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
2.2.1. Implementação do código 12
2.2.2. Resultados 13
3. Conclusão: **14**
4. Referências: **14**

1. Introdução:

O objetivo deste relatório é explorar o código de Hamming e suas propriedades, incluindo a construção de uma matriz geradora, a determinação da distância mínima e da distribuição de peso das palavras-código, a construção de uma matriz de verificação, a construção de uma tabela mensagem → palavra-código, a construção de uma tabela síndrome → padrão de erro, a determinação da distribuição de peso dos padrões de erro corrigíveis, e a implementação de uma simulação de desempenho de BER de um sistema de comunicação que utiliza o código de Hamming (8, 4) com decodificação via síndrome, modulação QPSK e canal AWGN.

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)
```

$$G_{7,4} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \quad (1)$$

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)))

# 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:

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

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)”	$\rightarrow$	**“Palavra código (8 bits)”**
“0 0 0 0”	$\rightarrow$	“0 0 0 0 0 0 0 0”
“0 0 0 1”	$\rightarrow$	“0 0 0 1 1 1 1 0”
“0 0 1 0”	$\rightarrow$	“0 0 1 0 0 1 1 1”
“0 0 1 1”	$\rightarrow$	“0 0 1 1 1 0 0 1”
“0 1 0 0”	$\rightarrow$	“0 1 0 0 1 0 1 1”
“0 1 0 1”	$\rightarrow$	“0 1 0 1 0 1 0 1”
“0 1 1 0”	$\rightarrow$	“0 1 1 0 1 1 0 0”
“0 1 1 1”	$\rightarrow$	“0 1 1 1 0 0 1 0”
“1 0 0 0”	$\rightarrow$	“1 0 0 0 1 1 0 1”
“1 0 0 1”	$\rightarrow$	“1 0 0 1 0 0 1 1”
“1 0 1 0”	$\rightarrow$	“1 0 1 0 1 0 1 0”
“1 0 1 1”	$\rightarrow$	“1 0 1 1 0 1 0 0”
“1 1 0 0”	$\rightarrow$	“1 1 0 0 0 1 1 0”
“1 1 0 1”	$\rightarrow$	“1 1 0 1 1 0 0 0”
“1 1 1 0”	$\rightarrow$	“1 1 1 0 0 0 0 1”
“1 1 1 1”	$\rightarrow$	“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.

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

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

```

```python
# 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.

```python
# 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)
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.

```
# 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):

```
# 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:

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

2.1.5. Construa uma tabela síndrome $\rightarrow$ padrão de erro.

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

```
# Inicializando matriz de padrões de erro com 16 linhas
errorMatrix = np.zeros((16, 2**k, n), dtype=int)

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

# Preenchendo a primeira coluna com a matriz identidade para padrões de erro de 1 bit
for i in range(1, min(9, 16)):
    errorMatrix[i, 0] = np.eye(n, dtype=int)[i-1]

# Gerando padrões de erro para 1 bit
for row in range(1, 9):  # Para as linhas 1 a 8
    for col in range(1, 2**k):  # Para todas as colunas
        errorMatrix[row, col] = (errorMatrix[0, col] + errorMatrix[row, 0]) % 2

# Gerando padrões de erro para 2 bits
two_bit_errors = list(it.combinations(range(n), 2))

# Contador para controlar o número de padrões de dois bits
two_bit_count = 0

# Laço para terminar com as linhas da tabela (16 linhas no total)
for pos in two_bit_errors:
    if 9 + two_bit_count >= 16:
        break

    error_pattern = np.zeros(n, dtype=int)
    error_pattern[list(pos)] = 1
```

```python
    errorMatrix[9 + two_bit_count, 0] = error_pattern

    # Aplicando a combinação de dois bits de erro na matriz
    for col in range(1, 2**k):
        errorMatrix[9 + two_bit_count, col] = (errorMatrix[0, col] + errorMatrix[9 + two_bit_count, 0]) % 2

    two_bit_count += 1  # Incrementa o contador de padrões de dois bits

# Preencher as linhas restantes com padrões aleatórios, se necessário
for i in range(9 + two_bit_count, 16):
    for col in range(2**k):
        errorMatrix[i, col] = np.random.randint(2, size=n)  # Gera 0s e 1s aleatórios

# Inicializando a matriz de pesos
w_matrix = np.zeros((16, 2**k), dtype=int)

# Calculando os pesos para cada padrão de erro
for row in range(16):
    for col in range(2**k):
        w_matrix[row, col] = sum(errorMatrix[row, col])

# Impressão da matriz de padrões de erro
print("Matriz de Padrões de Erro (ap):")
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

00000000	**10111010**	**01001100**	**00000110**	**10110010**	**00011000**	**10000000**	**00000110**	**01011011**	**11110111**	**00001010**	**10000010**	**10101001**	**01000101**	**01101001**	**00110101**
10000000	00111010	11001100	10000110	00110010	10011000	00000000	10000110	11011011	01110111	10001010	00000010	00101001	11000101	11101001	10110101
01000000	11111010	00001100	01000110	11110010	01011000	11000000	01000110	00011011	10110111	01001010	11000010	11101001	00000101	00101001	01110101
00100000	10011010	01101100	00100110	10010010	00111000	10100000	00100110	01111011	11010111	00101010	10100010	10001001	01100101	01001001	00010101
00010000	10101010	01011100	00010110	10100010	00001000	10010000	00010110	01001011	11100111	00011010	10010010	10111001	01010101	01111001	00100101
00001000	10110010	01000100	00001110	10111010	00010000	10001000	00001110	01010011	11111111	00000010	10001010	10100001	01001101	01100001	00111101
00000100	10111110	01001000	00000010	10110110	00011100	10000100	00000010	01011111	11110011	00001110	10000110	10101101	01000001	01101101	00110001
00000010	10111000	01001110	00000100	10110000	00011010	10000010	00000100	01011001	11110101	00001000	10000000	10101011	01000111	01101011	00110111
00000001	10111011	01001101	00000111	10110011	00011001	10000001	00000111	01011010	11110110	00001011	10000011	10101000	01000100	01101000	00110100
11000000	01111010	10001100	11000110	01110010	11011000	01000000	11000110	10011011	00110111	11001010	01000010	01101001	10000101	10101001	11110101
10100000	00011010	11101100	10100110	00010010	10111000	00100000	10100110	11111011	01010111	10101010	00100010	00001001	11100101	11001001	10010101
10010000	00101010	11011100	10010110	00100010	10001000	00010000	10010110	11001011	01100111	10011010	00010010	00111001	11010101	11111001	10100101
10001000	00110010	11000100	10001110	00111010	10010000	00001000	10001110	11010011	01111111	10000010	00001010	00100001	11001101	11100001	10111101
10000100	00111110	11001000	10000010	00110110	10011100	00000100	10000010	11011111	01110011	10001110	00000110	00101101	11000001	11101101	10110001
10000010	00111000	11001110	10000100	00110000	10011010	00000010	10000100	11011001	01110101	10001000	00000000	00101011	11000111	11101011	10110111
10000001	00111011	11001101	10000111	00110011	10011001	00000001	10000111	11011010	01110110	10001011	00000011	00101000	11000100	11101000	10110100

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 $\rightarrow$ padrão de erro, temos que o resultado apresentado abaixo:

Tabela 5: Elaborada pelo Autor

index	**syndrome**	**Padrão de erro**
0	0000	00000000
1	1101	10000000
2	1011	01000000
3	0111	00100000
4	1110	00010000
5	1000	00001000
6	0100	00000100
7	0010	00000010
8	0001	00000001
9	0110	11000000
10	1010	10100000
11	0011	10010000
12	0101	10001000
13	1001	10000100
14	1111	10000010
15	1100	10000001

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.

```python
# Verificação dos erros em cada bit
```

```python
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.

2.2.1. Implementação do código

Para implementar a simulação do 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, utilizamos a biblioteca `komm` para gerar o código de Hamming (8, 4) e a modulação QPSK.

```
# Criação de um dicionário para mapear as síndromes para os padrões de erro
syndrome_error = {tuple(s): e for s, e in zip(e_s["syndrome"], e_s["error"])}


# Criando um vetor de 100 mensagens de 4 bits aleatórias
u = np.random.randint(0, 2, (1000, 4))

# Codificação das mensagens de 4 bits em palavras código de 8 bits usando a matriz geradora estendida
v = (u @ G_input) % 2

# Inicializando o modulador QPSK e o canal AWGN
qpsk_mod = komm.PSKModulation(4)


# Inicializando o modulador QPSK e o canal AWGN para o código de Hamming
SNR = range(-1, 8)  # dB

# Inicializando os vetores de BER para o código de Hamming e sem código
ber = np.zeros(len(SNR))
ber_hamm = np.zeros(len(SNR))


# Loop para calcular a BER para cada valor de SNR
for i, snr in enumerate(SNR):

    # calculando o valor de dBm de volta para lienar
```

```
    snr_lin = 10 ** (snr / 10)

    # Inicializando o canal AWGN e aplicando a SNR em linear
    awgn = komm.AWGNChannel(signal_power="measured", snr=snr_lin)

    # Modulação, transmissão, recepção e demodulação para o código de Hamming
    v_mod = qpsk_mod.modulate(v.flatten())
    vb = awgn(v_mod)
    v_demod = qpsk_mod.demodulate(vb).reshape(-1, 8)

    # Decodificação do código de Hamming
    s = ((H @ v_demod.T) % 2).T

    # Calculando a palavra código corrigida para cada síndrome usando a tabela de síndromes e padrões de erro
    errors = np.array([syndrome_error[tuple(x)] for x in s])
    v_hat = (v_demod + errors) % 2

    # Calculando a BER para o código de Hamming
    ber_hamm[i] = np.sum(v_hat.reshape(-1) != v.reshape(-1)) / 1000

    # Modulação, transmissão, recepção e demodulação sem código de Hamming
    u_mod = qpsk_mod.modulate(u.flatten())
    ub = awgn(u_mod)
    u_hat = qpsk_mod.demodulate(ub).reshape(-1, 4)

    # Calculando a BER sem código de Hamming
    ber[i] = np.sum(u_hat.reshape(-1) != u.reshape(-1)) / 1000
```

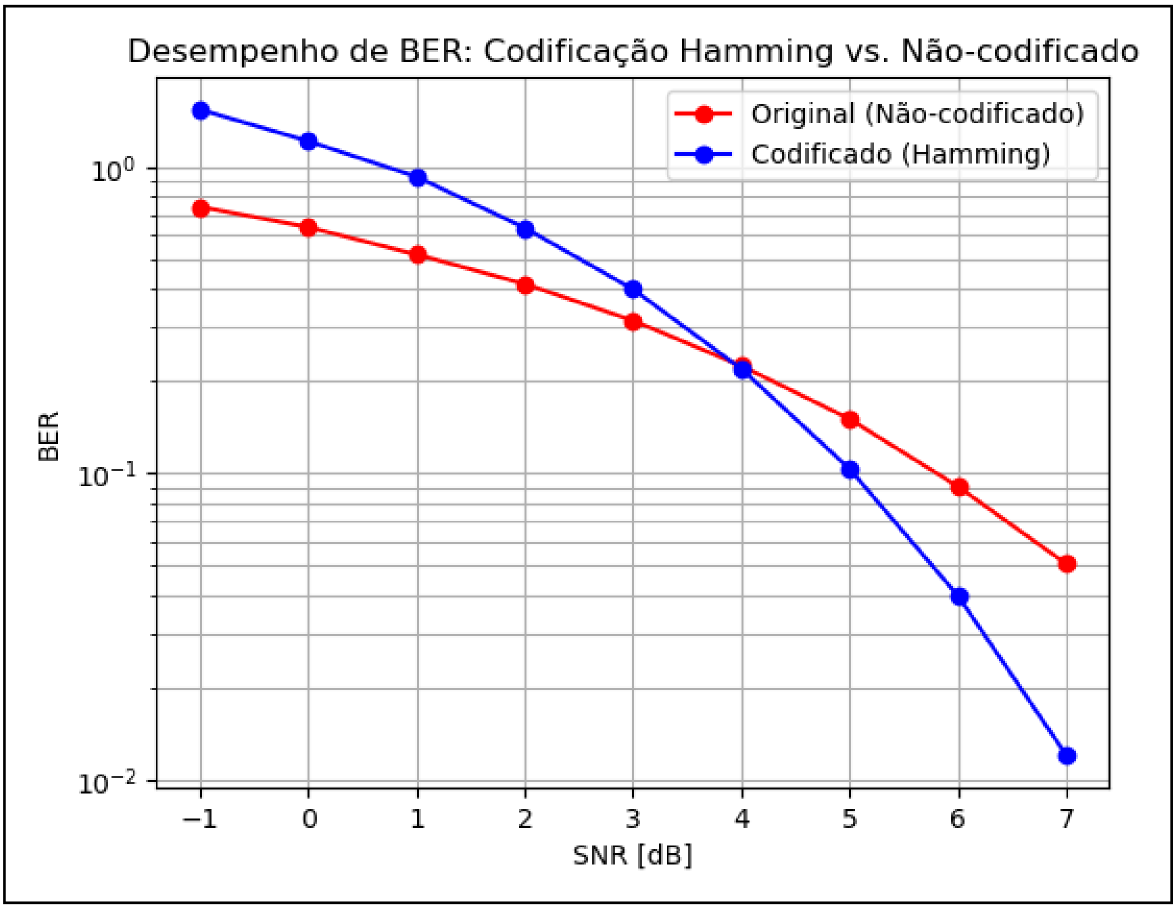
2.2.2. Resultados

Para verificar o resultado podemos realizar o plot da BER para o código de Hamming e sem código.

```
# Plotando BER original e codificado
plt.figure()

plt.semilogy(SNR, ber, label="Original (Não-codificado)", marker="o", color="red")
plt.semilogy(SNR, ber_hamm, label="Codificado (Hamming)",marker="o", color="blue")

plt.xlabel("SNR [dB]")
plt.Figure(figsize=(10, 6))
plt.ylabel("BER")
plt.title("Desempenho de BER: Codificação Hamming vs. Não-codificado")

plt.legend()
plt.grid(True, which="both")  # Grid em ambas escalas
plt.show()
```

Dessa forma, temos que o gráfico de desempenho de BER para o código de Hamming (8, 4) e sem código é apresentado abaixo:

Figura 1: Elaborada pelo Autor



3. Conclusão:

A partir dos conceitos vistos e resultados obtidos anteriormente, podemos concluir que o código de Hamming (8, 4) é capaz de corrigir um único erro e detectar até dois erros. Além disso, a implementação do código de Hamming (8, 4) em um sistema de comunicação, juntamente com a modulação QPSK e o canal AWGN, mostrou uma melhoria significativa no desempenho de BER em comparação com o sistema sem codificação. Portanto, o código de Hamming (8, 4) é uma técnica eficaz para melhorar a confiabilidade da comunicação em sistemas de comunicação digital.

4. Referências:

Códigos de Bloco - R. W. Nobrega