FÁBIO RIZENTAL COUTINHO

# DECODIFICAÇÃO POR DECISÃO SUAVE DE CÓDIGOS DE REED-SOLOMON NA PRESENÇA DE SÍMBOLOS APAGADOS

Proposta de Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, Setor de Tecnologia, Universidade Federal do Paraná.
Orientador: Prof. Dr. Evelio Martín García Fernández

CURITIBA

2007

FÁBIO RIZENTAL COUTINHO

# DECODIFICAÇÃO POR DECISÃO SUAVE DE CÓDIGOS DE REED-SOLOMON NA PRESENÇA DE SÍMBOLOS APAGADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Engenharia Elétrica, Setor de Tecnologia, Universidade Federal do Paraná.
Orientador: Prof. Dr. Evelio Martín García Fernández

CURITIBA

2007

FÁBIO RIZENTAL COUTINHO

# DECODIFICAÇÃO POR DECISÃO SUAVE DE CÓDIGOS DE REED-SOLOMON NA PRESENÇA DE SÍMBOLOS APAGADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Engenharia Elétrica, Setor de Tecnologia, Universidade Federal do Paraná.
Orientador: Prof. Dr. Evelio Martín García Fernández

CURITIBA

2007

FÁBIO RIZENTAL COUTINHO

# DECODIFICAÇÃO POR DECISÃO SUAVE DE CÓDIGOS DE REED-SOLOMON NA PRESENÇA DE SÍMBOLOS APAGADOS

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Paraná, pela Comissão formada pelos professores:

Orientador: Prof. Dr. Evelio Martín García Fernández
Departamento de Engenharia Elétrica, UFPR

Prof. Dr. Bartolomeu Ferreira Uchôa Filho
Departamento de Engenharia Elétrica, UFSC

Prof. Dr. Eduardo Parente Ribeiro
Departamento de Engenharia Elétrica, UFPR

Prof. Dr. Marcelo de Oliveira Rosa
Departamento de Engenharia Elétrica, UFPR

Curitiba, 21 de setembro de 2007

# SUMÁRIO

# RESUMO

A busca por uma melhoria do desempenho de erro de algoritmos de decodificação algébrica por decisão suave de códigos de Reed-Solomon de altas taxas motivou o desenvolvimento desta dissertação numa tentativa de se determinar a real capacidade de correção de erros da decodificação algébrica por decisão suave na presença de apagamentos. É mostrado através de simulações como é possível conseguir aumento de desempenho significativo quando os bits não confiáveis recebidos na saída do canal são considerados como bits apagados. Uma estratégia de atribuição de multiplicidades através do mapeamento das probabilidades a posteriori do canal é desenvolvida no sentido de se atribuir multiplicidades iguais para símbolos com o mesmo padrão de bits apagados. Na direção oposta é mostrado através de simulações que tentar declarar como apagamentos símbolos não confiáveis recebidos do canal não leva a nenhum ganho de desempenho podendo inclusive levar a perdas de desempenho.

# ABSTRACT

The search to increase the error performance of algebraic soft-decision decoding of high rate Reed-Solomon codes motivates the development of this work in an attempt to determine the ultimate error-correcting capabilities of algebraic soft-decision decoding with erasures. It is shown trough simulations that a significant performance improvement can be obtained when the unreliable bits received from channel output are declared as erased bits. A multiplicity assignment strategy is developed through the mapping of the a posteriori channel probabilities in order to assign equal multiplicity for symbols with the same erased bits patterns. In the opposite way, it is shown through simulations that trying to declare the unreliable symbols received from the channel as erasures does not lead to any performance gain and in some cases it would lead to a decrease of the decoder performance.

# CAPÍTULO 1

# INTRODUÇÃO

Em junho de 1960, Irving Reed e Gustave Solomon publicaram o artigo "Polynomial codes over certain finite fields"[1], onde descreveram uma nova classe de códigos corretores de erros que se baseam no mapeamento de um espaço vetorial de dimensão $k$ sobre um corpo finito $\mathbb{F}$ em um espaço vetorial de dimensão maior que $k$ sobre o mesmo corpo. Através desse mapeamento a distância de Hamming entre cada palavra código tornou-se a maior possível em relação aos demais códigos aumentando assim a capacidade de correção de erros do código, além do fato de tratar os símbolos como grupos de bits, o que tornou o código particularmente bom para casos de erros em bits consecutivos. Então em 1968, Elwyn Berlekamp desenvolveu um algoritmo de decodificação que tornou possível a implementação dos códigos de Reed-Solomon (RS), como ficaram conhecidos atualmente, em aplicações práticas. Hoje, aparelhos de reprodução de DVD´s e CD´s, por exemplo, utilizam uma variação do algoritmo de Berlekamp para corrigir erros consecutivos de até 4000 bits; o que equivale a um risco na superfície de um CD de 2.5 milímetros de comprimento. Além de sistemas de armazenamento digital, podemos encontrá-los sendo usados em comunicações via satélites, comunicação sem fio de terceira geração (3G) e CDMA2000.

Embora recentes descobertas, como os códigos Turbo, por exemplo, e outros tenham atraído a atenção de vários pesquisadores, os códigos de Reed-Solomon são empregados em cerca de 75% dos circuitos de correção de erro em operação hoje.

Na recepção ou leitura de uma seqüência de dados binários modulados corrompidos por ruído, pode-se tomar dois tipos de decisão para escolher a seqüência mais provável transmitida ou gravada: decisão abrupta e decisão suave.

Na decisão abrupta, a saída do demodulador ou leitor é quantizada para o símbolo mais próximo, assim a decisão é feita baseada apenas no símbolo transmitido mais provável.

Devido a esse tipo de abordagem, a decisão abrupta acaba não usando toda a informação disponível no sinal recebido, deixando de lado o objetivo principal de um sistema de comunicação ou armazenamento que é utilizar o máximo das informações recebidas do canal.

Na decisão suave é levada em conta a confiabilidade, a probabilidade e a verossimilhança da informação recebida. Ou seja, nesse tipo de decisão, além da informação do símbolo mais provável transmitido, temos também o grau de confiabilidade que podemos ter dessa decisão. De maneira geral pode-se dizer que a decodificação por decisão suave pode ter até 3dB de ganho de codificação em relação a uma decodificação por decisão abrupta.

Apesar da sua grande aplicabilidade, a decodificação de códigos de Reed-Solomon ainda é feita utilizando decisão abrupta o que não explora toda a capacidade de correção do código.

Nos últimos anos novos avanços ocorreram nesse campo iniciando-se em 1997 quando Madhu Sudan utilizando-se da conseqüência do teorema de Bezout [2] mostrou em [3] que era possível fazer a decodificação por decisão abrupta de códigos de Reed-Solomon além da capacidade de correção clássica $(n-k)/2$ utilizando-se um método de decodificação por lista e abordando-o como um problema de interpolação de duas variáveis. Assim, para um código de comprimento $n$ e dimensão $k$, o algoritmo apresentado na referência [3] produz todas as palavras-código cuja distância do vetor recebido não excede $n-\sqrt{2kn}$. Uma análise mais cuidadosa mostrou que o método de decodificação apresentado em [3] era assintóticamente melhor que a decodificação padrão de até $(n-k)/2$ erros se a taxa do código (ou seja, $k/n$) fosse menor do que $1/3$ . Contudo códigos com taxas menores que $1/3$ não são de interesse do ponto de vista prático. Em 1999, Venkatesan Guruswami juntamente com Madhu Sudan [4] conseguiram melhorar o método de decodificação proposto anteriormente mostrando que poderiam se corrigir ainda mais erros se fizessem com que a interpolação em cada ponto ocorresse não apenas uma vez, como ocorria anteriormente, e sim $m$ vezes, onde $m$ é uma variável inteira arbitrária. Para $m \to \infty$ , o algoritmo de decodificação por lista de [4] corrige até $\lfloor n-\sqrt{nk} \rfloor$ erros, o que é melhor que o valor

$(n-k)/2$, e desta vez para todas as taxas $k/n$. Todavia, do ponto de vista prático, este novo aperfeiçoamento na decodificação tinha uma deficiência: o desempenho assintótico demonstrado em [4] era reduzido a zero quando aplicado a códigos de Reed-Solomon de altas taxas e comprimento finito. Logo depois, Ralph Koetter e Alexander Vardy [5], adaptaram a nova técnica de decodificação de Guruswami-Sudan mostrando como as múltiplas interpolações, ou multiplicidades, de [4] poderiam ser utilizadas para que se alcançasse a decodificação por decisão suave de códigos de Reed-Solomon, produzindo assim um algoritmo algébrico que permite utilizar todas as informações probabilísticas disponíveis no receptor. No algoritmo apresentado na referência [5] apresentou-se um esquema de atribuição de multiplicidades assintóticamente ótimo que maximiza a taxa de transmissão e que pode reduzir a probabilidade de erros a valores muito baixos desde que o comprimento do código tenda ao infinito. Entretanto é mostrado que o algoritmo de [5] já apresenta um substancial ganho de desempenho na decodificação de códigos RS de altas taxas e comprimento finito razoavelmente grandes (como por exemplo, o código RS (255,239) sobre um corpo de Galois $\mathbb{F}(2^8)$). Devido a esse aumento significativo de desempenho obtido, os algoritmos de decodificação algébrica por decisão suave, ou algoritmos de decodificação ASD (*Algebraic Soft Decision*), como ficaram conhecidos, despertaram um grande interesse de pesquisas de toda a comunidade acadêmica da área. Do ponto de vista prático, várias técnicas [6],[7],[8],[9],[10] e [11] foram propostas tentando conduzir a decodificação ASD no caminho de uma implementação que venha a ser uma alternativa à decodificação convencional de Berlekamp-Massey muito utilizada em aplicações práticas como CD, DVD, leitores de códigos de barras, telefones celulares, comunicação via satélite, televisão digital e modems DSL.

Entretanto ainda não se descobriu uma técnica de atribuição de multiplicidades ótima para a decodificação ASD. Nas referências [12],[13] e [14] apresentam-se esquemas de atribuição de multiplicidades otimizados para códigos RS de comprimento finito, através de algoritmos númericos.

Um apagamento é um erro no qual a localização do erro é conhecida, porém o valor é desconhecido. Eles podem acontecer de várias maneiras. Em alguns receptores, o sinal

correspondente a um ou mais símbolos recebidos é analisado e se estiver fora de limites aceitáveis, aquele símbolo(s) é declarado estar apagado.

Na referência [15] é caracterizado o raio da decodificação ASD sobre canais com apagamentos e o canal binário simétrico, e é investigado qual seria a estratégia de atribuição de multiplicidades para esses tipos de canais, além de mostrar como a decodificação ASD pode obter desempenhos melhores que a decodificação convecional de Berlekamp-Massey sobre canais com apagamentos.

Com o intuito de se buscar cada vez mais melhoras no desempenho da decodificação algébrica por decisão suave de códigos de Reed-Solomon de dimensão finita e altas taxas buscou-se, neste trabalho, implementar uma técnica para apagar bits não confiáveis no sentido de se obter ganhos de desempenho sobre as técnicas de decodificação ASD propostas atualmente. Baseado no trabalho de [15], propõe-se um meio de se apagar os bits não confiáveis e atribuir a mesma multiplicidade para os símbolos que têm a mesma configuração de bits apagados. A validade e ganhos de desempenho deste método são confirmados através da implementação de simulações. A proposição de se apagar os símbolos não confiáveis recebidos do canal, também é investigada através de simulações, entretanto mostra-se que devido à grande quantidade de possibilidades de símbolos que podemos ter em virtude de um apagamento não se consegue melhora no desempenho e sim uma piora do mesmo.

O resto dessa dissertação está organizado da seguinte maneira: no Capítulo 2, após uma breve introdução sobre a construção clássica de códigos de Reed-Solomon, comenta-se sobre os tipos de algoritmos de decodificação que podemos ter e em seguida é mostrada a idéia motivante que deu origem ao novo algoritmo de decodificação algébrica por lista de decisão abrupta de Guruswami-Sudan, através de exemplos informais e didáticos. No Capítulo 3, é feita uma revisão minuciosa nos conceitos fundamentais e de nosso interesse que envolvem polinômios de duas variáveis. São apresentados os Teoremas de Interpolação e Fatoração, fundamentais para a decodificação de GS, continuando-se na seqüência explorando em detalhes aspectos dessa decodificação, como sua capacidade de correção, número de polinômios retornados e algoritmos de interpolação e fatoração. No Capítulo

4, após uma breve introdução na decodificação por decisão suave, partimos para a revisão de conceitos de canais discretos sem memória e suas consequências nas probabilidades condicionais que serão os conceitos básicos para a introdução da matriz de confiabilidade. Uma vez visto estes conceitos iniciais partimos para uma análise mais aprofundada do algoritmo de Koetter-Vardy para decodificação ASD, bem como as principais mudanças que ocorreram nas fases de interpolação e fatoração do algoritmo de GS para que se atinja a decisão suave. No final do capítulo, apresenta-se o potencial do algoritmo de decodificação ASD sobre os outros algoritmos de decodificação convencional por decisão abrupta através de simulações de desempenho. No capítulo 5, é feita uma análise do desempenho do algoritmo de GS quando se tem apagamentos presentes mostrando como o mesmo tem seu desempenho degenerado quando o número de apagamentos vai crescendo. O canal binário simétrico com apagamentos é caracterizado e seu desempenho analisado para que mais adiante se possam extrapolar os resultados obtidos para um canal Gaussiano com erros e apagamentos. E assim por fim propor um método de atribuição de multiplicidades para o algoritmo de KV quando se tem a presença de apagamentos de bits. Os resultados obtidos para apagamentos de bits então são extendidos para apagamentos de símbolos. O desempenho da decodificação com o método de atribuição de multiplicidades proposto é obtido através de simulação computacional, mostrando-se como é possível ter ainda maior ganho de desempenho na decodificação ASD quando são declarados bits não confiáveis como bits apagados e também que se pode obter o mesmo desempenho ou até perda de desempenho se tentarmos apagar símbolos não confiáveis. As conclusões do trabalho bem como sugestões para trabalhos futuros são apresentadas no Capítulo 6.

# CAPÍTULO 2

# VISÃO GERAL DA TEORIA

## 2.1 Corpos de Galois

Os códigos de Reed-Solomon, em geral, são codificados e decodificados através do uso de matemática sobre corpos finitos. Os corpos finitos foram descobertos por Evariste Galois, e por isso, é geralmente utilizado o termo corpo de Galois quando nos referenciamos a eles. Um corpo finito com $q$ elementos é geralmente denominado como $GF(q)$. O número de elementos num corpo finito, $q$, deve ser da forma $p^m$, onde $p$ é um número inteiro primo e $m$ é um inteiro positivo [16],[17]. Um corpo de Galois $GF(q)$ sempre contém pelo menos um elemento chamado de elemento primitivo e denominado usualmente por $\alpha$. A ordem de um elemento $\alpha$ de um $GF(q)$ é o menor inteiro positivo $n$, tal que, $\alpha^n = 1$. O elemento primitivo $\alpha$ deve obrigatoriamente ter ordem igual a $(q-1)$. Assim podemos perceber que as $(q-1)$ potências de $\alpha$, ou seja, $\{1, \alpha, \alpha^2, ..., \alpha^{q-2}\}$, devem ser distintas, formando assim os elementos diferentes de zero que compõem o corpo de Galois $GF(q)$. A representação dos elementos do corpo de Galois na forma $\{0, 1, \alpha, \alpha^2, ..., \alpha^{q-2}\}$ é chamada de representação exponencial e é utilizada principalmente quando precisamos multiplicar dois elementos do corpo $GF(q)$ pois nela basta utilizarmos a regras de multiplicação à qual estamos acostumados: $\alpha^x \cdot \alpha^y = \alpha^{x+y}$.

Já para o caso da adição é necessário utilizar uma outra forma de representação dos elementos, chamada de representação polinomial. Nessa representação o elemento primitivo, no caso $\alpha$, deverá ser a raiz de um polinômio primitivo de grau $m$, $p(x)$. Considere um corpo $GF(2^3)$, se $p(x) = x^3 + x + 1$ for um polinômio primitivo binário (com coeficientes sobre $GF(2)$), e $\alpha$ o elemento primitivo e por conseqüência a raiz de $p(x)$, isto implica que $\alpha^3 + \alpha + 1 = 0$ ou $\alpha^3 = \alpha + 1$ (não existe subtração na aritmética binária). Usando esta última relação podemos obter a representação polinomial dos demais elementos do corpo, como mostra a tabela:

Tabela 2.1

| Representação Exponencial | Representação Polinomial | Representação Binária |
|---|---|---|
| 0 | 0 | 000 |
| 1 | 1 | 100 |
| $\alpha$ | $\alpha$ | 010 |
| $\alpha^2$ | $\alpha^2$ | 001 |
| $\alpha^3$ | $\alpha + 1$ | 110 |
| $\alpha^4$ | $\alpha^2 + \alpha$ | 011 |
| $\alpha^5$ | $\alpha^2 + \alpha + 1$ | 111 |
| $\alpha^6$ | $\alpha^2 + 1$ | 101 |

Então para calcular $\alpha+\alpha^5$ em $GF(8)$, basta substituir a representação exponencial para a representação polinomial, fazer a soma e reconverter para a representação exponencial:

$$\alpha + \alpha^5 = \alpha + \alpha^2 + \alpha + 1 = \alpha^2 + 1 = \alpha^6$$

## 2.2 Construção Clássica de Códigos de Reed-Solomon

Em 1960, Irving Reed e Gus Solomon, publicaram um artigo [1] que mostrava a descoberta de uma nova classe de códigos, que mais tarde ganhou a alcunha de seus descobridores. Assim o método proposto em [1] foi chamado de construção clássica de códigos de Reed-Solomon. Esta construção é muito importante e leva a uma série de generalizações que serão mostradas mais adiante.

Neste artigo, Reed e Solomon mostram que para se construir ou codificar basta que se tomem quaisquer $k$ símbolos de informação (equação 2.1) e transforme-os em coeficientes de um polinômio de grau até $k-1$ (equação 2.2) . Então determina-se o valor deste polinômio para uma quantidade fixa de $n$ elementos distintos de um corpo (equação 2.3). Os valores encontrados formarão a palavra código (equação 2.4).

$$u_0, u_1, ..., u_{k-1} \tag{2.1}$$

$$\Downarrow$$

$$f(x) = u_0 + u_1 x + ... + u_{k-1} x^{k-1} \tag{2.2}$$

$$\Downarrow$$

$$c_1 = f(x_1), c_2 = f(x_2), ..., c_n = f(x_n) \tag{2.3}$$

$$\Downarrow$$

$$(c_1, c_2, ..., c_n) \tag{2.4}$$

O número de símbolos de informação $k$ é chamado de dimensão do código. Este termo vem do fato de que as palavras código de Reed-Solomon formam um espaço vetorial de dimensão $k$ sobre o corpo $GF(q)$, enquanto que $n$ é denominado de comprimento do código. Assim, quando nos referimos a códigos de Reed-Solomon, caracterizamo-os pelo seu comprimento $n$ e dimensão $k$, utilizando a forma RS$(n, k)$.

Colocando de maneira formal temos:

**Definição 2.2.1** *Seja $\alpha$ um elemento primitivo de um corpo de Galois $\mathbb{F}(q^m)$, com $n = q^m - 1$. Seja $\boldsymbol{u} = u_0, u_1, ..., u_{k-1} \in \mathbb{F}(q^m)^k$ um vetor de mensagem e $f(x) = u_0 + u_1 x + ... + u_{k-1}x^{k-1} \in \mathbb{F}(q^m)[x]$ o seu polinômio associado. Então o processo de codificação é definido pelo mapeamento $\rho : f(x) \mapsto \boldsymbol{c}$:*

$$(c_0, c_1, ..., c_{n-1}) \triangleq \rho(f(x)) = (f(1), f(\alpha), f(\alpha^2), ..., f(\alpha^{n-1})). \tag{2.5}$$

O conjunto $1, \alpha, \alpha^2, ..., \alpha^{n-1}$ formado por $n$ elementos distintos de $\mathbb{F}(q^m)$, menos o elemento 0, é denominado de conjunto suporte.

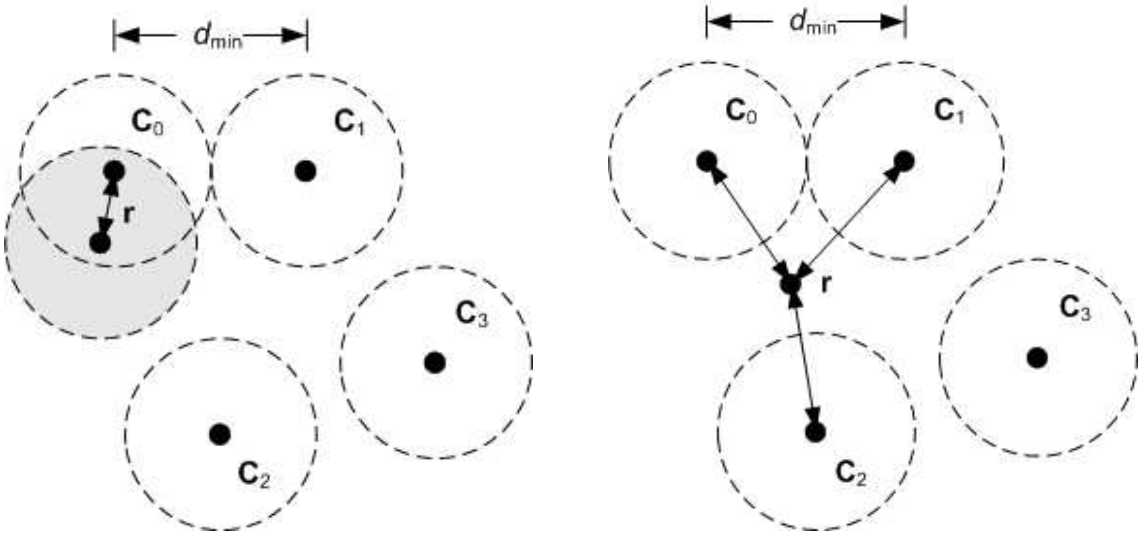
## 2.3 Paradigmas da Decodificação de Códigos de Reed-Solomon

Como visto na seção anterior cada palavra de um código de Reed-Solomon $C(n, k)$ nada mais é que os valores que o polinômio mensagem $f(x)$ assume para um conjunto de $n$ valores distintos de $x$. Então para resolver o problema de decodificação de códigos de Reed-Solomon basta encontrar a equação $f(x)$ que interpole os $n$ pontos $(x, y)$ da palavra código recebida. Como o grau de $f(x)$ é menor que $k$ é possível determinar $f(x)$ utilizando

apenas $k$ pontos dos $n$ que formam a palavra código, dessa forma utilizando interpolação numa equação de uma variável podem-se corrigir até $n-k$ símbolos apagados ou $(n-k)/2$ erros.

A decodificação vista acima também pode ser interpretada de outra maneira; o decodificador busca por palavras códigos na esfera de Hamming de raio $t_0 = (n-k)/2$ com centro no vetor recebido. Se no interior desta esfera se encontrar somente uma palavra código, esta será a palavra transmitida e a decodificação terá sido realizada com sucesso. Caso contrário haverá uma falha na decodificação. Este paradigma de decodificação é denominado de decodificação por distância limitante, (ver figura 2.1(a)), no qual se baseia a maior parte dos algoritmos de decodificação por decisão abrupta de códigos de Reed-Solomon, como o algoritmo de Berlekamp-Massey [18], por exemplo. Contudo podemos também utilizar um decodificador por lista (figura 2.1(b)), que irá retornar todas as palavras código que estão a uma distância de Hamming $r$ da palavra recebida. Neste caso podemos tentar aumentar o raio de decodificação $t_0$, e é possível que dentro da esfera de decodificação tenhamos mais de uma palavra código e consequentemente o decodificador irá falhar. Entretanto se analisarmos a probabilidade de que a esfera contenha mais de uma palavra, ao invés da possibilidade, pode-se chegar numa conclusão diferente. Considere, por exemplo, o código RS(32,8), sobre o corpo $\mathbb{F}(2^5)$, que tem raio de decodificação $t_0 = 12$. Se aumentarmos o raio de decodificação para $t = 13$, e a palavra transmitida sofrer 13 erros é possível que a esfera de decodificação contenha mais de uma palavra código: a palavra transmitida e uma outra palavra código qualquer com distância 12 ou 13 da palavra recebida. Entretanto se assumirmos que todos os padrões de 13 erros são igualmente prováveis, pode ser mostrado que a probabilidade disso acontecer é $2,08437 \times 10^{-12}$. Ou seja, pode-se dizer que o código é capaz de corrigir praticamente todos os padrões de 13 erros, apesar dele ter uma capacidade de correção convencional de somente 12 erros.

Entretanto mesmo sabendo que é possível atingir uma capacidade de correção maior que a convencional, não se consegue atingir essa capacidade utilizando a estratégia de interpolação numa equação de uma variável descrita inicialmente.




(a) Decodificação por Distância Limitante (b) Decodificação por Lista

**Figura 2.1:** *Paradigmas de Decodificação: Em (a) a palavra código deve estar na mesma esfera de Hamming para que ocorra sucesso na decodificação. Já em (b) todas as palavras códigos que estão a uma distância de Hamming r da palavra recebida são retornadas.*

## 2.4 Ultrapassando a Capacidade de Correção Usual

Para ultrapassar a capacidade de correção usual continua-se utilizando a interpolação, porém agora passamos a usar uma equação de duas variáveis ao invés de uma variável apenas. Essa foi a solução encontrada por Sudan em [3]. Neste trabalho, Sudan mostra que a palavra recebida pode ser representada pela equação algébrica $y - f(x)$ de grau $k - 1$ e a partir daí ele se baseia no teorema de Bézout [2]:

**Teorema 2.4.1** *Duas curvas algébricas de graus d e $\delta$ se interceptam em $d\delta$ pontos, e não podem concorrer em mais de $d\delta$ pontos a não ser que as equações que as definam contenham um fator comum.*

Então se pudermos construir uma equação de duas variáveis $Q(x, y) \in \mathbb{F}_q[x, y]$ que tenha grau $\delta$ e intercepte $y - f(x)$ em mais do que $(k - 1)\delta$ pontos, incluindo pontos no $\infty$, então $y - f(x)$ pode ser recuperado como um fator de $Q(x, y)$.

Para se compreender melhor como isso acontece, considere o seguinte exemplo. Suponha que a dimensão do código seja dois ($k = 2$), dessa maneira os polinômios mensagens serão simples retas, ou melhor dizendo, polinômios de grau um do tipo $f(x) = ax + b$, onde $a$ e $b$ pertencem ao corpo dos reais. Digamos que tenhamos 14 pontos recebidos e que

queremos encontrar todas as retas (polinômios mensagem de grau um) que passam por pelo menos cinco desses pontos. Para fazer isso Sudan mostra que primeiramente deve-se obter um polinômio de duas variáveis de grau menor que o número de interseções, cinco, e que passe por todos os 14 pontos, e ele prova que esse polinômio sempre vai existir. Então podemos, por exemplo, encontrar o polinômio:

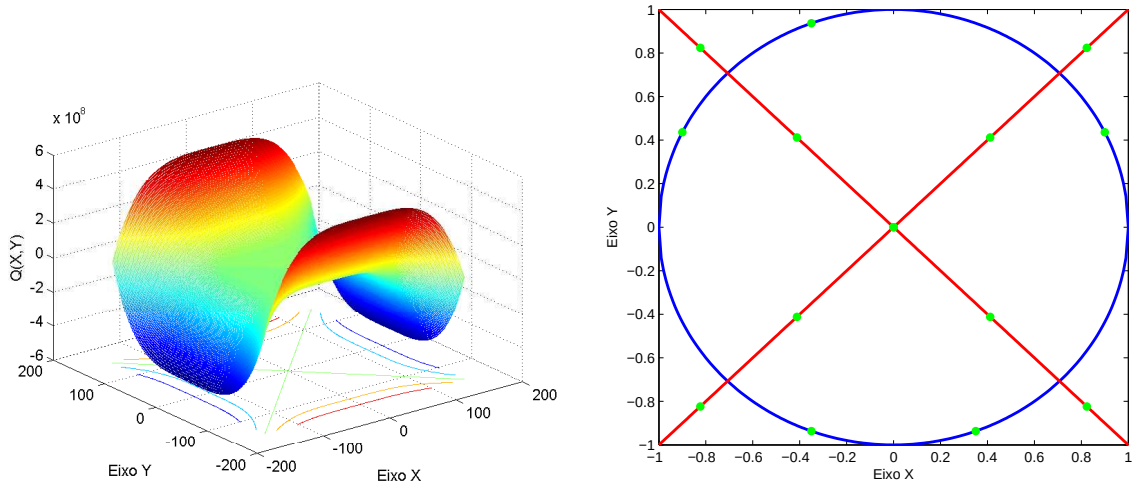
$$Q(x, y) = y^4 - x^4 - y^2 + x^2 \tag{2.6}$$

Pelo próprio critério da construção da equação 2.6 sabe-se que ela passa por todos os 14 pontos; porém ela tem mais zeros além desses 14. Na figura 2.2(a) temos a representação gráfica da equação 2.6. Já na figura 2.2(b) temos o gráfico de todas as raízes do polinômio, e nesse gráfico vemos que aparecem as retas, ou os polinômios mensagens; isto acontece porque a equação 2.6 pode ser fatorada, resultando em :

$$Q(x, y) = (y + x)(y - x)(x^2 + y^2 - 1) \tag{2.7}$$

Como pode se ver na equação 2.7, temos os polinômios de cada uma das retas e a equação de uma circunferência. Isto ocorre devido ao Teorema de Bézout, pois o produto dos graus do polinômio mensagem (grau um) e da equação 2.6 é igual a quatro, e como as duas equações se interceptam em mais de quatro pontos elas obrigatoriamente devem ter um fator comum, e como o fator comum do polinômio mensagem é ele próprio, pois não tem como ser fatorado, ele aparece como raiz da equação de duas variáveis.

Apesar dessa nova abordagem de decodificação de códigos RS, proposta por Sudan, parecer ser muito promissora, esse resultado foi recebido com muita cautela na época, pois esse novo método de decodificação utilizando interpolação de polinômios de duas variáveis somente apresentava aumento na capacidade de correção de erros para códigos de baixas taxas, mais especificamente códigos de taxa inferiores a 1/3, os quais não têm importância prática nenhuma, pois carregam muito mais redundância do que informação propriamente dita. Como o número exato de erros que podem ser corrigidos nesse algoritmo é difícil de ser determinado [19], é definido um limitante inferior para o número de erros corrigidos,




(a) Gráfico de Q(x,y)

(b) Raízes: Pontos a serem interpolados

**Figura 2.2:** *Gráfico de Q(x,y) e corte do mesmo no plano z=0, mostrando a suas raízes e os pontos a serem interpolados.*

que é $n - \sqrt{2kn}$ (ver figura 2.3).

Dois anos mais tarde, Sudan em parceria com Guruswami mostravam em um novo artigo [4] como conseguiram melhorar o método anterior fazendo com que o desempenho se mantivesse para todas as taxas de códigos. A solução encontrada continuava com a idéia de utilizar interpolação dos pontos em uma equação de duas variáveis. Contudo agora já não bastava apenas que o gráfico da equação passa-se por cada um dos pontos desejados, mas sim que cada ponto fosse uma singularidade da equação. De maneira informal, uma singularidade é um ponto onde a curva $Q(x, y) = 0$ interseciona a si mesma. Esse novo requisito faz com que cada um dos fatores que compõem $Q$ fosse elevado a uma potência $m$, denominado multiplicidade, que numa análise inicial pode parecer que serve apenas para elevar o grau de $Q(x, y)$; entretanto quando tentarmos fatorar $Q$, sabemos que o polinômio mensagem irá passar por muito mais singularidades de $Q$ do que somente pontos. Neste caso precisaremos apenas de metade das singularidades em comparação aos pontos e é daí que vem a vantagem, pois isso faz com que a capacidade de correção se mantenha mesmo para códigos com altas taxas, atingindo a capacidade de corrigir até $n - \sqrt{nk}$ erros, que é melhor para todas as taxas, superando a capacidade de correção do algoritmo de Berlekamp-Massey (figura 2.3), que é o algoritmo de decodificação mais

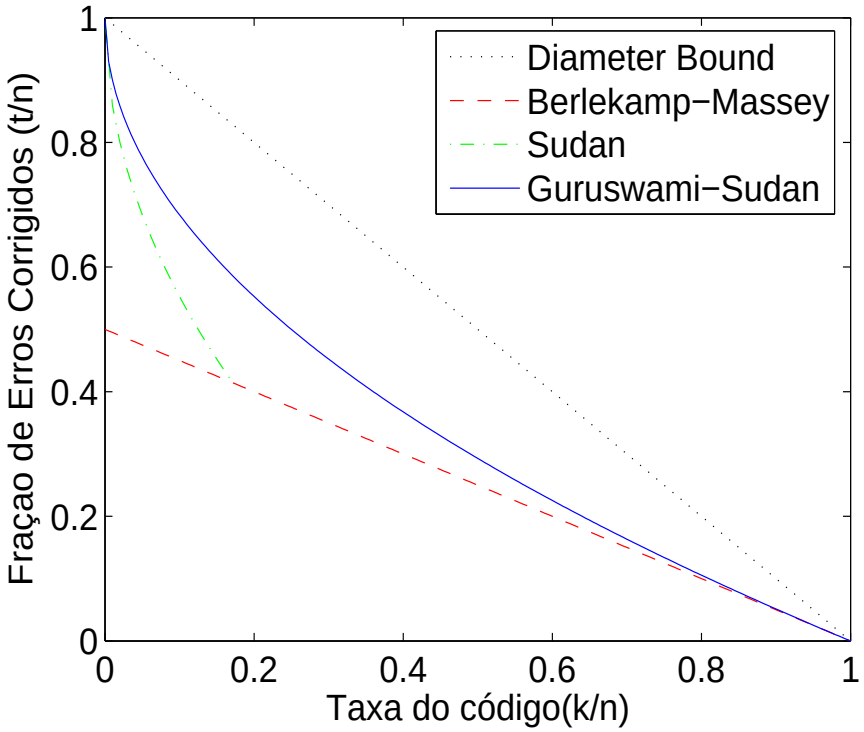



Figura 2.3: Comparativo da fração de erros corrigidos x taxa do códigos para vários tipos de algoritmos de decodificação de códigos de RS.

utilizado atualmente.

Com o término dessa visão geral do algoritmo de decodificação de GS, podemos perceber os fatores motivantes desse novo algoritmo de decodificação que vieram para mudar completamente os paradigmas de decodificação de códigos de Reed-Solomon até então existentes, e que fez do mesmo uma fonte para derivação de várias pesquisas sobre o assunto. E como veremos mais para frente será o passo inicial para atingir a decodificação por decisão suave de códigos de Reed-Solomon com forte embasamento algébrico. No próximo capítulo veremos com mais profundidade a teoria por trás da decodificação de Guruswami-Sudan.

# CAPÍTULO 3

# O ALGORITMO DE DECODIFICAÇÃO DE GURUSWAMI-SUDAN

## 3.1 A Correção de Erros por Interpolação

Como dito anteriormente, a idéia fundamental do algoritmo proposto por Guruswami-Sudan é que, através da construção clássica dos códigos de RS, um conjunto de pontos $(x_i, c_i), i = 1, ..., n$ são gerados através da relação $c_i = f(x_i)$, para um polinômio mensagem $f(x)$ qualquer, mas que tem que ter grau no máximo $k-1$. Os pontos $c_i$ serão utilizados como base para a modulação dos símbolos de canal os quais serão corrompidos por algum processo de ruído, então teremos um novo conjunto de pontos $(x_i, y_i), i = 1, ..., n$, no qual poderemos ter até um número $e$ de erros em $y_i$. O problema agora é encontrar um polinômio $p(x)$ de grau $< k$, que passe por todos os pontos de dados, tal que $p(x_i) = y_i$. Entretanto, como podemos ter pontos que estão com valores corrompidos pelo ruído; na realidade devemos buscar um polinômio $p(x)$ que passe pelo menos por $n - e$ pontos. Assim este polinômio que passa pelos pontos corretos tenderá a passar também pelos pontos mais prováveis de representar os símbolos transmitidos, e dessa maneira os pontos em erros serão recuperados.

O algoritmo de decodificação de Guruswami-Sudan é baseado na idéia de interpolação. O polinômio interpolante é construído utilizando-se um polinômio de duas variáveis, $Q(x, y)$, que satisfaz a condição $Q(x_i, y_i) = 0$. Além de simplesmente interpolar em cada ponto, um número inteiro $m_i$, denominado multiplicidade de interpolação é introduzido, o qual define a ordem de interpolação para cada ponto. Ele é equivalente a dizer o valor da função naquele ponto e das suas $m_i - 1$ derivadas. Esta multiplicidade de interpolação irá aumentar a capacidade de correção para todas as taxas de código. Além disso, será visto posteriormente que a multiplicidade será fundamental para atingir a decodficação

de códigos de RS por decisão suave. A partir do polinômio de duas variáveis, $Q(x, y)$, os polinômios $p(x)$ são obtidos através de fatoração, e irão satisfazer a condição $p(x_i) = y_i$ para uma quantidade suficiente de pontos $(x_i, y_i)$. Então cada polinômio $p(x)$ representa uma possível palavra transmitida, e o conjunto de polinômios é uma lista de possíveis palavras-código decodificadas.

## 3.2 Polinômios em Duas Variáveis

Nesta seção será apresentada uma série de conceitos algébricos associados a polinômios de duas variáveis que serão fundamentais para o entendimento do algoritmo de decodificação de códigos de RS por decisão suave. Estes conceitos básicos vão desde ordenação de monômios, grau de polinômios de duas variáveis até raízes múltiplas de polinômios de duas variáveis.

### 3.2.1 Grau Ponderado e Ordenação de Monômios

Para um polinômio de uma variável a noção de grau é intuitiva. Entretanto quando se passa a trabalhar com polinômios de duas variáveis, fica bem mais complexo pois agora temos duas incógnitas e temos que levar em conta a contribuição de cada uma delas para se determinar o grau do polinômio. E essa discussão das contribuições de cada variável faz com que existam vários métodos ou algoritmos para se obter o grau de uma equação a duas variáveis.

Antes de entrar mais fundo nessa questão devemos definir o corpo e o formato geral da equação de duas variáveis:

Seja $\mathbb{F}$ um corpo, e seja $\mathbb{F}[x, y]$ o anel comutativo de polinômios nas variáveis $x$ e $y$ com coeficientes em $\mathbb{F}$. Um polinômio $Q(x, y) \in \mathbb{F}[x, y]$ é por definição a soma finita de monômios com coeficientes diferentes de zeros da forma:

$$Q(x, y) = \sum_{i,j \geq 0} a_{i,j} x^i y^j \tag{3.1}$$

Para esclarecer a dificuldade da obtenção do grau de um polinômio de duas variáveis considere o seguinte exemplo: Seja $\mathbb{F} = \mathbb{R}$ e

$$Q(x, y) = 2x^4y + 3xy^3 + 7x^3 \tag{3.2}$$

Qual deve ser o grau desse polinômio? Qual é o termo de maior grau de $Q(x, y)$? E como ordenaríamos os monômios dessa equação de forma que fiquem em ordem crescente de grau?

Não é difícil de perceber que se resolvermos a última questão as outras estarão respondidas, pois se encontrarmos um meio de ordenarmos os monômios, o termo de maior grau e o grau do polinômio serão obtidos através do coeficiente e expoente do último termo, por isso vamos nos ater a entender o conceito de ordenação de monômios nesse primeiro momento. A ordenação de monômios é um assunto bem extenso e serão abordados somente os tópicos imprescindíveis para o entendimento desta dissertação; sugere-se que se leia a referência [20] caso se deseje aprofundar mais no assunto. A questão da ordenação de monômios com duas incógnitas não é nada mais do que dizer que $x^{i_1}y^{j_1}$ é menor que $x^{i_2}y^{j_2}$, através da comparação dos graus dos coeficientes de $x$ e $y$. De maneira intuitiva podemos ordenar os monômios utilizando a seguinte regra: se $i_1 < i_2$ e $j_1 < j_2$ então $x^{i1}y^{j1} < x^{i_2}y^{j_2}$. Porém quando temos expoentes iguais ou para a variável $x$ ou para $y$, fica a dúvida de como ficaria a ordenação. Com o intuito de criar um critério de desempate para esse tipo de situação foi criado uma ordenação de monômios por pesos ponderados, usualmente denominada por ordem WD (*WD: weighted degree*). A ordenação WD é caracterizada por se dar pesos para a contribuição de cada um dos expoentes das variáveis $x$ e $y$. Esses pesos são determinados pelo par ordenado $(u, v)$, onde $u$ e $v$ são inteiros não negativos que representam os pesos dos expoentes das incógnitas $x$ e $y$, respectivamente. O peso ponderado, $w$, de um monômio qualquer é definido como:

$$deg_w x^i y^j = ui + vj; \tag{3.3}$$

Esse tipo de ordenação ainda não é suficiente para ordenar todas as variações de

monômios pois existem casos onde teremos graus ponderados iguais; basta lembrar que a ordenação intuitiva analisada anteriormente é um caso particular da ordem WD quando $(u, v) = (1, 1)$. Existem duas formas que são de nosso interesse que criam critérios de desempate que possibilitam ordenar totalmente os monômios de uma equação de duas variáveis. A primeira é denominada ordem *w-lexicographic*, ou abreviadamente w-lex, que estabelece que se $ui_1 + vj_1 = ui_2 + vj_2$, então $x^{i_1}y^{j_1} < x^{i_2}y^{j_2}$ se $i_1 < i_2$. A segunda é denominada ordem *w-reverse lexicographic*, ou w-revlex, que estabele o critério inverso da segunda, ou seja, se $ui_1 + vj_1 = ui_2 + vj_2$, então $x^{i_1}y^{j_1} < x^{i_2}y^{j_2}$ se $i_1 > i_2$.

Por exemplo, consideremos como pesos $(u, v) = (1, 3)$ e seja $\phi_1(x, y) = x^3y^2$ um monômio de grau ponderado igual a 9, $deg_{(1,3)}\phi_1 = 9$. Seja também $\phi_2(x, y) = x^6y$ um monômio de grau ponderado igual a 9, $deg_{(1,3)}\phi_2 = 9$. Então na ordenação w-lex, $\phi_1 <_{wlex} \phi_2$, já se for utilizada a ordenação w-revlex, $\phi_2 <_{wrevlex} \phi_1$. A tabela 3.1 mostra os primeiros 26 monômios, $\phi_j(x, y)$, ordenados de acordo com o critério w-lex, junto com o seu grau ponderado e seu índice na ordenação. Os primeiros 26 monômios ordenados pelo critério w-revlex são mostrados na tabela 3.2.

Tabela 3.1: Monômios Ordenados segundo a ordem w-lex com $(u, v) = (1, 3)$

| Monômio | $1$ | $x$ | $x^2$ | $y$ | $x^3$ | $xy$ | $x^4$ | $x^2y$ | $x^5$ | $y^2$ | $x^3y$ | $x^6$ | $xy^2$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Peso | 0 | 1 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 6 | 7 |
| Índice ($j$) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Monômio | $x^4y$ | $x^7$ | $x^2y^2$ | $x^5y$ | $x^8$ | $y^3$ | $x^3y^2$ | $x^6y$ | $x^9$ | $xy^3$ | $x^4y^2$ | $x^7y$ | $x^{10}$ |
| Peso | 7 | 7 | 8 | 8 | 8 | 9 | 9 | 9 | 9 | 10 | 10 | 10 | 10 |
| Índice ($j$) | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |

Tabela 3.2: Monômios Ordenados segundo a ordem w-revlex com $(u, v) = (1, 3)$

| Monômio | $1$ | $x$ | $x^2$ | $x^3$ | $y$ | $x^4$ | $xy$ | $x^5$ | $x^2y$ | $x^6$ | $x^3y$ | $y^2$ | $x^7$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Peso | 0 | 1 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 6 | 7 |
| Índice ($j$) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Monômio | $x^4y$ | $xy^2$ | $x^8$ | $x^5y$ | $x^2y^2$ | $x^9$ | $x^6y$ | $x^3y^2$ | $y^3$ | $x^{10}$ | $x^7y$ | $x^4y^2$ | $xy^3$ |
| Peso | 7 | 7 | 8 | 8 | 8 | 9 | 9 | 9 | 9 | 10 | 10 | 10 | 10 |
| Índice ($j$) | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |

Se adotarmos qualquer um dos métodos de ordenação relacionados acima, podemos representar qualquer polinômio com duas variáveis do tipo exposto na equação 3.1, da

seguinte forma:

$$Q(x,y) = \sum_{j=0}^{J} a_j \phi_j(x,y) \tag{3.4}$$

Onde os coeficientes $a_j \in \mathbb{F}$, e $a_j \neq 0$. O inteiro $j$ representa o índice do monômio de acordo com as tabelas 3.1 ou 3.2, dependendo da ordem adotada. O inteiro $J$ indica o último monômio da equação que é denominado de monômio líder. Assim o grau ponderado de uma equação $Q(x,y)$ é dado pelo grau ponderado do seu monômio líder $\phi_j$.

Por exemplo, sejam os pesos $(u,v) = (1,3)$, e considere o ordenação utilizando o critério *w-lexicographic*, então a equação:

$$Q(x,y) = 1 + y + x^4 + xy + x^4y + x^3y + x^6 + x^2y \tag{3.5}$$

depois de ordenada de maneira crescente, utilizando o critério w-lex, fica:

$$Q(x,y) = 1 + y + xy + x^4 + x^2y + x^3y + x^6 + x^4y \tag{3.6}$$

Assim o grau ponderado de $Q$, ou $degree_{1,3}Q$, é igual ao grau ponderado do monômio líder, no caso $x^4y$, então o $degree_{1,3}Q = 1 \cdot 4 + 3 \cdot 1 = 7$.

### 3.2.2 Número de Monômios

Com a intenção de se caracterizar o desempenho do decodificador, é necessário conhecer quantos monômios de até um certo grau ponderado podem existir. A fórmula abaixo permite encontrar o valor exato do número de monômios de grau ponderado menor ou igual a $\delta$, para o caso especifíco (e de nosso interesse) onde os pesos dos coeficientes $x$ e $y$, é $(u,v) = (1,v)$, ou seja, fixando o peso 1 para os expoentes da incógnita $x$.

$$N_{1,v}(\delta) = \left( \left\lfloor \frac{\delta}{v} \right\rfloor + 1 \right) \left( \delta + 1 - \frac{v}{2} \left\lfloor \frac{\delta}{v} \right\rfloor \right) \tag{3.7}$$

Esta fórmula provêm diretamente da análise da ordenação de monômios e é independente do critério utilizado (w-lex ou w-revlex). A prova da mesma pode ser encontrada

em [4].

A equação 3.7 pode ser simplificada na seguinte inequação que nos dá um limitante inferior:

$$N_{1,v}(\delta) > \frac{\delta^2}{2v} \tag{3.8}$$

Utilizando as equações 3.7 e 3.8 para o exemplo dado anteriormente, que tem grau ponderado 7, e peso $(1, 3)$, teremos:

$$N_{1,3}(7) = 15 \tag{3.9}$$

$$N_{1,3}(7) > 8 \tag{3.10}$$

Resultado que pode ser verificado através das tabelas 3.1 ou 3.2.

### 3.2.3 Raízes Múltiplas

O conceito fundamental por trás do algoritmo de Guruswami-Sudan para decodificação de códigos RS está em se obter um polinômio que tenha raízes de ordem múltipla nos pontos recebidos. Nesta seção veremos o que é uma raiz de ordem múltipla e como fazemos para reconhecê-la.

Em Matemática, uma raiz (ou um zero) de uma função $f$ é um elemento $x$ do domínio de $f$ tal que $f(x) = 0$. Seja $Q(x)$ um polinômio de uma variável e $\alpha$ uma raiz de $Q$, então:

$$Q(x) = (x - \alpha)^m P(x) \tag{3.11}$$

Onde $m$ é um inteiro não negativo e $P(x)$ é um polinômio qualquer com $P(\alpha) \neq 0$. Diz-se então que $\alpha$ é uma raiz de $Q$ com multiplicidade $m$, e para o caso particular em que $m = 1$ então $\alpha$ é denominado como uma raiz simples de $Q$.

Considere o polinômio do exemplo abaixo:

$$Q(x) = x^3 + x^2 \tag{3.12}$$

Colocando $x^2$ em evidência e ressaltando os fatores que a compõem, temos:

$$Q(x) = x^2(x+1) = (x-0)^2(x+1) \tag{3.13}$$

Analisando a equação 3.13, podemos dizer que ela tem raízes em $x = -1$ e $x = 0$, pois se substituirmos qualquer um dos dois valores em $Q$ o resultado da expressão será zero. E podemos dizer também que a raiz no ponto $x = -1$ tem multiplicidade 1, ou seja é uma raiz simples, enquanto que a raiz em $x = 0$ tem multiplicidade 2.

Um polinômio $Q(x)$ que possui uma raiz de multiplicidade $m$ maior que 1 num ponto $\alpha$ apresenta uma característica muito importante, que é o fato de que se aplicarmos $m-1$ vezes o operador derivada de forma sucessiva em $Q(x)$, as equações resultantes continuarão tendo $\alpha$ como raiz. Essa propriedade é muito importante e será bastante explorada mais adiante. Voltando ao exemplo da equação 3.13; podemos notar que se calcularmos o valor da derivada primeira do fator $(x+0)^2$, a derivada primeira no ponto $x = 0$ será 0 $(d(x^2)/dx = 2x = 2 \cdot 0 = 0)$; isto acontece pelo fato da raiz ter multiplicidade 2.

Para se estender os conceitos expostos até aqui para polinômios de duas variáveis, primeiro irá se estudar como se obtêm a multiplicidade de uma raíz para um polinômio qualquer de uma variável e depois isto será estendido para polinômios em duas variáveis.

**Definição 3.2.1** *Para $m \leq n$, o polinômio:*

$$Q(x) = a_m x^m + a_{m+1} x^{m+1} + \cdots + a_n x^n = \sum_{r=m}^{n} a_r x^r, \tag{3.14}$$

*onde os coeficientes $a_0 = a_1 = \cdots = a_{m-1} = 0$, é dito ter uma raiz de multiplicidade ou ordem $m$ no ponto $x = 0$.*

Seja $D_r$ o operador formal de derivada de ordem $r$, então pode-se afirmar que:

$$Q(0) = D_1 Q(0) = \cdots = D_{m-1} Q(0) = 0 \tag{3.15}$$

Onde podemos notar que a multiplicidade da raiz pode ser expressa em termos das derivadas do polinômio no ponto da raiz.

A Definição 3.2.1 somente trata do caso para uma raiz na origem. Pode-se utilizar dela para se obter um resultado geral onde podemos ter raízes de multiplicidade $m$ em qualquer lugar do eixo $x$.

Seja $Q(x)$ um polinômio qualquer, para analisar se ele possui uma raiz com multiplicidade $m$ em $x = \alpha$, basta verificar se a sua versão transladada para a origem, ou seja, $Q(x + \alpha)$, tem uma raiz de multiplicidade $m$ em $x = 0$. Essa solução a primeira vista parece simples, contudo analisando-se mais detalhadamente pode-se perceber a dificuldade para se obter $Q(x + \alpha)$, pois se simplesmente aplicarmos uma substituição da variável $x$ por $x + \alpha$ na equação 3.14, será altamente trabalhoso para casos onde $n$ é muito grande. Para resolver este problema irá recorrer-se ao Teorema de Hasse [21], que equaciona a translação de um polinômio qualquer com coeficientes vindos de um corpo $\mathbb{F}$ qualquer.

**Teorema 3.2.1** *Se* $Q(x) = \sum_i a_i x^i \in \mathbb{F}[x]$*, então para qualquer* $\alpha \in \mathbb{F}$*, temos*

$$Q(x + \alpha) = \sum_{r=0}^{n} Q_r(\alpha) x^r, \tag{3.16}$$

*onde*

$$Q_r(x) = \sum_{i=0}^{n} \binom{i}{r} a_i x^{i-r} \tag{3.17}$$

*e se* $r > i$ *então faz-se* $\binom{i}{r} = 0$.

A equação 3.17 é chamada de $r$-ésima derivada formal de Hasse de $Q(x)$. Por isso denomina-se $Q_r(x)$ por $D_r Q(x)$. É importante ressaltar que o operador formal de derivada não é igual ao operador derivada utilizado normalmente, mas pode ser relacionado ao operador usual através da seguinte equação:

$$Q_r(x) = D_r Q(x) = \frac{1}{r!} \frac{d^r}{dx^r} Q(x) \tag{3.18}$$

onde pode-se perceber que $D_r$ parece muito com o operador derivada usual tendo a diferença na introdução de um fator de escalonamento $1/r!$.

Pode-se então escrever que:

$$Q(x+\alpha) = \sum_{r=0}^{n} D_r Q(\alpha) x^r. \tag{3.19}$$

Assim se $Q(x)$ tiver uma raiz em $\alpha$ de ordem $m$, então devem-se ter os primeiros $m$ coeficientes desta soma iguais a zero:

$$Q(\alpha) = D_1 Q(\alpha) = ... = D_{m-1} Q(\alpha) = 0 \tag{3.20}$$

Desta forma pode-se estender a Definição 3.2.1 para:

**Definição 3.2.2** *Um polinômio $Q(x)$ tem uma raiz de multiplicidade $m$ em $x = \alpha$ se:*

$$Q(\alpha) = D_1 Q(\alpha) = \cdots = D_{m-1} Q(\alpha) = 0 \tag{3.21}$$

Os conceitos apresentados aqui podem ser facilmente estendidos para polinômios de duas variáveis.

**Definição 3.2.3** *Seja um polinômio $Q(x,y) \in \mathbb{F}[x,y]$, e sejam $\alpha$ e $\beta$ tais que $Q(\alpha,\beta) = 0$. Então dizemos que $Q$ tem uma raiz em $(\alpha,\beta)$.*

*Seja $Q(x,y) = \sum_{i,j\geq 0} a_{i,j} x^i y^j$. Dizemos que $Q$ tem uma raiz de multiplicidade ou ordem $m$ em $(0,0)$ se os coeficientes $a_{i,j} = 0$ para todos os $i$ e $j$ tais que $i + j < m$.*

*De maneira similar dizemos que $Q(x,y)$ tem uma raiz de multiplicidade ou ordem $m$ em $(\alpha,\beta)$, se $Q(x+\alpha, y+\beta)$ tiver uma raiz de ordem $m$ em $(0,0)$.*

Considere o seguinte exemplo, seja $Q(x,y) = x^4y + x^3y^2 + x^4y^4$ então utilizando a definição 3.2.3, podemos concluir que este polinômio tem uma raiz em $(0,0)$ de multiplicidade 5, pois a equação não tem nenhum monômio cuja soma dos seus expoentes seja menor que 5. Já o polinômio $Q(x,y) = x + y$ tem uma raiz na origem com multiplicidade 1. A equação $Q(x,y) = (x-\alpha)^4(y-\beta) + (x-\alpha)^3(y-\beta)^2 + (x-\alpha)^4(y-\beta)^4$ tem uma raiz de ordem 5 em $(\alpha,\beta)$.

Uma importante relação que pode ser observada é que para se ter uma raiz de ordem $m$ é preciso que $m(m+1)/2$ coeficientes sejam zero. Por exemplo, para uma raiz de

multiplicidade 3 em $(0,0)$ é necessário que $3(3+1)/2 = 6$ coeficientes sejam zero, ou seja, que $a_{0,0} = a_{0,1} = a_{0,2} = a_{1,0} = a_{1,1} = a_{2,0} = 0$.

Pode-se agora estender o Teorema de Hasse (Teorema 3.2.1) para o caso em que tenhamos um polinômio de duas variáveis.

**Teorema 3.2.2** *Seja um polinômio* $Q(x,y) = \sum_{i,j} a_{i,j} x^i y^j$, *então*

$$Q(x+\alpha, y+\beta) = \sum_{r,s} Q_{r,s}(\alpha,\beta) x^r y^s \tag{3.22}$$

*onde*

$$Q_{r,s}(x,y) = \sum_i \sum_j \binom{i}{r}\binom{j}{s} a_{i,j} x^{i-r} y^{j-s} \tag{3.23}$$

A equação 3.23 é chamada de derivada parcial mista formal de Hasse de $Q(x,y)$. Por isso denomina-se $Q_{r,s}(x,y)$ por $D_{r,s}Q(x,y)$, e sua relação com o operador de derivada parcial é dada por:

$$Q_{r,s}(x,y) = D_{r,s}Q(x,y) = \frac{1}{r!s!}\frac{\partial^{r+s}}{\partial^r \partial^s} Q(x,y) \tag{3.24}$$

Assim baseado nesta notação pode-se estender o resultado de que a multiplicidade $m$ de uma raiz $(\alpha,\beta)$ pode ser expressa pela condição:

$$D_{r,s}Q(\alpha,\beta) = 0, \text{ para todos } r,\ s \text{ tais que } r+s < m \tag{3.25}$$

Mudando no fato que deve ser dividido por um fator de escalonamento $1/(r!s!)$

E o número de combinações que podemos ter entre $r$ e $s$, é dado em função de $m$, por

$$\binom{m+1}{2} = m(m+1)/2. \tag{3.26}$$

## 3.3 Os Teoremas de Interpolação e Fatoração

Com os conceitos expostos nas seções anteriores se pode descrever com detalhes como funciona o decodificador de Guruswami-Sudan (GS). Para uma código RS $(n,k)$ sobre um

corpo $\mathbb{F}$, com um conjunto suporte $(x_1, x_2, ..., x_n)$ e um inteiro positivo $m$, o decodificador de GS recebe um vetor $\mathbf{y} = (y_1, y_2, ..., y_n) \in \mathbb{F}^n$ como entrada e produz como saída uma lista de polinômios $(p_1, p_2, ..., p_L)$ através de duas fases:

### 3.3.1 A Fase de Interpolação

O decodificador constrói um polinômio de duas variáveis da forma:

$$Q(x, y) = \sum_{j=0}^{C} a_j \phi_j(x, y) \tag{3.27}$$

com o menor grau ponderado $(1, v)$ possível, e que tem raízes com multiplicidade $m$ em cada um dos pontos $(x_i, y_i)$, $i = 1, 2, ..., n$. Na equação anterior, $\phi_j(x, y)$ representa monômios da forma $x^p y^q$, ordenados de acordo com o critério $(1, v)$ revlex, tal que, $\phi_0 < \phi_1 < \cdots$. Questões fundamentais que surgem em relação a esse passo são: Como podemos contruir tal polinômio? E será que tal polinômio sempre existirá?

### 3.3.2 A Fase de Fatoração

O polinômio $Q(x, y)$ é fatorado para se encontrar um conjunto de polinômios $p(x)$ tal que $y - p(x)|Q(x, y)$, ($y - p(x)$ divide $Q(x, y)$). Assim é formado um conjunto de polinômios denominados de raízes de $y$ de $Q(x, y)$,

$$L = \{p(x) \in \mathbb{F}[x] : (y - p(x))|Q(x, y)\} \tag{3.28}$$

Questões relacionadas a esse passo: Como isso se relaciona com a capacidade de correção de erros do código? Como se deve proceder para fatorar $Q(x, y)$? Quantos polinômios em $L$ devem existir?

### 3.3.3 Teorema da Interpolação

Este teorema tem o objetivo de fornecer as condições de existência do polinômio interpolante.

**Teorema 3.3.1** *Seja*

$$Q(x,y) = \sum_{i,j} a_{i,j} x^i y^j = \sum_{i=0}^{C} a_i \phi(x,y), \tag{3.29}$$

*onde os monômios são ordenados de acordo com uma ordenação monomial arbitrária. Então existe um polinômio $Q(x,y)$, diferente de zero, que interpola os pontos $(x_i, y_i), i = 1, 2, ..., n$ com multiplicidade $m$ em cada ponto se*

$$C = \binom{m+1}{2} \tag{3.30}$$

**Prova 3.3.1** *Só existirá uma raiz de ordem $m$ em $(x_i, y_i)$ se*

$$D_{r,s}Q(x_i, y_i) = 0, \text{ para todo (r,s) tal que } 0 \leq r + s < m. \tag{3.31}$$

*A equação 3.31 pode ser escrita de outra forma através da equação de derivada parcial de Hasse 3.23:*

$$\sum_i \sum_j \binom{i}{r} \binom{j}{s} a_{i,j} x^{i-r} y^{j-s} = 0 \tag{3.32}$$

*onde $i = 1, 2, ..., n$ e $r + s < m$.*

*Existem $\binom{m+1}{2}$ combinações que podem ser feitas com $(r, s)$ para cada $(x_i, y_i)$, o que totaliza $n\binom{m+1}{2}$ equações homogêneas. Note que nessas equações, as incógnitas são os coeficientes $a_{k,j}$. Então se $C = n\binom{m+1}{2}$, haverá $C+1$ variáveis $a_0, a_1, ..., a_C$ em 3.29. Por isso tem que haver pelo menos uma solução diferente de zero para o conjunto de equações lineares de 3.31, que irá corresponder ao polinômio $Q(x,y)$ da forma de 3.29.*

### 3.3.4 Teorema da Fatoração

O resultado fundamental da fase de fatoração vem do seguinte Teorema:

**Teorema 3.3.2** *Seja $Q(x,y)$ um polinômio interpolante com grau ponderado $\leq \delta$,com $(u,v) = (1,v)$, e tal que $D_{r,s}Q(x_i, y_i) = 0$ para todo $i = 1, 2, ..., n$ com $r + s < m$. Ou seja*

*cada ponto $(x_i, y_i)$ é interpolado com multiplicidade $m$. Seja $p(x)$ um polinômio com grau no máximo $v$, tal que $y_i = p(x_i)$, para pelo menos $K_m$ valores de $i$. Se $mK_m > \delta$, então $(y - p(x))|Q(x, y)$.*

A prova pode ser encontrada em [18]. Mas não é difícil perceber a validade deste Teorema pois o mesmo não deixa de ser um caso particular de aplicação do Teorema de Bézout [2].

O grau de $p(x)$ no Teorema 3.3.2 é no máximo $v$. Como $p(x)$ deve interpolar pontos nos quais $y_i = p(x_i)$ e existe, pelo próprio processo de codificação de Reed-Solomon, uma relação polinomial de grau $< k$ entre o conjunto suporte e as palavras códigos, devemos ter então que o grau de $p(x) < k$. Por isso fazemos:

$$v = k - 1 \tag{3.33}$$

que estabelece o peso da ordenação de monômios utilizado pelo algoritmo.

## 3.4 Capacidade de Correção

Uma vez apresentados os teoremas fundamentais na decodificação de Guruswami-Sudan, é necessário obter uma equação que caracterize a capacidade de correção deste tipo de decodificação em função da multiplicidade $m$, do comprimento da palavra código, $n$, e da quantidade de símbolos de informação, $k$. O ponto fundamental do Teorema de Interpolação é que o número de coeficientes do polinômio interpolante deve exceder o número de equações, que é $n\binom{m+1}{2}$. Lembrando que o número de coeficientes é igual ao número de monômios de um polinômio, e que para um polinômio de grau ponderado $\delta$, com pesos $(1, v)$ este número é dado pela equação 3.7. Então:

$$n\binom{m+1}{2} < N_{1,v}(\delta) \tag{3.34}$$

Pelo Teorema da Fatoração (Teorema 3.3.2) a seguinte condição deve ser respeitada:

$$mK_m > \delta \text{ ou } mK_m \geq \delta + 1 \text{ ou } mK_m - 1 \geq \delta \tag{3.35}$$

Como a função $N_{1,v}(\delta)$ sempre cresce com o aumento de $\delta$, pode-se substituir $\delta$ por um valor maior, no caso $mK_m - 1$, sem que a inequação 3.34 perca a validade, então temos:

$$N_{1,v}(mK_m - 1) > n\binom{m+1}{2} \tag{3.36}$$

Para $m \geq 1$, então $K_m$ será o menor valor para o qual a equação 3.36 se torna verdadeira:

$$K_m = min\left\{K : N_{1,v}(mK_m - 1) > n\binom{m+1}{2}\right\} \tag{3.37}$$

Do Teorema de Fatoração (Teorema 3.3.2), $K_m$ é o número de pontos do vetor $\mathbf{y}$ que concordam com a palavra código, então $t_m = n - K_m$ é a distância entre $\mathbf{y}$ e a palavra código transmitida, ou seja, é a capacidade de correção de erros do código. Para $m = 0$, define-se $K_m$ como $n - t_0 = n - \lfloor (n-k)/2 \rfloor = \lceil (n+v+1)/2 \rceil$. No exemplo que se segue, mostra-se que $K_m$ nunca decresce com o aumento de $m$.

Seja o código de Reed-Solomon C(32,8). A figura 3.1 mostra o comportamento de $K_m$ com a variação de $m$. Pode-se notar, no início da curva, que há abruptos decréscimos em $K_m$ para pequenos acréscimos em $m$, seguida de uma região longa sem variações. Em $m = 120$, $K_m$ diminui para seu valor final, e a partir daí ele estabiliza e não decresce mais.

Existe um valor de multiplicidade, $m_0$, além do qual $K_{m_0} = K_{m_0+1} = \cdots$, ou seja, onde $K_m$ atinge o seu menor valor possível, este valor é denominado de $K_\infty$. Como $K_m$ nunca cresce com o aumento de $m$, então com $t_m = n - K_m$ ocorre o contrário, $t_m$ nunca descresce com o aumento de $m$. Isto é, aumentando a multiplicidade $m$, é possível aumentar a capacidade de correção do código, até que se atinge o ponto $t_\infty = n - K_\infty$, que é chamado de capacidade assintótica de decodificação do decodificador de Guruswami-Sudan.

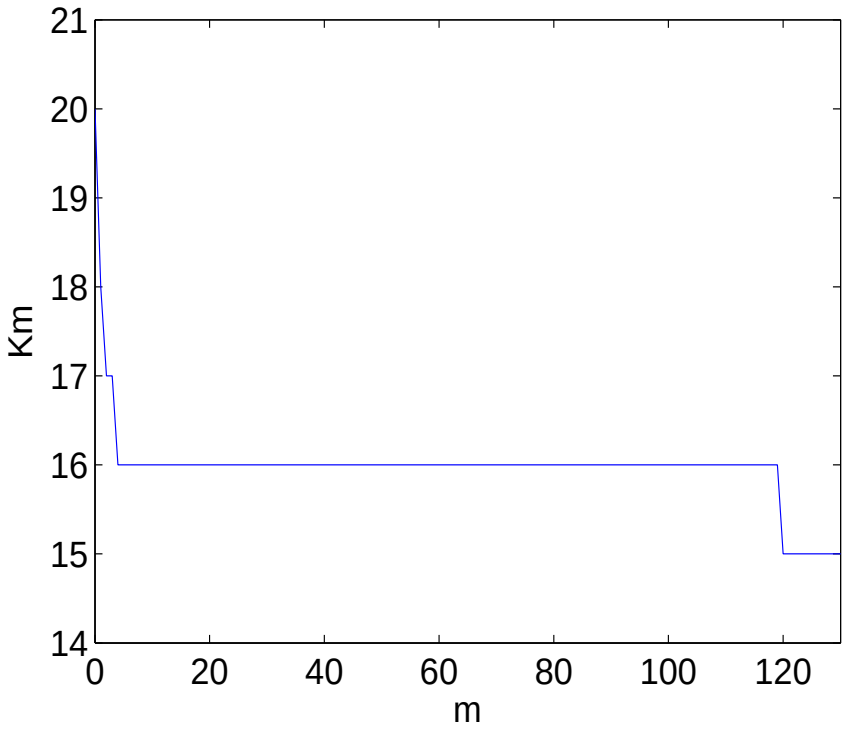



Figura 3.1: $K_m$ em função de $m$ para uma código de RS C(32,8).

Substituindo o lado esquerdo da equação 3.36 pela equação 3.8 e isolando $m$ temos:

$$\begin{aligned} \frac{m^2 K_m^2}{2v} &> n\binom{m+1}{2} \\ \frac{m^2 K_m^2}{2v} &> \frac{nm(m+1)}{2} \\ mK_m^2 &> nvm + nv \\ m &> \left(\frac{K_m^2}{nv} - 1\right)^{-1} \end{aligned} \tag{3.38}$$

Como a multiplicidade tem que ser um número inteiro maior que zero, o termo do lado direito da equação 3.38, $m$, tem que ser maior que zero, assim:

$$\left(\frac{K_m^2}{nv} - 1\right) > 0$$
$$K_m^2 > nv$$
$$K_m > \lfloor\sqrt{nv}\rfloor$$
$$K_m \geq \lfloor\sqrt{nv}\rfloor + 1 \tag{3.39}$$

Suponha que $K_m$ fosse menor, por exemplo, $K_m = \lfloor\sqrt{nv}\rfloor$. Então substituindo $K_m$ na equação 3.38, ficaria $(\lfloor\sqrt{nv}\rfloor)^2/nv - 1)$ que resultaria num $m$ negativo. Portanto $K_m = \lfloor\sqrt{nv}\rfloor + 1$ é o menor valor possível para $K_m$.

Considere a seguinte relação algébrica [22]:

$$\sqrt{n(v+1)} > \sqrt{nv} \tag{3.40}$$

E considere que se $x$ e $y$ forem números reais então:

$$x > y \text{ implica } \lceil x\rceil \geq \lfloor y\rfloor + 1 \tag{3.41}$$

Assim, combinando 3.41 com 3.40 temos:

$$\lceil\sqrt{n(v+1)}\rceil \geq \lfloor\sqrt{nv}\rfloor + 1 \tag{3.42}$$

Portanto a capacidade de correção assintótica de um código de Reed-Solomon utilizando a decodificação de Guruswami-Sudan será:

$$t_m = n - K_m$$
$$K_m = \lceil\sqrt{n(v+1)}\rceil$$
$$t_m = n - \lfloor\sqrt{n(v+1)}\rfloor \tag{3.43}$$

Através da relação 3.33, substituimos $v$ por $k-1$ obtendo então a equação que permite

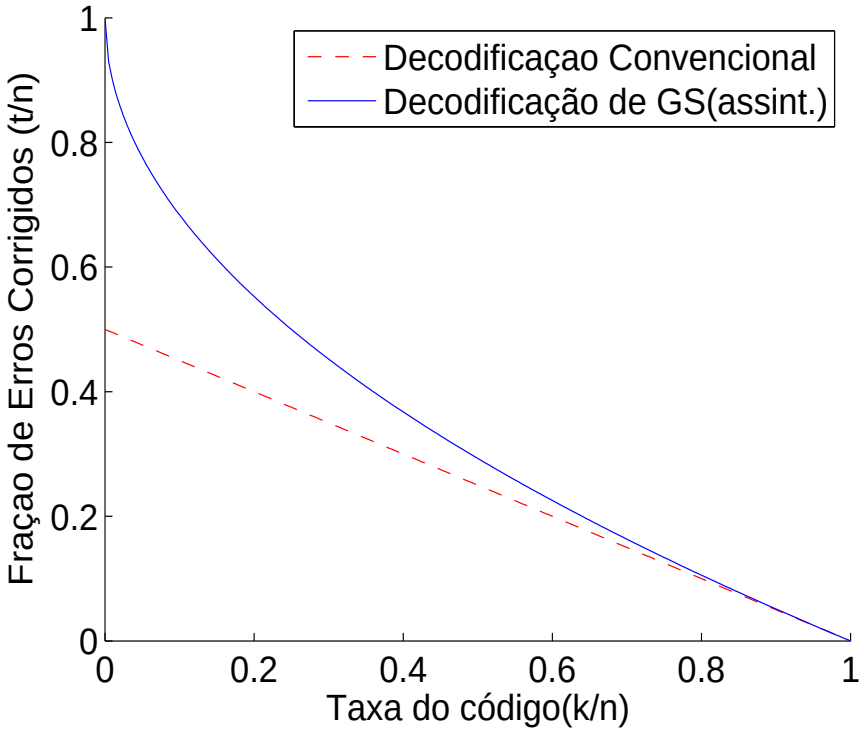



Figura 3.2: Ganho de desempenho do decodificador de GS

se obter a capacidade de correção de erros em função da dimensão do código:

$$t_m = \lfloor n - \sqrt{nk} \rfloor \tag{3.44}$$

Para um código $C(n,k)$, com um decodificador capaz de corrigir até $t$ erros, seja $\tau = t/n$ a fração de erros corrigidos e seja $R = k/n$ a taxa do código. No caso de um decodificador convencional onde $t = t_0$, a fração de erros corrigidos é (assintóticamente) $\tau_0 = (1-R)/2$. Utilizando o algoritmo de Guruswami-Sudan para decodificação, então $\tau = 1 - \sqrt{R}$ (resultado assintótico). A figura 3.2 mostra a melhora no desempenho da fração de erros corrigidos em função da taxa do código. O aumento da capacidade de decodificação é notável, particularmente para códigos de baixa taxa.

## 3.5 Números de Polinômios na Lista do Decodificador

O algoritmo de GS utiliza o paradigma de decodificação por lista, portanto sua saída é uma lista de polinômios $L = \{p_1(x), p_2(x), ..., p_L(x)\}$. A palavra código transmitida estará

em $L$ se o número de erros inseridos pelo canal for $< t_m$. Estabelecer o número máximo de polinômios que desejamos ter na saída do algoritmo de decodificação é muito importante, pois quanto mais polinômios forem retornados, pior será o desempenho tanto em termos de tempo de execução, que irá aumentar bastante, quanto em termos de capacidade de correção [22]. Relembrando que $Q(x, y)$ é um polinômio de grau ponderado (com peso $(1, v) \leq \delta$) e que $p(x)$ são os polinômios para os quais $(y - p(x))|Q(x, y)$. O número máximo de polinômios retornados é, então, o grau de $y$ de $Q(x, y)$.

O grau de $y$ de $Q(x, y)$ é definido como o grau de $Q(x, y)$ quando fazemos $x = 1$, ou seja, é o grau do polinômio $Q(1, y)$, de uma variável em y. Seja $N_{1,v}(L)$ o número de monômios de um polinômio com maior monômio igual a $y^L$, com respeito à ordenação revlex com pesos $(1, v)$. Utilizando a tabela 3.2 temos:

| L | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $N_{1,v}(L)$ | 0 | 4 | 11 | 21 | 34 | 50 |

Então podemos dizer que o número máximo de polinômios retornados pelo algoritmo de decodificação de GS ($L_m$) é dado pela relação:

$$L_m = max\left\{L : N_{1,v}(L) \leq n\binom{m+1}{2}\right\} \tag{3.45}$$

A partir da equação 3.45 pode-se chegar na seguinte fórmula (o desenvolvimento foi omitido e pode ser encontrado em [18]):

$$L_m = \left\lfloor \sqrt{\left(\frac{v+2}{2v}\right)^2 + \left(\frac{nm(m+1)}{v}\right)} - \left(\frac{v+2}{2v}\right) \right\rfloor \tag{3.46}$$

Pode-se também obter uma fórmula mais simples que fornece um limitante superior para $L_m$:

$$L_m < \left(m + \frac{1}{2}\right)\sqrt{\frac{n}{v}} \tag{3.47}$$

## 3.6 Algoritmo de Interpolação de Koetter

Como visto no Teorema da Interpolação (Teorema 3.3.1), o polinômio interpolante pode ser encontrado se resolvermos o conjunto de

$$C = n\binom{m+1}{2} \tag{3.48}$$

equações lineares com um número de incógnitas $> C$, como na equação 3.32. A primeira solução para o sistema de equações seria utilizar a eliminação de Gauss, porém teria uma complexidade muito alta (de ordem cúbica - $O(n^3)$) o que invibializaria aplicações práticas. O algoritmo de Koetter, inspirado no algoritmo de Feng-Tzeng [23], consegue resolver o problema da interpolação de forma eficiente e com complexidade aceitável ($O(n^2m^4)$), sendo portanto o mais utilizado. Por esta razão, veremos seus conceitos principais (para provas e mais detalhes do algoritmo recomenda-se consultar [18], [22] e [24]). Outros algoritmos que implementam a fase de interpolação podem ser encontrados em [25] e [26].

O operador derivada formal definido em 3.24 pode atuar como um funcional linear. Este mapeamento a ser definido é a base para o entendimento do algoritmo de Koetter.

**Definição 3.6.1** *Um mapeamento $D : \mathbb{F}[x, y] \rightarrow \mathbb{F}$ é denominado um funcional linear se para quaisquer polinômios $Q(x, y)$ e $P(x, y) \in \mathbb{F}[x, y]$ e quaisquer constantes $a, b \in \mathbb{F}$,*

$$D(aQ(x, y) + bP(x, y)) = aDQ(x, y) + bDP(x, y). \tag{3.49}$$

Vamos reformular o problema da interpolação resolvendo-o de forma mais geral como um problema envolvendo funcionais lineares: encontre $Q(x, y)$ que satisfaça um conjunto de equações da forma:

$$D_iQ(x, y) = 0,\ i = 1, 2, ..., C \tag{3.50}$$

onde cada $D_i$ é um funcional linear. Para o problema da interpolação, cada $D_i$ corresponde a um $D_{r,s}$, de acordo com a ordenação monomial escolhida.

Seja $\mathbb{F}_L[x, y] \subset \mathbb{F}[x, y]$ o conjunto de polinômios cujo grau de $y \leq L$. Então qualquer

$Q(x,y) \in \mathbb{F}_L[x,y]$ pode ser escrito na forma:

$$Q(x,y) = \sum_{k=0}^{L} q_k(x) y^k \tag{3.51}$$

onde $q_k(x) \in \mathbb{F}[x]$

Para um funcional linear $D$, denomina-se $K_D$ como o núcleo de $D$:

$$K_D = \{Q(x,y) \in \mathbb{F}[x,y] : DQ(x,y) = 0\} \tag{3.52}$$

Para um conjunto de funcionais lineares $D_1, D_2, ..., D_C$ definidos em $\mathbb{F}_L[x,y]$, e sendo $K_1, K_2, ..., K_C$ os núcleos correspondentes, então:

$$K_i = \{Q(x,y) \in \mathbb{F}[x,y] : D_i Q(x,y) = 0\} \tag{3.53}$$

Então a solução para o problema

$$D_i Q(x,y) = 0 \text{ para todo } i = 1, 2, ..., C \tag{3.54}$$

está na intersecção dos núcleos $K = K_1 \cap K_2 \cap \cdots \cap K_C$. Assim para resolver o problema da interpolação basta encontrar o polinômio de mínimo grau ponderado em $K$. Para encontrar as interseções de maneira gradativa, vamos empregar núcleos cumulativos, definidos como $\bar{K}_0 = \mathbb{F}_L[x][y]$, onde $\mathbb{F}_L[x][y] \subset \mathbb{F}[x][y]$ e denomina o conjunto de polinômios cujo grau da variável $y$ é $\leq L$.

$$\bar{K}_i = \bar{K}_{i-1} \cap K_i = K_1 \cap \cdots \cap K_i \tag{3.55}$$

Isto é, $\bar{K}_i$ é o conjunto das primeiras $i$ soluções para o problema da equação 3.54. A solução da interpolação é o polinômio de mínimo $(1, v)$ grau ponderado em $\bar{K}_C$.

Vamos particionar os polinômios em $\mathbb{F}_L[x,y]$ de acordo com o expoente de $y$. Seja

$$S_j = \{Q(x,y) \in \mathbb{F}_L[x,y] : LM(Q) = x^i y^j\} \text{ para um } i \text{ qualquer} \tag{3.56}$$

onde $LM$ indica o monômio líder ou o monômio de maior grau de $Q(x,y)$, segundo a ordenação de graus ponderados. Assim $S_j$ é o conjunto de todos os polinômios $Q(x,y)$ cujo monômio de maior grau possui a incógnita $y$ com expoente $j$. Seja $g_{i,j}$ o polinômio de menor grau ponderado do conjunto $\bar{K}_i \cap S_j$. Então $\{g_{C,j}\}_{j=0}^{L}$ é o conjunto de todos os polinômios que satisfazem todas as condições da equação 3.54.

O algoritmo de Koetter gera um conjunto de polinômios $(G_0, G_1, ...G_C)$,

$$G_i = (g_{i,0}, g_{i,1}, ..., g_{i,L}) \tag{3.57}$$

onde $g_{i,j}$ é o elemento de menor grau de $\bar{K}_i \cap S_j$, isto é que satisfaz as primeiras $i$ condições para grau de $y$ igual a $j$. Então a saída do algoritmo é o elemento de $G_C$ de menor ordem, que tem polinômios que satisfazem as $C$ equações:

$$Q(x,y) = min_{0 \leq j \leq L} g_{C,j}(x,y) \tag{3.58}$$

Antes de prosseguir é importante apresentar algumas propriedades importantes associadas aos funcionais lineares. O funcional linear será utilizado no algoritmo de interpolação para o cálculo das discrepâncias, que será visto mais adiante. Para um dado funcional linear $D$, o mapeamento $[\cdot,\cdot]_D : \mathbb{F}[x,y] \times \mathbb{F}[x,y] \mapsto \mathbb{F}[x,y]$ é definido por:

$$[P(x,y), Q(x,y)]_D = (DQ(x,y))P(x,y) - (DP(x,y))Q(x,y) \tag{3.59}$$

**Teorema 3.6.1** *Para todo $P(x,y), Q(x,y) \in \mathbb{F}[x,y]$, $[P(x,y), Q(x,y)]_D \in K_D$ e considerando que $P(x,y) > Q(x,y)$, com respeito a alguma ordenação monomial e $Q(x,y) \notin K_D$, então o rank de $\{[P(x,y), Q(x,y)]_D\} = rank\{P(x,y)\}$.*

O algoritmo é inicializado com

$$G_0 = (g_{0,0}, g_{0,1}, ..., g_{0,L}) = (1, y, y^2, ..., y^L). \tag{3.60}$$

Para calcular $G_{i+1}$, dado $G_i$, primeiro obtemos o seguinte conjunto:

$$J_i = \{j : D_{i+1}(g_{i,j}) \neq 0\} \tag{3.61}$$

como o conjunto dos polinômios em $G_i$ que não satisfazem a $i$-ésima + 1 condição. Se $J$ não for vazio, então existe uma discrepância e é necessário atualizar os valores. Neste caso, seja $j^*$ o índice do polinômio de rank mínimo,

$$j^* = argmin_{j \in J_i} g_{i,j} \tag{3.62}$$

e seja $f$ dado por,

$$f = min_{j \in J_i} g_{i,j} \tag{3.63}$$

A regra de atualização é:

$$y = \begin{cases} g_{i,j} & \text{se } j \notin J_i \\ [g_{i,j}, f]_{D_{i+1}} & \text{se } j \in J_i \text{ e } j \neq j^* \\ [xf, f]_{D_{i+1}} & \text{se } j = j^* \end{cases}$$

No final do algoritmo é selecionado o menor elemento de $G_C$ como $Q_0(x, y)$. O algoritmo de Koetter que resolve o problema de interpolação polinomial é mostrado no Algoritmo 3.6.1.

Para entender melhor o funcionamento do algoritmo considere o seguinte exemplo didático.

Os pontos $(1, \alpha^3)$, $(\alpha, \alpha^4)$, $(\alpha^2, \alpha^5)$, $(\alpha^3, \alpha^7)$ e $(\alpha^4, \alpha^8)$ devem ser interpolados através de um polinômio $Q(x, y)$ através da utilização do algoritmo 3.6.1, onde as operações devem ocorrer sobre o corpo $\mathbb{F} = GF(2^4)$ gerado pelo polinômio primitivo $1+\alpha+\alpha^4 = 0$. Usando multiplicidade $m = 1$ e critério de ordenação revlex com pesos $(u, v) = (1, 2)$.

Início: $G_0 : g_0(x, y) = 1, g_1(x, y) = y, g_2(x, y) = y^2, g_3(x, y) = y^3, g_4(x, y) = y^4$.

i=0: $(r, s) = (0, 0)$, $(x_i, y_i) = (1, \alpha^3)$

Discrepâncias: $\Delta_0 = g_0(1, \alpha^3) = 1$, $\Delta_1 = g_1(1, \alpha^3) = \alpha^3$, $\Delta_2 = g_2(1, \alpha^3) = \alpha^6$, $\Delta_3 =$

**Algoritmo 3.6.1** Algoritmo de Interpolação de Koetter

**Entrada:** Pontos $(x_i, y_i), i = 1, ..., n$; ordem de interpolação $m$; os pesos $(1, v)$ da ordenação monomial; número máximo de polinômios de resposta $L = L_m$

**Saída:** $Q_0(x, y)$ que interpola os pontos de entrada com multiplicidade $m$.

**Inicialização:** $g_j = y^j$ para $j = 0$ a $L$.

**for** $i = 1$ a $n$ **do** ▷ Percorre os $n$ pontos a serem interpolados
  $C \leftarrow m(m+1)/2$ ▷ Calcula quantas equações de derivadas deve-se ter
  **for** $(r, s) = (0, 0)$ até $(m-1, 0)$ utilizando ordenação lex (u,v)=$(m-1, 1)$ **do** ▷ De 0 a C
    **for** $j = 0$ até $L$ **do**
      $\Delta_j = D_{r,s} g_j(x_i, y_i)$ ▷ Calcula a discrepância
    **end for**
    $J = \{j : \Delta_j \neq 0\}$ ▷ Conjunto de discrepâncias diferentes de zero
    **if** $J \neq \emptyset$ **then**
      $j^*$ =arg min$\{g_j : j \in J\}$ ▷ Polinômio de menor grau ponderado
      $f = g_{j^*}$
      $\Delta = \Delta_{j^*}$
      **for** $j \in J$ **do**
        **if** $j \neq j^*$ **then**
          $g_j = \Delta g_j - \Delta_j f$
        **else**
          **if** $j = j^*$ **then**
            $g_j = (x - x_i) f$
          **end if**
        **end if**
      **end for**
    **end if**
  **end for**
**end for**
$Q_0(x, y) = min_j\{g_j(x, y)\}$

$g_3(1,\alpha^3)=\alpha^9$, $\Delta_4=g_4(1,\alpha^3)=\alpha^{12}$. $J=\{0,1,2,3,4\}$, $j^*=0$, $f=1$, $\Delta=1$

$G_1: g_0=1+x, g_1=\alpha^3+y, g_2=\alpha^6+y^2, g_3=\alpha^9+y^3, g_4=\alpha^{12}+y^4$.

i=1: $(r,s)=(0,0)$, $(x_i,y_i)=(\alpha,\alpha^4)$

Discrepâncias: $\Delta_0=\alpha^4$, $\Delta_1=\alpha^7$, $\Delta_2=\alpha^{14}$, $\Delta_3=\alpha^8$, $\Delta_4=\alpha^{13}$. $J=\{0,1,2,3,4\}$, $j^*=0$, $f=1+x$, $\Delta=\alpha^4$

$G_2$:$g_0=\alpha+\alpha^4x+x^2$, $g_1=\alpha^7x+\alpha^4y$, $g_2=(\alpha^{11}+\alpha^{14}x)+\alpha^4y^2$, $g_3=(\alpha^3+\alpha^8x)+\alpha^4y^3$, $g_4=(\alpha^{12}+\alpha^{13}x)+\alpha^4y^4$.

i=2: $(r,s)=(0,0)$, $(x_i,y_i)=(\alpha^2,\alpha^5)$

Discrepâncias: $\Delta_0=\alpha^{13}$, $\Delta_1=0$, $\Delta_2=\alpha^8$, $\Delta_3=\alpha^6$, $\Delta_4=\alpha^2$. $J=\{0,2,3,4\}$, $j^*=0$, $f=\alpha+\alpha^4x+x^2$, $\Delta=\alpha^{13}$

$G_3$:$g_0=\alpha^3+\alpha^{11}x+\alpha^{10}x^2+x^3$, $g_1=\alpha^7x+\alpha^4y$, $g_2=\alpha^8x^2+\alpha^2y^2$, $g_3=(\alpha^{14}+\alpha^7x+\alpha^6x^2)+\alpha^2y^3$, $g_4=(\alpha^{12}+\alpha x+\alpha^2x^2)+\alpha^2y^4$.

i=3: $(r,s)=(0,0)$, $(x_i,y_i)=(\alpha^3,\alpha^7)$

Discrepâncias: $\Delta_0=\alpha^{14}$, $\Delta_1=\alpha^{14}$, $\Delta_2=\alpha^7$, $\Delta_3=\alpha^2$, $\Delta_4=\alpha^3$. $J=\{0,1,2,3,4\}$, $j^*=1$, $f=\alpha^7x+\alpha y^4$.

$G_4$:$g_0=\alpha^2+\alpha^7x+\alpha^9x^2+\alpha^{14}x^3+\alpha^3y$, $g_1=(\alpha^{10}x+\alpha^7x^2)+(\alpha^7+\alpha^4x)y$, $g_2=(\alpha^{14}x+\alpha^7x^2)+\alpha^{11}y+\alpha y^2$, $g_3=(\alpha^{13}+\alpha^5x+\alpha^5x^2)+\alpha^6y+\alpha y^3$, $g_4=(\alpha^{11}+\alpha^5x+\alpha x^2)+\alpha^7y+\alpha y^4$.

i=4: $(r,s)=(0,0)$, $(x_i,y_i)=(\alpha^4,\alpha^8)$

Discrepâncias: $\Delta_0=\alpha^{11}$, $\Delta_1=\alpha^8$, $\Delta_2=\alpha^{11}$, $\Delta_3=\alpha^2$, $\Delta_4=\alpha^{10}$. $J=\{0,1,2,3,4\}$, $j^*=0$, $f=(\alpha^2+\alpha^7x+\alpha^9x^2+\alpha^{14}x^3)+\alpha^3y$.

$G_5$:$g_0=(\alpha^6+\alpha^9x+\alpha^5x^2+\alpha x^3+\alpha^{14}x^4)+(\alpha^7+\alpha^3x)y$, $g_1=(\alpha^9+\alpha^8x+\alpha^9x^2+\alpha^6x^3)+(\alpha^{12}+x)y$, $g_2=(\alpha^{13}+\alpha^{12}x+\alpha^{11}x^2+\alpha^{10}x^3)+\alpha y+\alpha^{12}y^2$, $g_3=(\alpha^{14}+\alpha^3x+\alpha^6x^2+\alpha x^3)+\alpha y+\alpha^{12}y^3$, $g_4=(\alpha^2+\alpha^5x+\alpha^6x^2+\alpha^9x^3)+\alpha^8y+\alpha^{12}y^4$.

Resultado:$Q(x,y) = g_1 = (\alpha^9 + \alpha^8 x + \alpha^9 x^2 + \alpha^6 x^3) + (\alpha^{12} + x)y$

## 3.7 Algoritmo de Fatoração de Roth-Ruckenstein

Uma vez obtido o polinômio $Q(x,y)$ que interpola o conjunto de pontos $(x_i, y_i), i = 1, 2, ..., n$, o próximo passo no algoritmo de decodificação de Guruswami-Sudan é a determinação de todos os fatores, da forma $y - p(x)$ (onde $p(x)$ é um polinômio de grau $\leq v$), que a compõem, ou seja, $(y - p(x))|Q(x,y)$. Para entendimento do algoritmo devem-se considerar os seguintes Teoremas (as provas serão omitidas e podem ser encontradas em [18]):

**Teorema 3.7.1** *$(y - p(x))|Q(x,y)$ se, e somente se, $Q(x, p(x)) = 0$.*

Este é um caso análogo ao que ocorre com polinômios de uma variável que diz :

$$(x - a)|g(x) \text{ se e somente se, } g(a) = 0. \tag{3.64}$$

Assim $p(x)$ é dito como a raiz de $y$ de $Q(x,y)$. O algoritmo descrito nesta seção, (o algoritmo de Roth-Ruckenstein [27]), encontra as raízes de $y$ do polinômio. Outros algoritmos que fazem isso de maneira igualmente eficiente podem ser encontrados em [28].

A notação $\langle\langle Q(x,y)\rangle\rangle$ é utilizada para denominar a divisão de $Q(x,y)$ pela incógnita $x$ elevada ao maior expoente de $x$ de $Q(x,y)$. Ou seja, se $x^r|Q(x,y)$ mas $x^{r+1} \nmid Q(x,y)$; ($x^{r+1}$ não divide $Q(x,y)$), assim:

$$\langle\langle Q(x,y)\rangle\rangle = \frac{Q(x,y)}{x^r} \tag{3.65}$$

para um $r \geq 0$.

Por exemplo, se $Q(x,y) = xy$ então $\langle\langle Q(x,y)\rangle\rangle = y$. Se $Q(x,y) = x^3y^4 + x^4y^5$ então $\langle\langle Q(x,y)\rangle\rangle = y^4 + xy^5$. Se $Q(x,y) = 1 + xy^3 + x^4y^6$ então $\langle\langle Q(x,y)\rangle\rangle = Q(x,y)$.

Seja $p(x) = a_0 + a_1x + a_2x^2 + ... + a_vx^v$ uma raiz de $y$ de $Q(x,y)$, então o algoritmo de Rock-Ruckenstein irá determinar os coeficientes de $p(x)$ um de cada vez. O coeficiente $a_0$ é encontrado através do seguinte teorema:

**Teorema 3.7.2** *Seja* $Q_0(x,y) = \langle\langle Q(x,y)\rangle\rangle$. *Se* $(y - p(x))|Q(x,y)$, *então* $p(0) = a_0$ *é uma raiz da equação* $Q_0(0,y) = 0$.

Deste teorema pode-se notar que o conjunto de possíveis valores para $a_0$ serão as raízes do polinômio $Q_0(x,y)$. A partir daí o algoritmo funciona de maneira indutiva, ou seja, translada-se a origem de $Q(x,y)$ para $a_0$ e assim aplica-se novamente o Teorema 3.7.2, obtendo-se agora como raízes o coeficiente $a_1$ e assim por diante. A operação do algoritmo é mostrado no Algoritmo 3.7.1.

**Algoritmo 3.7.1** Algoritmo de Fatoração de Rock-Ruckenstein

**Entrada:** $Q(x,y)$ e $k$, o máximo grau que $p(x)$ pode ter
**Saída:** Lista de polinômios $p(x)$ de grau $\leq k$
**Inicialização:** Faça $p(x) = 0$, $u = grau(p) = -1$, $k$ deve ser uma variável global interna
Cria uma lista encadeada onde os polinômio serão salvos
Faça $v = 0$ ▷ $v$ indica o número do nó (variável global)
Chama $rothrucktree(Q(x,y), u, p)$

**Algoritmo 3.7.2** Algoritmo de Fatoração de Rock-Ruckenstein - Função Rothrucktree

**Entrada:** $Q(x,y)$, $p(x)$ e $u$, grau de $p(x)$
**Saída:** Lista de polinômios $p(x)$ de grau $\leq k$

$v = v + 1$ ▷ Vai para o próximo nó
**if** $Q(x,0) = 0$ **then**
  Adiciona $p(x)$ na lista de saída
**else**
  **if** $u < k$ **then** ▷ Tenta outro ramo da árvore
    $R$ =lista de raízes de $Q(0,y)$
    **for** Para cada raiz $\alpha$ em $R$ **do**
      $Q_{new}(x,y) = Q(x, xy + \alpha)$ ▷ Translada o polinômio para a raiz
      $p_{u+1} = \alpha$ ▷ Novo coeficiente de $p(x)$
      Chama $rothrucktree(\langle\langle Q_{new}(x,y)\rangle\rangle, u+1, p)$ ▷ Chamada recursiva
    **end for**
  **else** ▷ Chegou ao fim de um ramo da árvore com um polinômio diferente de zero
    Não gera saída
  **end if**
**end if**

Exemplo, seja $Q(x,y) = (4 + 4x^2 + 2x^3 + 3x^4 + x^5 + 3x^6 + 4x^7) + (1 + 2x + 2x^2 + 3x^4 + 3x^6)y + (1 + x + 2x^2 + x^3 + x^4)y^2 + (4 + 2x)y^3$ um polinômio com coeficientes em $\mathbb{F} = GF(5)$ e seja o grau máximo de $p(x)$, igual a 2 ($k = 2$).

1. No nó 1, obtêm-se o polinômio $Q(0, y) = 4 + y + y^2 + 4y^3$ e suas raízes são calculadas $\{1, 4\}$

2. No nó 1, a raiz $\{1\}$ é selecionada. O nó 2 é chamado com $\langle\langle Q(x, xy + 1)\rangle\rangle$.

3. No nó 2, obtêm-se o polinômio $Q(0, y) = 3 + 3y^2$ com raízes $\{2, 3\}$.

4. No nó 2, a raiz $\{2\}$ é selecionada. O nó 3 é selecionado com $\langle\langle Q(x, xy + 2)\rangle\rangle$.

5. No nó 3, o polinômio $Q(x, 0) = 0$, então a lista de raízes selecionadas para estes nós $\{1, 2\}$ formam o seguinte polinômio $p(x) = 1 + 2x$.

6. Volta-se para o nó 2, onde a raiz $\{3\}$ é selecionada, e o nó 4 é chamado com $\langle\langle Q(x, xy + 3)\rangle\rangle$.

7. No nó 4, obtêm-se o polinômio $Q(0, y) = 2 + 3y$ com raiz $\{1\}$

8. No nó 4, a raiz $\{1\}$ é selecionada. O nó 5 é chamado com $\langle\langle Q(x, xy + 1)\rangle\rangle$.

9. No nó 5, como o nível na árvore é 3 que é maior que $k$, assim nenhum polinômio de saída foi obtido

10. Devido a recursão retorna-se ao nó 1, onde a raiz $\{4\}$ é selecionada e o nó 6 é chamado com $\langle\langle Q(x, xy + 4)\rangle\rangle$.

11. No nó 6, obtêm-se o polinômio $Q(0, y) = 2 + y$, com raiz $\{3\}$ e o nó 7 é chamado com $\langle\langle Q(x, xy + 3)\rangle\rangle$.

12. No nó 7, obtêm-se o polinômio $Q(0, y) = 3 + y$, com raiz $\{2\}$. O nó 8 é chamado com $\langle\langle Q(x, xy + 2)\rangle\rangle$.

13. No nó 8, obtêm-se $Q(x, 0) = 0$, então a lista de raízes selecionadas até este nó que foram $\{4, 3, 2\}$ formam um polinômio de saída $p(x) = 4 + 3x + 2x^2$

O conjunto de polinômios produzido pelo algoritmo é $1 + 2x, 4 + 3x + 2x^2$.

Neste capítulo foi mostrado em detalhes como Guruswami e Sudan chegaram na decodificação algébrica de códigos de Reed-Solomon, e que a mesma pode ultrapassar a

capacidade de correção convencional para códigos de taxa $< 1/3$. Vamos ver no próximo capítulo que podemos atingir a decodificação por decisão suave de códigos de RS se ao invés de aplicarmos a mesma multiplicidade de interpolação para cada ponto recebido do canal, atribuirmos multiplicidades diferentes para cada ponto fazendo essa ponderação através dos valores suaves calculados para cada ponto recebido do canal.

# CAPÍTULO 4

# A DECODIFICAÇÃO DE CÓDIGOS DE RS POR DECISÃO SUAVE

## 4.1 Introdução

Nos primórdios do desenvolvimento da teoria de codificação, era mais simples e conveniente modelar os canais de comunicação de maneira que tivessem um conjunto finito de símbolos em sua saída. Assim os efeitos causados pelo ruído no canal eram representados pela recepção ocasional de um outro símbolo diferente daquele que foi transmitido. Este modelamento ou abstração da realidade permite que se aplique uma série de artifícios algébricos, probabilísticos e combinatórios que facilitam o desenvolvimento de códigos e de algoritmos de decodificação. O código de Reed-Solomon é um exemplo de código que foi concebido através dessa abstração da realidade.

O que acontece na prática, entretanto, é que o ruído inserido no canal está sempre ocorrendo, de maneira menos ou mais pronunciada, mas está continuamente influenciando o sinal recebido. Desse jeito, mesmo que esteja se transmitindo a informação através de um conjunto finito de símbolos, no receptor o que se tem são valores que podem assumir infinitos níveis, devido à influência do canal.

Essa percepção levou à pesquisa e desenvolvimento de algoritmos de decodificação que aceitassem vetores de amostras reais da saída do receptor e assim tentar estimar qual símbolo foi transmitido. Este paradigma ficou conhecido como decodificação por decisão suave. Já o decodificador por decisão abrupta caracteriza o algoritmo que somente aceita que sua entrada seja de símbolos tirados de um conjunto finito. Uma descoberta que motivou a pesquisa no campo da decodificação por decisão suave é que se podem obter ganhos de desempenho da ordem de 2 dB em relação à decisão abrupta para canais Gaussianos.

Apesar do conhecimento de que se poderiam obter ganhos razoáveis de decodificação através da decisão suave, ela não foi, em primeiro momento, cogitada a ser utilizada na decodificação de códigos de Reed-Solomon por dois motivos: o primeiro era devido à natureza algébrica do código que não tornava simples a adaptação para decisão suave e o segundo porque muitos achavam que era desnecessário, devido à característica destes códigos de ter a máxima distância possível entre as palavras códigos.

Porém com o rápido avanço da área de telecomunicações e armazenamento digital, surgia cada vez mais a necessidade de se operar em ambientes de comunicação extremamente hostis, e com taxas de transmissão cada vez mais altas. Assim a decodificação por decisão suave era a melhor forma de se melhorar o desempenho de códigos de RS. Além disso, o advento de esquemas de modulação $M$-árias, como a modulação QAM, não tinham como ser usadas de maneira efetiva a não ser que fosse utilizada decisão suave [29].

Assim teve início uma grande onda de pesquisas para encontrar algoritmos de decodificação de códigos de RS por decisão suave. Contudo, apesar da grande quantidade de pesquisas na área, nenhum dos algoritmos apresentados conseguia aproveitar a estrutura algébrica do código para realizar a decodificação por decisão suave. Mais detalhes sobre o histórico da decodificação de códigos de Reed-Solomon por decisão suave, recomenda-se consultar [30]. Até que em 2003, Alexander Vardy e Ralf Kötter publicaram o artigo "Algebraic Soft-Decision Decoding of Reed-Solomon Codes"[5], mostrando que através de uma modificação no algoritmo de Guruswami-Sudan podia-se atingir a decodificação de códigos de Reed-Solomon por decisão suave com ganhos de desempenho da ordem de 1,5 dB e fazendo isso utilizando a estrutura algébrica do código. Este novo algoritmo ficou conhecido pelo nomes de seus autores, como o algoritmo de Koetter-Vardy (KV) para decodificação de códigos de Reed-Solomon por decisão suave. No capítulo anterior mostrou-se que o algoritmo de Guruswami-Sudan possui um parâmetro $m$ que representa a multiplicidade da interpolação. Contudo esse parâmetro é fixo para todos os pontos interpolados pelo algoritmo. No algoritmo de Koetter-Vardy, esse paramêtro passa a ser variável, mudando de valor de acordo com o ponto a ser interpolado. O valor de $m$ para cada ponto aumenta ou diminui de acordo com as informações suaves observadas no ca-

nal, ou seja, se o ponto tiver um alto grau de confiabilidade a multiplicidade $m$ aumenta, caso contrário ela diminui. Uma vez estabelecidas as multiplicidades para cada ponto o algoritmo de KV passa pela fase de interpolação seguida da fase de fatoração, como no algoritmo de GS.

Será apresentado aqui apenas o algoritmo de KV para canais sem memória, que é mais que suficiente para mostrar o desempenho contundente desse novo algoritmo. Para resultados em canais com memória ou canais concatenados sugere-se que se consulte [5].

## 4.2 Canais sem Memória e Probabilidade Condicional

Um canal sem memória [28] é aquele cuja saída, $y_n$, em um determinado instante $n$ depende somente da entrada, $x_n$, ocorrida no mesmo instante $n$, sem importar as entradas nos instantes anteriores ou posteriores a $n$. Assim, dado uma entrada no instante $n$, $x_n$, a saída no mesmo instante de tempo $y_n$ é estatisticamente independente das saídas em outros instantes de tempo. Isto é, para uma seqüência de sinais recebidos

$$\boldsymbol{Y} = \{y_1, y_2, ..., y_n\} \tag{4.1}$$

e uma seqüência de sinais transmitidos

$$\boldsymbol{X} = \{x_1, x_2, ..., x_n\}, \tag{4.2}$$

diz-se que a probabilidade de se observar $\boldsymbol{Y}$ na saída dado que $\boldsymbol{X}$ foi transmitido é dada pela equação:

$$f(y_1, y_2, ..., y_n | x_1, x_2, ..., x_n) \tag{4.3}$$

que pela característica de cada saída ser estaticamente independentes dos outros instantes de tempo, pode ser reescrita como:

$$f(y_1, y_2, ..., y_n | x_1, x_2, ..., x_n) = \prod_{i=1}^{n} f(y_i | x_i) \tag{4.4}$$

Onde $f(\cdot|x)$ é a função de densidade de probabilidade. Assume-se que $X$ seja uma variável aleatória uniformemente distribuída, ou seja, que cada palavra código em $X$ seja selecionada com igual probabilidade. Assim, se quisermos saber a probabilidade de um símbolo transmitido $x = \alpha$ dado que foi observado um símbolo $y$ na saída, pode-se encontrar essa probabilidade utilizando o teorema de Bayes:

$$P(x = \alpha|y) = \frac{f(y|\alpha)P(x = \alpha)}{\sum_{x \in X} f(y|x)} \tag{4.5}$$

## 4.3 Matriz de Confiabilidade

Para um código de Reed-Solomon, o conjunto de símbolos de entrada vêm do corpo $\mathbb{F}$ sobre o qual os símbolos estão definidos, dessa maneira

$$X = GF(q) = \{\alpha_1, \alpha_2, ..., \alpha_q\}. \tag{4.6}$$

E seja

$$\boldsymbol{Y} = (y_1, y_2, ..., y_n) \tag{4.7}$$

o vetor de observações.

Então podemos definir a probabilidade a posteriori como:

$$\pi_{i,j} = P(X = \alpha_i|Y = y_j),\ i = 1, 2, ..., q,\ j = 1, 2, ..., n, \tag{4.8}$$

de acordo com a equação 4.5, que irá formar uma matriz $\Pi$ com $q$ linhas e $n$ colunas, denominada de matriz de confiabilidade [18]. Então as linhas de $\Pi$ representarão todos os elementos que compõem o corpo $\mathbb{F}$, enquanto que as colunas representam as posições ocupadas por estes elementos no vetor recebido. Por exemplo, se estivermos trabalhando com o código RS(7,3) sobre o corpo $\mathbb{F} = GF(2^3)$ então a matriz de confiabilidade terá 8 linhas que estarão representando os elementos $\{0, 1, \alpha, \alpha^2, ..., \alpha^6\}$, e terá 7 colunas que representam as posições dos elementos do corpo no vetor recebido, como mostrado na matriz abaixo.

$$\mathbf{\Pi} = \left(\begin{array}{c|cccc} & y_1 & y_2 & \cdots & y_n \\ \hline 0 & \pi_{1,1} & \pi_{1,2} & \cdots & \pi_{1,7} \\ 1 & \pi_{2,1} & \pi_{2,2} & \cdots & \pi_{2,7} \\ \alpha & \pi_{3,1} & \pi_{3,2} & \cdots & \pi_{3,7} \\ \alpha^2 & \pi_{4,1} & \pi_{4,2} & \cdots & \pi_{4,7} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \alpha^6 & \pi_{8,1} & \pi_{8,2} & \cdots & \pi_{8,7} \end{array}\right)$$

Pode-se ver que a matriz de confiabilidade nada mais é que relacionar as probabilidades de cada um dos símbolos possíveis (linha) com sua posição no vetor recebido (coluna).

Assim se o elemento $\pi_{3,2} = \pi_{\alpha,2} = 0.8$, isto quer dizer que o símbolo $\alpha$ tem 80% de probabilidade de ser o símbolo transmitido para a segunda posição do vetor recebido. Pode-se perceber que se somarmos os valores de todos os elementos que compõem a coluna da matriz de confiabilidade temos que ter um valor tendendo a 1 (só será igual a 1 se a matriz de confiabilidade tiver infinitas casas decimais para representar os valores que a compõem).

Assim vamos assumir como sendo $\Pi$ a entrada do algoritmo de decodificação por decisão suave; este tipo de consideração é bem razoável pois em muitas aplicações ([31], [32], [33]) é a matriz de confiabilidade que está disponível na entrada do decodificador ao invés do vetor $\boldsymbol{Y}$. Uma vez inserida a matriz $\Pi$, o algoritmo de decodificação algébrica por decisão suave converte esta matriz em pontos a serem interpolados e suas respectivas multiplicidades que servirão então de entrada para o algoritmo de KV.

## 4.4 O algoritmo de Koetter-Vardy para Decodificação por Decisão Suave

O algoritmo de KV [5] pode ser dividido em 3 fases distintas para a sua melhor compreensão como mostra a figura 4.1.

A primeira fase, denominada de Atribuição das Multiplicidades, recebe da saída do

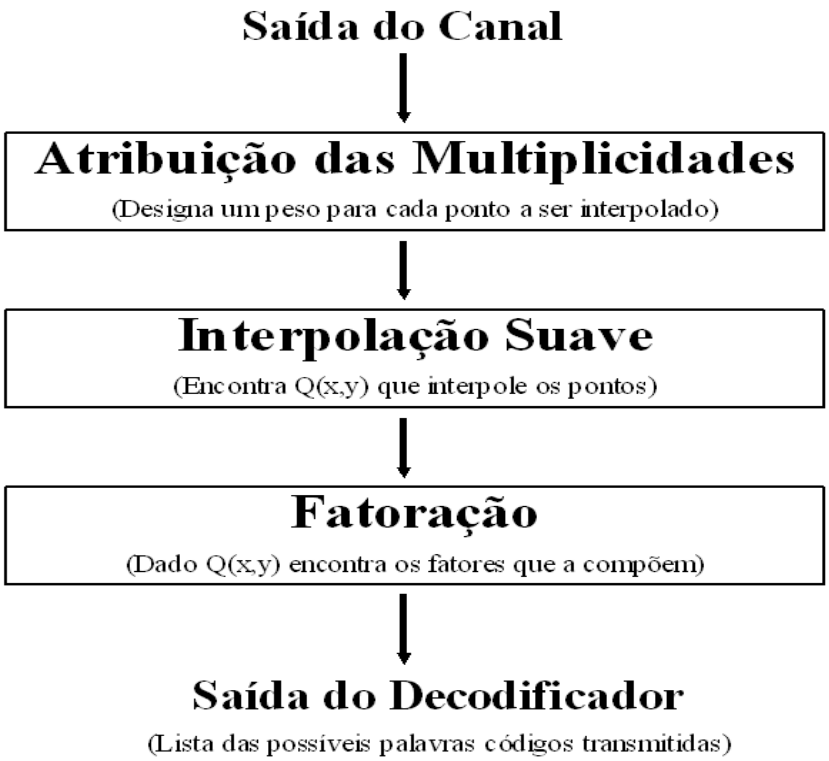



Figura 4.1: Visão Geral do Algoritmo de Koetter-Vardy

canal a matriz de confiabilidade $\Pi$ e a converte em uma matriz de multiplicidade $M$. A matriz de multiplicidade tem a mesma dimensão que a matriz $\Pi$ $(q \times n)$, e seus elementos $m_{i,j}$ devem ser números inteiros não negativos. Os valores dessa matriz que forem maiores que zero formarão os pontos a serem interpolados com a respectiva ordem de multiplicidade atribuída. Esta fase é a mais importante do algoritmo de KV, pois é ela que determina o desempenho final do decodificador.

A segunda fase realiza a interpolação suave obtendo um polinômio de duas variáveis $Q(x, y)$ através da matriz $M$ obtida na fase anterior. Esta fase difere da interpolação de GS no fato de não precisar ter valores de multiplicidades iguais para cada ponto a ser interpolado, além de que agora podemos ter mais de $n$ pontos a serem interpolados, diferente do algoritmo de GS que limita em exatos $n$ pontos.

Na terceira fase, faz-se a fatoração, ou seja, encontram-se todos os fatores do tipo $y - p(x)$ que compõem $Q(x, y)$. Na sua saída teremos a lista de polinômios que serão as possíveis palavras código transmitidas. Esta fase em conjunto com a interpolação determinarão a complexidade do algoritmo de decodificação.

No resto dessa seção, iremos caracterizar as condições nas quais o decodificador irá produzir a palavra código transmitida, para um dado conjunto de pontos a serem interpolados e suas respectivas multiplicidades, isto é, para uma dada matriz $M$. O resultado disso será uma expressão que permitirá a simulação teórica do algoritmo de KV, ou seja, poderemos obter o desempenho do algoritmo sem ter a necessidade de implementá-lo.

Para uma matriz $M$ de dimensão $q \times n$, composta por valores inteiros não negativos, $m_{i,j}$, define-se o custo de $M[5]$ como:

$$C_M \triangleq \frac{1}{2} \sum_{i=1}^{q} \sum_{j=1}^{n} m_{i,j}(m_{i,j}+1) \tag{4.9}$$

Como foi mostrado no capítulo anterior, achar a equação $Q(x,y)$ que interpole todos os pontos é equivalente a resolver um sistema de equações lineares do tipo da equação 3.23. Como uma raiz de ordem $m$ faz com que se tenham $m(m+1)/2$ derivadas de $Q(x,y)$, nesta raiz, iguais a zero, o custo de $M$, é exatamente o número total de equações lineares. Como mostrado na equação 3.34, pode-se sempre encontrar um polinômio $Q(x,y)$ diferente de zero se o passo de interpolação prover um polinômio de grau ponderado $\delta$, com critério de peso $(1, k-1)$, de maneira que o número de coeficientes dos monômios seja maior que o custo de $M$, então:

$$N_{1,k-1}(\delta) > C_M \tag{4.10}$$

Dessa maneira, o número de graus de liberdade é maior que o número de equações lineares. Assim, pode-se definir a função:

$$\Delta_{w_x,w_y}(z) \triangleq min\{\delta \in \mathbb{Z} : N_{w_x,w_y}(\delta) > z\} \tag{4.11}$$

E lembrando que $\Delta$ fornece o grau ponderado mínimo que precisamos ter em $Q(x,y)$ para que a inequação 4.10 seja verdadeira. Observe que se utilizarmos o limitante inferior dado na equação 3.8, podemos escrever a equação 4.11 como[5]:

$$\Delta_{1,k-1}(z) < \sqrt{2(k-1)z} \tag{4.12}$$

Entrentanto a equação 4.12 é apenas um limitante inferior, por isso em [12] é mostrado que a função $\Delta_{1,k-1}(z)$ pode ser aproximada por:

$$\Delta_{1,k-1}(z) \approx \left\lfloor \sqrt{2(k-1)z} - \frac{k-1}{2} \right\rfloor \tag{4.13}$$

Vamos representar o vetor de observações $\boldsymbol{Y}$ como uma matriz $[\boldsymbol{Y}]$ de dimensão $q \times n$. Esta matriz será composta de zeros e uns, e cada coluna só poderá ter um 1, que estará indicando a linha da matriz que contém o símbolo de $GF(q)$ observado naquela posição do vetor. Por exemplo, para o código RS(7,3) sobre um corpo $\mathbb{F} = GF(2^3)$, seja o vetor de observações $\boldsymbol{Y} = (\alpha, 0, \alpha^2, \alpha^4, 0, \alpha^2, \alpha^6)$ então:

$$[\boldsymbol{Y}] = \left(\begin{array}{c|ccccccc} & y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \alpha & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \alpha^2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ \alpha^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \alpha^4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \alpha^5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \alpha^6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

Onde as primeiras linha e coluna são apenas para mostrar o símbolo e a posição no vetor que cada elemento da matriz representa. Para duas matrizes $A$ e $B$, o produto interno é definido como:

$$\langle A, B \rangle = AB^t = \sum_{i=1}^{q} \sum_{j=1}^{n} a_{i,j} b_{i,j} \tag{4.14}$$

Assim usando estes dois conceitos apresentados pode-se definir o escore de um vetor de observações $\boldsymbol{Y} = (y_1, y_2, ..., y_n)$ com relação a uma matriz de multiplicidade $M$ dada como[5]:

$$S_M(\boldsymbol{Y}) = \langle M, [\boldsymbol{Y}] \rangle \tag{4.15}$$

Então o escore representa a soma das multiplicidades de todos os pontos associados ao vetor $\boldsymbol{Y}$. Por exemplo, a matriz de multiplicidade genérica para o código RS(7,3) do exemplo anterior fica:

$$\boldsymbol{M} = \begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} & m_{1,5} & m_{1,6} & m_{1,7} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} & m_{2,5} & m_{2,6} & m_{2,7} \\ m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} & m_{3,5} & m_{3,6} & m_{3,7} \\ m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} & m_{4,5} & m_{4,6} & m_{4,7} \\ m_{5,1} & m_{5,2} & m_{5,3} & m_{5,4} & m_{5,5} & m_{5,6} & m_{5,7} \\ m_{6,1} & m_{6,2} & m_{6,3} & m_{6,4} & m_{6,5} & m_{6,6} & m_{6,7} \\ m_{7,1} & m_{7,2} & m_{7,3} & m_{7,4} & m_{7,5} & m_{7,6} & m_{7,7} \\ m_{8,1} & m_{8,2} & m_{8,3} & m_{8,4} & m_{8,5} & m_{8,6} & m_{8,7} \end{pmatrix}$$

Então o escore para o vetor $\boldsymbol{Y}$ do exemplo anterior será:

$$S_M(\boldsymbol{Y}) = \langle M, [\boldsymbol{Y}] \rangle = M[\boldsymbol{Y}]^t = m_{3,1} + m_{1,2} + m_{4,3} + m_{6,4} + m_{1,5} + m_{4,6} + m_{8,7} \tag{4.16}$$

Com os conceitos apresentados até aqui podemos agora enunciar o seguinte teorema[5]:

**Teorema 4.4.1** *Seja $C_M$ o custo de uma matriz de multiplicidade $M$ dada. Então o polinômio $Q(x, y)$ tem fatores do tipo $y - p(x)$, onde $p(x)$ determina o valor da palavra código $\boldsymbol{c} \in C(n, k)$, se o escore de $\boldsymbol{c}$ for maior que o grau ponderado mínimo determinado por $C_M$ na equação 4.10.*

$$S_M(\boldsymbol{c}) > \Delta_{1,k-1}(C_M) \tag{4.17}$$

Para prova deste Teorema sugere-se consultar [5]. Ainda mais importante, no sentido de utilidade prática, que o Teorema 4.4.1 apresentado, é a implicação do mesmo:

**Corolário 4.4.1** *Seja $C_M$ o custo de uma matriz de multiplicidade $M$ dada. Então $Q(x, y)$ tem um fator do tipo $y - p(x)$, onde $p(x)$ determina o valor da palavra código $\boldsymbol{c} \in C(n, k)$, se*

$$S_M(\boldsymbol{c}) > \left\lfloor \sqrt{2(k-1)C_M} - \frac{k-1}{2} \right\rfloor \tag{4.18}$$

Esse corolário sai direto do Teorema 4.4.1 e da equação 4.13.

A equação 4.18 é muito importante porque permite que se avalie o desempenho do algoritmo de Koetter-Vardy sem que se precise passar pelas duas fases de alta complexidade, que são as fases de interpolação e fatoração. Ou seja, uma vez calculada a matriz de confiabilidade de um vetor recebido do canal, procede-se para a primeira fase do algoritmo de KV onde se atribuem as multiplicidades que resultam na matriz de multiplicidades. Tendo $M$, obtêm-se $C_M$ através da equação 4.9, e, sabendo qual a palavra código transmitida $\boldsymbol{c}$ pode-se saber se o algoritmo terá sucesso na decodificação testando a condição da equação 4.18.

## 4.5 Atribuição das Multiplicidades

Essa fase do algoritmo é a mais importante pois ela determinará o desempenho do algoritmo. Nesse passo a idéia principal é tentar encontrar uma matriz de multiplicidade $M$ que maximize o escore $S_M$, que por conseqüência melhorará o desempenho do decodificador que é dado pela equação 4.18. Porém como o decodificador não sabe qual a palavra código recebida, o melhor que se pode fazer é tentar maximizar o escore para uma palavra transmitida escolhida de forma aleatória. Seja $\boldsymbol{X} = (X_1, X_2, ..., X_n)$, um vetor aleatório transmitido no canal. O escore desse vetor é $S_M = \langle M, [\boldsymbol{X}] \rangle$, assim se quisermos maximizar o escore, teremos que maximizar o seu valor médio ou esperado,

$$E[S_M(\boldsymbol{X})] = \sum_{\boldsymbol{x} \in \boldsymbol{X}^n} S_M P(\boldsymbol{x}) \tag{4.19}$$

de acordo com uma distribuição de probabilidades $P(\boldsymbol{x})$. Será adotada a distribuição de probabilidades determinada pela saída do canal, ou seja, usando o modelo para canais sem memória.

$$P(\boldsymbol{x}) = P(x_1, x_2, ..., x_n) = \prod_{j=1}^{n} P(\boldsymbol{X_j} = x_j | \boldsymbol{Y_j} = y_j) = \prod_{j=1}^{n} \Pi(x_j, j) \tag{4.20}$$

onde $\Pi$ é a matriz de confiabilidade. Entretanto este tipo de abordagem é extremamente difícil de ser calculado devido ao fato de $\boldsymbol{X}$ ser tirado de um conjunto de palavras de um código qualquer cujo modelo de distribuição das suas palavras não é suficientemente preciso [5]. Contudo pode-se sair utilizando um outro método que não é a melhor solução mas que pelo menos possibilita que se encontre uma expressão mais facilmente.

Reformulando o problema temos: Selecione a matriz $M \in \boldsymbol{M}$, com $\boldsymbol{M}$ sendo o conjunto de todas as possíveis matrizes $M$, que maximize o valor esperado $E[S_M(\boldsymbol{X})]$ com respeito à distribuição da equação 4.20. Iremos denominar a solução como $M(\Pi, C)$, onde

$$M(\Pi, C) = argmax_{M \in \boldsymbol{M}(C)} E[S_M(\boldsymbol{X})] \tag{4.21}$$

Assim considere o seguinte teorema[5]:

**Teorema 4.5.1** *O valor médio do escore com respeito à distribuição de probabilidades da equação 4.20 é igual ao produto interno entre a matriz de multiplicidade e a matriz de confiabilidade:*

$$E[S_M(\boldsymbol{X})] = \langle M, \Pi \rangle \tag{4.22}$$

Este teorema deriva da seguinte constatação:

$$E[S_M(\boldsymbol{X})] = E[\langle M, [\boldsymbol{X}] \rangle] = \langle M, E[\boldsymbol{X}] \rangle = \langle M, \Pi \rangle \tag{4.23}$$

A matriz $M(\Pi, C)$ é construída iterativamente, iniciando por uma matriz com todos os elementos iguais a zero, e aumentando cada um dos valores da matriz a cada iteração. O resultado é o algoritmo abaixo que faz o mapeamento da matriz de confiabilidade $\Pi$ para uma matriz de multiplicidades $M$.

O parâmetro $S$ do algoritmo apresentado controla a complexidade do mesmo, pois quanto maior o número $S$, maior o número de iterações. Pode ser mostrado [5], que quando fazemos $S \to \infty$ a matriz de multiplicidade $M \to \Pi$. Outra maneira de se perceber isso

**Algoritmo 4.5.1** Algoritmo de Koeter-Vardy para Atribuição de Multiplicidades

**Entrada:** Matriz de confiabilidade, $\Pi$, o número inteiro positivo $S$ indicando o número de iterações do algoritmo

**Saída:** A matriz de multiplicidades, $M$.

**Inicialização:** Faça $\Pi^* = \Pi$, e zere todos os valores de $M$

**while** $S > 0$ **do**

Encontre a posição $(i, j)$ do elemento de maior valor $\pi^*_{i,j}$ de $\Pi^*$

$\pi^*_{i,j} \leftarrow \frac{\pi^*_{i,j}}{m_{i,j}+2}$

$m_{i,j} \leftarrow m_{i,j} + 1$

$S \leftarrow S - 1$

**end while**

basta considerar uma matriz $\bar{M}$, como a matriz de multiplicidades $M$ (obtida através do algoritmo apresentado) normalizada, ou seja, normalizando $M$ por um fator que faça com que o somatório de cada coluna de $\bar{M}$ seja igual a 1. Pode ser mostrado que se isso for feito $\bar{M} \rightarrow \Pi$. Assim podemos dizer que o algoritmo produz uma matriz que quando normalizada parece muito com a matriz de confiabilidade. Como a cada iteração ocorre um incremento na multiplicidade, então o escore é aumentado de forma linear enquanto que o custo de $M$ cresce de forma quadrática com $S$.

Com isso, podemos concluir que a matriz $M$ deve ser escolhida de forma a ser proporcional a $\Pi$, que é exatamente o que acontece quando fazemos o número de iterações do algoritmo 4.5.1, $S$, tender a infinito. Pode ser mostrado [5] que para um número real $\lambda \geq 0$, existe uma matriz de multiplicidade $M$, gerada pelo algoritmo 4.5.1 com parâmetro $S$ tal que:

$$M = \lfloor \lambda \Pi \rfloor \tag{4.24}$$

Isto sugere um meio de se obter a matriz de multiplicidades escalonando a matriz $\Pi$ por um número real e depois truncando o resultado para que se obtenha uma matriz com números inteiros. Para matrizes de confiabilidade que são quantizadas para valores finitos, o resultado de 4.24 não será válido, isto é existem algumas matrizes que são geradas pelo algoritmo 4.5.1 que não podem ser geradas independente do valor de $\lambda$ em 4.24. Um exemplo disto é quando dois valores em $\Pi$ são exatamente iguais. Observe que para um canal gaussiano com valores de quantização infinitos, dois valores de $\Pi$ tem probabilidade

zero de serem iguais. Assim, na prática, apesar de não ser estritamente equivalente ao algoritmo 4.5.1, o seguinte algoritmo de complexidade reduzida com apenas uma iteração, sempre produz uma matriz de multiplicidade válida.

**Algoritmo 4.5.2** Algoritmo de Complexidade Reduzida de Koeter-Vardy para Atribuição de Multiplicidades

**Entrada:** Matriz de confiabilidade, $\Pi$, o número real positivo $\lambda$
**Saída:** A matriz de multiplicidades, $M$.

**for** $j = 0$ até $n - 1$ **do**
  **for** $i = 0$ até $q - 1$ **do**
    $m_{i,j} \leftarrow \lfloor \lambda \pi_{i,j} \rfloor$
  **end for**
**end for**

## 4.6 Fase de Interpolação Suave

A fase de interpolação do algoritmo de GS passa a ser chamada de fase de interpolação suave, ou também chamada de interpolação ponderada pois essa fase recebe como entrada uma matriz $M$ de multiplicidades obtida através dos dados contidos na matriz de confiabilidade, $\Pi$. Assim deixamos de ter como entradas o conjunto suporte $(x_1, x_2, ..., x_n)$(que nada mais é que todos os elementos do corpo de Galois excetuando-se o zero) e o vetor recebido através de decisão abrupta $\mathbf{r} = (y_1, y_2, ..., y_n)$, para utilizar todos elementos da matriz $M$ que forem diferentes de zero como pontos a serem interpolados.

Por exemplo, seja o código RS $(7, 3)$ sobre o corpo $\mathbb{F} = GF(2^3)$. Podemos ter depois da fase de atribuição de multiplicidades a seguinte matriz:

$$\boldsymbol{M} = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 & 3 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Então o conjunto de pontos a serem interpolados são:

$$\{(1,0),(\alpha,1),(\alpha,\alpha),(\alpha,\alpha^2),(\alpha^2,\alpha),(\alpha^3,1),(\alpha^4,\alpha^2),(\alpha^5,\alpha^2),(\alpha^5,\alpha^4),(\alpha^6,\alpha)\}$$

Com as respectivas multiplicidades $\{3,1,1,1,3,2,3,1,2,3\}$.

Então utilizando uma abordagem mais formal dizemos que a interpolação suave ou interpolação ponderada consiste em: Dado um conjunto de pontos $D = \{x_1, x_2, ...x_n\}$, que são os elementos que compõem o corpo $\mathbb{F}$ onde foi definido o código, e uma matriz de multiplicidades $M = [m_{i,j}]$, encontre o polinômio de duas variáveis $Q(x,y)$ de grau mínimo ponderado pelos pesos $(1, k-1)$, que possui raízes de multiplicidades $m_{i,j}$, nos pontos $(x_j, \alpha^i)$ para todo $i, j$ tais que $m_{i,j} \neq 0$.

## 4.7 Fase de Fatoração

Esta fase permanece idêntica a fase de Fatoração do algoritmo de Guruswami-Sudan; entretanto a expressão que determina o parâmetro $L_m$, que é o número máximo de polinômios retornados pelo algoritmo de decodificação de GS, ou seja, o número máximo de possíveis palavras códigos transmitidas, muda devido à utilização de multiplicidades diferentes para cada ponto. Este parâmetro é muito importante, pois é ele que determina a complexidade dessa fase.

O número de palavras código contidas na lista produzida por um decodificador de decisão suave para uma dada matriz de multiplicidades $M$ não pode exceder o seguinte valor:

$$L_m = \frac{\sqrt{\langle M, M\rangle + \langle M, 1\rangle}}{\sqrt{k-1}} \tag{4.25}$$

A análise e dedução da equação anterior não é o objeto deste trabalho e pode ser encontrada em [5].

## 4.8 Desempenho Assintótico

A melhor forma de se avaliar um algoritmo implementado na prática é comparar o seu desempenho com o desempenho assintótico do decodificador, que é o melhor desempenho que se pode esperar daquele método de decodificação, ou desempenho ideal do decodificador.

Como foi visto anteriormente, a parte que determina o desempenho do algoritmo de decodificação de Koetter-Vardy é a fase de atribuição de multiplicidades. Assim para atingir o desempenho assintótico é necessário que se obtenha uma matriz de multiplicidades $M$ através do algoritmo 4.5.1 fazendo com que $S \to \infty$. A dedução da fórmula é extensa e sugere-se que se consulte [5]. Assim o decodificador algébrico por decisão suave irá retornar uma palavra código $\boldsymbol{c} \in C(n,k)$ se:

$$\frac{\langle \Pi, [\boldsymbol{c}]\rangle}{\sqrt{\langle \Pi, \Pi\rangle}} \geq \sqrt{k-1} \tag{4.26}$$

## 4.9 Resultados de Simulações

Nesta seção é mostrado o ganho de desempenho obtido utilizando o algoritmo de decisão suave de Koetter-Vardy, para diferentes códigos. Todas as simulações foram feitas considerando um canal AWGN.

Foram feitas simulações através dos limitantes teóricos para o algoritmo de KV utilizando a equação 4.18. Para o algoritmo de Guruswami-Sudan e Berleykamp-Welch foram

utilizadas respectivamente as equações 3.44 e $t_0 = (n-k)/2$. Devido à linearidade do código, optou-se por utilizar um vetor de informação com todos os símbolos iguais a zero. Essa palavra, com todos os símbolos iguais a zero, é codificada, e transmitida no canal. No receptor calcula-se o vetor de símbolos da palavra recebida gerado por decisão abrupta que servirá para avaliar o desempenho dos algoritmos de GS e BW e uma matriz de confiabilidade sem quantização é calculada para os algoritmos de decisão suave de KV, KV com complexidade reduzida e KV assintótico. O escore é calculado utilizando a matriz de confiabilidade e com o conhecimento da palavra código transmitida; se o limiar imposto por (4.18) for satisfeito então é garantido que a decodificação irá acontecer com sucesso. Como a decodificação ainda pode ter sucesso, pois a equação (4.18) é apenas um limitante inferior, os resultados dessa simulação podem ficar um pouco abaixo do desempenho prático do algoritmo. Para o desempenho assintótico utilizou-se a equação (4.26).

A figura 4.2 mostra a comparação de desempenho de vários algoritmos de decodificação para o código RS$(15, 11)$. Como a taxa desse código é de aproximadamente 0.7, o desempenho dos algoritmos de decodificação por decisão abrupta de Berlekamp-Welch e Guruswami-Sudan aparecem na mesma curva na figura, pois têm desempenho igual. Já para o algoritmo de decodificação por decisão suave de KV três curvas foram obtidas. A primeira (SOFT-KV S=45) emprega o algoritmo 4.5.1 para obter a matriz de multiplicidade utilizando $S = 45$, ou seja, fazendo 45 iterações no algoritmo. Na segunda (SOFT-KV Lambda=4.2), empregou-se o algoritmo de 4.5.2 para se obter a matriz de multiplicidades utilizando o parâmetro $\lambda = 4.2$. Na terceira (SOFT-KV Assintótico) foi utilizada a relação dada pela equação (4.26), utilizando-se apenas a matriz de confiabilidade. Analisando os resultados para uma taxa de erro de palavra código de $10^{-4}$, pode-se perceber que o desempenho assintótico do algoritmo de KV tem um ganho um pouco maior de 1 dB em relação aos algoritmos de decodificação por decisão abrupta.

O desempenho da decodificação por decisão suave para o código RS(31,25) é mostrado na figura 4.3. Da mesma forma que na figura anterior devido a taxa desse código ser de aproximadamente 0.8, o ganho de desempenho da decodificação por decisão abrupta de Guruswami-Sudan se degenera tornando-se igual ao da decodificação conven-

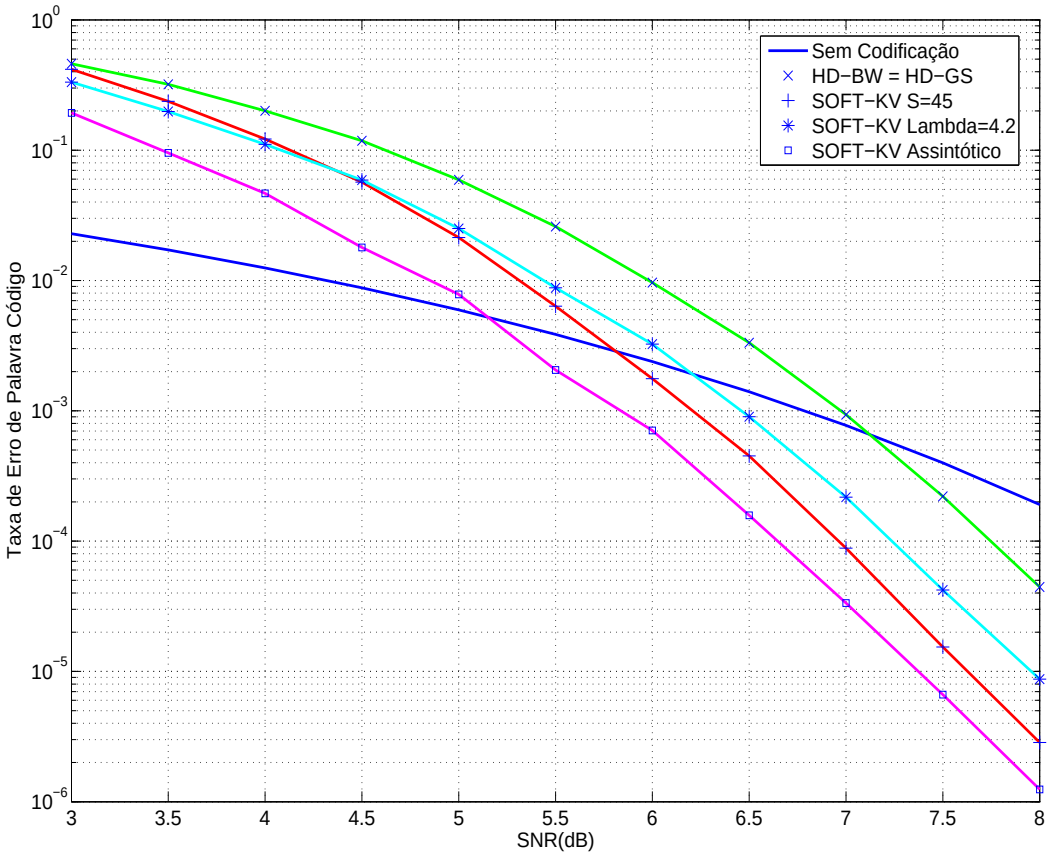



Figura 4.2: Comparação de desempenho teórico de diferentes algoritmos decodificação para o código RS(15,11), utilizando modulação BPSK num canal AWGN

cional (Berlekamp-Welch). Já o desempenho do algoritmo de complexidade reduzida de KV (Algoritmo 4.5.2) representado na curva SOFT-KV com $\lambda = 6.2$ se degenera a partir da relação sinal-ruído (SNR) de 5 dB em comparação com o algoritmo 4.5.1 (curva SOFT-KV com $S = 93$). O desempenho do algoritmo 4.5.2 pode ser melhorado através da mudança do valor de $\lambda$, contudo, não existe uma fórmula ou método para encontrar o valor ótimo de $\lambda$, assim o ajuste deve ser feito manualmente [7]. Ambos os algoritmos estão utilizando valores de $\lambda$ e $S$ escolhidos para que possuam custos de interpolação semelhantes, de forma a equiparar a complexidade e tempo de execução dos mesmos. Para esse código podemos concluir que o desempenho assintótico (obtido através da equação 4.26) do algoritmo de KV (curva SOFT-KV Assintótico) mostra que podemos ter ganhos de aproximadamente 1,5 dB para uma taxa de erro de palavra código da ordem de $10^{-3}$ em relação aos algoritmos de decodificação por decisão abrupta.

No decorrer deste capítulo foi mostrado como se atingiu a decodificação ASD e que podemos obter ganhos de desempenho significativos sobre a decodificação convencional. Deve-se salientar, que para se atingir ganhos de desempenho práticos próximos ao ganho assintótico deve-se trabalhar com códigos de dimensões razoavelmente grandes, como por exemplo, RS(255,239), e com custos de interpolação altos, ou seja, atribuindo multiplicidades altas aos pontos recebidos, o que torna o tempo de execução muito alto. Vamos mostrar no próximo capítulo que se apagarmos os bits recebidos do canal que não são confiáveis e atribuirmos a mesma multiplicidade para símbolos com o mesmo padrão de bits apagados podemos chegar próximos do desempenho assintótico utilizando códigos de pequenas dimensões e utilizando um custo de interpolação razoável.

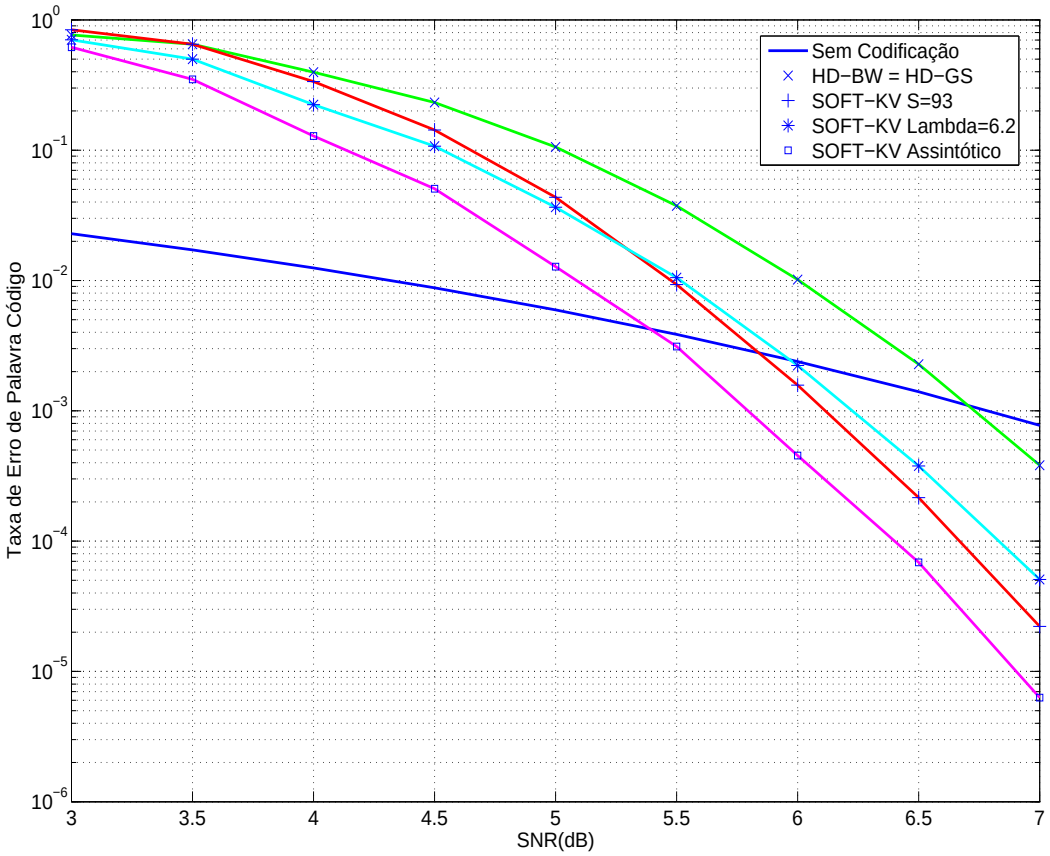



Figura 4.3: Comparação de desempenho teórico de diferentes algoritmos decodificação para o código RS(31,25), utilizando modulação BPSK num canal AWGN

# CAPÍTULO 5

# DECODIFICAÇÃO POR DECISÃO SUAVE DE CÓDIGOS DE RS NA PRESENÇA DE SÍMBOLOS APAGADOS

Uma informação corrompida de maneira mais leve do que um erro numa transmissão de dados é denominada de um apagamento. Neste caso o símbolo ou bit transmitido ou foi simplesmente perdido ou foi recebido de tal maneira que não há dúvidas que está corrompido. Neste capítulo são desenvolvidos os conceitos necessários para que se possa definir o comportamento da decodificação por decisão suave de códigos de RS na presença de apagamentos, mostrando qual deve ser a melhor estratégia de atribuição de multiplicidades para esse caso. No final é mostrada nossa contribuição ao assunto através da implementação de algoritmos de simulação, onde pode-se concluir que podemos obter ganhos razoáveis no desempenho da decodificação algébrica por decisão suave de códigos de RS se propositalmente apagarmos os bits não confiáveis recebidos do canal.

## 5.1 Decodificação de Guruswami-Sudan na Presença de Apagamentos

Podemos analisar a presença de apagamentos na decodificação de GS sobre dois enfoques distintos. Quando podemos ter apagamento em nível de bit, e quando temos apagamentos em nível de símbolos (conjuntos de bits).

Decodificadores por decisão abrupta utilizam estratégias de se gerar palavras candidatas quando ocorrem apagamentos em bits. Ou seja, é gerado um conjunto de palavras códigos com as combinações possíveis de zeros e uns nos locais onde estes foram apagados, e então é passada cada palavra do conjunto pelo decodificador e espera-se que somente uma das palavras tenha sucesso na decodificação, porque caso contrário terá que se adotar um processo de escolha. De qualquer modo, no caso de apagamentos de bits, o

decodificador de GS continua o mesmo.

Já para o caso de símbolos apagados é preciso se analisar o que ocorrerá com o algoritmo de decodificação e como ficará sua capacidade de correção.

Suponha que a palavra código recebida do canal tenha sido corrompida por $s$ símbolos apagados, que para efeito de notação assumimos que os mesmos se encontram nas últimas posições do vetor, ou seja, nas posições $n-s+1, ..., n$. Então a palavra recebida pode ser representada como $(\beta_1, ..., \beta_{n-s}., ?, ?, ..., ?)$ onde o símbolo '?' representa um apagamento. Neste caso o decodificador de Guruswami-Sudan constrói um polinômio com raízes de multiplicidades $m$ em cada um dos $n-s$ pontos $(\alpha_1, \beta_1), ..., (\alpha_{n-s}, \beta_{n-s})$. Então pode-se notar que os fatores do tipo $y - p(x)$ que compõem $Q(x, y)$ irão concordar com a palavra código transmitida em pelo menos $n-s$ posições. Assim o problema de decodificação se resume à reconstrução de um polinômio que passa por $n' = n - s$ pontos. Substituindo $n$ por $n'$ na equação 3.44 temos a seguinte expressão para a capacidade de correção de erros de um código $C(n, k)$ na presença de apagamentos:

$$t_m = \lfloor n' - \sqrt{n'k} \rfloor \tag{5.1}$$

Assim podemos formular o seguinte teorema [4]:

**Teorema 5.1.1** *O problema de decodificação por lista de um código de Reed-Solomon $C(n, k)$, que pode ter até e erros e s apagamentos, tem como ser resolvido através de interpolação polinomial, desde que:*

$$e + s < n - \sqrt{(n-s)k} \tag{5.2}$$

A equação 5.2 caracteriza a condição necessária para que se saiba quais as combinações de erros e apagamentos que se pode ter para que o decodificador de GS consiga encontrar a palavra código transmitida. Fazendo um paralelo com algoritmos convencionais que utilizam o paradigma de decodificação por distância limitante como, por exemplo, o algoritmo de decodificação de Berlekamp-Massey, esta condição fica:

$$2e + s < n - k + 1 \tag{5.3}$$

Por exemplo, seja o código RS$(31, 7)$ sobre o corpo $\mathbb{F} = GF(2^5)$. Se a palavra código é corrompida por $s$ apagamentos então temos na tabela 5.1 relacionados quantos erros o decodificador convencional consegue corrigir, $e_0$, e quantos erros o decodificador de GS, $e_{GS}$, consegue corrigir, a medida que a quantidade de símbolos apagados, $s$, vai aumentando.

Tabela 5.1: Combinações de correção de erros e apagamentos para o Decodificador Convencional e o de GS

| s | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $e_0$ | 12 | 11 | 11 | 10 | 10 | 9 | 9 | 8 | 8 | 7 | 7 | 6 | 6 | 5 | 5 | 4 | 4 |
| $e_{GS}$ | 17 | 16 | 15 | 14 | 13 | 13 | 12 | 11 | 11 | 10 | 9 | 8 | 8 | 7 | 6 | 6 | 5 |
| s | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | | | | | | | | | |
| $e_0$ | 3 | 3 | 2 | 2 | 1 | 1 | 0 | 0 | | | | | | | | | |
| $e_{GS}$ | 5 | 4 | 3 | 2 | 2 | 1 | 1 | 0 | | | | | | | | | |

A figura 5.1 mostra o gráfico da capacidade de correção combinada (erros + apagamentos) com a variação do número de símbolos apagados tanto para o algoritmo de decodificação convencional quanto para o de Guruswami-Sudan. Pode-se perceber que a medida que o número de apagamentos vai aumentando o ganho obtido com o algoritmo de GS se degenera ficando igual ao convencional.

## 5.2 Canal Binário Simétrico com Apagamento (*BSEC: Binary Symmetric Erasure Channel*)

Uma das formas de se modelar um canal discreto sem memória é como um canal binário com apagamentos (*BEC-Binary Erasure Channel*), que se caracteriza pelo transmissor poder somente mandar um bit de informação, 0 ou 1, e no receptor pode-se receber ou o bit transmitido ou uma mensagem dizendo que o bit não foi recebido, que no caso seria considerado como um bit apagado (ou então um *erasure*). O canal binário com apagamento foi introduzido pela primeira vez por Elias em 1956 [34], entretanto, nessa êpoca foi considerado apenas de interesse teórico.

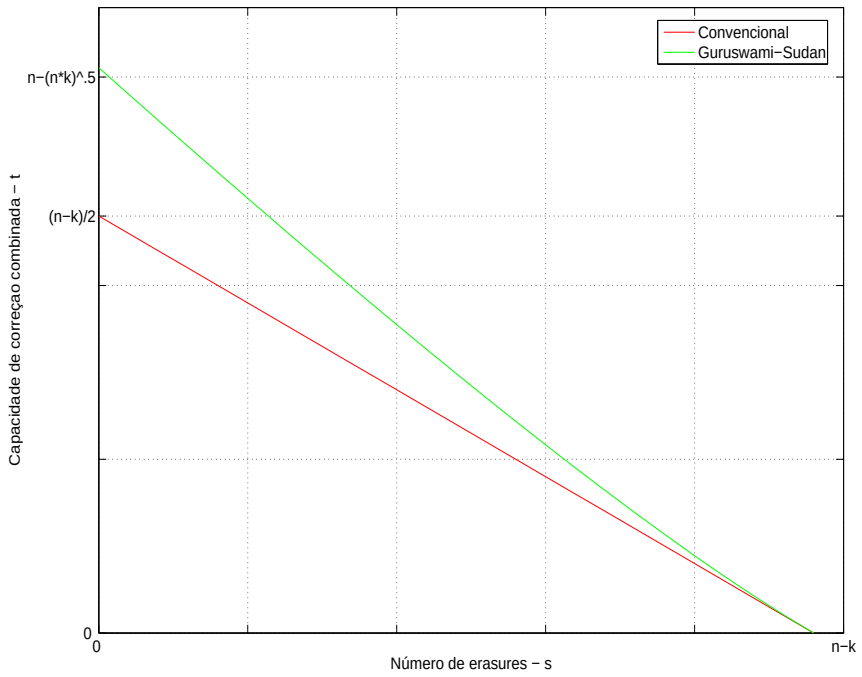



Figura 5.1: Comparação entre a capacidade de correção de apagamentos + erros do algoritmo de decodificação convencional com o algoritmo de GS

Um canal binário com apagamento é caracterizado por uma probabilidade de apagamento $p$, por uma variável aleatória de entrada $X$ que pode assumir os valores $\{X_1, X_2\} = \{0, 1\}$ e uma variável aleatória de saída $Y$ que pode assumir os seguintes valores $\{Y_1, Y_2, ?\} = \{0, 1, s\}$, onde ? denomina o símbolo apagado. Então o canal tem as seguintes probabilidades condicionais, e pode ser descrito graficamente como na figura 5.2.

$$Pr(Y = 0|X = 0) = 1 - p$$
$$Pr(Y =?|X = 0) = p$$
$$Pr(Y = 1|X = 0) = 0$$
$$Pr(Y = 0|X = 1) = 0$$
$$Pr(Y =?|X = 1) = p$$
$$Pr(Y = 1|X = 1) = 1 - p$$

Observe que neste modelamento de canal a probabilidade de se transmitir um 1 e receber um 0 é zero, assim como é zero a probabilidade de se transmitir um 0 e receber um 1, isto quer dizer que neste tipo de canal qualquer tipo de corrupção do sinal transmitido

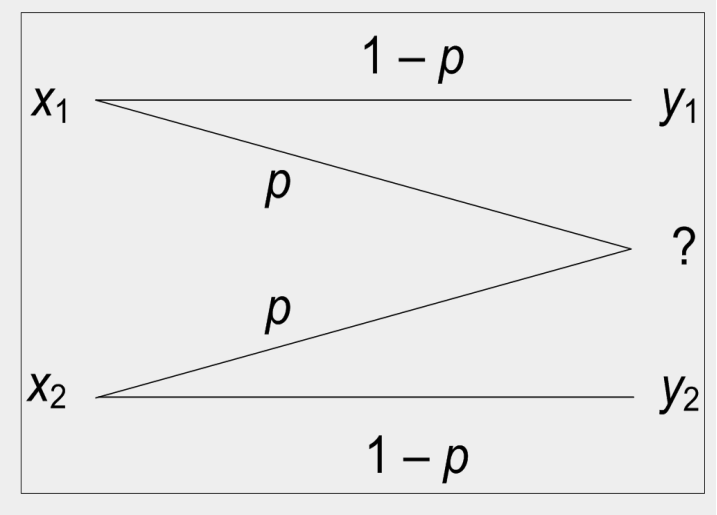



Figura 5.2: Diagrama do Canal Binário com Apagamentos

é mapeado para um símbolo apagado.

## 5.3 Análise do Desempenho da Decodificação por Decisão Suave de Códigos de RS sobre um BSEC

Antes de procedermos à análise da decodificação ASD na presença de símbolos apagados, vamos considerar o caso mais simples onde as palavras do código de RS são transmitidas como bits através de um BSEC com probabilidade de apagamento $p$. Neste caso quando estamos na fase de atribuição das multiplicidades, se tivermos apagamento em um bit, por exemplo, como não teremos as informações de probabilidade a posteriori, teremos que considerar que dois símbolos (um com o bit apagado como 0 e outro em 1) são igualmente prováveis. Além disso temos que considerar da mesma forma que em [5],[14] e [15] que os símbolos das palavras do código são distribuídos de forma uniforme, ou seja, não existe preferência de um símbolo sobre o outro, portanto podemos atribuir a mesma multiplicidade para dois símbolos igualmente prováveis.

Dessa forma iremos definir, conforme [15], que um símbolo é do tipo $i$, se tiver $i$ bits apagados. Assim, para um código de RS sobre um corpo $\mathbb{F} = GF(2^m)$, podemos ter até $m+1$ diferentes tipos de símbolos, por exemplo, para códigos de RS sobre $\mathbb{F} = GF(2^5)$, onde cada símbolo é formado por 5 bits, podemos ter símbolos com 0, 1, 2, 3, 4 ou 5 bits apagados, ou seja 6 tipos de símbolos. Seja o número de símbolos do tipo $i$ numa palavra código recebida denominado de $a_i$. Como foi definido anteriormente iremos atribuir a mesma multiplicidade para símbolos do mesmo tipo, apesar que essa multiplicidade atribuída pode variar de acordo com a palavra código atribuída. Vamos definir a multiplicidade designada para cada símbolo do tipo $i$ como $m_i$. Assim a multiplicidade total atribuída para todos os símbolos do tipo $i$ será $2^i m_i$. De acordo com as considerações feitas até agora podemos redefinir o escore e custo da matriz de multiplicidades como[15]:

$$S = \sum_{i=0}^{m} a_i m_i \tag{5.4}$$

$$C = \sum_{i=0}^{m} a_i 2^i \binom{m_i + 1}{2} \tag{5.5}$$

Onde $m$ é o número de bits que formam o símbolo. Para valores altos de $m_i$ a equação 5.5 pode ser aproximada por[15]:

$$C \approx \frac{1}{2} \sum_{i=0}^{m} a_i 2^i m_i^2 \tag{5.6}$$

A equação 4.18 definida no capítulo anterior, caracteriza a condição para que o algoritmo ASD tenha sucesso; utilizando-a como premissa podemos caracterizar qual a melhor técnica de atribuição de multiplicidades para o canal BSEC. É fácil de perceber que temos que tentar maximizar o escore, com a restrição do custo da matriz de multiplicidades, $M$. Supondo multiplicidades infinitas, o problema pode ser expresso como[15]:

$$max_{m_i} \sum_{i=0}^{m} a_i m_i \tag{5.7}$$

sujeito a :

$$C = \frac{1}{2}\sum_{i=0}^{m} a_i 2^i m_i^2 \tag{5.8}$$

Este problema de otimização pode ser resolvido usando multiplicadores de Lagrange e o desenvolvimento pode ser obtido em [15]. O que nos interessa é o resultado final que diz:

$$m_i \propto 2^{-i} \tag{5.9}$$

Onde se conclui que o melhor tipo de estratégia de atribuição de multiplicidades para canais BSEC é a atribuição proporcional à matriz $\Pi$.

## 5.4 Canal Binário Simétrico com Erros e Apagamentos

Podemos também acrescentar ao canal binário com apagamentos (BEC) a possibilidade de se ocorrerem erros, onde teremos um modelo de canal hibrído, ou um canal binário simétrico com erros e apagamentos. Neste tipo de canal de comunicação só se podem enviar dois tipos de símbolos, $\{X_1, X_2\} = \{0, 1\}$, e na saída do canal podemos ter qualquer um dos símbolos $\{Y_1, Y_2, ?\} = \{0, 1, s\}$, dependendo somente da probabilidade de transição do canal. Então o canal tem as seguintes probabilidades condicionais, e pode ser descrito graficamente como na figura 5.3.

$$Pr(Y = 0|X = 0) = 1 - p - q$$
$$Pr(Y = 0|X = 1) = p$$
$$Pr(Y = 1|X = 0) = p$$
$$Pr(Y = 1|X = 1) = 1 - p - q$$
$$Pr(Y = s|X = 0) = q$$
$$Pr(Y = s|X = 1) = q$$

É interessante observar que agora neste modelamento de canal as probabilidades de transição de 0 para 1 não tem mais probabilidades zero, ou seja, agora um bit transmitido pode ser tanto recebido certo, errado ou apagado.

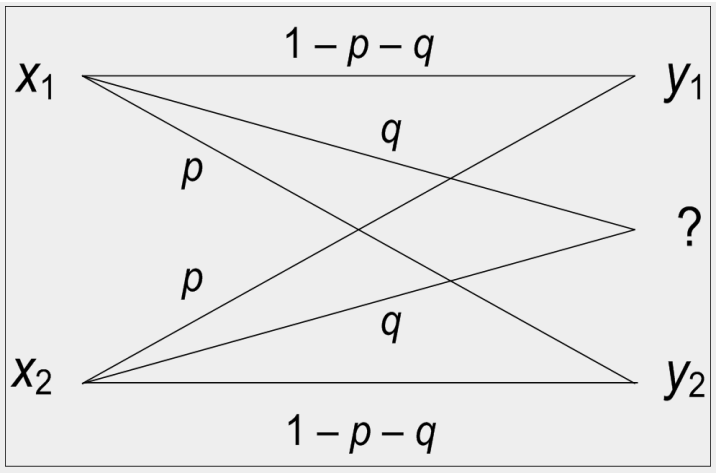



Figura 5.3: Diagrama do Canal Binário Simétrico com Erros e Apagamentos

## 5.5 O Canal Gaussiano e o Canal Binário Simétricos com Erros e Apagamentos

Num certo nível de abstração podemos dizer que o sistema de modulação e demodulação com o ruído Gaussiano aditivo pode ser visto como uma canal binário simétrico com erros[18]. Para isso, basta que as probabilidade de transição sejam iguais às probabilidades a posteriori do canal Gaussiano. Contudo fica uma variável em aberto: a probabilidade de transição de apagamento.

Vamos considerar uma transmissão binária num canal Gaussiano utilizando modulação BPSK (*Binary Phase Shift Keying*). Nesse caso a constelação de sinais consiste nos pontos $S = \{\sqrt{E_b}, -\sqrt{E_b}\}$, onde $E_b$ é a energia de um bit, correspondendo, respectivamente, aos bits 1 e 0, com funções de densidade de probabilidade condicional dadas por[18]:

$$p(r|s = \sqrt{E_b}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(r-\sqrt{E_b})^2} \tag{5.10}$$

correspondente à transmissão do bit 1, e

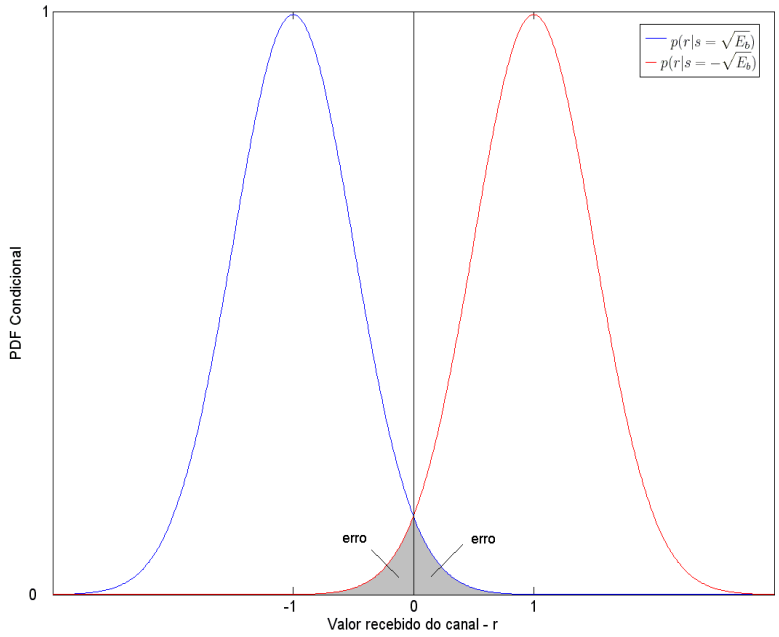



Figura 5.4: Gráfico das funções de densidade de probabilidade condicional das equações 5.10 e 5.11, com $E_b = 1$.

$$p(r|s=-\sqrt{E_b}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(r+\sqrt{E_b})^2} \tag{5.11}$$

correspondente à transmissão do bit 0.

Nas duas equações anteriores $\sigma^2$ é a variância do ruído e $r$ é a variável aleatória que representa o valor da saída do canal. Os gráficos das equações 5.10 e 5.11 são mostrados na figura 5.4 onde foi considerado que $E_b = 1$. Quando se utiliza um decodificador por decisão abrupta, se na saída do canal tivermos um sinal com valor maior que 0, o mesmo será demodulado como 1, enquanto que um sinal com valor menor que 0 será demodulado como 0. A região hachurada da figura 5.4, marca a região na qual o sinal recebido tem maior probabilidade de ter sido corrompido por ruído e que será demodulado para um valor binário diferente da valor transmitido.

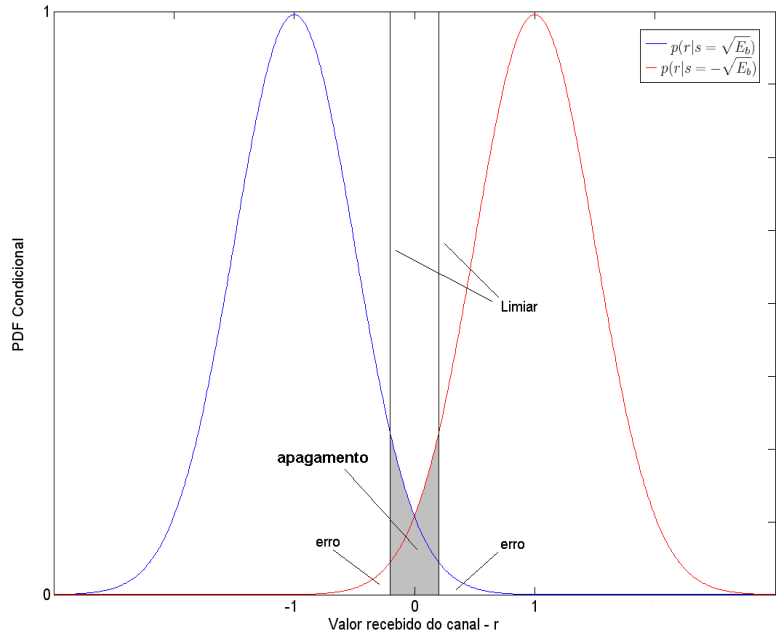



Figura 5.5: Gráfico das funções de densidade de probabilidade condicional das equações 5.10 e 5.11, com $E_b = 1$ e com limiar de apagamento $t = 0.2$

Vamos mudar esse modelo para que possamos introduzir um terceiro símbolo na saída, o apagamento. Para isso iremos introduzir dois tipos de limiares ($-t$ e $+t$), e declarar os sinais recebidos que tiverem valores dentro deste intervalo definidos como apagamentos. Na saída o novo demodulador agora estimará :

$$p(r|S=s) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(r+s)^2} & \text{, se } r < -t \text{ ou } \ r > +t \\ apagamento & \text{, se } -t \leq r \leq +t \end{cases}$$

Onde $S = \{\sqrt{E_b}, -\sqrt{E_b}\}$, e $s$ indica um elemento qualquer do conjunto $S$. Dessa maneira estamos mapeando todos os sinais que caírem na região delimitada por $t$ como bits apagados, como mostra a figura 5.5, onde $E_b$ é a energia correspondente a um bit que é igual a 1 e $t = 0.1$.

## 5.6 Atribuição de Multiplicidades na Presença de Bits Apagados

Sabendo que a atribuição proporcional é a mais indicada para canais binários simétricos com apagamentos, e que ela também é assintóticamente a mais indicada para canais Gaussianos, como mostrado no capítulo anterior, podemos inferir que a atribuição proporcional deve ser a mais indicada para o caso de um canal binário simétrico com erros e apagamentos.

Assim como definido na seção 5.3, quando temos um ou mais bits apagados num símbolo devemos considerar que todos os símbolos candidatos a serem o símbolo transmitido devem ter a mesma probabilidade e por consequência a mesma multiplicidade.

Contudo a tarefa de atribuir as multiplicidades a partir de uma matriz de confiabilidade que tem símbolos que podem ter bits apagados fica complexa. Para resolver esse problema e facilitar a implementação é proposto que se atue direto na função de densidade de probabilidade condicional do canal, de tal forma que os bits apagados sejam mapeados para uma probabilidade condicional onde a probabilidade destes bits serem 0 ou 1 é a mesma. Ou seja:

$$p(r|S=s) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(r+s)^2} & \text{, se } r < -t \text{ ou } r > +t \\ \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(E_b)} & \text{, se } -t \leq r \leq +t \end{cases}$$

Assumindo que nosso modelo de canal é sem memória, podemos afirmar que a probabilidade de ocorrência de cada bit é independente dos bits anteriores, dessa forma podemos escrever que a função de densidade de probabilidade condicional de um símbolo genérico $\alpha$ (composto por $m$ bits) pertencente a um corpo $\mathbb{F} = GF(2^m)$ é:

$$f(y|\alpha) = \prod_{i=1}^{m} p(r|S=s) \tag{5.12}$$

Com a equação 5.12 basta aplicá-la na equação 4.5 e em seguida na equação 4.8, para que se obtenham cada um dos elementos da matriz de confiabilidade $\Pi$.

Definido o mapeamento entre o canal e a matriz de confiabilidade, basta então utilizar

o algoritmo de Koetter-Vardy na sua forma original.

Agora a mudança é que os pontos que estiverem dentro da região de apagamento terão a mesma probabilidade. Assim, na fase de atribuição de multiplicidades, esses pontos terão a mesma multiplicidade, que é a idéia fundamental deste mapeamento visto que um apagamento tem a mesma chance de ser um 1 ou um 0.

## 5.7 Canais $M$-ários com Apagamentos

Podemos estender os resultados da seção 5.3 para canais $M$-ários com apagamentos[15], isto é, $\log_2 M$ bits são agrupados e transmitidos utilizando $M$ símbolos, através de formatos de modulações como a $M$-QAM ou $M$-PSK. O canal irá apagar o símbolo com uma probabilidade de apagamento $p$, em nível de símbolo. Modelos práticos deste canal são discutidos em [35]. Na seção 5.3, também foi mostrado que o esquema de atribuição de multiplicidades proporcional à matriz de confiabilidade era a solução ótima para o BSEC. Pode-se dizer que o canal $M$-ário com apagamentos nada mais é que um subconjunto dos padrões de apagamentos do BSEC. Portanto podemos concluir que a atribuição de multiplicidade proporcional a $\Pi$ é também ótima para canais $M$-ários com apagamentos.

## 5.8 Canal Gaussiano e Canal $M$-ário com Apagamentos e Erros

Da mesma forma que dissemos que, do ponto de vista do par codificador-decodificador, o canal Gaussiano pode ser visto como um canal binário simétrico com erros e apagamentos desde que se estabeleçam os mapeamentos necessários das probabilidades de transição, podemos utilizar o mesmo raciocínio para um canal $M$-ário com apagamentos e erros, utilizando a idéia de estipular um limiar $(-t, +t)$ dentro da função de densidade de probabilidade condicional do símbolo para que declaremos que o mesmo está apagado.

## 5.9 Implementação do Algoritmo de KV

Como o raio de decodificação de algoritmos por decisão suave é difícil de se encontrar [5], ainda mais se incluirmos apagamentos, ficaria muito complexo encontrar expressões

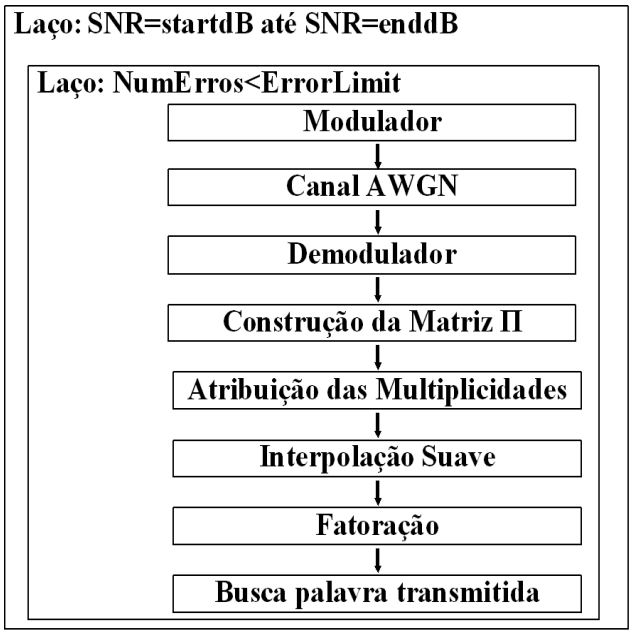



Figura 5.6: Visão Geral do Programa de Simulação

teóricas para simular o desempenho do algoritmo. Então, para analisar o desempenho do algoritmo de KV adaptado para símbolos apagados e para bits apagados, foi implementado um programa que simula na prática o algoritmo de KV, em liguagem C++, para que se pudesse levantar as curvas de desempenho para cada um dos casos. A figura 5.6 mostra uma visão geral do código implementado.

Como na decodificação de códigos de Reed-Solomon as palavras código são representadas como polinômios, e por consequência várias operações matemáticas são feitas sobre elas, então, para auxiliar na implementação do simulador utilizou-se a classe PolynomialT, desenvolvida por Todd K. Moon e modificada por Stewart Weber em [18], que implementa todas as funções aritméticas com polinômios, como por exemplo, soma, multiplicação, divisão, etc. Além disso, essa classe foi criada pensando-se que os coeficientes dos polinômios pudessem ser de um corpo $\mathbb{F}$ qualquer, através do uso do especificador "template"para o parâmetro coeficiente, permitindo assim declará-lo como sendo "float"(para o caso de pertecer ao corpo dos $\mathbb{R}$), ou como sendo de uma classe que implementa um corpo de Galois $GF(2^m)$, que é o caso utilizado na simulação.

O modulador e demodulador utilizado foi o BPSK, sendo escolhido pela facilidade de implementação e por permitir que se conseguissem gerar apagamentos de bits. Para a etapa de apagamento de bits, quando um bit chegava do canal com uma densidade de probabilidade condicional dentro da região de apagamento, o demodulador atribui a probabilidade do ponto 0, onde se tem densidades condicionais iguais para o bit 0 e o 1.

A construção da matriz $\Pi$ foi feita através da aplicação direta das equações 5.12, 4.5 e 4.8.

Para a etapa de atribuição de multiplicidades, foi utilizado o método reduzido de KV, que consiste em atribuir multiplicidades proporcionais a um fator $\lambda$, de acordo com a equação 4.24. Para o caso de análise de apagamentos em símbolos, essa fase foi alterada para atribuir multiplicidade zero para a coluna correspondente à posição do símbolo da palavra código que estava errado.

A interpolação suave e a fatoração foram implementadas de acordo com [5]. São as fases que mais consomem tempo de execução do algoritmo. Para diminuir o tempo de execução, foi utilizada a técnica descrita em [7], que consiste em fazer uma decodificação por decisão abrupta inicialmente, e caso ela fracasse, então procede-se para a decodificação por decisão suave.

Na etapa de busca pela palavra transmitida procedeu-se da seguinte forma: se a lista retornada tiver somente uma palavra código, a decisão é simples e basta ver se ela corresponde à palavra transmitida. Porém, se a lista retornar mais de uma palavra código, adota-se um procedimento de exclusão das palavras que estiverem além do raio de decodificação por decisão abrupta, ou seja, calcula-se a distância de Hamming de cada uma das palavras da lista em relação à palavra código demodulada por decisão abrupta; se esta distância for maior que a capacidade de correção então a palavra é descartada. Nessas exclusões espera-se que sobre somente uma palavra dentro do raio de decodificação; se sobrar mais de uma, não há como se decidir e será declarado um erro. Contudo justifica-se em [36] que na média somente uma palavra é retornada na lista.

## 5.10 Resultados das Simulações

Nesta seção são mostrado os resultados das simulações obtidos utilizando o algoritmo de decisão suave de Koetter-Vardy com a modificação proposta na construção das probabilidades a posteriori do canal para que possamos atribuir a mesma probabilidade para símbolos candidatos com os mesmos padrões de apagamento. Foram feitas simulações para dois códigos o RS(15,11) e o RS(31,25). Eles foram escolhidos devido ao comprimento do código ser pequeno, caso contrário o tempo de simulação ficaria muito alto. Todas as simulações foram feitas considerando um canal AWGN. O código fonte utilizado nas simulações pode ser encontrado no Apêndice A.

Para comparar com o algoritmo de Guruswami-Sudan e Berleykamp-Welch foram utilizadas respectivamente as equações 3.44 e $t_0 = (n-k)/2$; observe que para ambos os códigos escolhidos (RS(15,11) e RS(31,25)) o desempenho do decodificador de GS fica igual ao de BW pois os dois códigos escolhidos são de alta taxa. Devido à linearidade do código optou-se por utilizar um vetor de informação com todos os símbolos iguais a zero, esse vetor é codificado, e transmitido no canal. No receptor calcula-se o vetor de símbolos da palavra recebida gerado por decisão abrupta que servirá para avaliar o desempenho dos algoritmos de GS e BW e uma matriz de confiabilidade é calculada para se obter o desempenho assintótico do algoritmo de KV, obtido através da equação 4.26, e também para se obter o desempenho de KV utilizando uma atribuição de multiplicidades proporcional ao parâmetro $\lambda$, de acordo com a equação 4.24: esta será a decodificação por decisão suave de KV sem apagamentos. Uma outra matriz de confiabilidade é calculada onde os apagamentos têm a mesma probabilidade de acordo com o limiar $t$ de apagamento imposto.

Os resultados da simulação do algoritmo de KV com a alteração proposta para contemplar o apagamento de bits para o código de Reed-Solomon RS$(15, 11)$, de taxa 0.733, estão mostrados na figura 5.7. Para a fase de atribuição de multiplicidades foi utilizado o parâmetro $\lambda = 3.99$, esse valor foi escolhido pois segundo [7], é o que apresenta maior ganho de desempenho. O limiar de decisão de apagamento utilizado foi $t = 0.2$, valor esse obtido através de ajuste manual, que apresentou melhor resultados de desempenho.

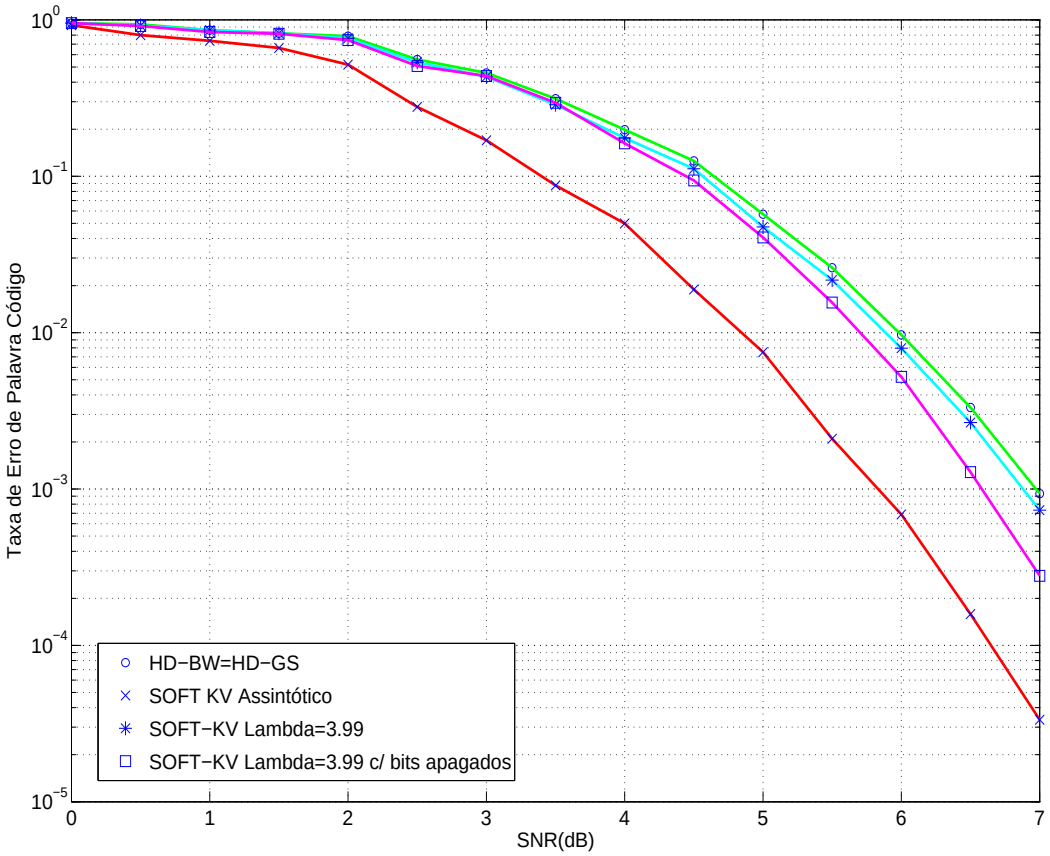



Figura 5.7: Comparação de desempenho prático simulado do algoritmo de KV sem apagamento e com apagamento de bits para o código RS(15,11), utilizando modulação BPSK num canal AWGN, $\lambda = 3.99$ e $t = 0.2$

Pode-se notar um ganho da ordem de 0.3 dB da decodificação algébrica por decisão suave com bits apagados para a decodificação ASD sem apagamentos para uma taxa de erro de palavra código da ordem de $10^{-3}$.

Para o código RS(31,25) com taxa aproximada de 0.8 pode-se notar um resultado ainda melhor para a decodificação por decisão suave na presença de apagamento de bits. Desta vez foi utilizado um custo de interpolação maior ($\lambda = 4.99$), esse valor foi escolhido por tentativa e erro, visando otimizar o desempenho do código sem aumentar demais a complexidade e tempo de execução(valores acima de 6 para $\lambda$ levam o tempo de simulação para dezenas de horas ou até dias), visto que não existe nenhum método que possa encontrar o valor de $\lambda$ ótimo. Pode-se notar que se chega próximo da decodificação algébrica por decisão suave assintótica, resultado esse que só é conseguido por outros autores quando se

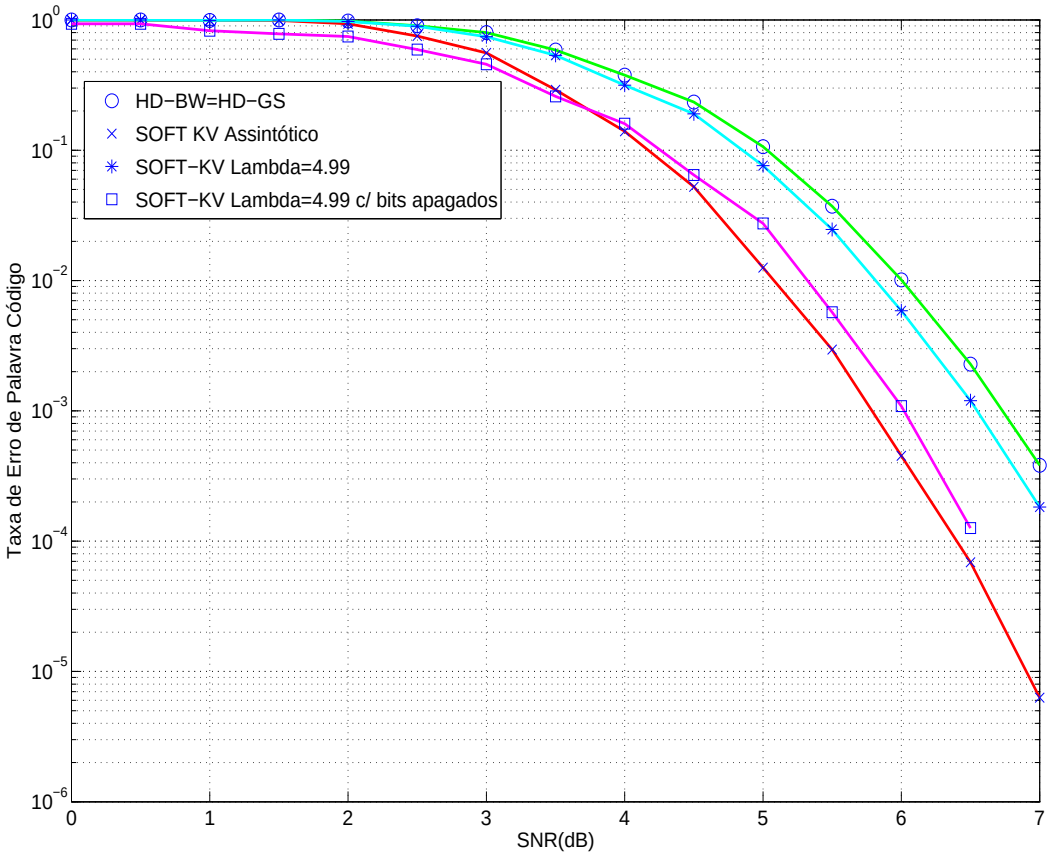



Figura 5.8: Comparação de desempenho prático simulado do algoritmo de KV sem apagamento e com apagamento de bits para o código RS(31,25), utilizando modulação BPSK num canal AWGN, $\lambda = 4.99$ e $t = 0.2$

utilizam códigos de grandes dimensões como, por exemplo, o código RS(255,239). Além disso, atingiram-se ganhos de desempenho de 0.5 dB em relação à decodificação por decisão suave sem apagamento de bits para uma taxa de erro de palavra código da ordem de $10^{-3}$. O limiar de decisão de apagamento utilizado foi igual ao da simulação anterior pelos motivos já explicados $t = 0.2$.

Os resultados da simulação do algoritmo de KV adaptado para apagamento em símbolos para o código de Reed-Solomon RS$(31, 25)$, de taxa 0.806, estão mostrados na figura 5.9. Para a fase de atribuição de multiplicidades foi utilizado o parâmetro $\lambda = 4.99$. O limiar de decisão de apagamento utilizado foi $t = 0.01$, pois lembrando que como as simulações foram feitas para modulação BPSK o valor recebido do canal para um símbolo qualquer formado por $u$ bits, será a multiplicação dos valores individuais recebidos para

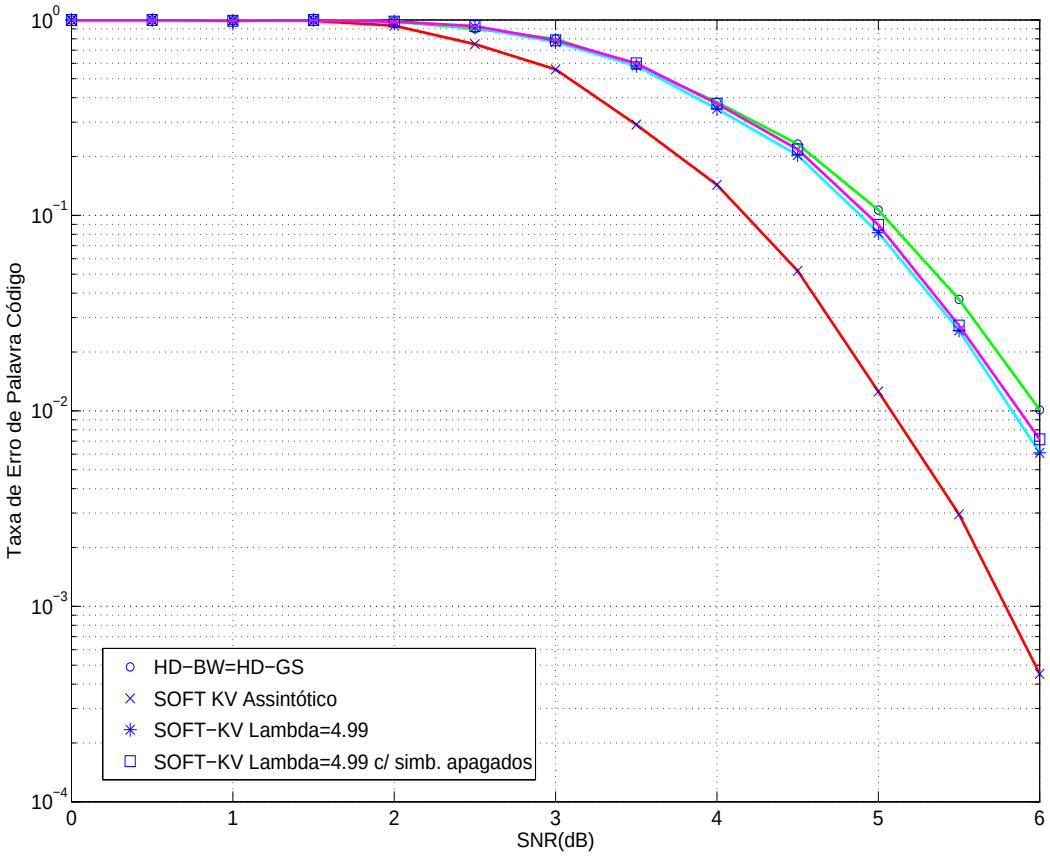



Figura 5.9: Comparação de desempenho prático simulado do algoritmo de KV sem apagamentos e com apagamentos de símbolos para o código RS(31,25), utilizando modulação BPSK num canal AWGN, $\lambda = 4.99$ e $t = 0.015$

cada um dos $u$ bits, e o resultado dessa multiplicação de valores menores que 1, leva a se escolher valores menores de limiar de apagamento de símbolo. Pode-se notar que diferentemente do desempenho para bits apagados temos agora uma perda de desempenho pois agora quando um símbolo é declarado como apagado, o que acontece é que toda uma coluna da matriz de confiabilidade, que corresponde à posição do símbolo na palavra código, fica com as mesmas probabilidades, e por consequência acabamos aumentando mais o custo da matriz de multiplicidade do que o escore, o que acaba por resultar em falhas na decodificação, como nos diz a equação 4.18.

# CAPÍTULO 6

# CONCLUSÃO

Com este trabalho foi possível demonstrar através de simulação computacional que a estratégia de declarar os bits não confiáveis recebidos do canal como bits apagados, atribuindo assim multiplicidades iguais para símbolos com o mesmo padrão de bits apagados, pode ser usada em códigos de Reed-Solomon de altas taxas e pequenas dimensões de forma eficiente, e ainda obter ganhos significativos de desempenho próximos ao desempenho assintótico elaborado por Koeter e Vardy, com baixo custo de interpolação e por consequência com menor tempo de execução. A contribuição dada nessa dissertação mostra o grande potencial que a decodificação ASD pode ter quando se consideram os bits não confiáveis como bits apagados, obtendo-se ganhos não somente de desempenho como também de complexidade, e sugere que ainda pode-se tentar encontrar algoritmos que busquem otimizar ao máximo essa capacidade de correção de apagamentos de bits.

Já quando se tenta estender o resultado obtido para apagamento de bits para o caso de se declarar os símbolos não confiáveis como símbolos apagados, pode-se comprovar através de simulações que dificilmente obteremos ganhos de desempenho utilizando o mesmo valor de multiplicidade para todos os possíveis símbolos que seriam candidatos a serem o símbolo transmitido que foi apagado. Pode-se pensar em tentar investigar se o caso misto de apagamento, onde declaramos bits e símbolos apagados dentro de uma palavra recebida, poderia vir a possibilitar novos ganhos de desempenho em relação ao apagamento de bits somente.

Ainda há vários pontos que podem ser investigados, como a análise do desempenho da decodificação ASD com apagamento de códigos de RS com grandes dimensões, como por exemplo, o código RS(255,239), e também para códigos com baixa taxa de codificação. Outra análise importante que pode ser estudada em trabalhos futuros é a análise do desempenho da decodificação ASD com apagamentos sobre o canal com memória, canal

de Rayleigh, entre outros. O uso de concatenação e matrizes de entrelaçamento em códigos de RS é bem aplicado na prática, o que torna bem interessante uma análise detalhada de desempenho para esse tipo de caso quando se utiliza a decodificação ASD de códigos de RS com apagamento.

# BIBLIOGRAFIA

[1] I.S. Reed e G. Solomon. Polynomial codes over certain finite fields. *J. Soc. Ind. Appl. Math*, 8:300–304, 1960.

[2] I.R. Shafarevich. *Basic Algebraic Geometry*. Springer-Verlag, 1995.

[3] M. Sudan. Decoding of Reed-Solomon codes beyond the error correction bound. *Journal of Complexity*, 12:180–193, 1997.

[4] V. Guruswami e M. Sudan. Improved decoding of Reed-Solomon and algebraic-geometry codes. *IEEE Trans. Inform. Theory*, 45(6):1755–1764, setembro de 1999.

[5] R. Kötter e A. Vardy. Algebraic soft-decision decoding of Reed-Solomon codes. *IEEE Trans. Inform. Theory*, 49(11):2809–2825, novembro de 2003.

[6] A. Ahmed, R. Koetter, e N. Shanghag. Reduced complexity interpolation for soft decoding of Reed-Solomon codes. *Proc. ISIT*, junho de 2004.

[7] W. J. Gross, F. R. Kschischang, R. Koetter, e P. G. Gulak. Applications of algebraic soft-decision decoding of Reed-Solomon codes. *IEEE Trans. Info. Theory*, 49(11):2809–2825, 2003.

[8] W. J. Gross, F. R. Kschischang, R. Koetter, e P. G. Gulak. Towards a VLSI architecture for interpolation-based soft-decision Reed-Solomon decoders. *Journal of VLSI Signal Processing*, 39:93–111, 2005.

[9] H. Xia e J. R. Cruz. A reliability-based foward recursive algorithm algorithm for algebraic soft-decision decoding of Reed-Solomon codes. *Proc. ISITA*, outubro de 2004.

[10] H. Xia. *Error-correction coding for high-density magnetic recording channels*. Tese de Doutorado, University of Oklahoma, 2004.

[11] J. Bellorado e A. Kavcic. A low-complexity method for chase-type decoding of Reed-Solomon codes. *Proc. ISIT*, julho de 2006.

[12] M. El-Khamy e R. J. McEliece. Interpolation multiplicity assignment algorithms for algebraic soft-decision decoding of Reed-Solomon codes. *AMS-DIMACS*, 68:99–120, 2000.

[13] F. Parvaresh e A. Vardy. Multiplicity assignments for algebraic soft-decoding of Reed-Solomon codes. *Proc. Int. Symp. Inf. Theory*, páginas 205, julho de 2003.

[14] N. Ratnakar e R. Koetter. Exponential error-bounds for algebraic soft-decision decoding of Reed-Solomon codes. *IEEE Trans. Info. Theory*, 51:3899–3917, 2005.

[15] J. Jiang e K. R. Narayanan. Perfomance analysis of algebraic soft decoding of Reed-Solomon codes over binary symmetric and erasure channels. *Proc. ISIT*, setembro de 2005.

[16] S. B. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice Hall, 1994.

[17] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic, 1987.

[18] T.K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley, 2005.

[19] M. Sudan. Decoding of Reed-Solomon codes beyond the error-correction diameter. *Proc. 35th Annu. Allerton Conf. Comm., Control and Comp.*, 1997.

[20] D. Cox, Little J., e O'Shea D. *Ideals, Varieties, and Algorithms*. Springer-Verlag, New York, 1992.

[21] H. Hasse. Theorie der höheren differentale in einem algebraishen funcktionenkörper mit vollkommenem konstantenkörper nei beliebeger charakteristic. *J. Reine. Ang. Math.*, 175:50–54, 1936.

[22] R. J. McEliece. The Guruswami-Sudan decoding algorithm for Reed-Solomon codes. *IPN Progress Report*, páginas 42–153, maio de 2003.

[23] G. L. Feng e K. K. Tzeng. A generalization of the Berlekamp-Massey algorithm for multisequence shift-register synthesis. *IEEE Trans. Info. Theory*, 37(5):1274–1287, setembro de 1991.

[24] R. Kötter. Fast generalized minimum distance decoding of algebraic-geometry and Reed-Solomon codes. *IEEE Trans. Info. Theory*, 42(3):721–737, maio de 1996.

[25] R. Kötter. *On Algebraic Decoding of Algebraic-Geometric and Cyclic Codes*. Tese de Doutorado, University of Linkoping, 1996.

[26] V. Olshevesky e A. Shokrollahi. A displacement structure approach to efficient decoding of Reed-Solomon and algebraic-geometric codes. *Proc. 31st ACM Symp. Theory of Computing*, maio de 1999.

[27] R. M. Roth e G. Ruckenstein. Efficient decoding of Reed-Solomon code beyond half the minimum distance. *IEEE Trans. Info. Theory*, 46(1):246–256, janeiro de 2000.

[28] D. Joyner. *Coding Theory and Cryptography*. Springer-Verlag, 1999.

[29] Mitchell C. *Cryptography and Coding II*. Clarendon Press, 1992.

[30] S. B. Wicker e Bhargava V. K. *Reed-Solomon Codes and Their Applications*. IEEE Press, 1994.

[31] Berlekamp E. R. e McEliece R. J. The application of error control to communications. *IEEE Commun. Mag.*, 25:44–57, 1987.

[32] A. Vardy e Y. Be'ery. Bit-level soft-decision decoding of Reed-Solomon codes. *IEEE Trans. Commun.*, 39(3):440–445, 1991.

[33] Haccoun D., Wu W. W., Peile R. E., e Hirata Y. Coding for satellite communication. *IEEE J. Select. Areas Commun.*, 5:724–785, maio de 1987.

[34] P. Elias e A. Shokrollahi. Coding for two noisy channels. *Information Theory*, páginas 61–74, 1956.

[35] R. D. Wesel, X. Liu, e W. Shi. Trellis codes for periodic erasures. *IEEE Trans. Communications*, 48:938–947, 200.

[36] R. J. McElliece. On the average list size for the Guruswami-Sudan decoder. *ISCTA03*, 2003.

# APÊNDICE A

# CÓDIGO FONTE DO PROGRAMA DE SIMULAÇÃO

Neste apêndice temos a listagem dos códigos fontes usados nas simulações do Capítulo 5.

```
////////////////////////////////////////////////////////////////////////
// c1511.cpp
// Código fonte que obtém o desempenho simulado do algoritmo de decodifi-
// cação de KV com apagamento de bits para o código RS(15,11)
////////////////////////////////////////////////////////////////////////

#define TYPE GFNUM2m
#include "polynomialT.cpp"
//#define DO_PRINT
#if TYPE==ModAr
#include "ModAr.h"
#endif
#if TYPE==GFNUM2m
#include "GFNUM2m.h"
#include "GFNUM2m.cc"
#endif
#include<math.h>
#include "rothruck.h"
#include <iostream>
#include <fstream>

template class polynomialT<TYPE>;
template class polytemp<TYPE>;
template class polynomialT<polynomialT<TYPE> >;


/* Protótipos de Funções */
double gran(void);

int computetm(int n,int k,int m);

int computeLm(int n,int k, int m);

polynomialT<polynomialT<GFNUM2m> >
kotter(int n,int L,GFNUM2m *xi, GFNUM2m *yi,int *mi, int *wdeg,int rlex,
   int m1);

GFNUM2m evaluate(polynomialT<polynomialT<GFNUM2m> > &Q,GFNUM2m a,
 GFNUM2m b);

TYPE computeD(int r,int s,const polynomialT<polynomialT<TYPE> > &Q,
  TYPE a, TYPE b);

int hammdist(const polynomialT<TYPE> &p, const polynomialT<TYPE> &r,
 const TYPE *ss, int n);

/* Programa principal */
main()
{
  const int m = 4;
  GFNUM2m::initgf(m);
  const int n = 15;
  const int k = 11;
  int d = n-k+1;
  int t = int(floor((d-1)/2));
  int m1 = 2;
  int tm = computetm(n,k,m1);
  const int km = tm;
  GFNUM2m sc[n]; // conjunto suporte

  cout << "n=" << n << "  k=" << k << "  d=" << d << "  t0=" << t <<
 "  m=" << m1 << "  tm=" << tm <<
```

```
 "  Lm=" << computeLm(n,k,m1) << endl;
  for (int i = 0 ; i < n ; i++ )
  {
    sc[i] = A^i;
  }
  polynomialT<GFNUM2m> xi(n-1,sc);
#ifdef DO_PRINT
  cout << "xi=" << xi << endl;
#endif
  /* inicializações do decodificador */
  int wdeg[2] = {1,k-1}; // peso ponderado
  int rlex = 1; // rlex=1 para revlex; rlex=0 para lex
  int *mi = NULL;     //vetor de multiplicidades;
  int Lm = computeLm(n,k,m1);
  double SNRdBstart = 0;
  double SNRdBend = 10;
  double SNRdBstep = 0.5;
  double SNRdB,SNR;
// variancia do ruido (sigma2- sigma ao quadrado) e desvio padrão (sigma)
  double sigma2,sigma;
// define o num. de erros de bits máx. que deve contar para estimar o BER
  int numerrstocount2[21]={100, 100, 100, 100, 100, 100 ,100, 100, 100,
    100, 100, 100, 100, 100, 100, 100 , 100 ,100, 100 ,100 ,100};
  /* contador de palavras de código errados */
  long codeworderrbw=0;
  long codewordcont=0;
  long codeworderrteo=0;
  long codeworderrprat=0;
  /* taxa do código */
  double R = double(k)/double(n);
  double Ecsqrt=1;
  /* variável que indica a ocorrência de um ou mais erros em um símbolo*/
  int simberr;
 // variável que armazena a quantidade de simbolos errados numa codeword
  int simberrcont;
  /* vetor que armazena a codeword recebida depois de quantizada */
  GFNUM2m recebido[n];
  /* vetor que armazena a codeword recebida antes de ser quantizada */
  double r3[n][m];
  /* polinômio recebido (com coeficientes zero suprimidos ) */
  polynomialT<GFNUM2m> r;
  /* polinômio recebido (sem supressão dos coeficientes = zero) */
  polynomialT<GFNUM2m> r2;
  polynomialT<polynomialT<GFNUM2m> > Q;
  /* variável que indica se encontrou o polinômio mensagem na lista de
    pol. decod.*/
  int polyreceived=0;
  GFNUM2m  aux2;
  /* cria matriz de multiplidades */
  long int mstar[n+1][n];
  long int L = 0;
  double   mxc=0;
  double   mxm=0;
  long int mx1 = 0;
  mi=new int[2*n*(n+1)];
  GFNUM2m *alpha = new GFNUM2m [2*n*(n+1)];
  GFNUM2m *beta = new GFNUM2m [2*n*(n+1)];
  int pont=0;
  int ve;
  int e=tm;
  const int s=n;
  GFNUM2m aux=0;
  double f[n][n+1], somaf[n], pi[n+1][n];
  int linha, coluna, bitindex;
  const float lim=0.2;
  double simbrealib=1;
  int simberasureindex[n], simberasurecont=0, biterasurecont=0;
  long erasuretotal=0;
  /* abre arquivos para armazenar os resultados */
  ofstream myfile1;
  ofstream myfile2;
  ofstream myfile3;
  myfile1.open ("CER-BW.txt");
```

```
 myfile2.open ("CER-KVLinf.txt");
 myfile3.open ("CER-KVL.txt");
 /* loop que percorre os diferente valores de SNR*/
 for(SNRdB = SNRdBstart; SNRdB <= SNRdBend; SNRdB += SNRdBstep)
 {
   // Converte de dB: SNR=Eb/N0
 SNR = pow(10.,SNRdB/10.);
   // reseta o contador de codewords recebidas
   codewordcont=0;
   // reseta o contador de codewords erradas
   codeworderrbw=0;
   codeworderrteo=0;
   codeworderrprat=0;
   sigma2 = Ecsqrt*Ecsqrt/(2.*SNR*R); // calcula a variância do SNR
   sigma = sqrt(sigma2);
// assume que 0 é o bit de entrada (transmite -sqrt(Ec) em cada posição)
   // fica no while enquanto não encontrar um número de erros suficiente
   while( (codeworderrbw < numerrstocount2[(int)(SNRdB*2)])
              || (codeworderrteo < numerrstocount2[(int)(SNRdB*2)])
              || (codeworderrprat < numerrstocount2[(int)(SNRdB*2)]) )
   {
     // reseta flag de indicacao de erro de simbolo
     simberr=0;
     // reseta contador de erros de simbolo em uma codeword
     simberrcont = 0;
     simberasurecont=0;
     // percorre os n simbolos da codeword
   for(int i2 = 0; i2 < n; i2++)
     {
       recebido[i2]=0x00;
       biterasurecont=0;
       // percorre os m bits que formam o simbolo
       simbrealib=1;
       for(int i3=0; i3<m; i3++)
       {
         // adiciona ruído
         r3[i2][i3] = -Ecsqrt + sigma*gran();
         simbrealib *= r3[i2][i3];
         // se houve erro no bit recebido
         if ( r3[i2][i3] > 0 )
         {
           // sinaliza erro no simbolo
           simberr=1;
           // vai montando o simbolo recebido
           recebido[i2]+= (0x01 << i3);
         }
         if( r3[i2][i3] < lim && r3[i2][i3] > -lim )
         {
             r3[i2][i3] = 0;
             erasuretotal++;
         }

       // incrementa o contador de bits transmitidos
       } // próximo bit
       // caso o simbolo tenha sido corrompido pelo ruido
         if ( simberr == 1)
         {
           // reseta o flag de simbolo com erro
           simberr=0;
     // incrementa o contador de simbolos errados detectados na codeword
           simberrcont++;
         }
     }
     if ( (2*simberrcont + simberasurecont) >= (n-k+1) ){
             codeworderrbw++;    //}

     // Constroi matriz pi de confianca
     // Varre a probabilidade por posição na codeword
     for ( coluna = 0 ; coluna < n ; coluna++ )
     {
       somaf[coluna] = 0;
       //calcula a probabilidade de cada simbolo nesta posicao
       for ( linha = 0 ; linha < (n+1) ; linha++ )
```

```
        {
          f[coluna][linha] = 1;
          if (linha == 0)
            aux = 0;
          else
            aux = A^(linha-1);
// Passa por todos os bits de cada simbolo para calcular a prob do simb.
          for ( bitindex = 0 ; bitindex < m ; bitindex++ )
          {
            if ( ( ( aux.getv() >> bitindex ) & 0x01 ) )
            {
              f[coluna][linha] *= ( 1 / (2 * 3.1416) ) *
              exp( ( -1 / ( 2 * sigma2 ) )
              * ( r3[coluna][bitindex] - Ecsqrt )
              *( r3[coluna][bitindex] - Ecsqrt )  );
            }
            else
            {
              f[coluna][linha] *= ( 1 / (2 * 3.1416) )
              * exp( ( -1 / ( 2 * sigma2 ) )
              * ( ( r3[coluna][bitindex] + Ecsqrt )
              * ( r3[coluna][bitindex] + Ecsqrt) ) );
            }
          }
          somaf[coluna] += f[coluna][linha];
        }
        for ( linha = 0 ; linha < (n+1) ; linha++ )
        {
          pi[linha][coluna] = f[coluna][linha]/somaf[coluna];
        }
      }
      // faz o mapeamento de pi para mi
      L=2.99; mxc=0; mxm=0; mx1=0;
      int simberasurecont2=0;
      for (linha = 0 ; linha < (n+1) ; linha ++ )
      {
        for (coluna = 0; coluna < n; coluna ++ )
        {
          if(coluna != simberasureindex[simberasurecont2]
          || simberasurecont2 == simberasurecont)
          {
            mstar[linha][coluna]=((long)(L)*pi[linha][coluna]);
            mxm += pi[linha][coluna]*pi[linha][coluna];
            mx1 += mstar[linha][coluna]*1;
            if( linha == 0  )
            {
              mxc += pi[linha][coluna];
            }
          }
          else
          {
            simberasurecont2++;
            mstar[linha][coluna]=((long)(L/2)*pi[linha][coluna]);
          }
        }
      }
      ve=k-1;
      if( (mxc/sqrt(mxm)) < (sqrt(ve)) )
      {
        codeworderrteo++;
      }
      pont=0;
      for (coluna = 0; coluna < n ; coluna ++ )
      {
        for ( linha = 0 ; linha < (n+1) ; linha ++ )
        {
          if ( mstar[linha][coluna] != 0  )
          {
            mi[pont] = mstar[linha][coluna];
            alpha[pont]=A^coluna;
            if(linha==0)
              beta[pont]=0;
            else
```

```
            beta[pont]=A^(linha-1);
#ifdef DO_PRINT
            cout << "recebido" << "[" << coluna << "]= "
              << recebido[coluna] << endl;
            cout << "mi" << "[" << pont << "]= " << mi[pont]<< endl;
#endif
            pont++;
          }
        }
      }
      Lm = (int)(sqrt(mxm+mx1))/(sqrt(k-1));
      //cout << "Lm=" << Lm << endl;
      // Inicia decodificação da codeword recebida
      // Monta um polinômio de código com o vetor recebido
#ifdef DO_PRINT
      cout << "r2=" << r2 << endl;
      system("pause");
#endif

      // chama rotina de interpolação
      Q = kotter(pont,Lm,alpha,beta,mi, wdeg,rlex,m1);
      r.setc(recebido,n-1);
      // Expande o polinômio de código para o seu máximo grau
        //(com intuito de facilitar a decodificação)
      // Inicializa ponteiro para receber a lista de prováveis
      // polinômios mensagem
      rpolynode *rptr,*rptr1;
      // fatora Q
      rptr = rothruck(Q,k-1); // find the y-roots of Q
      // se não encontrou nenhum polinômio
      if( ( rptr ) == NULL )
      {
        // atualiza o contador de erro de bit com codificação GS
        codeworderrprat++;
      }
      else
      {
        // inicializa variável que indica se encontrou o polinômio
        //mensagem na lista de pol.
        polyreceived=1;
        // percorre a lista de prováveis polinômios mensagens obtidos
        for(rptr1 = rptr; rptr1 != NULL; rptr1 = rptr1->next)
        {
          //cout << "rptr=" << rptr1->next << endl;
        // se encontrar um polinômio de grau zero, que equivale a p(x)=0;
          if(  rptr1->f.getdegree() != 0  )
          {
            if(hammdist(rptr1->f,r,sc,n) <= tm)
            {
              polyreceived=1;
            }
          }
          else
          {
            polyreceived=0;
          }
        }
        // verifica se depois de percorrer a lista de polinômios foi
        //encontrado o pol. transmitido
        if (polyreceived == 1)
        {
          // como não encontrou o pol. mensagem, atualiza contagem de
          // codewords errados pela decod. gs
          codeworderrprat++;
        }
      }
      }
      codewordcont++;

    }// proxima codeword

    // imprime na tela os resultados e  progresso da simulação
    cout << "SNR(dB)=" << SNRdB << " CER(BW)=" <<
```

```
   double(codeworderrbw)/double(codewordcont) << flush << endl;
    cout << "SNR(dB)=" << SNRdB << " CER(KV L=inf" <<")=" <<
   double(codeworderrteo)/double(codewordcont) << flush << endl;
    cout << "SNR(dB)=" << SNRdB << " CER(KV L=" << L << ")=" <<
   double(codeworderrprat)/double(codewordcont) << flush << endl;
    cout << "Total de Erasures: "  << erasuretotal << " Erasure rate: "
     << double(erasuretotal)/double(n*m*codewordcont) << flush << endl;
    erasuretotal=0;
    // escreve os resultados nos arquivos
    myfile1 <<  double(codeworderrbw)/double(codewordcont)
      << flush << endl;
    myfile2 <<  double(codeworderrteo)/double(codewordcont)
      << flush << endl;
    myfile3 <<  double(codeworderrprat)/double(codewordcont)
      << flush << endl;
  }
  // encerra arquivos e termina programa
  myfile1.close();
  myfile2.close();
  myfile3.close();
  cout << "Termino " << endl;
  system ("pause");
}

//Função para calcular a distância de hamming entre duas palavras códigos
int hammdist(const polynomialT<TYPE> &p, const polynomialT<TYPE> &r,
 const TYPE *ss, int n)
{
   TYPE codesym;
   int hdist = 0;
   int j;
   for(j=0; j <= r.getdegree(); j++)
   {
  codesym = p(ss[j]);
  if(codesym != r[j]) hdist++;
   }
   for(  ; j < n; j++)
   {
  codesym = p(ss[j]);
  if(codesym != 0) hdist++;
   }
   return hdist;
}


////////////////////////////////////////////////////////////////////////
// c3125.cpp
// Código fonte que obtém o desempenho simulado do algoritmo de decodifi-
// cação de KV com apagamento de bits para o código RS(31,25)
////////////////////////////////////////////////////////////////////////

#define TYPE GFNUM2m
#include "polynomialT.cpp"
//#define DO_PRINT
#if TYPE==ModAr
#include "ModAr.h"
#endif
#if TYPE==GFNUM2m
#include "GFNUM2m.h"
#include "GFNUM2m.cc"
#endif
#include<math.h>
#include "rothruck.h"
#include <iostream>
#include <fstream>

template class polynomialT<TYPE>;
template class polytemp<TYPE>;
template class polynomialT<polynomialT<TYPE> >;


/* Protótipos de Funções */
double gran(void);
```

```
int computetm(int n,int k,int m);

int computeLm(int n,int k, int m);

polynomialT<polynomialT<GFNUM2m> >
kotter(int n,int L,GFNUM2m *xi, GFNUM2m *yi,int *mi, int *wdeg,int rlex,
   int m1);

GFNUM2m evaluate(polynomialT<polynomialT<GFNUM2m> > &Q,GFNUM2m a,
 GFNUM2m b);

TYPE computeD(int r,int s,const polynomialT<polynomialT<TYPE> > &Q,
  TYPE a, TYPE b);

int hammdist(const polynomialT<TYPE> &p, const polynomialT<TYPE> &r,
 const TYPE *ss, int n);

/* Programa principal */
main()
{
  const int m = 5;
  GFNUM2m::initgf(m);
  const int n = 31;
  const int k = 25;
  int d = n-k+1;
  int t = int(floor((d-1)/2));
  int m1 = 2;
  int tm = computetm(n,k,m1);
  const int km = tm;
  GFNUM2m sc[n]; // conjunto suporte

  cout << "n=" << n << "  k=" << k << "  d=" << d << "  t0=" << t <<
 "  m=" << m1 << "  tm=" << tm <<
 "  Lm=" << computeLm(n,k,m1) << endl;
  for (int i = 0 ; i < n ; i++ )
  {
    sc[i] = A^i;
  }
  polynomialT<GFNUM2m> xi(n-1,sc);
#ifdef DO_PRINT
  cout << "xi=" << xi << endl;
#endif
  /* inicializações do decodicador */
  int wdeg[2] = {1,k-1}; // peso ponderado
  int rlex = 1; /// rlex=1 para revlex; rlex=0 para lex
  int *mi = NULL; //vetor de multiplicidades;
  int Lm = computeLm(n,k,m1);

  double SNRdBstart = 0;
  double SNRdBend = 10;
  double SNRdBstep = 0.5;
  double SNRdB,SNR;
// variancia do ruido (sigma2- sigma ao quadrado) e desvio padrão (sigma)
  double sigma2,sigma;
// define o num. de erros de bits máx que deve contar para estimar o BER
  //int numerrstocount = 3000;
  int numerrstocount2[21]={100, 100, 100, 100, 100, 100 ,100, 100, 100,
            100, 100, 100, 100, 100, 100, 100 , 100 ,100, 100 ,100 ,100};
  /* contador de palavras de código errados */
  long codeworderrbw=0;
  long codewordcont=0;
  long codeworderrteo=0;
  long codeworderrprat=0;
  /* taxa do código */
  double R = double(k)/double(n);
  double Ecsqrt=1;
/* variável que indica a ocorrência de um ou mais erros em um símbolo */
  int simberr;
/* variável que armazena a quantidade de simbolos errados numa codeword*/
  int simberrcont;
  /* vetor que aramazena a codeword recebida depois de quantizada */
  GFNUM2m recebido[n];
```

```
  /* vetor que armazena a codeword recebida antes de ser quantizada */
  double r3[n][m];
  /* polinômio recebido (com coeficientes zero suprimidos ) */
  polynomialT<GFNUM2m> r;
  /* polinômio recebido (sem supressão dos coeficientes = zero) */
  polynomialT<GFNUM2m> r2;
  polynomialT<polynomialT<GFNUM2m> > Q;
//variável que indica se encontrou o polin mensagem na lista de pol. decod.
  int polyreceived=0;
  GFNUM2m  aux2;
  /* cria matriz de multiplidades */
  long int mstar[n+1][n];
  long int L = 0;
  double   mxc=0;
  double   mxm=0;
  long int mx1 = 0;
  mi=new int[2*n*(n+1)];
  GFNUM2m *alpha = new GFNUM2m [2*n*(n+1)];
  GFNUM2m *beta = new GFNUM2m [2*n*(n+1)];
  int pont=0;
  int ve;
  int e=tm;
  const int s=n;
  GFNUM2m aux=0;
  double f[n][n+1], somaf[n], pi[n+1][n];
  int linha, coluna, bitindex;
  const float lim=0.2;
  double simbrealib=1;
  int simberasureindex[n], simberasurecont=0, biterasurecont=0;
  long erasuretotal=0;
  /* abre arquivos para armazenar os resultados */
  ofstream myfile1;
  ofstream myfile2;
  ofstream myfile3;
  myfile1.open ("CER-BW.txt");
  myfile2.open ("CER-KVLinf.txt");
  myfile3.open ("CER-KVL.txt");
  // percorre os valores de SNR
  for(SNRdB = SNRdBstart; SNRdB <= SNRdBend; SNRdB += SNRdBstep)
  {
    // Converte de dB: SNR=Eb/N0
  SNR = pow(10.,SNRdB/10.);
    // reseta o contador de codewords recebidas
    codewordcont=0;
    // reseta o contador de codewords erradas
    codeworderrbw=0;
    codeworderrteo=0;
    codeworderrprat=0;
    sigma2 = Ecsqrt*Ecsqrt/(2.*SNR*R);
    sigma = sqrt(sigma2);
 // assume que 0 é o bit de entrada (transmite -sqrt(Ec) em cada posição)
    // fica no while enquanto não encontrar um número de erros suficiente
    while( (codeworderrbw < numerrstocount2[(int)(SNRdB*2)])
                || (codeworderrteo < numerrstocount2[(int)(SNRdB*2)])
               || (codeworderrprat < numerrstocount2[(int)(SNRdB*2)]) )
    {
      // reseta flag de indicacao de erro de simbolo
      simberr=0;
      // reseta contador de erros de simbolo em uma codeword
      simberrcont = 0;
      simberasurecont=0;
      // percorre os n simbolos da codeword
    for(int i2 = 0; i2 < n; i2++)
      {
        recebido[i2]=0x00;
        biterasurecont=0;
        // percorre os m bits que formam o simbolo
        simbrealib=1;
        for(int i3=0; i3<m; i3++)
        {
          // adiciona ruído
          r3[i2][i3] = -Ecsqrt + sigma*gran();
          simbrealib *= r3[i2][i3];
```

```
          // se houve erro no bit recebido
          if ( r3[i2][i3] > 0 )
          {
            // sinaliza erro no simbolo
            simberr=1;
            // vai montando o simbolo recebido
            recebido[i2]+= (0x01 << i3);
          }
          if( r3[i2][i3] < lim && r3[i2][i3] > -lim )
          {
              r3[i2][i3] = 0.01;
              erasuretotal++;
          }
          if ( simberr == 1)
          {
            // reseta o flag de simbolo com erro
            simberr=0;
    // incrementa o contador de simbolos errados detectados na codeword
            simberrcont++;
          }
      }
      if ( (2*simberrcont + simberasurecont) >= (n-k+1) ){
              codeworderrbw++;    //}

      // Constroi matriz pi de confianca
      // Varre a probabilidade por posição na codeword
      for ( coluna = 0 ; coluna < n ; coluna++ )
      {
        somaf[coluna] = 0;
        //calcula a probabilidade de cada simbolo nesta posicao
        for ( linha = 0 ; linha < (n+1) ; linha++ )
        {
          f[coluna][linha] = 1;
          if (linha == 0)
            aux = 0;
          else
            aux = A^(linha-1);

// Passa por todos os bits de cada simbolo para calcular a prob do simbolo
          for ( bitindex = 0 ; bitindex < m ; bitindex++ )
          {
            if ( ( ( aux.getv() >> bitindex ) & 0x01 ) )
            {
              f[coluna][linha] *= ( 1 / (2 * 3.1416) )
              * exp( ( -1 / ( 2 * sigma2 ) )
              * ( r3[coluna][bitindex] - Ecsqrt )
              *( r3[coluna][bitindex] - Ecsqrt )  );
            }
            else
            {
              f[coluna][linha] *= ( 1 / (2 * 3.1416) )
              * exp( ( -1 / ( 2 * sigma2 ) )
              * ( ( r3[coluna][bitindex] + Ecsqrt )
              * ( r3[coluna][bitindex] + Ecsqrt) ) );
            }
          }
          somaf[coluna] += f[coluna][linha];
        }
        for ( linha = 0 ; linha < (n+1) ; linha++ )
        {
          pi[linha][coluna] = f[coluna][linha]/somaf[coluna];
        }
      }
      // faz o mapeamento de pi para mi
      L=4.99; mxc=0; mxm=0; mx1=0;
      int simberasurecont2=0;
      for (linha = 0 ; linha < (n+1) ; linha ++ )
      {
        for (coluna = 0; coluna < n; coluna ++ )
        {
          if(coluna != simberasureindex[simberasurecont2]
          || simberasurecont2 == simberasurecont)
          {
```

```
            mstar[linha][coluna]=((long)(L)*pi[linha][coluna]);
            mxm += pi[linha][coluna]*pi[linha][coluna];
            mx1 += mstar[linha][coluna]*1;
            if( linha == 0  )
            {
              mxc += pi[linha][coluna];
            }
          }
          else
          {
            simberasurecont2++;
            mstar[linha][coluna]=((long)(L/2)*pi[linha][coluna]);
          }
        }
      }
      ve=k-1;
      if( (mxc/sqrt(mxm)) < (sqrt(ve)) )
      {
        codeworderrteo++;
      }
      pont=0;
      for (coluna = 0; coluna < n ; coluna ++ )
      {
        for ( linha = 0 ; linha < (n+1) ; linha ++ )
        {
          if ( mstar[linha][coluna] != 0  )
          {
            mi[pont] = mstar[linha][coluna];
            alpha[pont]=A^coluna;
            if(linha==0)
              beta[pont]=0;
            else
              beta[pont]=A^(linha-1);
#ifdef DO_PRINT
            cout << "recebido" << "[" << coluna << "]= " <<
            recebido[coluna] << endl;
            cout << "mi" << "[" << pont << "]= " << mi[pont]<< endl;
#endif
            pont++;
          }
        }
      }
      Lm = (int)(sqrt(mxm+mx1))/(sqrt(k-1));
      //cout << "Lm=" << Lm << endl;
      // Inicia decodificação da codeword recebida
      // Monta um polinômio de código com o vetor recebido
#ifdef DO_PRINT
      cout << "r2=" << r2 << endl;
      system("pause");
#endif

      // chama rotina de interpolação
      Q = kotter(pont,Lm,alpha,beta,mi, wdeg,rlex,m1);
      r.setc(recebido,n-1);
// Inicializa ponteiro para receber a lista de prováveis polin.s mensagem
      rpolynode *rptr,*rptr1;
      // fatora Q
      rptr = rothruck(Q,k-1); // find the y-roots of Q
      // se não encontrou nenhum polinômio
      if( ( rptr ) == NULL )
      {
        // atualiza o contador de erro de bit com codificação GS
        codeworderrprat++;
      }
      else
      {
        // inicializa variável que indica se encontrou o polinômio
        //mensagem na lista de pol.
        polyreceived=1;
        // percorre a lista de prováveis polinômios mensagens obtidos
        for(rptr1 = rptr; rptr1 != NULL; rptr1 = rptr1->next)
        {
          //cout << "rptr=" << rptr1->next << endl;
```

```
        // se encontrar um polinômio de grau zero, que equivale a p(x)=0;
          if(  rptr1->f.getdegree() != 0  )
          {
            if(hammdist(rptr1->f,r,sc,n) <= tm)
            {
              polyreceived=1;
            }
          }
          else
          {
            polyreceived=0;
          }
        }
        // verifica se depois de percorrer a lista de polinômios
        //foi encontrado o pol. transmitido
        if (polyreceived == 1)
        {
          // como não encontrou o pol. mensagem, atualiza contagem de
          // codewords errados pela decod. gs
          codeworderrprat++;
        }
      }
      }
      codewordcont++;

    }// proxima codeword

    // imprime na tela os resultados e  progresso da simulação
    cout << "SNR(dB)=" << SNRdB << " CER(BW)=" <<
   double(codeworderrbw)/double(codewordcont) << flush << endl;
    cout << "SNR(dB)=" << SNRdB << " CER(KV L=inf" <<")=" <<
   double(codeworderrteo)/double(codewordcont) << flush << endl;
    cout << "SNR(dB)=" << SNRdB << " CER(KV L=" << L << ")=" <<
   double(codeworderrprat)/double(codewordcont) << flush << endl;
    cout << "Total de Erasures: "  << erasuretotal << " Erasure rate: "
     << double(erasuretotal)/double(n*m*codewordcont) << flush << endl;
    erasuretotal=0;
    // escreve os resultados nos arquivos
    myfile1 <<  double(codeworderrbw)/double(codewordcont)
    << flush << endl;
    myfile2 <<  double(codeworderrteo)/double(codewordcont)
    << flush << endl;
    myfile3 <<  double(codeworderrprat)/double(codewordcont)
    << flush << endl;
  }
  // encerra arquivos e termina programa
  myfile1.close();
  myfile2.close();
  myfile3.close();
  cout << "Termino " << endl;
  system ("pause");
}

////////////////////////////////////////////////////////////////////////
// c31252.cpp
// Código fonte que obtém o desempenho simulado do algoritmo de decodifi-
// cação de KV com apagamento de símbolos para o código RS(31,25)
////////////////////////////////////////////////////////////////////////

#define TYPE GFNUM2m
#include "polynomialT.cpp"
//#define DO_PRINT
#if TYPE==ModAr
#include "ModAr.h"
#endif
#if TYPE==GFNUM2m
#include "GFNUM2m.h"
#include "GFNUM2m.cc"
#endif
#include<math.h>
#include "rothruck.h"
#include <iostream>
#include <fstream>
```

```
template class polynomialT<TYPE>;
template class polytemp<TYPE>;
template class polynomialT<polynomialT<TYPE> >;


/* Protótipos de Funções */
double gran(void);

int computetm(int n,int k,int m);

int computeLm(int n,int k, int m);

polynomialT<polynomialT<GFNUM2m> >
kotter(int n,int L,GFNUM2m *xi, GFNUM2m *yi,int *mi, int *wdeg,int rlex,
   int m1);

GFNUM2m evaluate(polynomialT<polynomialT<GFNUM2m> > &Q,GFNUM2m a,
 GFNUM2m b);

TYPE computeD(int r,int s,const polynomialT<polynomialT<TYPE> > &Q,
  TYPE a, TYPE b);

int hammdist(const polynomialT<TYPE> &p, const polynomialT<TYPE> &r,
 const TYPE *ss, int n);

/* Programa principal */
main()
{
  const int m = 5;
  GFNUM2m::initgf(m);
  const int n = 31;
  const int k = 25;
  int d = n-k+1;
  int t = int(floor((d-1)/2));
  int m1 = 2;
  int tm = computetm(n,k,m1);
  const int km = tm;
  GFNUM2m sc[n];

  cout << "n=" << n << "  k=" << k << "  d=" << d << "  t0=" << t <<
 "  m=" << m1 << "  tm=" << tm <<
 "  Lm=" << computeLm(n,k,m1) << endl;
  for (int i = 0 ; i < n ; i++ )
  {
    sc[i] = A^i;
  }
  polynomialT<GFNUM2m> xi(n-1,sc);
#ifdef DO_PRINT
  cout << "xi=" << xi << endl;
#endif
  /* inicializações do decodicador */
  int wdeg[2] = {1,k-1}; // peso ponderado
  int rlex = 1; // rlex=1 para revlex; rlex=0 para lex
  int *mi = NULL; //vetor de multiplicidades;
  int Lm = computeLm(n,k,m1);

  double SNRdBstart = 0;
  double SNRdBend = 10;
  double SNRdBstep = 0.5;
  double SNRdB,SNR;
// variancia do ruido (sigma2- sigma ao quadrado) e desvio padrão (sigma)
  double sigma2,sigma;
/* define o num de erros de bits máx que deve contar para estimar o BER*/
  int numerrstocount2[21]={100, 100, 100, 100, 100, 100 ,100, 100, 100,
  100, 100, 100, 100, 100, 100, 100 , 100 ,100, 100 ,100 ,100};
  /* contador de palavras de código errados */
  long codeworderrbw=0;
  long codewordcont=0;
  long codeworderrteo=0;
  long codeworderrprat=0;
  /* taxa do código */
  double R = double(k)/double(n);
```

```
  double Ecsqrt=1;
/* variável que indica a ocorrência de um ou mais erros em um símbolo */
  int simberr;
/* variável que armazena a quantidade de simbolos errados numa codeword */
  int simberrcont;
  /* vetor que aramazena a codeword recebida depois de quantizada */
  GFNUM2m recebido[n];
  /* vetor que armazena a codeword recebida antes de ser quantizada */
  double r3[n][m];
  /* polinômio recebido (com coeficientes zero suprimidos ) */
  polynomialT<GFNUM2m> r;
  /* polinômio recebido (sem supressão dos coeficientes = zero) */
  polynomialT<GFNUM2m> r2;
  polynomialT<polynomialT<GFNUM2m> > Q;
  /* variável que indica se encontrou o polinômio mensagem na lista
  de pol. decod.*/
  int polyreceived=0;
  GFNUM2m  aux2;
  /* cria matriz de multiplidades */
  long int mstar[n+1][n];
  long int L = 0;
  double   mxc=0;
  double   mxm=0;
  long int mx1 = 0;
  mi=new int[2*n*(n+1)];
  GFNUM2m *alpha = new GFNUM2m [2*n*(n+1)];
  GFNUM2m *beta = new GFNUM2m [2*n*(n+1)];
  int pont=0;
  int ve;
  int e=tm;
  const int s=n;
  GFNUM2m aux=0;
  double f[n][n+1], somaf[n], pi[n+1][n];
  int linha, coluna, bitindex;
  const float lim=0.2;
  double simbrealib=1;
  int simberasureindex[n], simberasurecont=0, biterasurecont=0;
  long erasuretotal=0;
  int simberasurecont2=0;
    int simberasurecont3=0;
  /* abre arquivos para armazenar os resultados */
  ofstream myfile1;
  ofstream myfile2;
  ofstream myfile3;
  myfile1.open ("CER-BW.txt");
  myfile2.open ("CER-KVLinf.txt");
  myfile3.open ("CER-KVL.txt");
  /* loop que percorre os diferente valores de SNR*/
  for(SNRdB = SNRdBstart; SNRdB <= SNRdBend; SNRdB += SNRdBstep)
  {
    // Converte de dB: SNR=Eb/N0
  SNR = pow(10.,SNRdB/10.);
    // reseta o contador de codewords recebidas
    codewordcont=0;
    // reseta o contador de codewords erradas
    codeworderrbw=0;
    codeworderrteo=0;
    codeworderrprat=0;
    sigma2 = Ecsqrt*Ecsqrt/(2.*SNR*R); // calcula a variância do SNR
    sigma = sqrt(sigma2);
 // assume que 0 é o bit de entrada (transmite -sqrt(Ec) em cada posição)
    // fica no while enquanto não encontrar um número de erros suficiente
    while( (codeworderrbw < numerrstocount2[(int)(SNRdB*2)])
               || (codeworderrteo < numerrstocount2[(int)(SNRdB*2)])
               || (codeworderrprat < numerrstocount2[(int)(SNRdB*2)]) )
    {
      // reseta flag de indicacao de erro de simbolo
      simberr=0;
      // reseta contador de erros de simbolo em uma codeword
      simberrcont = 0;
      simberasurecont=0;
      // percorre os n simbolos da codeword
    for(int i2 = 0; i2 < n; i2++)
```

```
      {
        recebido[i2]=0x00;
        biterasurecont=0;
        // percorre os m bits que formam o simbolo
        simbrealib=1;
        for(int i3=0; i3<m; i3++)
        {
          // adiciona ruído
          r3[i2][i3] = -Ecsqrt + sigma*gran();
          simbrealib *= r3[i2][i3];
          // se houve erro no bit recebido
          if ( r3[i2][i3] > 0 )
          {
            // sinaliza erro no simbolo
            simberr=1;
            // vai montando o simbolo recebido
            recebido[i2]+= (0x01 << i3);
          }
        // incrementa o contador de bits transmitidos
        } // próximo bit
        if( simbrealib < 0.01 && simbrealib  > -0.01 )
        {
          simberasureindex[simberasurecont] = i2;
          simberasurecont++;
          erasuretotal++;
        }
          if ( simberr == 1)
          {
            // reseta o flag de simbolo com erro
            simberr=0;
    // incrementa o contador de simbolos errados detectados na codeword
            simberrcont++;
          }
      }
      if ( (2*simberrcont + simberasurecont) >= (n-k+1) ){
              codeworderrbw++;    }
      // Constroi matriz pi de confianca
      // Varre a probabilidade por posição na codeword
      simberasurecont2=0;
     simberasurecont3=0;
      for ( coluna = 0 ; coluna < n ; coluna++ )
      {
        somaf[coluna] = 0;
        //calcula a probabilidade de cada simbolo nesta posicao
        for ( linha = 0 ; linha < (n+1) ; linha++ )
        {
          f[coluna][linha] = 1;
          if (linha == 0)
            aux = 0;
          else
            aux = A^(linha-1);

//Passa por todos os bits de cada simbolo para calcular a prob do simbolo
          for ( bitindex = 0 ; bitindex < m ; bitindex++ )
          {
            if( simberasureindex[simberasurecont2] != coluna
            || simberasurecont2 == simberasurecont )
            {
              if ( ( ( aux.getv() >> bitindex ) & 0x01 ) )
              {
                f[coluna][linha] *= ( 1 / (2 * 3.1416) )
                * exp( ( -1 / ( 2 * sigma2 ) )
                * ( r3[coluna][bitindex] - Ecsqrt )
                *( r3[coluna][bitindex] - Ecsqrt )  );
              }
              else
              {
                f[coluna][linha] *= ( 1 / (2 * 3.1416) )
                * exp( ( -1 / ( 2 * sigma2 ) )
                * ( ( r3[coluna][bitindex] + Ecsqrt )
                * ( r3[coluna][bitindex] + Ecsqrt) ) );
              }
            }
```

```
            else
            {
               f[coluna][linha] *= ( 1 / (2 * 3.1416) )
               * exp( ( -1 / ( 2 * sigma2 ) )
               * ( ( Ecsqrt ) * ( Ecsqrt) ) );
               simberasurecont3=1;
            }
          }
          somaf[coluna] += f[coluna][linha];
        }
        for ( linha = 0 ; linha < (n+1) ; linha++ )
        {
          pi[linha][coluna] = f[coluna][linha]/somaf[coluna];
        }
        if(simberasurecont3==1)
        {
          simberasurecont2++;
          simberasurecont3=0;
        }
      }
      // faz o mapeamento de pi para mi
      L=4.99; mxc=0; mxm=0; mx1=0;
      int simberasurecont2=0;
      for (linha = 0 ; linha < (n+1) ; linha ++ )
      {
        for (coluna = 0; coluna < n; coluna ++ )
        {
            mstar[linha][coluna]=((long)(L)*pi[linha][coluna]);
            mxm += pi[linha][coluna]*pi[linha][coluna];
            mx1 += mstar[linha][coluna]*1;
            if( linha == 0  )
            {
              mxc += pi[linha][coluna];
            }
        }
      }

      ve=k-1;
      if( (mxc/sqrt(mxm)) < (sqrt(ve)) )
      {
        codeworderrteo++;
      }
      // constroe o vetor de multiplicidades para a codeword

      pont=0;
      for (coluna = 0; coluna < n ; coluna ++ )
      {
        for ( linha = 0 ; linha < (n+1) ; linha ++ )
        {
          if ( mstar[linha][coluna] != 0  )
          {
            mi[pont] = mstar[linha][coluna];
            alpha[pont]=A^coluna;
            if(linha==0)
              beta[pont]=0;
            else
              beta[pont]=A^(linha-1);
#ifdef DO_PRINT
            cout << "recebido" << "[" << coluna << "]= "
            << recebido[coluna] << endl;
            cout << "mi" << "[" << pont << "]= " << mi[pont]<< endl;
#endif
            pont++;
          }
        }
      }
      Lm = (int)(sqrt(mxm+mx1))/(sqrt(k-1));
      //cout << "Lm=" << Lm << endl;
      // Inicia decodificação da codeword recebida
      // Monta um polinômio de código com o vetor recebido
#ifdef DO_PRINT
      cout << "r2=" << r2 << endl;
      system("pause");
```

```
#endif

      // chama rotina de interpolação
      Q = kotter(pont,Lm,alpha,beta,mi, wdeg,rlex,m1);
      r.setc(recebido,n-1);
 // Inicializa ponteiro para receber a lista de prováveis polin. mensagem
      rpolynode *rptr,*rptr1;
      // fatora Q
      rptr = rothruck(Q,k-1); // find the y-roots of Q
      // se não encontrou nenhum polinômio
      if( ( rptr ) == NULL )
      {
        // atualiza o contador de erro de bit com codificação GS
        codeworderrprat++;
      }
      else
      {
        // inicializa variável que indica se encontrou o polinômio
        // mensagem na lista de pol.
        polyreceived=1;
        // percorre a lista de prováveis polinômios mensagens obtidos
        for(rptr1 = rptr; rptr1 != NULL; rptr1 = rptr1->next)
        {
          //cout << "rptr=" << rptr1->next << endl;
        // se encontrar um polinômio de grau zero, que equivale a p(x)=0;
          if(  rptr1->f.getdegree() != 0  )
          {
            if(hammdist(rptr1->f,r,sc,n) <= tm)
            {
              polyreceived=1;
            }
          }
          else
          {
            polyreceived=0;
          }
        }
        // verifica se depois de percorrer a lista de polinômios foi
        // encontrado o pol. transmitido
        if (polyreceived == 1)
        {
          // como não encontrou o pol. mensagem, atualiza contagem de
          //codewords errados pela decod. gs
          codeworderrprat++;
        }
      //}
      }
      codewordcont++;

    }// proxima codeword

    // imprime na tela os resultados e  progresso da simulação
    cout << "SNR(dB)=" << SNRdB << " CER(BW)=" <<
   double(codeworderrbw)/double(codewordcont) << flush << endl;
    cout << "SNR(dB)=" << SNRdB << " CER(KV L=inf" <<")=" <<
   double(codeworderrteo)/double(codewordcont) << flush << endl;
    cout << "SNR(dB)=" << SNRdB << " CER(KV L=" << L << ")=" <<
   double(codeworderrprat)/double(codewordcont) << flush << endl;
    cout << "Total de Erasures: "  << erasuretotal << " Erasure rate: "
     << double(erasuretotal)/double(n*m*codewordcont) << flush << endl;
    erasuretotal=0;
    // escreve os resultados nos arquivos
    myfile1 <<  double(codeworderrbw)/double(codewordcont) <<
    flush << endl;
    myfile2 <<  double(codeworderrteo)/double(codewordcont) <<
    flush << endl;
    myfile3 <<  double(codeworderrprat)/double(codewordcont) <<
    flush << endl;
  }
  // encerra arquivos e termina programa
  myfile1.close();
  myfile2.close();
  myfile3.close();
```

```
  cout << "Termino " << endl;
  system ("pause");
}
```

A seguir a listagem dos códigos fontes das funções comuns utilizadas pelos programas de simulação.

```
////////////////////////////////////////////////////////////////////////
// kotter.cpp
// Implementa o algoritmo de interpolação de Koetter
// Dado n pontos (xi,yi) com i=0,...,n-1, o valor dos pesos para a
// ordenação de monômios ponderada(wdeg), o tipo da ordenação
// (lex ou reverse lex), o ponteiro para um vetor *mi de multiplicidades
// para cada um dos n pontos, retornando o polinômio Q(x,y) de menor grau
// ponderado que passa pelos n pontos
////////////////////////////////////////////////////////////////////////

//#define DO_PRINT
#include "polynomialT.h"

#define TYPE GFNUM2m
#define BIGINT 65535

#if TYPE==ModAr
#include "ModAr.h"
#endif
#if TYPE==GFNUM2m
#include "GFNUM2m.h"
#endif


polynomialT<polynomialT<TYPE> >
kotter(int n,int L,TYPE *xi, TYPE *yi, int *mi,int *wdeg,int rlex,
   int m1)
{
  const int Laux=20;
  polynomialT<polynomialT<TYPE> > f;
  polynomialT<polynomialT<TYPE> > gj[Laux+1];
  TYPE lc[] = {1,1};
  polynomialT<TYPE> l1(1,lc);

  // declaração das funções locais de suporte ao algoritmo
  int computewdeg(const polynomialT<polynomialT<TYPE> > &Q,const int *wdeg,
   int &xi, int &yj);
  TYPE computeD(int r,int s,const polynomialT<polynomialT<TYPE> > &Q,
TYPE a, TYPE b);
  void findminwdeg(const polynomialT<polynomialT<TYPE> > *gj,
int *Jlist,int numinJ,int &jstar,int &xistar,int &yjstar,
int *wdeg,int rlex);


  int xi1,yj1;
  int i,j,j1,jstar,xistar,yjstar;
  polynomialT<TYPE> Deltaj[Laux+1]; // seta a discrepâncias como polinôm.
  polynomialT<TYPE> Delta;
  int Jlist[Laux+1];
  int numinJ,wd,wdmin;
  int r, s;
  int m;

  // Inicialização
  for(int j = 0; j <= L; j++)
  {
  gj[j].setc(j,polynomialT<TYPE>(1));
  gj[j].setvarname("y");
  gj[j].setbeforeafterprint("y","(",")");
  }
#ifdef DO_PRINT
  cout << endl;
#endif
  int C = 0;
  for(i = 0; i < n; i++)
```

```
  {  // loop que percorre os pontos a sere interpolados
  if(mi==NULL)// se mi for NULL usa a multiplic. m1 para todos os ptos
    {
    m=m1;
  }
  else
    {
  m = mi[i];
  }
  C = (m+1)*m/2;
#ifdef DO_PRINT
  cout << "i=" << i << "  C=" << C << endl;
#endif
    for ( r = 0 ; r < m ; r++ )
    {
      for ( s = 0 ; s < (m-r) ; s++ )
      {
#ifdef DO_PRINT
    cout << "   (r,s)=" << r << "," << s << endl;
      for(j = 0; j <= L; j++)
        {
      cout << "   gj[" << j << "]=" << gj[j] << endl;
      cout << "   xi[" << j << "]=" << xi[j] << endl;
        }
#endif
    numinJ = 0;
    for(j = 0; j <= L; j++)
        {
    // calcula a j-ésima discrepância
     Deltaj[j][0] = computeD(r,s,gj[j],xi[i],yi[i]);
     if(Deltaj[j][0] != 0)
          {
      Jlist[numinJ++] = j;
    }
    }
    if(numinJ)
        { // Se J != vazio
    findminwdeg(gj,Jlist,numinJ,jstar,xistar,yjstar,wdeg,rlex);
    f = gj[jstar];
    Delta[0] = Deltaj[jstar][0];
#ifdef DO_PRINT
    cout << "      J= ";
    for(j1 = 0; j1 < numinJ; j1++)cout<< Jlist[j1]<<" ";
          cout << endl;
      cout << "      ";
    for(j1 = 0; j1 < numinJ; j1++)
      cout << "Delta[" << Jlist[j1] << "]=" <<
  Deltaj[Jlist[j1]][0] << " ";
    cout << endl;
    cout << "      jstar= " << jstar << "  f=" << f << "  Delta=" <<
    Delta[0] << endl;
#endif
    for(j1 = 0; j1 < numinJ; j1++)
          {
      j = Jlist[j1];
      if(j != jstar)
            {
      gj[j] = gj[j]*Delta - f*Deltaj[j];
      }
            else if(j == jstar)
            {
      l1[0] = -xi[i];
      gj[j] = f*l1;
      }
      gj[j].setvarname("y");
    } // para j em J
    }  // se J != vazio
    } // for s
    } // for r
  } // for i

  // encontra o polinômio de menor grau ponderado
#ifdef DO_PRINT
```

```
  cout << "Polinômios finais:" << endl;
for(j = 0; j <= L; j++)
  {
  int xi, yj;
cout << "   gj[" << j << "]=" << gj[j] << "  wdeg=" <<
computewdeg(gj[j], wdeg, xi,yj) << endl;
}
#endif
  for(j = 0; j <= L; j++) Jlist[j] = j;
    numinJ = L+1;
  findminwdeg(gj,Jlist,numinJ,jstar,xistar,yjstar,wdeg,rlex);
  return gj[jstar];
}



// Função que encontra o polinômio de duas variáveis Q(x,y) de menor grau
// ponderado

void findminwdeg(const polynomialT<polynomialT<TYPE> > *gj,
 int *Jlist,int numinJ,int &jstar,int &xistar,int &yjstar,
 int *wdeg,int rlex)
{
  int wdmin,j1,j,wd,xi,yj;
  int computewdeg(const polynomialT<polynomialT<TYPE> > &Q,const int *wdeg,
int &xi, int &yj);

  wdmin = BIGINT;
  for(j1 = 0; j1 < numinJ; j1++)
  {
  j = Jlist[j1];
  wd = computewdeg(gj[j],wdeg,xi,yj);
  if(wd < wdmin)
    {
  jstar = j;
  xistar = xi;
  yjstar = yj;
  wdmin = wd;
  }
  else if(wd == wdmin)
    {
  if(rlex==0)
      { // ordenação tipo lex
  if(xi<xistar)
        {
    jstar = Jlist[j];
    xistar = xi;
    yjstar = yj;
  }
  }
  else
      { // ordenação revlex
  if(xi > xistar)
        {
    jstar = Jlist[j];
    xistar = xi;
    yjstar = yj;
  }
  }
  }
  }
}

//Função que calcula o grau ponderado de um polinômio de duas variáveis

int computewdeg(const polynomialT<polynomialT<TYPE> > &Q,const int *wdeg,
int &xi, int &yj)
{
  // this is an inefficient search, being exhaustive.
  // However, it gets the job done.
  int degy = Q.getdegree();
  int degx;
  int deg;
```

```
  int maxdeg = 0;
  xi = yj = 0;
  for(int i = 0; i <= degy; i++)
  {
  degx = Q[i].getdegree();
  for(int j = 0; j <= degx; j++)
    {
  if(Q[i][j] != 0)
      {
  deg = wdeg[0]*j + wdeg[1]*i;
  if(deg > maxdeg)
        {
    maxdeg = deg;
    xi = j;
    yj = i;
  }
      }
    }
  }
  return maxdeg;
}

// Funcão que calcula a derivada de Hasse

TYPE computeD(int r,int s,const polynomialT<polynomialT<TYPE> > &Q,
 TYPE a, TYPE b)
{
   int degy = Q.getdegree();
   TYPE Drs;
   int degx;
   int jchs, ichr;
   int binommod(int n, int k, int m);
   TYPE aij;
   int charac = a.character();

   for(int j = s; j <= degy; j++) {
  jchs = binommod(j,s,charac);
  if(jchs) {
 degx = Q[j].getdegree();
 for(int i = r; i <= degx; i++) {
aij = Q[j][i];
if(aij != 0) {
   ichr = binommod(i,r,charac);
   if(ichr) {
  Drs += aij*(a^(i-r))*(b^(j-s));
   }
}
 }
  }
   }
   return Drs;
}

int binommod(int n, int k, int character)
{
  double prod = 1;
  long int ipod;
  double i,j;
  if(k > n) return 0;
  if(k > n/2) k = n-k;
  if(k <= 1)
  {
  if(k == 0) return 1;
  if(k == 1) return n % character;
  }
  for(i = n-k+1, j=1; i <= n; i++,j++)
  {
  prod *= i/j;
  }
  ipod = (long int)(prod+0.5);
  return ipod % character;
}
```

```
////////////////////////////////////////////////////////////////////////
// rothruck.cpp
// Dado um polinômio de duas variáveis Q(x,y) esta função determina todos
// os polinômios do tipo y-f(x) de grau máximo determinado pelo parâmetro
// D, utilizando o algoritmo de fatoração de Roth-Ruck
////////////////////////////////////////////////////////////////////////

#include "rothruck.h"

static rpolynode *rpolystart = NULL;   // ponteiro para o início da lista
static rpolynode *rpolylist = NULL; // ponteiro da lista
static int rothD; // máximo grau
static int rothnodenum;  // conta os nós dos ramos
static int numinrrpool = 0; // número de soluções encontradas

//#define DO_PRINT


void
QxytoQxxya(const polynomialT<polynomialT<TYPE> > &Q, const TYPE &a,
   polynomialT<polynomialT<TYPE> > &P);
void rothrucktree(polynomialT<polynomialT<TYPE> >  &Qu, int u,
 TYPE *f);
void RootList(TYPE *R,const polynomialT<TYPE> &Q,int &numroots);
int binom(int n, int k);



rpolynode *
rothruck(polynomialT<polynomialT<TYPE> >&Q,int D)
{
   int u = 0;
   TYPE *f = new TYPE[D+1];  // reserva espaço para o polinômio retornado
   f[0]=0;      // inicializa factor com zero para no caso de px ser zero
   // inicializa a lista onde as respostas serão armazenadas
   numinrrpool = 0;
   rpolystart = NULL;
   rpolylist = rpolystart; // inicializa lista no início
   rothD = D; // seta o grau máximo
   rothnodenum = 0;
   // chama a função que irá se executar de forma recursiva
   rothrucktree(Q,u,f);
   return rpolystart;
}

void
rothrucktree(polynomialT<polynomialT<TYPE> > &Qu, int u, TYPE *f)
{

   rothnodenum++;
#ifdef DO_PRINT
   cout << "núm. nó=" << rothnodenum << endl;

   cout <<"Qu=" << Qu[0] << endl;
#endif
   int i;
   if(Qu[0] == 0) {
#ifdef DO_PRINT
  cout << "***** f=";
#endif
          if(rothnodenum==1) u=1; // para caso o p(x) for igual a zero
#ifdef DO_PRINT
  for(i = 0; i < u; i++) {
 cout << f[i] << " ";
  }
  cout << " ******" << endl;
#endif
  // adiciona o polinômio ã lista
  if(rpolystart == NULL) {
 rpolystart = new rpolynode[1];
 rpolylist = rpolystart;
 rpolylist->next = NULL;
  }
```

```
  else if(rpolylist->next == NULL) { // precisa adicionar ã lista
 rpolylist->next = new rpolynode[1];
 rpolylist = rpolylist->next;
 rpolylist->next = NULL;
  }
  (rpolylist->f).setc(f,u-1);
   }
   else if(u <= rothD) {  // tenta outro ramo
  int qdeg = Qu.getdegree();
  TYPE *R = new TYPE[qdeg];  // número max. de raízes possíveis
  int numroots;
  polynomialT<TYPE> *Q0y = polytemp<TYPE>::gettemppoly(qdeg);
  for(i = 0; i <= qdeg; i++) {
 Q0y->operator[](i) = Qu[i][0];
  }
  RootList(R,*Q0y,numroots);

#ifdef DO_PRINT
  cout << "Ramo da Árvore=" << u << endl;
  cout << "Q(0,y)=" << *Q0y << endl;
  cout <<"Raízes: ";
  for(i = 0; i < numroots; i++){
 cout << R[i] << " ";
  }
  cout << endl;
#endif
  polynomialT<polynomialT<TYPE> > Qv;
  for(i = 0; i < numroots; i++) {
#ifdef DO_PRINT
 cout << "Root=" << R[i] << endl;
#endif
 QxytoQxxya(Qu,  R[i], Qv);
 f[u] = R[i];
 rothrucktree(Qv, u+1, f);
  }
  delete [] R;
   }
   else {
#ifdef DO_PRINT
  cout << "Sem resultado\n";
#endif
   }
}

// Função para encontrar as raízes de um polinômio

void RootList(TYPE *R,const polynomialT<TYPE> &Q,int &numroots)
{
   int i = 0;
   numroots = 0;
   if(Q(0) == 0) R[numroots++] = 0;
   for(i = 0; i < Q[0].getN(); i++) {
  if(Q(A^i) == 0) R[numroots++] = A^i;
   }

}


void
QxytoQxxya(const polynomialT<polynomialT<TYPE> > &Q, const TYPE &a,
   polynomialT<polynomialT<TYPE> > &P)
{
   int i,k,j;
   TYPE nk;
   int mod = a.character();
   int ydeg = Q.getdegree();
   P = Q;
   for(i = 1; i <= ydeg; i++) P[i] = polynomialT<TYPE>(0);
   for(i = 1; i <= ydeg; i++) {
  for(k = 0; k <= i; k++) {
 nk = TYPE(binom(i,k) % mod);
 if(nk != 0) {
P[k] += ((Q[i] << k)*(a^(i-k)))*nk;
```

```
 }
  }
   }
   int m = P[0].getdegree();
   for(i = 0; i <= ydeg; i++) {
  for(j = 0; j <= P[i].getdegree(); j++) {
 if(P[i][j] != 0){
if(j < m) m = j;
break;
 }
  }
   }
   for(i = 0; i <= ydeg; i++) {
  P[i] >>= m;
   }
}

int binom(int n, int k)
{
   double prod = 1;
   long int ipod;
   double i,j;
   if(k > n) return 0;
   if(k > n/2) k = n-k;
   if(k <= 1) {
  if(k == 0) return 1;
  if(k == 1) return n;
   }
   for(i = n-k+1, j=1; i <= n; i++,j++) {
  prod *= i/j;
   }
   ipod = (long int)(prod+0.5);
   return ipod;
}

////////////////////////////////////////////////////////////////////////
// rothruck.h
// Define a lista encadeada que armazena todas as raízes de y de Q(x,y)
// encontradas pelo algoritmo de fatoração de roth-ruck
////////////////////////////////////////////////////////////////////////

#ifndef ROTHRUCK_H
#define ROTHRUCK_H

#define TYPE GFNUM2m
#include "polynomialT.h"
#include "GFNUM2m.h"

class rpolynode
{
public:
   rpolynode *next;
   polynomialT<TYPE> f;
   rpolynode() {
  next = NULL;
  f = TYPE(0);
   }
   ~rpolynode() {};
};

rpolynode *
rothruck(polynomialT<polynomialT<TYPE> >&Q,int D);

#endif

////////////////////////////////////////////////////////////////////////
// computeLm.cpp
// Calcula o comprimento máximo da lista de polinômios para o algoritmo
// de GS
////////////////////////////////////////////////////////////////////////

#include <math.h>
int computeLm(int n,int k, int m)
```

```
{
   double v = k-1;
   double t;
   if(m==0) return 1;
   t = (v+2)/(2*v);
   int Lm = int(floor(sqrt( n*m*(m+1)/v + t*t) - t));
   return int(Lm);
}

////////////////////////////////////////////////////////////////////////
// computetm.cpp
// Calcula a capacidade de correção de erros para um código RS(n,k)
// utilizando o algoritmo de decodificação de Guruswami-Sudan
////////////////////////////////////////////////////////////////////////
#include<math.h>

int fact(int n)
{
  if(n < 1)
    return (1);
  else
  return (n*fact(n-1));
}

int nchoosek(int n, int r)
{
  return fact(n)/(fact(r)*fact(n-r));
}

int computetm(int n,int k, int m)
{
  int v = k-1;
  int Km=0,tm=0,k1,l,lk,C,n1;
  if(m==0){
    Km = ceil((n+v+1)/2);
    tm = n-Km;
    return tm;
    }
   C = n*nchoosek(m+1,2);
   k1 = 0;
   while(1)
   {
     k1 = k1+1;
     l = (m*k1)-1;
     lk = floor(l/v);
     n1 = (l+1)*(lk+1) - v/2*lk*(lk+1);
     if(n1 > C) break;
   }
   Km = k1;
   tm = n-Km;
   return tm;
}


////////////////////////////////////////////////////////////////////////
// gran.cpp
// Gera um número aleatório
////////////////////////////////////////////////////////////////////////

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

static int graniset = 0;
static double grangset;

double gran(void)
{
   double rsq, v1, v2, fac;

   if(!graniset) {
  graniset = 1;
  do {
 v1 = 2*(rand()/(double)RAND_MAX) - 1;
```

```
 v2 = 2*(rand()/(double)RAND_MAX) - 1;
 rsq = v1*v1 + v2*v2;
  } while(rsq > 1 || rsq == 0);
  fac = sqrt(-2*log(rsq)/rsq);
  grangset = v1*fac;
  return v2*fac;
   }
   else {
  graniset = 0;
  return grangset;
   }
}
```

FÁBIO RIZENTAL COUTINHO

# DECODIFICAÇÃO POR DECISÃO SUAVE DE CÓDIGOS DE REED-SOLOMON NA PRESENÇA DE SÍMBOLOS APAGADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Engenharia Elétrica, Setor de Tecnologia, Universidade Federal do Paraná.
Orientador: Prof. Dr. Evelio Martín García Fernández

CURITIBA

2007