{com n elementos modulados em BPSK}\} \qquad \text{(VII.5)}$$

Dado um código de bloco linear com distância mínima de Hamming = d_{Hmin}, tomando duas palavras código $\mathbf{c_1}$ e $\mathbf{c_2}$ quaisquer, moduladas em BPSK (do inglês *Binary Phase Shift Keying*), com distância de Hamming *d* entre elas, o cosseno do ângulo entre ambas será, aplicando as equações (VII.2) a (VII.5), e observando que se a distância de Hamming entre as palavras é *d*, então elas coincidem em (*n* – *d*) e diferem em *d* elementos:

$$\cos(\delta) \;=\; \frac{\langle \mathbf{c}_1, \mathbf{c}_2 \rangle}{\|\mathbf{c}_1\| \cdot \|\mathbf{c}_2\|} \;=\; \frac{(n-d)-d}{n} \;=\; 1-\frac{2d}{n} \qquad \text{(VII.6)}$$

Desejando portanto encontrar o menor ângulo possível entre duas palavras código $\mathbf{c}_1$ e $\mathbf{c}_2$ deve-se maximizar o cosseno, o que é obtido minimizando *d*. Assim obtém-se:

$$(\cos(\delta))_{max} \;=\; \cos(\delta_{min}) \;=\; 1-\frac{2d_{Hmin}}{n} \;\Rightarrow\; \delta_{min} \;=\; \arccos\left(1-\frac{2d_{Hmin}}{n}\right) \qquad \text{(VII.7)}$$

Para aplicar o critério do cone será necessário impor que o ângulo formado entre um determinado vetor recebido e uma palavra código candidata, ambos modulados em BPSK (do inglês *Binary Phase Shift Keying*), seja inferior à metade do ângulo mínimo entre duas palavras quaisquer do código, conforme dado pela equação (VII.7).

Para determinar o valor da metade do ângulo mínimo basta aplicar a seguinte identidade trigonométrica:

$$\cos\left(\frac{\delta}{2}\right) \;=\; \sqrt{\frac{\cos(\delta)+1}{2}} \qquad \text{(VII.8)}$$

Portanto, utilizando a equação (VII.7) e aplicando a (VII.6) obtém-se:

$$\cos\left(\frac{\delta_{min}}{2}\right) \;=\; \sqrt{\frac{1-2d_{Hmin}/n+1}{2}} \;=\; \sqrt{1-d_{Hmin}/n} \qquad \text{(VII.9)}$$

De posse das relações (VII.1) a (VII.9), pode-se então demonstrar a validade do critério de parada do Cone:

Dados um vetor analógico recebido **v** e uma palavra código candidata **c**, a aplicação do critério de parada do cone consiste em se verificar se o ângulo α compreendido entre **v** e **c** é inferior a $\delta_{min}/2$, ou seja, se o $\cos(\alpha)$ supera o valor $\cos(\delta_{min}/2)$. Se superar então pode-se afirmar que **v** pertence à região de Voronoi de **c**. O teste portanto pode ser expresso como:

$$\cos(\alpha) \quad = \quad \frac{\langle \boldsymbol{c}, \boldsymbol{v} \rangle}{\sqrt{n} \cdot \|\boldsymbol{v}\|} \quad > \quad \cos\left(\frac{\delta_{min}}{2}\right) \quad = \quad \sqrt{1 - d_{Hmin}/n} \tag{VII.10}$$

ou ainda:

$$\langle \mathbf{c}, \mathbf{v} \rangle \quad > \quad \|\mathbf{v}\| \times \sqrt{n - d_{Hmin}} \tag{VII.11}$$

sendo que o termo sob a raiz no segundo membro da (VII.10) é sempre uma constante dependente dos parâmetros do código em questão.

Neste ponto a demonstração está completa. É entretanto interessante observar formas alternativas para a expressão (VII.11), que podem facilitar sua determinação em certas circunstâncias:

Godoy em (GODOY, 1991) observou que, para o caso particular de **c** = $\mathbf{c}_0$, onde $\mathbf{c}_0$ é a palavra código com todas as componentes nulas, ou iguais a – 1 quando modulada em BPSK (do inglês *Binary Phase Shift Keying*), tem-se que:

$$\langle \mathbf{c}_{0,} \mathbf{v} \rangle \quad = \quad -\sum_{i=1}^{n} v_i \tag{VII.12}$$

e portanto, nesse caso, o critério da (VII.11) se simplifica para:

$$\sum_{i=1}^{n} v_i \quad < \quad -\|\mathbf{v}\| \times \sqrt{n - d_{Hmin}} \tag{VII.13}$$

Utilizando então o conceito definido em 2.1.11 de soma híbrida (GODOY, 1991), como pela definição de soma híbrida ([+]) vale:

$$\|\mathbf{v}[+]\mathbf{c}\| = \|\mathbf{v}\| \tag{VII.14}$$

pode-se também escrever:

$$\sum_{i=1}^{n} SHv_i \quad < \quad -\|\mathbf{v}\| \times \sqrt{n - d_{Hmin}}, \quad \text{com} \quad \mathbf{SHv} \quad = \quad \mathbf{v}[+]\mathbf{c} \tag{VII.15}$$

ANEXO VIII
Demonstração de que minimizar o peso analógico de um vetor equivale a minimizar a soma de suas componentes

Dados **v** e **v'** dois vetores tais que suas componentes sejam:

$$|v_i| \leq 1 \qquad e \qquad |v'_i| \leq 1 \tag{VIII.1}$$

e tais que:

$$\sum_i v'_i < \sum_i v_i \quad \Rightarrow \quad \sum_i (v'_i - v_i) < 0 \tag{VIII.2}$$

deseja-se mostrar que:

$$W(\mathbf{v}') \; < \; W(\mathbf{v}) \tag{VIII.3}$$

com:

$$W(\mathbf{v}') = \sqrt{\sum_i (v'_i + 1)^2} \quad e \quad W(\mathbf{v}) = \sqrt{\sum_i (v_i + 1)^2} \tag{VIII.4}$$

demonstração:

$$\begin{aligned} W(\mathbf{v}')^2 - W(\mathbf{v})^2 &= \sum_i (v'_i + 1)^2 - \sum_i (v_i + 1)^2 \\ &= \sum_i [(v'_i + 1)^2 - (v_i + 1)^2] \\ &= \sum_i [v'^2_i - v_i^2 + 2(v'_i - v_i) + 1 - 1] \\ &= \sum_i (v'_i + v_i)(v'_i - v_i) + 2(v'_i - v_i) \\ &= \sum_i (v'_i - v_i)(v'_i + v_i + 2) \end{aligned} \tag{VIII.5}$$

mas, devido à (VIII.1) tem-se que:

$$0 \; \leq \; (v'_i + v_i + 2) \; \leq \; 4 \tag{VIII.6}$$

e portanto, com a hipótese (VIII.2), que exclui a possibilidade de igualdade na (VIII.6), tem-se que:

$$\sum_i (v'_i - v_i)(v'_i + v_i = 2) \; < \; 0 \tag{VIII.7}$$

o que pela (VIII.5) implica em:

$$W(\mathbf{v}')^2 - W(\mathbf{v})^2 < 0 \quad \Rightarrow \quad W(\mathbf{v}')^2 < W(\mathbf{v})^2 \tag{VIII.8}$$

e, consequentemente:

$$W(\mathbf{v}') < W(\mathbf{v}) \quad \text{(c.q.d)} \tag{VIII.9}$$

ANEXO IX
Demonstração do teorema do critério de parada BGW

Seja:

- C um código de blocos linear de comprimento n e distância mínima de Hamming d_{Hmin}
- **c** uma palavra-código pertencente a C, com componentes com valores pertencentes ao conjunto {-1,+1}.
- **y** um vetor recebido com n componentes analógicas y_i, com $|y_i| < 1 \ \forall \ i$.
- **y'** = **y** [+] **c** o vetor soma híbrida conforme definição em 2.1.11.
- S_+ a soma das d_{Hmin} componentes mais positivas de **y'**.

Teorema:

Se $S_+ < 0$ então $\mathbf{y} \in \mathcal{V}(\mathbf{c})$.

Demonstração:

Da definição de S_+ decorre que qualquer outra soma de d_{Hmin} componentes de **y'** será, necessariamente, menor que S_+. Como por hipótese $S_+ < 0$ então qualquer soma de d_{Hmin} componentes de **y'** será, necessariamente, negativa.

A demonstração do teorema será feita por redução ao absurdo, mostrando que se $\mathbf{y} \notin \mathcal{V}(\mathbf{c})$ então deve existir algum conjunto de d_{Hmin} componentes de **y'** cuja soma é positiva, o que contraria a hipótese.

Suponha-se que $\mathbf{y} \notin \mathcal{V}(\mathbf{c})$.

Então de acordo com a equação (2.14), na página 25

$$(\mathbf{y'} = \mathbf{y}\ [+]\ \mathbf{c}) \notin \mathcal{V}(\mathbf{c_0}),$$

e deve existir uma outra palavra-código candidata **c'**, diferente de **c** e pertencente ao código, tal que:

$$\mathbf{y} \in \mathcal{V}(\mathbf{c'}) \ \text{ e } \ (\mathbf{y''} = \mathbf{y}\ [+]\ \mathbf{c'}) \in \mathcal{V}(\mathbf{c_0}).$$

De acordo com a equação (2.16), na página 27, tem-se então que:

$$\left(\sum_{i=1}^{n} y''_i\right)_{min} \Rightarrow \sum_{i=1}^{n} y''_i < \sum_{i=1}^{n} y'_i = S_+ \qquad \text{(IX.1)}$$

Da definição de soma híbrida tem-se que:

$$c_i \neq c'_i \Rightarrow c_i = -c'_i \Rightarrow y'_i = -y''_i \qquad \text{(IX.2)}$$

Como tanto **c** como **c'** pertencem ao código, e este tem distância mínima de Hamming d_{Hmin}, deve existir um conjunto D de índices i, de cardinalidade igual ou superior a d_{Hmin}, para o qual $y'_i = -y''_i$. Esse conjunto será definido como:

$$D = \{i \mid y'_i = -y''_i\}, |D| \geq d_{Hmin} \qquad \text{(IX.3)}$$

logo para $i \notin D$ tem-se que $y'_i = y''_i$.

Aplicando a (IX.2) e a (IX.3) tem-se que:

$$\sum_{i \in D} y''_i = -\sum_{i \in D} y'_i \qquad \text{(IX.4)}$$

$$\sum_{i \notin D} y''_i = \sum_{i \notin D} y'_i \qquad \text{(IX.5)}$$

A equação (IX.1) pode então ser reescrita como:

$$\sum_{i=1}^{n} y''_i = \sum_{i \in D} y''_i + \sum_{i \notin D} y''i < \sum_{i=1}^{n} y'_i = \sum_{i \in D} y'_i + \sum_{i \notin D} y'i \qquad \text{(IX.6)}$$

sendo que após a eliminação dos termos comuns aos dois membros da desigualdade, fazendo uso da (IX.5) acima, resulta:

$$\sum_{i \in D} y''_i < \sum_{i \in D} y'_i \qquad \text{(IX.7)}$$

e utilizando a (IX.4) obtém-se:

$$-\sum_{i \in d} y'_i < \sum_{i \in D} y'_i \qquad \text{(IX.8)}$$

mas a (IX.8) acima implica que:

$$\sum_{i \in D} y'_i > 0 \qquad \text{(IX.9)}$$

ou seja, se $\mathbf{y} \notin V(\mathbf{c})$, conclui-se que deve ser possível encontrar pelo menos d_{Hmin} componentes de **y'** cuja soma resulta positiva, mas, como visto acima, isso contradiz a hipótese de que $S_+ < 0$, logo

$\mathbf{y} \in V(\mathbf{c})$ c.q.d.

REFERÊNCIAS

ARNOLD, C. Barry, BALAKRISHNAM, N., NAGARAJA, H. N., **A first Course in Order Statistics**. SIAM Classics In Applied Mathematics, Vol. 54, Philadelphia, 2008,

BARROS, Dulte. José de, GODOY, Walter. Jr., WILLE, Emilio C. G, **A New Approach to the Information Set Decoding Algorithm**. Computer Communications, Vol 20, pp. 302 – 308, 1997.

_____ . **Soft-Decision Decodierung langer Blockcodes mit Informationsmengen**. Tese de Doutorado. Technische Universität Darmstadt, Darmstadt, 2000.

BLAHUT, Richard E. **Introduction to Error Correcting Codes**. Cambridge University Press, New York, 1983.

_____ . **Algebraic Codes for Data Transmission**. Cambridge University Press, New York, 2003.

BRANTE, G. G. de O., MUNIZ, D. N., GODOY, W. Jr. **Information Set Based Soft-Decoding Algorithm for Block Codes.** IEEE Latin America Transactions, Vol 9, No. 4, 2011.

CHASE, D. **A Class of Algorithms for Decoding Block Codes with Channel Measurement Information**. IEEE Transactions on Information Theory, Vol IT-18 No 1, Jan. 1972.

CLARK, George C. Jr., CAIN, J. Bibb. **Error-Correction Coding for Digital Communications.** Plenum Press, New York, 1981.

COFFEY, John T., GOODMAN, Rodney M. **The Complexity of Information Set Decoding**. IEEE Transactions on Information Theory, Vol 36 No 5, Sept. 1990.

COSTA NETO, Pedro Luiz, **Estatística**. Editora Edgar Blücher Ltda. 1977, São Paulo.

DAVID, H. A, NAGARAJA, H. N. **Order Statistics**, John Wiley & Sons, 3rd ed. 2003, New York.

BALAKRISHNAM, N., RAO, C. R. **Handbook of Statistics – Vol 16 – Order Statistics – Theory and Methods**, Elsevier Science B. V., 1998

DORSCH, B. G. **A decoding Algorithm for Binary Block Codes and J-ary Output Channels**. IEEE Transactions on Information Theory, IT–20, pp. 391 – 394, May 1974.

FORNEY, D. G. **Generalized Minimum Distance Decoding**. IEEE Transactions on Information Theory, Vol. IT–12, No. 2, pp. 125 – 131, April 1966.

_____ , **Concatenated Codes**. Cambridge, Massachusets, MIT Press, 1966.

FOSSORIER, Marc P. C. **Decoding of linear block codes based on order statistics**. (Phd Dissertation), 202 f. University of Hawaii, Hawaii, 1994

_____ , LIN, Shu, **Soft-decision Decoding of Linear Block Codes Based on Order Statistics**. IEEE Transactions on Information Theory, IT– 41, No 5, pp. 1379 – 1396, Sept. 1995.

_____ , _____ , SNYDERS, J. **Reliability based syndrome decoding of linear block codes**. IEEE Transactions on Information Theory, IT – 44, No 1, pp. 388 – 398, Jan. 1998.

_____ . **Reliability based soft-decision decoding with iterative information set reduction.** IEEE Transactions on Information Theory, IT– 48, No 12, pp. 3101 – 3106, Dec. 2002.

FUNG, W. H. C., GORTAN, A., GODOY, W. Jr., **A Review Study on Image Digital Watermarking.** Tenth International Conference on Networks, St. Maarten, 2011.

GODOY, Walter Jr. **Esquemas de modulação codificada com códigos de bloco**. Editora Cefet Pr, 1991.

_____ , WILLE, Emilio C. G., **Proposal of Sub-optimum Decoding Algorithm with a Bound of Voronoi Region V(C_o)**. Computer Communications, Vol 21, pp. 736–740, 1998.

_____ , _____ , CUNHA, J. A. T., **Adaptive Decoding of Binary Linear Block Codes Using Information Sets and Erasures.** Third International Conference on Communication Theory, Reliability and Quality of Service, Athens/Glyfada, Greece, 2010a.

_____ , _____ , JASINSKI, R. P. **A Simple Algorithm for Decoding of Binary Block Codes Based on Information Sets.** International Communications Conference, Cape Town, South Africa, 2010b (ICC2010).

GORTAN, A. **Análise Comparativa de Limitantes da Região de Voronoi**. 2002 61 f. Monografia (Especialização em Teleinformática e Redes de Computadores) Universidade Tecnológica Federal do Paraná, Curitiba, 2002.

_____ , JANSINSKI, R. P., GODOY, W. Jr., PEDRONI, Volnei A. **Hardware Friendly Implementation of Soft Information Set Decoders.** International Telecommunications Symposium, Manaus, 2010a (ITS2010).

_____ , JANSINSKI, R. P., GODOY, W. Jr., PEDRONI, Volnei A. **Achieving Near-MLD Performance with Soft Information-Set Decoders Implemented in FPGAs.** 2010 Asia Pacific Conference on Circuits and Systems, Kuala Lumpur, Malaysia, 2010b (APCCAS2010).

_____ , JANSINSKI, R. P., GODOY, W. Jr., PEDRONI, Volnei A. **A Very Efficient, Hardware Oriented Acceptance Criterion for Soft Information-Set Decoders.** International Communications Conference, Otawa, Canada, 2012 (ICC2012 – submitted).

HUFFMAN, W. Cary, PLESS, Vera. **Fundamentals of Error Correcting Codes**. Cambridge University Press, New York, 2003.

JASINSKI, R. P., PEDRONI, V. A., GORTAN, A., GODOY, W. Jr. **GF(2) Matrix Inversion in Hardware with O(N) Time Complexity.** International Conference on Reconfigurable Computing and FPGAs, Cancun, Mexico, 2010 (ReConFig2010).

LIN, Shu, COSTELLO Jr., Daniel J., **Error Control Coding**, 2nd ed. Person Prentice Hall, Upper Saddle River, New Jersey, 2004.

MEYER, Carl D. **Matrix Analysis and Applied Linear Algebra**. SIAM, 2000.

PRANGE, E. **The Use of Information Sets in Decoding Cyclic Codes**. IRE Transactions on Information Theory, Vol. IT-8, pp. 5-9, Sept. 1962.

TAIPALE, Dana J., PURSLEY, Michael B. **An Improvement to Generalized Minimum Distance Decoding**. IEEE Transactions on Information Theory, IT–37, pp. 167–172, Jan. 1991.

WARREN, Henry S. **Hacker's Delight**. Addison Wesley, Boston, MA, 2003.

WIDROW, Bernard, KOLLÁR, István, **Quantization Noise – Round off Error in Digital Computation, Signal Processing, Control and Communications.** Cambridge University Press, New York, 2008.