Códigos convolucionais

Sistemas de Comunicação II

Arthur Cadore Matuella Barcella

08 de Dezembro de 2024

Engenharia de Telecomunicações - IFSC-SJ

Sumário

1. Introdução

Os códigos convolucionais são um tipo de código de correção de erro que são amplamente utilizados em sistemas de comunicação digital. Eles são caracterizados por serem códigos de bloco, onde a saída do codificador é uma função dos bits de informação e dos bits anteriores. Dessa forma, os códigos convolucionais são capazes de corrigir erros de transmissão, melhorando a confiabilidade do sistema de comunicação.

2. Desenvolvimento

2.1. Questão 1

Considere o código convolucional com matrizes geradoras dadas por:

$$G_0 = [1 \ \ 1 \ \ 1] \tag{1}$$

$$G_0 = [1 \ \ 1 \ \ 0] \tag{2}$$

$$G_0 = [0 \ \ 1 \ \ 1] \tag{3}$$

Determine:

2.1.1. Determine a taxa e a ordem de memória do código convolucional.

Para determinar a taxa do código convolucional, é necessário calcular a razão entre o número de bits de informação e o número de bits na saída do codificador de bits. Dessa forma, a formula para a taxa do código convolucional é dada por:

$$R = \frac{k}{n} \tag{4}$$

Onde:
- k é o número de bits de informação
- n é o número de bits na saída do codificador de bits
- R é a taxa do código convolucional

Dessa forma, para o cenário proposto, temos que:

$$R = \frac{1}{3} \tag{5}$$

Já para a ordem de memória do código convolucional, seu valor é dado pelo maior atraso de propagação de um bit de informação no codificador. Dessa forma, no pior caso, teremos um atraso de 2 bits, ou seja, a ordem de memória do código convolucional é 2. O código abaixo realiza o cálculo da taxa e da ordem de memória do código convolucional:

```
# Definindo as matrizes geradoras
G0 = np.array([1, 1, 1])
G1 = np.array([1, 1, 0])
```

```
G2 = np.array([0, 1, 1])

# Definindo os parâmetros do código
k = 1
n = 3

# Calculando a taxa e a ordem de memória
memory_order = len(G0) - 1
rate = k / n

print(f"Taxa: {rate}")
print(f"Ordem de Memória: {memory_order}")
```

2.1.2. Esboce o diagrama de blocos do codificador

Para esboçar o diagrama de blocos do codificador, primeiramente é necessário determinar a saída v_t do codificador para cada estado do código. Dessa forma, temos que:

$$v_t = u_t.G_0 + u_{\{t-1\}}.G_1 + u_{\{t-2\}}.G_2 \tag{6}$$

Onde:

- u_t é a saída do codificador no instante t
- $u_{\{t-1\}}$ é a saída do codificador no instante $t-1$
- $u_{\{t-2\}}$ é a saída do codificador no instante $t-2$

Dessa forma, para o cenário proposto, temos que:

$$v_t = u_t.[1\ \ 1\ \ 1] + u_{\{t-1\}}.[1\ \ 1\ \ 0] + u_{\{t-2\}}.[0\ \ 1\ \ 1] \tag{7}$$

$$v_t = [u_t\ \ u_t\ \ u_t] + [u_{\{t-1\}}\ \ u_{\{t-1\}}\ \ 0] + [0\ \ u_{\{t-2\}}\ \ u_{\{t-2\}}] \tag{8}$$

$$v_t = [u_t + u_{\{t-1\}}\ \ u_t + u_{\{t-1\}} + u_{\{t-2\}}\ \ u_t + u_{\{t-2\}}] \tag{9}$$

Dessa forma, temos que:

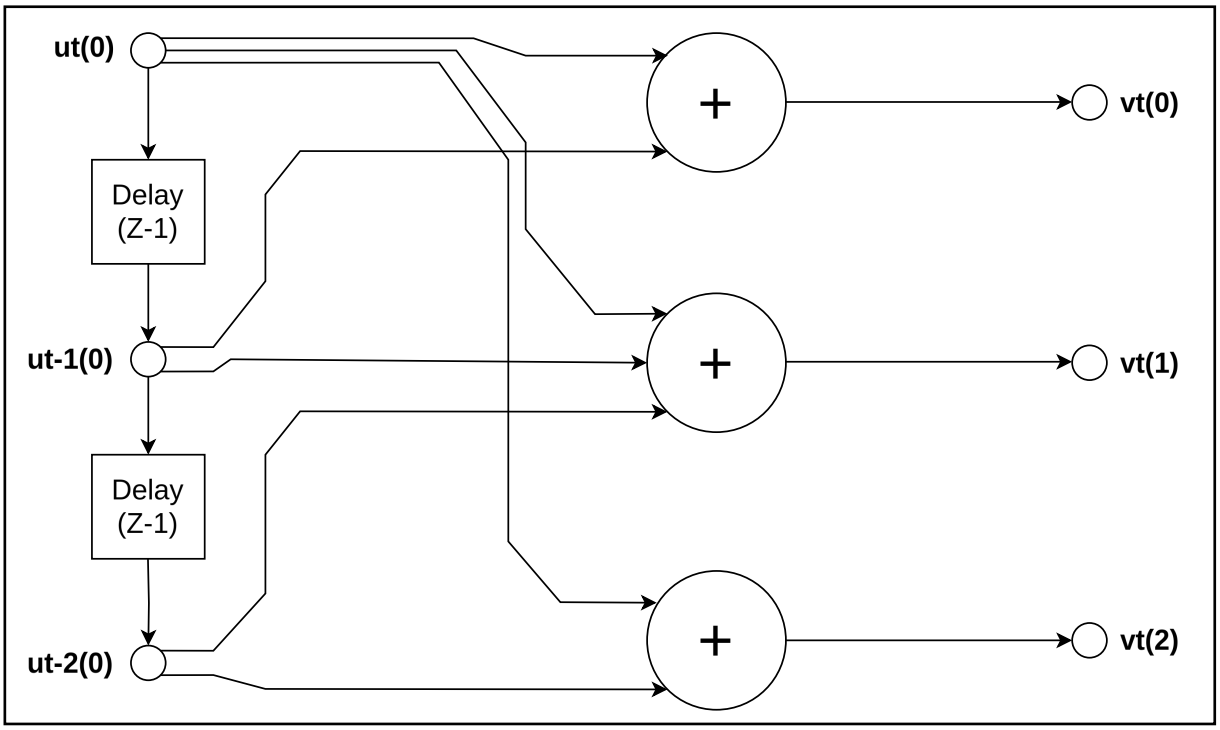
$$v_{t(0)} = u_t + u_{\{t-1\}} \tag{10}$$

$$v_{t(1)} = u_t + u_{\{t-1\}} + u_{\{t-2\}} \tag{11}$$

$$v_{t(2)} = u_t + u_{\{t-2\}} \tag{12}$$

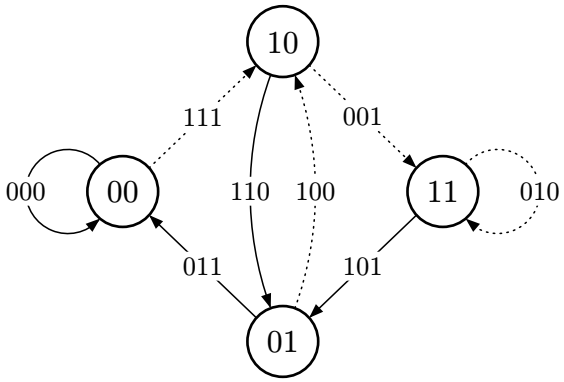
Portanto, o diagrama de blocos do codificador é dado por:

Figura 1: Elaborada pelo Autor



2.1.3. Esboce o diagrama de estados do código

Abaixo é apresentado o diagrama de estados do código convolucional proposto:



2.1.4. Determine os parâmetros (n', k') do código de bloco resultante.

Considere na questão:

- $h = 5$ blocos
- Terminação no estado zero.

Para determinar os parâmetros (n', k') do código de bloco resultante, é necessário calcular os valores de n' e k' através da relação:

$$n' = n.h \tag{13}$$

$$k' = k.h \tag{14}$$

Portanto, dado o valor de $h = 5$, temos que:

$$n' = 3.5 = 15 \tag{15}$$

$$k' = 1.5 = 5 \tag{16}$$

Dessa forma, os parâmetros (n', k') do código de bloco resultante são $n' = 15$ e $k' = 5$. O código abaixo realiza o cálculo dos parâmetros (n', k') do código de bloco resultante:

```
# Definindo o parâmetro h
h = 5


# Calculando os valores de n' e k'
n_line = n * h
k_line = k * h

# Imprimindo os valores de n' e k'
print(f"n': {n_line}")
print(f"k': {k_line}")
```

2.1.5. Codifique a mensagem 11101. Insira a cauda apropriada.

Nota: De acordo com o diagrama de estado do código exposto anteriormente, a maior distância entre o estado 00 e qualquer outro estado é de 2, ou seja, é necessário adicionar 2 bits de cauda para garantir que o código de bloco seja decodificado corretamente.

Inicialmente precisamos definir a mensagem que será codificada. Para isso, temos que a mensagem é dada pela função abaixo. A mesma irá receber a mensagem original, o valor de h e o array de polinômios geradores do código convolucional (estes representados em binário).

```
# Função para codificar a mensagem com o código convolucional
def encode_message(h, polinomial_array, message):
    # Criando o codificador convolucional
    conv_encoder = komm.ConvolutionalCode(
        feedforward_polynomials=polinomial_array)

    # Criando o código de bloco com terminação em zero
    block_code = komm.TerminatedConvolutionalCode(
        conv_encoder, h, mode='zero-termination')

    # Criando o codificador de bloco
    encoder = komm.BlockEncoder(
        block_code)

    # Codificando a mensagem
    encoded_message = encoder(message)

    return encoded_message
```

O mesmo processo é realizado para decodificar a mensagem. Para isso, temos que a mensagem é dada pela função abaixo. A mesma irá receber a matriz geradora do código convolucional, o valor de h e o array de polinômios geradores do código convolucional (estes representados em binário).

```

```

```python
# Função para decodificar a mensagem com o código convolucional
def decode_message(h, polinomial_array, encoded_message):
    # Criando o codificador convolucional
    conv_encoder = komm.ConvolutionalCode(
        feedforward_polynomials=polinomial_array)

    # Criando o código de bloco com terminação em zero
    block_code = komm.TerminatedConvolutionalCode(
        conv_encoder, h, mode='zero-termination')

    # Criando o decodificador de bloco
    decoder = komm.BlockDecoder(
        block_code, method='viterbi_hard')

    # Decodificando a mensagem
    decoded_message = decoder(encoded_message)

    return decoded_message
```

Para executar as funções, primeiramente é necessário definir a mensagem que será codificada, neste exemplo a mensagem é dada por 11101. Em seguida, a mensagem é codificada e decodificada, nos três passos é impresso o resultado obtido.

```python
# (5) Codifique a mensagem (1, 1, 1, 0, 1) usando o código convolucional e o código de bloco resultante.

# Definindo a mensagem
message = [1, 1, 1, 0, 1]
print("Mensagem Original:", ''.join(map(str, message)))

# Criando um array com os polinômios geradores do código convolucional
# É necessário que os polinômios sejam representados em binário apra que a biblioteca funcione corretamente.
polinomial_array = [[0b011 ,0b111, 0b101]]

# Codificando a mensagem e imprimindo o resultado
encoded_message = encode_message(h, polinomial_array, message)
print("Mensagem Codificada:", ''.join(map(str, encoded_message)))

# Decodificando a mensagem e imprimindo o resultado
decoded_message = decode_message(h, polinomial_array, encoded_message)
print("Mensagem Decodificada:", ''.join(map(str, decoded_message)))
```

Abaixo está o resultado obtido após a execução do código acima, note que a mensagem original, e a mensagem decodificada são iguais, o que indica que o processo de codificação e decodificação foi realizado corretamente.

```
Mensagem Original: 11101
Mensagem Codificada: 111001010101100110011
Mensagem Decodificada: 11101
```

A mensagem codificada acima também pode ser obtida seguindo o diagrama de estados do código código convolucional, neste caso a sequencia de estados seria a seguinte:

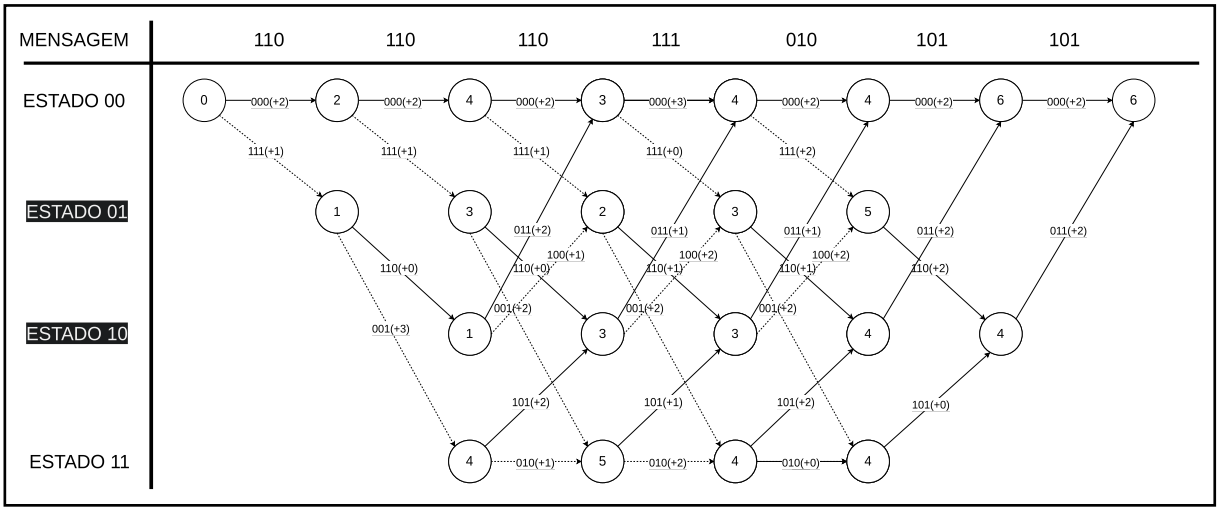
```
00 -> 10 -> 11 -> 01 -> 10 -> 01 -> 00
```

2.1.6. Decodifique a palavra recebida (x) utilizando o algoritmo de Viterbi.

Mensagem x: 11011011011101010101

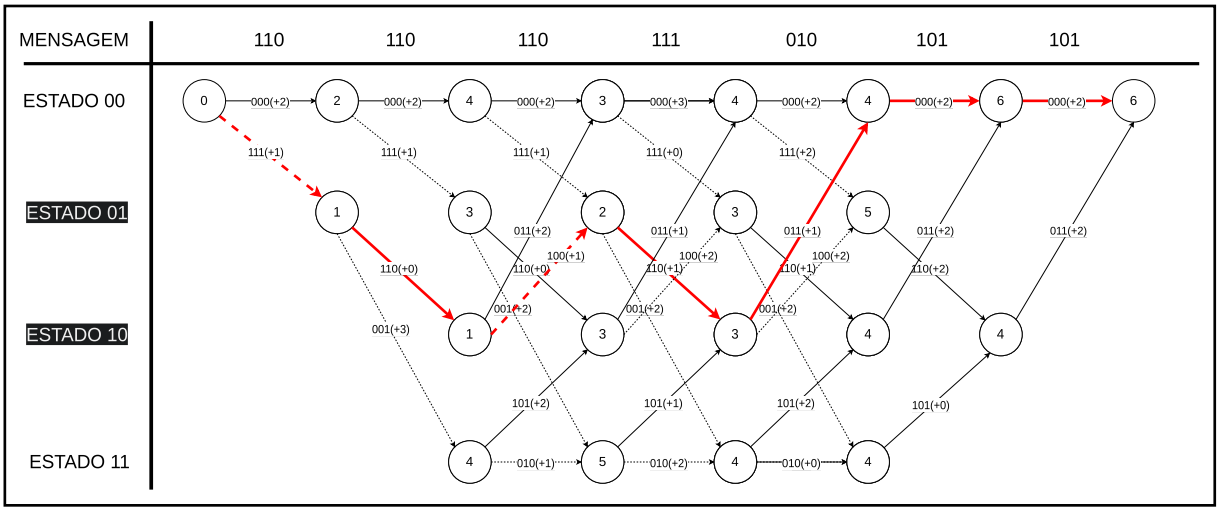
Inicialmente devemos montar a treliça correspondente ao diagrama de estados para a palavra recebida. A trelça é apresentada abaixo, para cada mudança de estado é apresentado o valor da mensagem recebida e um valor adicional de erro (+x) que representa a distância entre a mensagem recebida e a mensagem esperada.

Figura 2: Elaborada pelo Autor



Após analizar a treliça, é possivel determinar o melhor caminho para a decodificação da mensagem. A ilustração abaixo apresenta o caminho escolhido para a decodificação da mensagem.

Figura 3: Elaborada pelo Autor



$$\text{palavra-recebida} = 110 - 110 - 110 - 111 - 010 - 101 - 101 \tag{17}$$

$$\text{palavra-viterbi} = 111 - 110 - 100 - 110 - 011 - 000 - 000 \tag{18}$$

$$\text{estados} = 00 \rightarrow 01 \rightarrow 10 \rightarrow 01 \rightarrow 10 \rightarrow 00 \rightarrow 00 \rightarrow 00 \tag{19}$$

$$\text{mensagem-com-cauda} = 1011100 \tag{20}$$

$$\text{mensagem-decodificada} = 10111 \tag{21}$$

Para verificar o resultado obtido, o código abaixo realiza a decodificação da mensagem recebida utilizando o algoritmo de Viterbi.

```
# (7) Decodifique a palavra recebida 11011011011101010101 utilizando o algoritmo de Viterbi.

# Definindo a palavra recebida
received_word = [1,1,0,1,1,0,1,1,0,1,1,1,0,1,0,1,0,1,1,0,1]
print("Palavra Recebida:", ''.join(map(str, received_word)))

# Decodificando a palavra recebida e imprimindo o resultado
# Nota: Aqui é utilizado a mesma função de decodificação criada anteriormente e também seus parâmetros (exceto a mensagem)
decoded_message = decode_message(h, polinomial_array, received_word)
print("Mensagem Decodificada:", ''.join(map(str, decoded_message)))
```

Abaixo está o resultado obtido após a execução do código acima, note que a mensagem decodificada é igual a 10111, o que indica que o processo de decodificação foi realizado corretamente.

```
Palavra Recebida: 11011011011101010101
Mensagem Decodificada: 10111
```

2.1.7. Determine a distância livre do código através do método de Mason.

Dado o sistema linear, temos que:

$$\begin{cases} s_1 = D^3x + Ds_2 \\ s_2 = D^2s_1 + D^2s_3 \\ s_3 = Ds_1 + Ds_3 \\ y = D^2s_2 \end{cases} \tag{22}$$

Isolando s_1 na primeira equação, temos que:

$$s_3 = (1 - D) = Ds_1 \rightarrow s_1 = \frac{s_3}{1 - D} \tag{23}$$

Em seguida, substituindo s_1 na segunda equação, temos que:

$$s_2 = D^2s_3 + D^2\left[\frac{s_3(1 - D)}{D}\right] \tag{24}$$

Dessa forma, isolando y na última equação, temos que:

$$y = D^2\left[D^2s_3 + D^2\left[\frac{s_3(1 - D)}{D}\right]\right] \tag{25}$$

Isolando a função em x temos que:

$$x = s_3 \frac{1 - D^3}{D^4} \tag{26}$$

$$\frac{y}{x} = \frac{D^3 s_3}{s_3 \frac{1-D^3}{D^4}} = \frac{D^3 D^4 s_3}{1 - D^3 s_3} = \frac{D^7}{1 - D^3} \tag{27}$$

Portanto, a distância livre do código é dada por $\frac{D^7}{1-D^3}$.

2.2. Questão 2

Escreva um programa que simule o desempenho de BER de um sistema de comunicação que utiliza o código convolucional mostrado na figura abaixo com decodificação via algoritmo de Viterbi, modulação QPSK (com mapeamento Gray) e canal AWGN.

Considere a transmissão de 1000 quadros, cada qual contendo $h = 200$ blocos de informação e relação sinal-ruído de bit $\left(\frac{E_b}{N_0}\right)$ variando de -1 a 7 dB com passo 1 dB. Compare com o caso não codificado.

2.2.1. Definição dos parâmetros

Inicialmente para simular o desempenho de BER de um sistema de comunicação que utiliza o código convolucional, é necessário definir os parâmetros do sistema.

```python
# Importando as bibliotecas necessárias
import numpy as np
import komm as komm
import matplotlib.pyplot as plt

# Imprimindo a versão da biblioteca
print("Komm version: ", komm.__version__)
print("Numpy version: ", np.__version__)
```

Dessa forma, os parâmetros do sistema são definidos conforme o código abaixo:

```python
# Quantidade de blocos:
h = 200

# Polinômios geradores
generator_matrices = [[0b1001111, 0b1101101]]

# Ordem da modulação PSK
M = 4

# Número de quadros
Nframes = 1000

# Faixa de valores de Eb/N0, com passo 1 dB
# EBN0_Range = np.arange(-1, 7, 1)

# NOTA: Neste ponto foi utilizado um passo de 0.5 dB para melhorar a resolução do gráfico
EBN0_Range = np.arange(-1, 10, 0.5)
```

2.2.2. Codificador e Decodificador

Em seguida, é necessário definir o codificador e decodificador convolucionais para o sistema, além de instânciar o modulador PSK. Neste caso, redefini as variaveis `modulator`, `conv_encoder`, `block_code`, `encoder` e `decoder` para que possam ser utilizadas no laço de simulação diretamente, pois caso utiliza-se as funções criadas anteriormente, o tempo de execução do código seria muito mais alto.

```
# Modulação PSK
modulator = komm.PSKModulation(M, labeling='reflected')

# Codificador convolucional
conv_encoder = komm.ConvolutionalCode(
    feedforward_polynomials=generator_matrices)

# Código de bloco com terminação em zero
block_code = komm.TerminatedConvolutionalCode(
    conv_encoder, h, mode='zero-termination')

# Criando instância de codificador e decodificador de bloco
encoder = komm.BlockEncoder(block_code)
decoder = komm.BlockDecoder(block_code, method='viterbi_hard')
```

2.2.3. Laço de simulação

Uma vez com os parâmetros do sistema prontos, foi criado dois vetores de BER, um para a transmissão sem codificação, e outro para a transmissão com codificação convolucional.

Em seguida, foi criado um laço para a simulação varrer a faixa de valores de $\frac{E_b}{N_0}$ e calcular a BER para cada correspondente valor. A simulação é realizada gerando um vetor de bits aleatórios, codificando a mensagem e a modulando, adicionando um ruido (simulando o canal de comunicação) e em seguida, realizando o processo de demodulação / decodificação da mensagem. Ao final, a BER e calculada para os dois sinais, o codificado e o não codificado.

```
# Inicializando o vetor de BER para transmissão sem codificação
BER_original = np.zeros(len(EBN0_Range))
# Inicializando o vetor de BER para o código convolucional
BER_conv = np.zeros(len(EBN0_Range))

# Loop para cada valor de Eb/N0
for i, SNR in enumerate(EBN0_Range):
    # Criando um vetor de bits aleatórios
    bits = np.random.randint(0, 2, h * Nframes)

    # Codificação da mensagem
    encoded_bits = encoder(bits)

    # Modulação da mensagens codificada e original
    modulated_bits = modulator.modulate(encoded_bits)
    modulated_bits_original = modulator.modulate(bits)

    # Calculando a SNR
    snr = 10 ** (SNR / 10)
    awgn = komm.AWGNChannel(snr=snr, signal_power="measured")
```

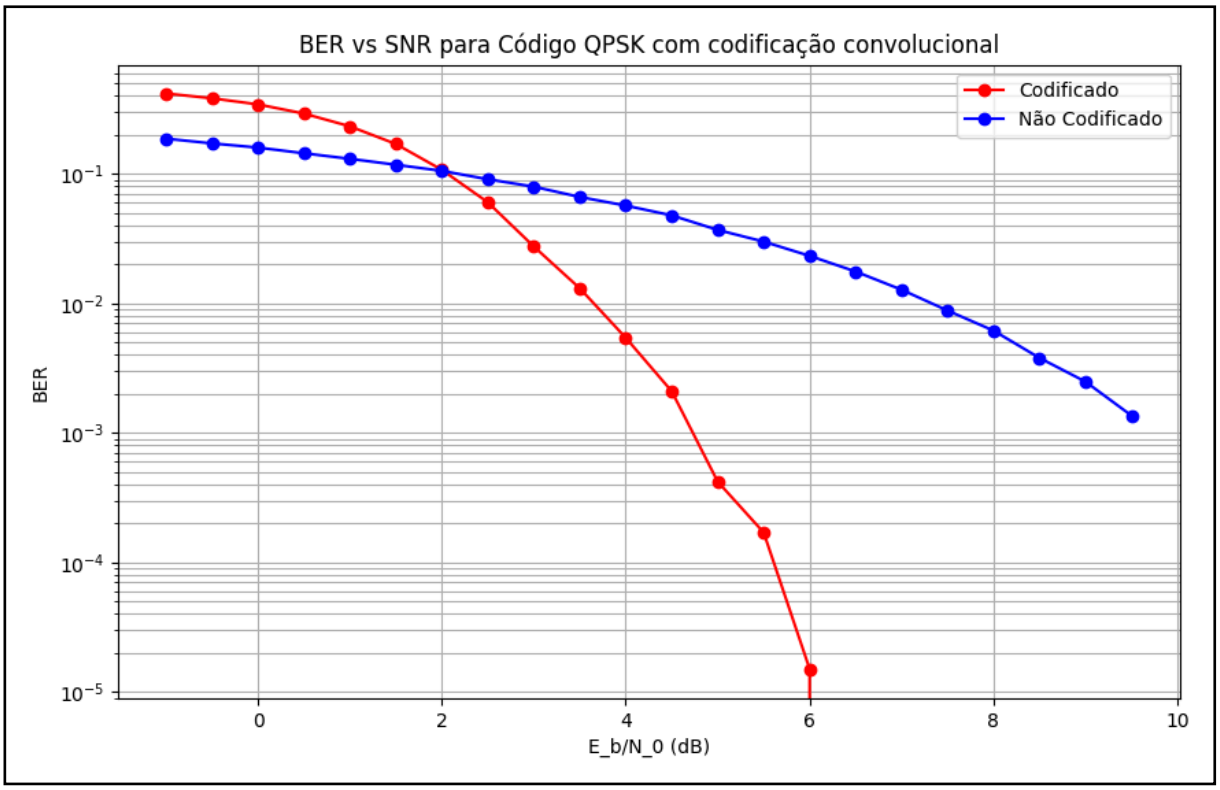
```
21
22      # Adicionando ruído aos sinais modulados
23      noisy_signal = awgn(modulated_bits)
24      noisy_signal_original = awgn(modulated_bits_original)
25
26      # Demodulando o sinal recebido
27      demodulated_signal = modulator.demodulate(
28          noisy_signal)
29      demodulated_signal_original = modulator.demodulate(
30          noisy_signal_original)
31
32      # Decodificando a mensagem (codificada)
33      decoded_bits = decoder(demodulated_signal)
34
35      # Calculando a BER para os sinais codificados e originais
36      BER_conv[i] = np.mean(
37          bits != decoded_bits[:len(bits)])
38      BER_original[i] = np.mean(
39          bits != demodulated_signal_original[:len(bits)])
40
41
42  print("BER_conv     | BER_original")
43  for conv, original in zip(BER_conv, BER_original):
44      print(f"{conv:<12} | {original}")
```

2.2.4. Gráfico da BER

Por fim, foi plotado um gráfico da BER em função da relação sinal-ruido de bit $\frac{E_b}{N_0}$ para o sistema, comparando a transmissão com e sem codificação convolucional.

```
1  # Plot dos resultados
2  plt.figure(figsize=(10, 6))
3  plt.semilogy(EBN0_Range, BER_conv, '-o', label='Codificado', color="red")
   plt.semilogy(EBN0_Range,  BER_original,  '-o',  label='Não  Codificado',
4  color="blue")
5
6  plt.xlabel('E_b/N_0 (dB)')
7  plt.ylabel("BER")
8  plt.title('BER vs SNR para Código QPSK com codificação convolucional')
9
10 plt.legend()
11 plt.grid(True, which="both")  # Grid em ambas escalas
12 plt.show()
```

Figura 4: Elaborada pelo Autor



3. Conclusão:

A partir dos conceitos vistos em aula, foi possível realizar a análise de um código convolucional, determinando sua taxa e ordem de memória, esboçando o diagrama de blocos do codificador, o diagrama de estados do código, determinando os parâmetros (n', k') do código de bloco resultante, codificando e decodificando mensagens, determinando a distância livre do código através do método de Mason, e simulando o desempenho de BER de um sistema de comunicação que utiliza o código convolucional com decodificação via algoritmo de Viterbi, modulação QPSK e canal AWGN.

Os resultados obtidos foram satisfatórios, e permitiram a compreensão dos conceitos abordados em aula, bem como a aplicação prática dos mesmos.

4. Referências

Rw Nobrega - Códigos Convolucionais