INSTITUTO
FEDERAL
Santa Catarina

Câmpus
São José

Códigos de bloco

Sistemas de Comunicação II

Arthur Cadore Matuella Barcella

03 de Novembro de 2024

Engenharia de Telecomunicações - IFSC-SJ

Sumário

1. Introdução: **3**
2. Desenvolvimento: **3**
2.1. Questão 1 3
2.1.1. Determine uma matriz geradora G para código. 3
2.1.2. Construa uma tabela mensagem $\rightarrow$ palavra-código. 4
2.1.3. Determine a distância mínima e a distribuição de peso das palavras-código. 5
2.1.4. Determine uma matriz de verificação H para código. 7
2.1.5. Construa uma tabela síndrome $\rightarrow$ padrão de erro. 8
2.1.6. Determine a distribuição de peso dos padrões de erro corrigíveis. 10
2.2. Questão 2 12
3. Conclusão: **12**
4. Referências: **12**

1. Introdução:

2. Desenvolvimento:

2.1. Questão 1

Considere o código de Hamming estendido (8,4), obtido a partir do código de Hamming (7,4) adicionando um "bit de paridade global" no final de cada palavra de código. (Dessa forma, todas as palavras-código terão um número par de bits 1.)

2.1.1. Determine uma matriz geradora G para código.

Para montar uma matriz geradora para o código de Hamming (8,4), partimos da matriz geradora do código de Hamming (7,4). A matriz geradora do código de Hamming (7,4) é dada por:

```python
# Import das bibliotecas do Python
import komm
import numpy as np
import itertools as it
from fractions import Fraction
from itertools import product

# Cria um objeto do código de Hamming (7,4)
hamm74 = komm.HammingCode(3)
(n, k) = (hamm74.length, hamm74.dimension)

# Imprime o código de Hamming (7,4)
print("Código de Hamming (7,4):")
print(n, k)

# Cria e Imprime a matriz geradora G (7,4)
G = hamm74.generator_matrix
print("Matriz geradora G (7,4):")
print(G)
```

$$G_{7,4} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{1}$$

Em seguida, adicionamos uma coluna contendo um bit de paridade global, que é a soma dos bits de dados. Dessa forma, a matriz geradora do código de Hamming (8,4) é dada por:

```python
# Calcula o bit de paridade para cada linha e adiciona à matriz

# Calcula a paridade (soma módulo 2 de cada linha)
parity_column = np.sum(G, axis=1)


# Adiciona a coluna de paridade
G_extended = np.hstack((G, parity_column.reshape(-1, 1)))
```

```python

# Imprime a matriz geradora estendida (8,4)
print("\nMatriz geradora estendida G (8,4):")
print(G_extended)
```

Dessa forma, temos que a matriz geradora estendida G para o código de Hamming (8,4) é dada por:

$$G_{8,4} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \quad (2)$$

2.1.2. Construa uma tabela mensagem → palavra-código.

Para construir a tabela mensagem → palavra-código, basta multiplicar a matriz geradora G pelo vetor de mensagem m. Dessa forma, a tabela mensagem → palavra-código é dada pela seguinte função:

```python
def encode_message(m, G_extended):
    # Converte a mensagem em um array numpy, caso ainda não seja
    m = np.array(m)

    # Multiplica a mensagem pela matriz geradora
    codeword = np.dot(m, G_extended)
    return codeword

# Exemplo de uso, mensagem de 4 bits
m = [1, 0, 1, 1]
codeword = encode_message(m, G_extended)

print("Mensagem de entrada (4 bits):", m)
print("Palavra código gerada (8 bits):", codeword)
```

Para chamar a função é necessário um laço de repetição para todas as mensagens possíveis de 4 bits. Dessa forma, a tabela mensagem → palavra-código é dada por:

```python
# Gera todas as mensagens de 4 bits possíveis
all_messages = list(product([0, 1], repeat=4))  # Gera combinações de 4 bits

# Exibe cada mensagem e sua palavra código correspondente
print("Mensagem (4 bits) -> Palavra código (8 bits)")
for m in all_messages:
    codeword = encode_message(m, G_extended)
    print(f"{m} -> {codeword}")
```

Dessa forma, temos que:

Tabela 1: Elaborada pelo Autor

“Mensagem (4 bits)”	→	**“Palavra código (8 bits)”**
“0 0 0 0”	→	“0 0 0 0 0 0 0 0”
“0 0 0 1”	→	“0 0 0 1 1 1 1 0”
“0 0 1 0”	→	“0 0 1 0 0 1 1 1”
“0 0 1 1”	→	“0 0 1 1 1 0 0 1”
“0 1 0 0”	→	“0 1 0 0 1 0 1 1”
“0 1 0 1”	→	“0 1 0 1 0 1 0 1”
“0 1 1 0”	→	“0 1 1 0 1 1 0 0”
“0 1 1 1”	→	“0 1 1 1 0 0 1 0”
“1 0 0 0”	→	“1 0 0 0 1 1 0 1”
“1 0 0 1”	→	“1 0 0 1 0 0 1 1”
“1 0 1 0”	→	“1 0 1 0 1 0 1 0”
“1 0 1 1”	→	“1 0 1 1 0 1 0 0”
“1 1 0 0”	→	“1 1 0 0 0 1 1 0”
“1 1 0 1”	→	“1 1 0 1 1 0 0 0”
“1 1 1 0”	→	“1 1 1 0 0 0 0 1”
“1 1 1 1”	→	“1 1 1 1 1 1 1 1”

Tabela de resultados da implementação

2.1.3. Determine a distância mínima e a distribuição de peso das palavras-código.

Para calcular a distância mínima e a distribuição de peso das palavras-código, utilizamos a matriz geradora estendida G e a função de distância de Hamming.

Dessa forma, a distância mínima é dada pela menor distância entre todas as palavras-código, enquanto que a distribuição de peso é dada pela quantidade de palavras-código de cada peso. Para isso, inicialmente precisamos calcular todas as palavras-código possíveis com seus respectivos pesos.

```python
# Calcula o peso de Hamming de uma palavra código

# Vetores para armazenar as palavras-código e seus pesos
codewords = []
weights = []

# Calcula o peso de Hamming de cada palavra código
print("Mensagem (4 bits) -> Palavra código (8 bits) -> Peso")
for m in all_messages:
    codeword = encode_message(m, G_extended)
    weight = np.sum(codeword)  # Calcula o peso (número de bits 1)
    codewords.append(codeword)
```

```
    weights.append(weight)
    print(f"{m} -> {codeword} -> {weight}")
```

Dessa forma, temos que:

Tabela 2: Elaborada pelo Autor

“Mensagem (4 bits)”	→	**“Palavra código (8 bits)”**	→	**“Peso”**
“0 0 0 0”	→	“0 0 0 0 0 0 0 0”	→	“0”
“0 0 0 1”	→	“0 0 0 1 1 1 1 0”	→	“4”
“0 0 1 0”	→	“0 0 1 0 0 1 1 1”	→	“4”
“0 0 1 1”	→	“0 0 1 1 1 0 0 1”	→	“4”
“0 1 0 0”	→	“0 1 0 0 1 0 1 1”	→	“4”
“0 1 0 1”	→	“0 1 0 1 0 1 0 1”	→	“4”
“0 1 1 0”	→	“0 1 1 0 1 1 0 0”	→	“4”
“0 1 1 1”	→	“0 1 1 1 0 0 1 0”	→	“4”
“1 0 0 0”	→	“1 0 0 0 1 1 0 1”	→	“4”
“1 0 0 1”	→	“1 0 0 1 0 0 1 1”	→	“4”
“1 0 1 0”	→	“1 0 1 0 1 0 1 0”	→	“4”
“1 0 1 1”	→	“1 0 1 1 0 1 0 0”	→	“4”
“1 1 0 0”	→	“1 1 0 0 0 1 1 0”	→	“4”
“1 1 0 1”	→	“1 1 0 1 1 0 0 0”	→	“4”
“1 1 1 0”	→	“1 1 1 0 0 0 0 1”	→	“4”
“1 1 1 1”	→	“1 1 1 1 1 1 1 1”	→	“8”

Tabela de resultados da implementação

Em seguida, calculamos a distância mínima e a distribuição de peso das palavras-código.

```
# Calcula a distância mínima
def hamming_distance(codeword1, codeword2):
    return np.sum(codeword1 != codeword2)

min_distance = float('inf')

for i in range(len(codewords)):
    for j in range(i + 1, len(codewords)):
        dist = hamming_distance(codewords[i], codewords[j])
        if dist < min_distance:
            min_distance = dist

print("\nDistância mínima entre as palavras-código:", min_distance)

# Distribuição de pesos (Peso varia de 0 a 8)
```

```python
weight_distribution = {i: weights.count(i) for i in range(9)}

print("Distribuição de pesos:")
for weight, count in weight_distribution.items():
    if count > 0:
        print(f"Peso {weight}: {count} palavra(s) código")
```

A distribuição de peso das palavras-código é dada por:

Tabela 3: Elaborada pelo Autor

Peso	"0"	"1"	"2"	"3"	"4"	"5"	"6"	"7"	"8"
Palavras Código	"1"	"0"	"0"	"0"	"14"	"0"	"0"	"0"	"1"

Tabela de resultados da implementação

Quanto a distância mínima, temos que o código de Hamming (8,4) possui distância mínima de 4, visto que a menor distância entre as palavras-código é 4.

2.1.4. Determine uma matriz de verificação H para código.

Para determinar a matriz de verificação H para o código de Hamming (8,4), utilizamos a matriz geradora estendida G e a propriedade de que $H = [I_k \mid G^T]$, onde I_k é a matriz identidade de ordem k.

```python
# Função para gerar a matriz de verificação H
def generate_check_matrix(G):
    k = G.shape[0]  # Número de linhas (k)
    n = G.shape[1]  # Número de colunas (n)

    # A submatriz P é a parte de paridade de G
    P = G[:, k:]  # Colunas correspondentes à parte de paridade
    P_T = P.T  # Transponha P

    # Cria a matriz identidade I_{n-k}
    I_n_minus_k = np.eye(n - k, dtype=int)

    # Combina para formar H
    H = np.hstack((P_T, I_n_minus_k))  # Matriz H: [P^T | I_{n-k}]
    return H

# Gerar a matriz de verificação
H = generate_check_matrix(G_input)

print("Matriz de verificação H (4, 8):")
print(H)
```

Ao inserirmos uma matriz geradora G de dimensões $4, 8$, obtemos a matriz de verificação H para o código de Hamming (8,4):

```python
# Matriz (8,4) geradora do código de Hamming
G_input = np.array([
```

```
            [1, 0, 0, 0, 1, 1, 0, 1],
            [0, 1, 0, 0, 1, 0, 1, 1],
            [0, 0, 1, 0, 0, 1, 1, 1],
            [0, 0, 0, 1, 1, 1, 1, 0]])
```

Dessa forma, temos que a matriz de verificação H para o código de Hamming (8,4) é dada por:

$$G_{\text{input}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} = [\text{I . P}] \rightarrow H = [P^T \mid I] = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad (3)$$

2.1.5. Construa uma tabela síndrome → padrão de erro.

Para construir a tabela síndrome → padrão de erro, utilizamos a matriz de verificação H e a propriedade de que a síndrome é dada por $s = \text{H}e^T$, onde e é o vetor recebido.

```
# Inicializando matriz de padrões de erro
errorMatrix = np.zeros((2**m_lenght, 2**k, n), dtype=int)

# Gerando palavras código aleatórias para cada coluna da primeira linha
for col in range(2**k):
     errorMatrix[0, col] = np.random.randint(2, size=n)  # Gera 0s e 1s aleatórios para cada coluna

# Inicializando a primeira linha com os dados de paridade
errorMatrix[1:9, 0] = np.eye(n)[:8]  # Preenchendo as primeiras 8 linhas da coluna 0 com a matriz identidade

# Gerar padrões de erro para 1 e 2 bits de erro
for row in range(1, 9):
    for col in range(1, 2**k):
        errorMatrix[row, col] = (errorMatrix[0, col] + errorMatrix[row, 0]) % 2

# Criando padrões de erro para 1 e 2 bits
error1 = []
error2 = []

# Calculando padrões de erro para um bit de erro em cada posição
for pos in range(n):
    error_pattern = np.zeros(n, dtype=int)
    error_pattern[pos] = 1
    error1.append(error_pattern)

# Calculando padrões de erro para dois bits de erro em cada posição
positions = it.combinations(range(n), 2)
for pos in positions:
    error_pattern = np.zeros(n, dtype=int)
    error_pattern[list(pos)] = 1
    error2.append(error_pattern)

# Preenchendo a matriz de padrões de erro com padrões de erro de 1 bit
line_index = 0
```

```python
for pattern in error1:
    if line_index < 2**m_lenght:
        errorMatrix[line_index, 0] = pattern
        line_index += 1

# Preenchendo a matriz de padrões de erro com padrões de erro de 2 bits
for pattern in error2:
    if line_index < 2**m_lenght:
        errorMatrix[line_index, 0] = pattern
        line_index += 1

# Inicializando a matriz de pesos
w_matrix = np.zeros((2**m_lenght, 2**k), dtype=int)

# Calculando os pesos para cada padrão de erro
for row in range(0, 2**m_lenght):
    for col in range(0, 2**k):
        w_matrix[row, col] = sum(errorMatrix[row, col])

# Impressão da matriz de padrões de erro
print("Matriz de Padrões de Erro (errorMatrix):")
for i in range(errorMatrix.shape[0]):
    for j in range(errorMatrix.shape[1]):
        print(f"{''.join(map(str, errorMatrix[i, j]))}", end=" ")
    print()
```

Tabela 4: Elaborada pelo Autor

10000000	01010100	10111000	00101001	11010111	00001111	10000110	11001111	10000011	01001000	00110100	11101001	10010001	00100111	10011011	01111010
01000000	11010100	00111000	10101001	01010111	10001111	00000110	01001111	00000011	11001000	10110100	01101001	00010001	10100111	00011011	11111010
00100000	00010100	11111000	01101001	10010111	01001111	11000110	10001111	11000011	00001000	01110100	10101001	11010001	01100111	11011011	00111010
00010000	01110100	10011000	00001001	11110111	00101111	10100110	11101111	10100011	01101000	00010100	11001001	10110001	00000111	10111011	01011010
00001000	01000100	10101000	00111001	11000111	00011111	10010110	11011111	10010011	01011000	00100100	11111001	10000001	00110111	10001011	01101010
00000100	01011100	10110000	00100001	11011111	00000111	10001110	11000111	10001011	01000000	00111100	11100001	10011001	00101111	10010011	01110010
00000010	01010000	10111100	00101101	11010011	00001011	10000010	11001011	10000111	01001100	00110000	11101101	10010101	00100011	10011111	01111110
00000001	01010110	10111010	00101011	11010101	00001101	10000100	11001101	10000001	01001010	00110110	11101011	10010011	00100101	10011001	01111000
11000000	01010101	10111001	00101000	11010110	00001110	10000111	11001110	10000010	01001001	00110101	11101000	10010000	00100110	10011010	01111011
10100000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10010000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10001000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10000100	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10000010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10000001	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
01100000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Tabela de resultados da implementação

Em seguida calculamos a síndrome para cada padrão de erro.

```python
# Calcula a síndrome para cada padrão de erro
syndrome = (H @ errorMatrix[:, 0, :].T) % 2
```

```

# Criação da para as síndromes e padrões de erro
e_s = pd.DataFrame(columns=["syndrome", "error"])
e_s["syndrome"] = ["".join(map(str, s)) for s in syndrome.T]
e_s["error"] = ["".join(map(str, err)) for err in errorMatrix[:, 0, :]]

# Filtrar apenas as entradas únicas
e_s = e_s.drop_duplicates()

# Exibir o DataFrame resultante
print(e_s)
```

Com a tabela síndrome → padrão de erro, temos que o resultado apresentado abaixo:

Tabela 5: Elaborada pelo Autor

index	syndrome	Padrão de erro
0	1101	10000000
1	1011	01000000
2	0111	00100000
3	1110	00010000
4	1000	00001000
5	0100	00000100
6	0010	00000010
7	0001	00000001
8	0110	11000000
9	1010	10100000
10	0011	10010000
11	0101	10001000
12	1001	10000100
13	1111	10000010
14	1100	10000001
15	1100	01100000

Tabela de resultados da implementação

2.1.6. Determine a distribuição de peso dos padrões de erro corrigíveis.

Para determinar a distribuição de peso dos padrões de erro corrigíveis, utilizamos a matriz de verificação H e a propriedade de que um padrão de erro é corrigível se a síndrome correspondente for diferente de zero.

```
# Verificação dos erros em cada bit
print("\nVerificação de Erros em Cada Bit:")
for index, row in e_s.iterrows():
```

```
    syndrome_value = row["syndrome"]
    error_value = row["error"]

    # Verificando onde os erros ocorrem
    error_bits = [i for i in range(len(error_value)) if error_value[i]
== '1']

    if len(error_bits) > 0:  # Se houver um ou mais erros
        print(f"Síndrome: {syndrome_value}, Erros detectados nos bits:
{', '.join(str(bit + 1) for bit in error_bits)}")
```

Dessa forma, temos o seguinte resultado para a tabela de sindrome apresentada anteriormente:

Tabela 6: Elaborada pelo Autor

syndrome	**Erros Detectados**
0000	bits:
1101	bits: 1
1011	bits: 2
0111	bits: 3
1110	bits: 4
1000	bits: 5
0100	bits: 6
0010	bits: 7
0001	bits: 8
0110	bits: 1, 2
1010	bits: 1, 3
0011	bits: 1, 4
0101	bits: 1, 5
1001	bits: 1, 6
1111	bits: 1, 7
1100	bits: 2, 3

Tabela de resultados da implementação

Realizando a contagem dos bits corrigíveis, temos que a distribuição de peso dos padrões de erro corrigíveis é dada por:

Tabela 7: Elaborada pelo Autor

Peso	**Quantidade de Padrões de Erro Corrigíveis**
0	1
1	8
2	7

Tabela de resultados da implementação

2.2. Questão 2

Escreva um programa que simule o desempenho de BER de um sistema de comunicação que utiliza o código de Hamming (8, 4) com decodificação via síndrome, modulação QPSK (com mapeamento Gray) e canal AWGN. Considere a transmissão de 100000 palavras-código e relação sinal-ruído de bit $\left(\frac{Eb}{N0}\right)$ variando de -1 a 7 dB, com passo de 1 dB. Compare com o caso não-codificado.

3. Conclusão:

4. Referências: