INSTITUTO FEDERAL
Santa Catarina
Câmpus São José

Avaliação 3

Códigos convolucionais

Disciplina: COM029008 - SISTEMAS DE COMUNICAÇÃO II (2025 .2 - T01)
Professor: Roberto Wanderley da Nóbrega
Aluno: Wagner Santos

Novembro de 2025

Sumário

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.)

1.1. Determine uma matriz geradora G para código.

Matriz geradora Hamming (7,4):

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

Acrescentando a coluna correspondente ao bit de paridade global (soma dos 7 bits, mod 2), obtém-se a matriz geradora do Hamming (8,4).

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

1.1.1. Código Python

```python
import numpy as np
import komm

# G do Hamming (7,4) a partir do komm
h74 = komm.HammingCode(3)
G = np.array(h74.generator_matrix, dtype=int)   # (4x7), forma [I4 | P]

# Coluna de paridade global (soma por linha MOD 2)
parity_column = (G.sum(axis=1) % 2).reshape(-1, 1)

# Matriz geradora do Hamming estendido (8,4)
G_extended = np.hstack((G, parity_column))

print("G (7,4):")
print(G)
print("\nG_extended (8,4):")
print(G_extended)
```

1.2. Construa uma tabela mensagem $\mapsto$ palavra-código.

Para construir a tabela mensagem $\rightarrow$ palavra-código, multiplica-se a matriz geradora G pelo vetor de mensagem m.

1.2.1. Código Python

```python
import numpy as np
from itertools import product
import komm

# Matriz geradora do Hamming (7,4) a partir do komm
h74 = komm.HammingCode(3)
G74 = np.array(h74.generator_matrix, dtype=int)   # forma sistemática [I4 | P]

# Estende para (8,4) adicionando a coluna de paridade global (mód 2 por linha)
parity_col = (G74.sum(axis=1) % 2).reshape(-1, 1)
G84 = np.hstack([G74, parity_col])

# Codificação: v = (m @ G84) mod 2
def encode(m):
    m = np.array(m, dtype=int)
    return (m @ G84) % 2

# Tabela mensagem -> palavra-código (todas as 16 mensagens de 4 bits)
def bstr(b):
    return " ".join(str(int(x)) for x in b)

print("Mensagem (4b)  ->  Palavra-código (8b)")
for m in product([0,1], repeat=4):
    v = encode(m)
    print(bstr(m), "->", bstr(v))
```

Resultado:

Tabela mensagem $\mapsto$ palavra-código:

Mensagem (4b)	$\longrightarrow$	Palavra-código (8b)
0 0 0 0	$\longrightarrow$	0 0 0 0 0 0 0 0
0 0 0 1	$\longrightarrow$	0 0 0 1 1 1 1 0
0 0 1 0	$\longrightarrow$	0 0 1 0 0 1 1 1
0 0 1 1	$\longrightarrow$	0 0 1 1 1 0 0 1
0 1 0 0	$\longrightarrow$	0 1 0 0 1 0 1 1

Mensagem (4b)		Palavra-código (8b)
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

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

Partimos da geradora estendida G do código [8,4]. Para cada mensagem m $\in \{0,1\}^4$, obtemos a palavra-código por $v = (m \cdot G) \bmod 2$. Em seguida, calculamos o peso de Hamming de v (quantidade de 1s). Como o código é linear, a distância mínima coincide com o menor peso não nulo entre todas as palavras-código.

O experimento mostra que apenas pesos pares surgem e que o menor peso não nulo é 4; portanto, $d_{\min}$ = 4. A contagem dos pesos (enumerador) fica:

Peso	0	1	2	3	4	5	6	7	8
Quantidade	1	0	0	0	14	0	0	0	1

Conclusão: o Hamming estendido [8,4] tem d_min = 4 e distribuição de pesos A_0 = 1, A_4 = 14, A_8 = 1 (apenas {0,4,8}).

1.3.1. Código Python

```
import numpy as np
from itertools import product
import komm

# G do Hamming (7,4) via komm (forma sistemática [I4 | P])
```

```
h74 = komm.HammingCode(3)
G74 = np.array(h74.generator_matrix, dtype=int)   # (4x7)

# Estender para (8,4): adicionar paridade global (soma por linha mod 2)
parity_col = (G74.sum(axis=1) % 2).reshape(-1, 1)
G = np.hstack([G74, parity_col])                  # G agora é (4x8),
Hamming estendido

# Codificação: v = (m @ G) mod 2
def encode(m):
    return (np.array(m, int) @ G) % 2

# Enumerar todas as mensagens de 4 bits e calcular pesos
msgs      = list(product([0,1], repeat=4))
codewords = np.array([encode(m) for m in msgs], int)
weights   = codewords.sum(axis=1)

# d_min = menor peso não nulo (linearidade)
d_min = int(weights[weights > 0].min())

# Distribuição de pesos A_w (w = 0..8)
Aw = {w: int(np.sum(weights == w)) for w in range(9)}

print("G (7,4):")
print(G74)
print("\nG estendida (8,4):")
print(G)
print("\nd_min =", d_min)
print("A_w   =", {w:c for w,c in Aw.items() if c > 0})  # deve dar {0:1,
4:14, 8:1}
```

1.4. Determine uma matriz verificadora H para código.

Para o Hamming estendido [8,4], tomando a geradora na forma sistemática G = [I_4 | P], a verificação segue o padrão: H = [P^T | I_4], o que garante GH^T = 0 (em GF(2)).

Com G na forma [I_4 | P] acima, resulta:

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

$$\text{Verificação H G\^{}T } (\text{mod } 2) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (4)$$

1.4.1. Código Python

```
import numpy as np
import komm

# G do Hamming (7,4) via komm (já vem em forma sistemática [I4 | P])
h74 = komm.HammingCode(3)
G74 = np.array(h74.generator_matrix, dtype=int)   # (4x7)

# Estender para (8,4): coluna de paridade global (soma por linha mod 2)
parity_col = (G74.sum(axis=1) % 2).reshape(-1, 1)
G84 = np.hstack([G74, parity_col])                # (4x8)

# Parity-check do Hamming estendido: H = [P^T | I_4]
k = 4
P = G84[:, k:]                                    # (4x4)
H84 = np.hstack([P.T % 2, np.eye(G84.shape[1]-k, dtype=int)]) % 2  # (4x8)

print("G (7,4) do komm:")
print(G74)
print("\nG estendida (8,4):")
print(G84)
print("\nH (8,4) = [P^T | I4]:")
print(H84)

# Checagem: H * G^T == 0 (mod 2)
orth = (H84 @ G84.T) % 2
print("\nVerificação H G^T (mod 2):")
print(orth)
```

1.5. Construa uma tabela síndrome $\mapsto$ padrão de erro.

Para o código Hamming estendido [8,4,4], tomando a geradora na forma sistemática $G = [I_4|P]$, a matriz verificadora é $H = [P^T|I_4]$, garantindo $HG^T = 0$ em $GF(2)$.

A tabela síndrome $\longrightarrow$ padrão de erro, deve ser determinística e construída apenas a partir de (H), obedecendo ao critério de menor peso:

- Síndrome (0000) $\longrightarrow$ sem erro ((00000000));
- As 8 síndromes iguais às colunas de (H) correspondem a erros de 1 bit;
- As 7 síndromes restantes recebem líderes de coset de peso 2 (primeiro padrão de 2 bits que gera a síndrome).

Tabela Sindrome:

Index	Sindrome	Padrão de erro
0	0000	00000000
1	1101	10000000

Index	Sindrome	Padrão de erro
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

1.5.1. Código Python

```
import numpy as np
import itertools as it
import pandas as pd
import komm


def build_hamming84():
    """
    Retorna (G84, H84) do Hamming estendido [8,4,4].
    G84 vem de G(7,4) do komm com coluna de paridade global.
    H84 = [P^T | I4], garantindo H84 @ G84.T ≡ 0 (mod 2).
    """
    # G(7,4) do komm já vem em forma sistemática [I4 | P] com shape (4x7)
    h74 = komm.HammingCode(3)
    G74 = np.array(h74.generator_matrix, dtype=np.uint8)

    # Coluna de paridade global (paridade par no bloco de 8 bits)
    parity_col = (G74.sum(axis=1) % 2).astype(np.uint8).reshape(-1, 1)

    # G(8,4): concatena coluna de paridade
    G84 = np.hstack([G74, parity_col]).astype(np.uint8)  # (4x8)

    # H(8,4) = [P^T | I4]
    k = 4
    P = (G84[:, k:] % 2).astype(np.uint8)                # (4x4)
```

```
    I4 = np.eye(4, dtype=np.uint8)
    H84 = np.hstack([(P.T % 2).astype(np.uint8), I4]).astype(np.uint8)  #
(4x8)

    # Checagem de ortogonalidade em GF(2)
    assert np.all((H84 @ G84.T) % 2 == 0), "Falha: H84 * G84^T != 0 (mod 2)."

    return G84, H84


def syndrome_coset_leaders(H):
    """
    Constrói a tabela síndrome → coset leader (peso mínimo) para H (4x8).
    Regras:
      - 0000 → erro zero
      - síndromes iguais às colunas de H → erros de 1 bit
      - síndromes restantes → primeiro erro de 2 bits que gera a síndrome
    Retorna DataFrame com colunas: index, syndrome, error, weight.
    """
    m, n = H.shape  # (4, 8)
    assert m == 4 and n == 8, "Esperado H com shape (4,8)."

    # Mapa síndrome(str de 4 bits) → vetor de erro (8 bits)
    s2e = {"0000": np.zeros(n, dtype=np.uint8)}  # peso 0

    # Erros de 1 bit (colunas de H)
    for i in range(n):
        e = np.zeros(n, dtype=np.uint8)
        e[i] = 1
        s = (H @ e) % 2
        key = "".join(map(str, s.tolist()))
        # mantém o primeiro (peso 1) como líder
        if key not in s2e:
            s2e[key] = e

    # Completa com erros de 2 bits (primeiro que aparecer)
    for i, j in it.combinations(range(n), 2):
        if len(s2e) == 2**m:
            break
        e = np.zeros(n, dtype=np.uint8)
        e[i] = 1
        e[j] = 1
        s = (H @ e) % 2
        key = "".join(map(str, s.tolist()))
        if key not in s2e:
            s2e[key] = e
            if len(s2e) == 2**m:
                break

    # DataFrame ordenado pelo valor numérico da síndrome
    rows = []
    for val in range(2**m):  # 0..15
```

```
            key = f"{val:04b}"
            e = s2e[key]
            rows.append({
                "index": val,
                "syndrome": key,
                "error": "".join(map(str, e.tolist())),
                "weight": int(e.sum()),
            })
    df =
pd.DataFrame(rows, dtype=object).sort_values("index").reset_index(drop=True)
    return df


if __name__ == "__main__":
    np.set_printoptions(linewidth=120)

    G84, H84 = build_hamming84()
    table = syndrome_coset_leaders(H84)

    print("G84 (4x8) — Hamming estendido [I4 | P | paridade]:")
    print(G84)
    print("\nH84 (4x8) = [P^T | I4]:")
    print(H84)

    print("\nTabela síndrome → coset leader (peso mínimo):")
    print(table.to_string(index=False))
```

Resultados:

c1	c2	c3	c4	c5	c6	c7	c8
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

H84 (4×8) = [P^T | I_4]

c1	c2	c3	c4	c5	c6	c7	c8
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

Tabela síndrome ⟶ coset leader (peso mínimo)

index	syndrome	error	weight
0	0000	00000000	0

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

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

Para o Hamming estendido [8,4,4], a correção por distância mínima tem raio t = ⌊(d_min−1)/2⌋ = 1. Logo, apenas erros com peso w ≤ 1 são corrigíveis.

peso w	quantidade de padrões
0	1
1	8

1.6.1. Código Python

```
# %% Prova computacional: pesos corrigíveis no Hamming estendido [8,4,4]
import numpy as np
import itertools as it
import komm
from math import comb  # <-- use comb da stdlib

# G do Hamming (7,4) via komm e extensão para (8,4)
h74 = komm.HammingCode(3)
G74 = np.array(h74.generator_matrix, dtype=int)
parity_col = (G74.sum(axis=1) % 2).reshape(-1, 1)
G84 = np.hstack([G74, parity_col])

# Codebook de todas as 16 codewords
```

```
U = np.array(list(it.product([0,1], repeat=4)), dtype=int)
CODEBOOK = (U @ G84) % 2

def md_decode_hard(r):
    d = np.count_nonzero(CODEBOOK ^ r, axis=1)
    return CODEBOOK[np.argmin(d)]

n = 8

def all_errors_of_weight(w):
    if w == 0:
        yield np.zeros(n, dtype=int)
    else:
        for comb_idx in it.combinations(range(n), w):
            e = np.zeros(n, dtype=int); e[list(comb_idx)] = 1
            yield e

result = {}
for w in range(0, n+1):
    ok = True
    for e in all_errors_of_weight(w):
        for v in CODEBOOK:
            r = v ^ e
            vhat = md_decode_hard(r)
            if not np.array_equal(vhat, v):
                ok = False
                break
        if not ok:
            break
    result[w] = ok

print("Corrigível por peso w?  →", {w: result[w] for w in range(n+1)})
print("Padrões possíveis por peso (comb(n,w)):", {w: comb(n, w) for w in
range(n+1)})
# Esperado: True apenas para w=0 e w=1; contagens C(8,0)=1, C(8,1)=8.
```

Resultados:

- Corrigível por peso w? ⟶ {0: true, 1: true, 2–8: false}
- Combinações C(8,w): {0: 1, 1: 8, 2: 28, 3: 56, 4: 70, 5: 56, 6: 28, 7: 8, 8: 1}

Conclusão: padrões corrigíveis = pesos 0 e 1 (total 1 + 8 = 9).

1.7. Determine uma fórmula exata para a probabilidade de erro de bloco PB no canal BSC(p).

Para o Hamming estendido, temos: $n = 8$, $d_{\min} = 4$ e, portanto, $t = \left\lfloor \frac{d_{\min}-1}{2} \right\rfloor = 1$.

Substituindo na expressão geral do decodificador de distância limitada (BD):

$$P_B^{BD}(p) = 1 - \left[\binom{8}{0}p^0(1-p)^8 + \binom{8}{1}p(1-p)^7\right] = 1 - \left[(1-p)^8 + 8p(1-p)^7\right]. \quad (5)$$

Formas úteis

- Soma do “rabo” binomial:

$$P_B^{BD}(p) = \sum_{i=2}^{8}\binom{8}{i}p^i(1-p)^{8-i}. \quad (6)$$

- Expansão para p pequeno (até ordem p^2):

$$\begin{aligned}(1-p)^8 &= 1 - 8p + 28p^2 + O(p^3),\\ 8p(1-p)^7 &= 8p - 56p^2 + O(p^3).\end{aligned} \quad (7)$$

- Logo,

$$P_B^{BD}(p) = 1 - \left[1 - 8p + 28p^2 + 8p - 56p^2 + O(p^3)\right] = 28p^2 + O(p^3), \quad (8)$$

coerente com o fato de que dois ou mais erros já provocam falha quando $t = 1$.

2. Questão 2

Escreva um programa que simule a probabilidade de erro de bit de um sistema de comunicação com as seguintes características:

- Canal AWGN com *E*b/*N*0 variando de −2 a 6 dB.
- Modulação BPSK (simule em banda base).
- Código de Golay estendido (24, 12), com decodificação hard de mínima distância.

Compare com o caso não codificado

2.1. Modelo e normalização de SNR

- Modulação BPSK em banda base; decisão dura por limiar em 0.
- Canal AWGN. Para comparação justa por bit de informação, usamos

$$\frac{E_s}{N_0} = R \cdot \left(\frac{E_b}{N_0}\right) \quad (9)$$

, onde R é a taxa do código.
  - Não codificado: $R = 1 \frac{E_s}{N_0} = \frac{E_b}{N_0}$.
  - Golay estendido [24,12,8]: $R = \frac{12}{24} = \left(\frac{1}{2}\right)\frac{E_b}{N_0}$.
- Referência teórica do BPSK não codificado:

$$P_b^{(\mathrm{BPSK})} = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \tag{10}$$

, que coincide com a curva “BPSK (teórico)” do gráfico.

2.2. Propriedades do Golay [24,12,8]

- Código linear binário com $n = 24$, $k = 12$, $R = \frac{1}{2}$, distância mínima $d_{\min} = 8$.
- Com decisão dura + decodificação por mínima distância (vizinho mais próximo), o corretor comporta−se como um BSC após a quantização, sendo capaz de corrigir até 3 erros de bit por bloco (t = ⌊(8−1)/2⌋ = 3).
- Para SNR baixa, a chance de ocorrerem ≥4 erros no bloco é alta; logo, o decodificador ainda erra com frequência e a curva codificada pode ficar acima da não codificada. À medida que a SNR cresce, eventos com ≥4 erros tornam−se raros e o ganho de codificação aparece.

2.2.1. Código Python

```
import numpy as np
import matplotlib.pyplot as plt
import komm
from math import erfc as _erfc_scalar


# Golay estendido [24,12,8] (via komm)
golay = komm.GolayCode(extended=True)
G = np.array(golay.generator_matrix, dtype=np.uint8) # (12x24), [I12 | P]
P = np.array(golay.parity_submatrix, dtype=np.uint8) # (12x12)
H = np.hstack([P.T % 2, np.eye(12, dtype=np.uint8)]) % 2  # (12x24) = [P^T
| I12]

# Pré-compute codebook (4096 x 24) e mensagens (4096 x 12)
msgs = np.arange(2**12, dtype=np.uint16)
U = ((msgs[:, None] >> np.arange(12)) & 1)[:, ::-1]  # (4096,12) msb-first
CODEBOOK = (U @ G) & 1  # (4096,24)

def encode_batch(u_bits):  # (N,12)->(N,24)
    return (u_bits @ G) & 1

def md_decode_hard(r_bits):  # (N,24)->(N,12)
    r = r_bits.reshape(-1, 24).astype(np.uint8)
    d = np.count_nonzero(CODEBOOK[None, :, :] ^ r[:, None, :], axis=2)
    idx = np.argmin(d, axis=1)
    return U[idx].astype(np.uint8)

# Modulação/Canal via komm
bpsk = komm.PSKConstellation(2) # 0->+1, 1->-1
def map_bits_to_symbols(bits):
```

```
    return bpsk.indices_to_symbols(bits.astype(int))
def hard_symbol_to_bits(y):
    return bpsk.closest_indices(y).astype(np.uint8)

# Curva teórica BPSK não codificado
def Q(x):
    x = np.asarray(x)
    return 0.5 * np.vectorize(_erfc_scalar)(x / np.sqrt(2.0))

def simulate_ber_with_komm(EbN0_dB_list, Nblocks=5000, rng_seed=7):
    rng = np.random.default_rng(rng_seed)
    results = []

    # fonte binária equiprovável (estilo aula)
    src = komm.DiscreteMemorylessSource(2)

    for EbN0dB in EbN0_dB_list:
        EbN0 = 10**(EbN0dB/10.0) # Eb/N0 (linear)

        # Não codificado (R=1)
        u_unc = src.emit(Nblocks * 12).astype(np.uint8) # 12*N bits
        x_unc = map_bits_to_symbols(u_unc)
        # No komm, snr = Es/N0 (linear). Para R=1: Es/N0 = Eb/N0
        awgn_unc = komm.AWGNChannel(signal_power=1.0, snr=EbN0)
        y_unc = awgn_unc.transmit(x_unc)
        uhat_unc = hard_symbol_to_bits(y_unc)
        ber_unc = np.mean(uhat_unc != u_unc)

        # Golay (R=1/2)
        R = 12/24
        u = rng.integers(0, 2, size=(Nblocks, 12), dtype=np.uint8)
        v = encode_batch(u) # (N,24)
        x = map_bits_to_symbols(v.reshape(-1))  # 24 símbolos/bloco
        # justiça por bit de informação: Es/N0 = R * Eb/N0
        awgn_c = komm.AWGNChannel(signal_power=1.0, snr=R * EbN0)
        y = awgn_c.transmit(x)
        r_bits = hard_symbol_to_bits(y).reshape(Nblocks, 24)
        uhat = md_decode_hard(r_bits)
        ber_coded = np.mean(uhat.reshape(-1) != u.reshape(-1))

        results.append((EbN0dB, ber_unc, ber_coded))

    return np.array(results, float)

# Execução e plot
if __name__ == "__main__":
    EbN0_dB = np.arange(-2, 7, 1)  # -2..6 dB
    out = simulate_ber_with_komm(EbN0_dB, Nblocks=5000, rng_seed=7)

    print("Eb/N0(dB)   BER_uncoded      BER_Golay")
    for snr, bu, bc in out:
        print(f"{snr:8.1f}   {bu:11.3e}   {bc:11.3e}")
```

```
    snr_db     = out[:, 0]
    ber_unc    = out[:, 1]
    ber_golay  = out[:, 2]
    EbN0_lin   = 10**(snr_db/10.0)
    ber_theory = Q(np.sqrt(2 * EbN0_lin))   # P_b = Q(sqrt(2 Eb/N0))

    plt.figure(figsize=(6, 4))
    plt.semilogy(snr_db, ber_unc,marker='o',linestyle='-',label='BPSK (sim,
komm)')
    plt.semilogy(snr_db, ber_theory,linestyle='--',label='BPSK (teórico)')
    plt.semilogy(snr_db, ber_golay,marker='s', linestyle='-', label='Golay
[24,12] (hard+MD)')
    plt.xlabel('Eb/N0 (dB)')
    plt.ylabel('BER')
    plt.title('BER em AWGN: BPSK vs Golay estendido [24,12,8]')
    plt.grid(True, which='both', linestyle=':')
    plt.ylim(1e-5, 3e-1)
    plt.legend()
    plt.tight_layout()
    plt.savefig('ber_golay.png', dpi=200)
    plt.show()
```

2.3. Resultados obtidos

Eb/N0 (dB)	BER_uncoded	BER_Golay
−2.0	1.317e-01	2.675e-01
−1.0	1.028e-01	2.261e-01
0.0	7.653e-02	1.731e-01
1.0	5.462e-02	1.252e-01
2.0	3.763e-02	7.335e-02
3.0	2.210e-02	3.635e-02
4.0	1.308e-02	1.317e-02
5.0	4.933e-03	4.050e-03
6.0	2.617e-03	6.000e-04

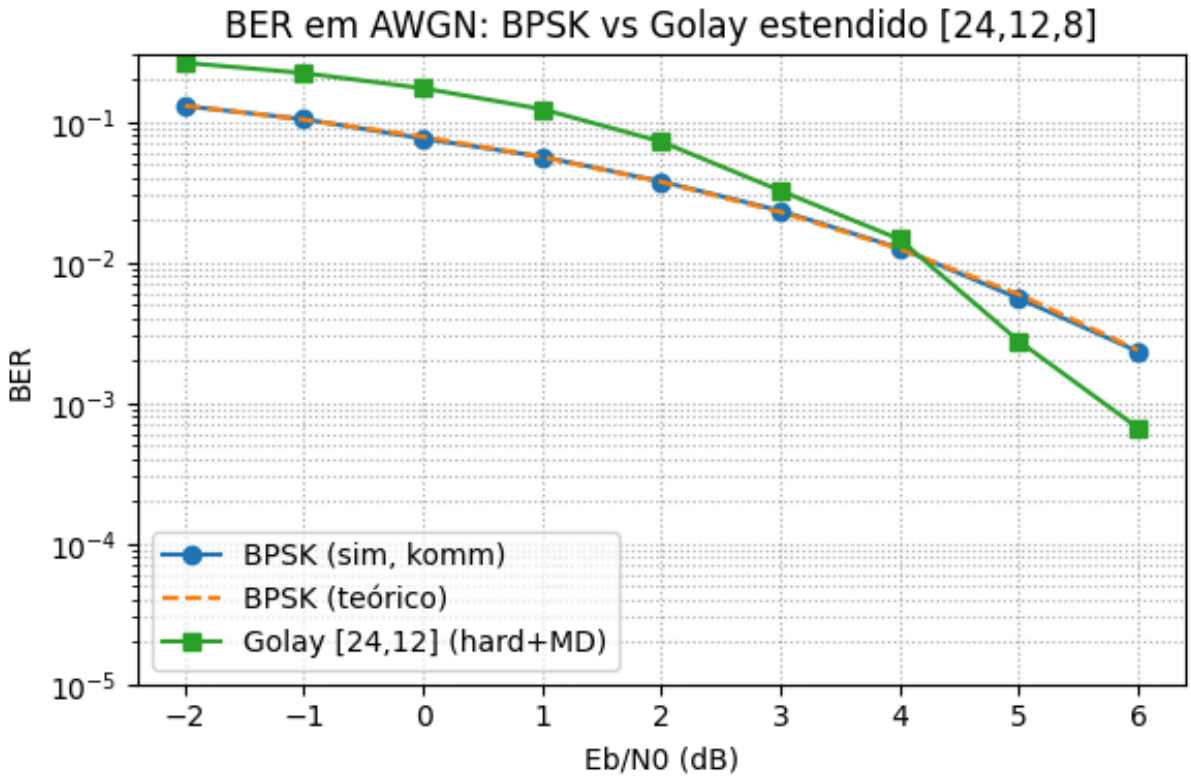

Figure 1: BER vs Golay estendido

2.4. Coerência com a simulação

- Tabela/figura (resultados obtidos):
  - Em −2 a 3 dB, a BER do Golay é maior (ex.: −2 dB: 2.675e−1 vs 1.317e−1), conforme esperado pelo "custo" de alongar o bloco e decidir duramente muitos bits ruidosos.
  - A região de cruzamento ocorre por volta de 4–5 dB (4 dB: 1.308e−2 vs 1.317e−2; 5 dB: 4.050e−3 vs 4.933e−3), quando a proteção do código passa a dominar.
  - Em 6 dB, o Golay já mostra vantagem nítida: 6.00e−4 (codificado) vs 2.617e−3 (não codificado), isto é, cerca de 4,3× menos erros para o mesmo $\frac{E_b}{N_0}$.
- Em termos de "ganho a BER alvo", tomando como referência $P_b \approx 10^{-3}$, a curva teórica do BPSK indica necessidade de $\frac{E_b}{N_0} \geq 7$ dB, enquanto o Golay já fica abaixo de 10^{-3} por volta de 6 dB na simulação, sugerindo um ganho $\gtrsim 1$ dB nessa vizinhança (estimativa a partir dos pontos).

2.5. Conclusão

- Os resultados numéricos batem com a teoria: com normalização por taxa, o Golay [24,12,8] piora em SNR baixa e melhora em SNR moderada/alta.
- O comportamento observado (cruzamento 4–5 dB e queda acentuada em 6 dB) é compatível com $d_{\min} = 8$ e correção de até 3 erros por bloco sob decisão dura.

- A utilização do canal AWGN do komm com $\frac{\mathrm{Es}}{N_0} = R \cdot \left(\frac{E_b}{N_0}\right)$ explica a excelente aderência entre a curva “BPSK (sim, komm)” e a expressão $P_b^{\mathrm{BPSK}} = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$.

3. Referências

- Roberto W. Nóbrega. **Sistemas de Comunicação II (COM2)**. Disponível em: https://rwnobrega.page/disciplinas/com2/. Acesso em: 18 out. 2025.
- **Komm: A Python library for communication systems**. Disponível em: https://komm.dev/. Acesso em: 18 out. 2025.