Códigos Cíclicos
(Dezembro 2017)

Dylan Nakandakari Sugimoto, Estudante, *ITA*, Gabriel Adriano de Melo, *Estudante, ITA.*

***Abstract*— This document is a study of channel coding using cyclic codes, which are codes whose codewords has the property of when rotated result in another codeword. The properties of these codes allow that it is not necessary to store a generator matrix to encode and a parity matrix to decode so that this coding and decoding system consumes less memory, but there is the complication of finding generator polynomials with a great minimum distance, and another problem is that the number of syndromes increases exponentially with increasing block size and number of correctable errors. The objective of this work is find generator polynomials for block sizes of 7, 15, 31, 63, 127 and 255 bits; and evaluate them in terms of probability of error.**

***Index Terms*—**Codification, Channel, Codes, Cycles

I. Introdução

Este documento é um estudo de codificação de canal utilizando códigos cíclicos, que são códigos cuja palavra código possuem a propriedade de quando rotacionadas resultam em outra palavra código. As propriedades desses códigos permitem que não seja necessário armazenar uma matriz geradora para codificar e uma matriz de paridade para decodificar de forma que esse sistema de codificação e decodificação consome menos memória, porém há a complicação de encontrar polinômios geradores com distância mínima grande, e a quantidade de síndromes cresce exponencialmente com o aumento do tamanho do bloco e do número de erros corrigíveis. Esse trabalho tem o objetivo de encontrar polinômios geradores para tamanho de bloco de 7, 15, 31, 63, 127 e 255 bits; e avaliá-los em relação à probabilidade de erro.

II. Medidas, Análise e Indicadores

A. Descrição do algoritmo de procura por bons polinômios geradores.

Em primeiro lugar, programou-se um algoritmo em Python que realizou a fatoração dos termos de D^{n+1}. A fatoração foi realizada por meio da divisão sucessiva do dividendo por fatores primos. Primeiramente, todos os fatores primos de grau até 8 foram encontrados e armazenados em uma lista. Para que um polinômio fosse um fator primo, ele não poderia ser divisível por nenhum outro nessa lista. Os polinômios foram armazenados na forma de uma lista de zeros e uns, e o seu incremento se dava assim como a soma de 1 em binário (com carry). Uma forma mais eficiente seria armazenar os polinômios na forma de vetores de bits, porém a sua manipulação é complicada o que dificulta a programação, por isso optou-se por representar na forma de lista, que é uma estrutura de fácil manipulação em Python.

Dados os fatores primos, realizou-se uma busca, por força bruta, a todos os 2^L -1 possíveis fatores de D^{n+1}. Como o código em Python se mostrou muito lento para essa tarefa para grande quantidade de fatores primos (L = 19, por exemplo), programou-se o algoritmo em C. Nessa versão em C utilizou-se um conjunto de instruções do tipo SIMD (Single Instruction Multiple Data) em processadores projetados pela Intel, para que fosse realizado em poucos ciclos de clock operações de multiplicação sem carry e peso de hamming.

Utilizou-se, para tal, registradores de 128 bits para realizar a multiplicação dos fatores. A representação dos polinômios foi feita por meio desses bits, com cada bit representando um coeficiente do polinômio, sendo o bit menos significativo o representante do coeficiente de D^0. Como o processador utilizado é de 64 bits, fez-se necessário utilizar um vetor de dois elementos *unsigned long long int* para representar 128 bits e 4 desses para representar os 256 bits. [1] Utilizou-se um inteiro de 64 bits para representar todas as possíveis combinações de até 64 fatores. Todos os 2^L -1 possíveis fatores foram percorridos, por meio do incremento desse número, cujos bits representavam se o primo da mesma posição do bit seria ou não multiplicado. O peso de hamming desse número foi um critério de decisão inicial que descartou mais da metade de todas as combinações possíveis. Para $D^{127} + 1$, o algoritmo demorou cerca de 20 milissegundos para encontrar o melhor polinômio gerador, e para $D^{255} + 1$, o algoritmo demorou cerca de 9 minutos.

B. Descrição do algoritmo de codificação

A codificação é realizada pela multiplicação do polinômio gerador encontrado com vetor de informação. O algoritmo de multiplicação, em Python, foi implementado como a soma, módulo 2, de um vetor com os seus deslocamentos indicados pelos bits do outro vetor. O algoritmo da divisão é realizado utilizando o método da divisão euclidiana de polinômios, levando-se em consideração a soma módulo 2. Uma forma mais eficiente de realizar essa multiplicação e divisão é utilizar a representação e manipulação binário de inteiros. Assim, em C, a multiplicação foi feita por uma instrução, em Assembly, que multiplicou dois números de 64 bits e devolveu o resultado em um registrador

S. N. Dylan é estudante no Instituto Tecnológico de Aeronáutica, São José dos Campos, SP 12228-900 BRL (e-mail: dylan-ns@hotmail.com).

M. A. Gabriel, é estudante no Instituto Tecnológico de Aeronáutica, São José dos Campos, SP 12228-900 BRL. (e-mail: gaadrime.melo@gmail.com)

de 128 bits "PCLMULLQLQDQ xmmreg,xmmrm", utilizando a diretiva intrínseca _mm_clmulepi64_si128. Como os fatores só tinha no máximo 64 bits, foi necessárias várias multiplicações e deslocamentos para multiplicar fatores de 128 bits. [2]

Para as instruções de Assembly, escritas em C, utilizou-se cabeçalhos que encapsularam o código de Assembly emmintrin.h, wmmintrin.h, smmintrin.h e immintrin.h [3]. As instruções utilizadas foram _mm_unpackhi_epi64, _mm_cvtsi128_si64 e _mm_store_si128para manipulaçao de 128 bits; _mm_popcnt_u64 para o cálculo do peso de hamming para uma palavra de 64 bits; _mm_clmulepi64_si128 para a multiplicação sem carry de 64 bits; _mm_xor_si128 para a soma módulo 2 de uma palavra de 128 bits; _mm_slli_si128 para o deslocamento de uma palavra de 128 bits [4]. Essas instruções fazem parte do conjunto de intruções SSE (Streaming SIMD Extensions) da Intel ou 3DNow da AMD, sendo necessários um processador da 3ª geração da Intel (Ivy Bridge) ou equivalente da AMD para que o programa possa ser executado. Utilizou-se o compilador GCC versão 7.2 com as flags -save-temps-O3-mpclmul -mavx -static -march=native habilitadas, que geram um código otimizado e utilizam as instruções citadas anteriormente. Durante este trabalho, utilizou-se um processador i7-4510U, com 8 GB de RAM, no sistema operacional Ubuntu 17.10, utilizando o IDE NetBeans 8.2 para programação em C.

C. *Descrição do algoritmo de decodificação*

A decodificação foi realizada utilizando o algoritmo simplificado apresentado em aula. É importante notar que não necessariamente a síndrome encontrada está no conjunto de síndromes possíveis, mesmo após a realização de todos os giros. Para tal, verifica-se que caso fosse efetuado mais de um giro completo e não fosse identificado alguma síndrome pertencente ao conjunto de síndromes de erro no último bit, o processo da busca das síndromes é interrompido e o quociente calculado anteriormente é utilizado como resposta.

O algoritmo simplificado apresentado em sala consiste de, em primeiramente, dividir o vetor recebido pelo polinômio gerador. Caso o resto fosse zero, o quociente representava a palavra de informação. Caso o resto não fosse zero, ele é a síndrome. Foi realizados giros cíclicos com a síndrome até que esta estivesse dentro do conjunto associado de síndromes conhecidas, caso no qual o valor do último bit é trocado. Dessa forma recalculou-se a síndrome e repetiu-se esses passos até que o seu valor fosse zero. A quantidade de giros cíclicos foi guardado, para que depois, a palavra código fosse girada ciclicamente para a posição original, podendo ser assim, dividida novamente pelo polinômio gerador, sendo o quociente a palavra de informação. É importante notar que enquanto a palavra código foi girada ciclicamente em relação ao polinômio D^{n+1}, a síndrome foi girada ciclicamente com relação ao polinômio gerador. Uma forma eficiente de realizar a realimentação é verificar se o bit mais significativo é '0' ou '1', pois quando o bit é '0' então o primeiro é '0' independente do polinômio gerador, e a rotação da síndrome se torna uma rotação comum. Assim, apenas é necessário realizar as multiplicações (operações AND binária) e as somas (operações XOR binária), quando o último bit é '1'.

D. Apresentação e análise dos dados obtidos.

Gerou-se 1024000 de bits com distribuição uniforme, ou seja, igual probabilidade do resultado ser '0' ou '1', que foi dividido em vetores de tamanho igual a k, que é o tamanho da sequência de informação e que foi escolhido de modo que a taxa é próximo de meio e que o polinômio gerador gera um conjunto de palavras de maior distância mínima.

Após divisão desses bits em vetores, codificou-se esses vetores mensagens da forma descrita anteriormente (no tópico B) utilizando os polinômios geradores da Tabela A, sendo em seguida as mensagens codificadas passadas pelo canal binário simétrico com diferentes valores de probabilidade de erro para esse canal.

Por fim, as mensagens recebidas pelo canal binário simétrico foram decodificadas conforme descrito anteriormente (no tópico C), e comparadas com a mensagem original para contagem dos erros. Assim, a probabilidade de erro mostrada na Figura 1 foi calculada dividindo-se a quantidade de erro contada pela quantidade total de bits (1024000 bits).

Como esperado devido à propriedade dos polinômios geradores encontrados à medida que crescem consertarem mais erros ou por gerarem conjuntos de palavras códigos com distâncias mínimas maiores, observa-se na Figura 1 que a tendência é que os sistemas com polinômios geradores maiores tenham probabilidade de erro menores quando a probabilidade de erro do canal binário simétrico é suficientemente pequeno, ou seja, a partir de certo valor de probabilidade de erro do canal binário simétrico a probabilidade de erro diminui mais rapidamente para os sistemas que possuem os polinômios geradores maiores, sendo que no início, para probabilidade de erro do BSC próximas de 50%, esses sistemas são piores em termos de probabilidade de erro, pois é mais provável que haja mais erros do que sistema

de codificação e decodificação consegue consertar com palavras códigos maiores, por isso é observado essa “barriga para cima” na curva amarela da Figura 1.

Porém, não foi possível realizar esse processo de medição para todos os valores desejados de tamanho de bloco, pois a quantidade de síndromes de erro no bit mais significativo cresce exponencialmente, para tamanhos de bloco igual a 63, tem-se na ordem de centenas de milhões de síndromes para serem armazenadas o que inviabiliza esse processo (Tabela B), pois para realizar a decodificação é necessário armazenar todas as síndromes de erro no bit mais significativo o que ocupa mais memória do que a maioria dos computadores pessoais comerciais disponíveis atualmente. Outra possibilidade seria repassar essa carga da memória para o tempo de processamento, ou seja, em vez de armazenar as síndromes, o algoritmo poderia deixar para, no momento em fosse necessário saber se a síndrome calculada é ou não uma síndrome de erro de último bit, calcular as síndromes de erro bit mais significativo e comparar com a síndrome encontrada, porém isso retarda o processamento da decodificação e o tempo de processamento passa a inviabilizar o processo.

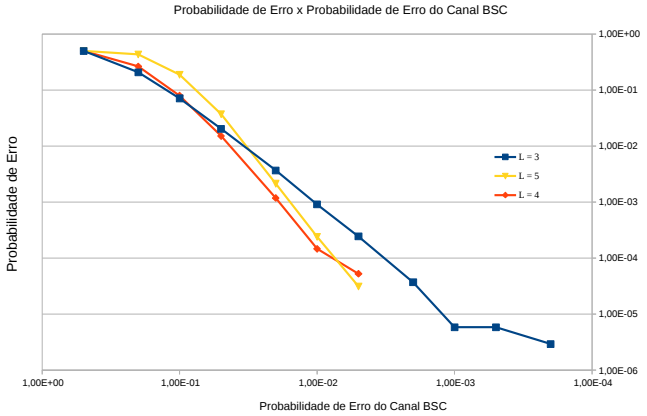

FIGURA 1: PROBABILIDADE DE ERRO OBTIDO EXPERIMENTALMENTE CODIFICANDO E DECODIFICANDO UMA AMOSTRA DE 1024000 DE BITS INTERMEDIADO PELO CANAL BSC.

Dessa forma, como uma forma alternativa de comparação do desempenho dos polinômios geradores encontrados em relação à probabilidade de erro, construiu-se o gráfico da Figura 2 considerando a distribuição binomial da quantidade de erros inseridos na mensagem transmitida pelo canal binário simétrico. Assim, dada a distância mínima do conjunto das palavras códigos pode-se calcular quantos erros são corrigíveis, e a partir disso calcular a probabilidade de que

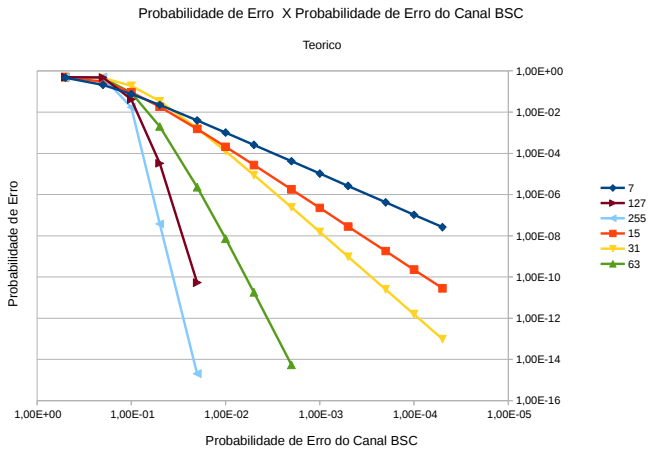

Figura 2: Probabilidade de erro teórico do sistema que utiliza os polinômios geradores da Tabela A, para o tamanho de bloco indicado pela legenda.

ocorra mais erros do que a quantidade de erros corrigíveis na passagem pelo canal BSC considerando a distribuição binomial. Porém, cada uma dessas probabilidades deve ser ponderada pela probabilidade de erro de bit após a decodificação dado que ocorreu essa quantidade de erro no canal.

A probabilidade de erro de bit na palavra de informação dado x erros na palavra codificada foi estimada utilizando um código em C, que a partir da codificação da palavra de informação nula (gerava palavra código com apenas zeros), introduzia x bits de erros e decodificava-a, obtendo, dessa forma, uma palavra de informação que era diferente de zero, caso a quantidade de bits de erros x introduzidos fosse maior do que a quantidade máxima de erros a serem consertados. Dessa forma, o peso de Hamming dessa palavra de informação era a quantidade de bits errados, uma vez que se partiu de uma palavra de informação nula. É importante notar que o fato de ter-se partido de uma palavra nula não modificou o resultado, uma vez que a codificação é linear, a única diferença seria ter que calcular a distância entre a palavra de informação obtida e a palavra de informação original.

Assim, por meio de um conjunto de cerca de 1 bilhão de palavras códigos com erros inseridos e decodificadas, estimou-se a probabilidade de erro de bit para cada valor de quantidades de erros x na palavra código. É importante notar, que pela dificuldade em se decodificar uma palavra código de tamanho de 63 bits ou mais, os erros inseridos, eram palavras códigos cuja decodificação era conhecida. Isto é, gerava-se uma palavra de informação aleatória, codificava-a e acrescentava-se uma quantidade de erros menor do que o máximo corrigível. Assim, utilizando-se uma sequência de erros que fosse igual a essa palavra código gerada anteriormente, o resultado da codificação seria justamente essa

palavra de informação gerada anteriormente. Dessa forma, a decodificação era realizada instantaneamente, dado que já se conhecia a palavra informação utilizada para a geração da palavra código, tratada como bits de erro em uma palavra código toda nula. Um exemplo dessas estimativas está exposto na Tabela 1, cuja codificação utilizava 7 bits.

Tabela 1: Estimativa para os erros nos bits da palavra informação decodificada em função dos erros nos bits para a palavra informação decodificada, utilizando-se uma codificação cíclica de 7 bits.

Erros na palavra código	Média de erros nos bits
1	0
2	1,86
3	2,07
4	2,07
5	2,29
6	3,00
7	3,00

Dessa forma, de posse dos valores estimados para os erros de bits na palavra informação decodificada, utilizou-se a distribuição binomial da quantidade de erros na palavra código, ponderada pela probabilidade de erro no bit na palavra informação, para o cálculo das probabilidades de erros teóricos encontrados na Figura 2. É interessante notar, que para palavras códigos com tamanho maior ou igual a 63 bits, observou-se que a probabilidade de erro de bit dado um erro incorrigível na palavra código ficou muito próxima de 50%, isto é, com valores que pertenciam ao intervalo de 47% até 53%.

III. Conclusão

Este documento apresentou um método de busca de polinômios geradores de códigos cíclicos de forma que a taxa é igual a 50%. Os polinômios encontrados geram um conjunto de códigos com grande distância mínima de forma que a probabilidade de erro de bit é razoavelmente boa, como se observou na Figura 1 e 2. Apesar dos códigos cíclicos terem simplificado a codificação e decodificação e diminuído o consumo de memória, uma vez que não é preciso armazenar uma matriz geradora e uma matriz de paridade, ainda há alguns empecilhos, como ainda é difícil encontrar um bom polinômio gerador, pois basicamente o método de procura é por “força bruta” com algumas restrições de projeto que acabam selecionando ou excluindo algumas opções, mas que para códigos com tamanho de bloco grande (por exemplo, maior que 255), essas restrições não eliminam candidatos suficientes a ponto de que o algoritmo de busca deve ter várias otimizações de hardware e de software para ser possível encontrar um bom candidato em um tempo de processamento razoável, como, por exemplo, uma otimização realizada nesse trabalho foi utilizar um conjunto de instruções específicas em Assembly dos processadores da Intel. Outro inconveniente é que o número de síndromes de erro no bit mais significativo é muito grande para códigos de tamanho grande (por exemplo, para 255, como observado na Tabela B) o que torna a decodificação um processo inviável do ponto de vista de consumo de memória (ou de tempo de processamento, conforme discutido anteriormente).

APÊNDICE

Tabela A: L é a quantidade de fatores primos, n é o tamanho da palavra código, R é a taxa, d_{min} e é a distância minima. O bit menos significativo representa o coeficiente de D^0 , e assim sucessivamente.

L	n	R (%)	d_{min}	g (em hexadecimal)
3	007	57	04	D
5	015	47	05	1D1
7	031	52	08	EE43
13	063	52	18	7A98 9F5B
19	127	50	36	ADCD2DB30F0EA75F
35	255	50	70	F69B2C73A079ADE5 CCAA3ACB13D1A93F

Tabela B: n é o tamanho da palavra código, Ne é quantidade de erros corrigíveis em uma palavra código e Q(s) é a quantidade de síndromes de erro no bit mais significativo.

n	Ne	Q(s)
7	1	1
15	2	15
31	3	466
63	8	560.339.292
127	17	8,324568906554627e+19
255	34	3,518090710711877e+41

O código em C e em Python produzido para este relatório está disponível em: <https://github.com/Gabrui/ele32_lab4>

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Shay Gueron, Michael E. “Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode”. Disponível em: <https://software.intel.com/sites/default/files/managed/72/cc/clmul-wp-rev-2.02-2014-04-20.pdf>.

[2] Vinodh Gopal et. al. “Fast CRC Computation for Generic Polynomials Using PCLMULQDQ Instruction”. Disponível em: <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/fast-crc-computation-generic-polynomials-pclmulqdq-paper.pdf>.

[3] Intel Corporation. “The Intel Intrinsics Guide”. Disponível em: <https://software.intel.com/sites/landingpage/IntrinsicsGuide>

[4] Orion Lawlor. “SSE for High Performance Programming”. Disponível em: <https://www.cs.uaf.edu/courses/cs301/2014-fall/notes/sse-performance/>.

Dylan Nakandakari Sugimoto estudante do 3º ano do curso de engenharia da computação no Instituto Tecnológico de Aeronáutica (ITA), São José dos Campos, SP, Brasil.

Gabriel Adriano de Melo estudante do 3º ano do curso de engenharia da computação no Instituto Tecnológico de Aeronáutica (ITA), São José dos Campos, SP, Brasil.