INSTITUTO TECNOLÓGICO DA AERONAUGRÁFICA (ITA)
DEPARTAMENTO DE TELECOMUNICAÇÕES - IEET
EET-50 - PRINCÍPIOS DE COMUNICAÇÕES

FELIPE KELLER BALTOR

EXAME

SÃO JOSÉ DOS CAMPOS/SP
2024

Introdução

O presente relatório visa explorar os princípios da comunicação digital na forma de exercícios, finalizando com a implementação de uma rotina de transmissão adaptativa. Serão consideradas as principais dimensões de performance de um sistema de comunicação, a saber, taxa de erro (por símbolo e por bit), razão sinal ruído e eficiência espectral. Os resultados teóricos apresentados durante o curso serão contrapostos à resultados simulados utilizando o Método de Monte Carlo.

Exercício 1

Estime, através de simulação computacional (método de Monte Carlo), a probabilidade de erro de simbolo (P_e) em função da razão E_b/N_0, para um canal AWGN de média zero e PSD $N_0/2$. Considere um modelo de simulação em banda base. Para esta questão não é permitido o uso de funções de modulação/demodulação já existentes no Matlab. (a) 2-PAM (BPSK); (b) 4-PAM; (c) 4-QAM (QPSK); (d) 2-FSK

Para o presente exercício iremos utilizar o Python. O preâmbulo a seguir importa as bibliotecas necessárias e defina a função Q.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import erfc

def Q(x):
    return 0.5 * erfc(x / np.sqrt(2))
```

(a) 2-PAM

Para gerar os dados simulados, utilizamos o trecho a seguir:

```
symbols_constellation = np.array([-1, 1])
M = 2
k = np.log2(M)
snr_db_range = np.arange(0, 11, 1)
symbol_sequence_length = int(1e6)

def simulate_ser_2pam(snr_db_range, symbol_sequence_length):
    ser = []
    for snr_db in snr_db_range:
        snr_linear = 10**(snr_db / 10.0)
        snr_symbol = snr_linear * k
```

```python
        energy_symbol = np.mean(symbols_constellation**2)
        sigma = np.sqrt(energy_symbol/(2*snr_symbol))
        noise = sigma * np.random.randn(symbol_sequence_length)
        signal = np.random.choice(symbols_constellation, symbol_sequence_length)
        signal_with_noise = signal + noise
        signal_demod = [1 if symbol > 0 else -1 for symbol in signal_with_noise]
        signal_demod = np.array(signal_demod)
        num_errors = np.sum(signal != signal_demod)
        ser.append(num_errors / symbol_sequence_length)
    return ser

simulated_ser = simulate_ser_2pam(snr_db_range, symbol_sequence_length)
```

Já para implementar o resultado teórico, foi utilizado o seguinte trecho:

```python
def theoretical_ser(M, snr_db):
    snr_linear = 10**(snr_db / 10.0)
    Q_arg = np.sqrt(((6*np.log2(M))/(M**2 - 1)) * snr_linear)
    return 2 * (1 - 1/M) * Q(Q_arg)

theoretical_ser = theoretical_ser(M, snr_db_range)
```

Os resultados foram plotados com a seguinte rotina:

```python
plt.figure(figsize=(10, 6))
plt.semilogy(snr_db_range, simulated_ser, 'bo-', label='Simulated 2-PAM')
plt.semilogy(snr_db_range, theoretical_ser, 'r-', label='Theoretical 2-PAM')
plt.grid(True, which='both')
plt.xlabel('Eb/N0 (dB)')
plt.ylabel('Symbol Error Rate (SER)')
plt.title('Symbol Error Rate vs. Eb/N0 for 2-PAM')
plt.legend()
plt.show()
```

O resultado obtido foi então:

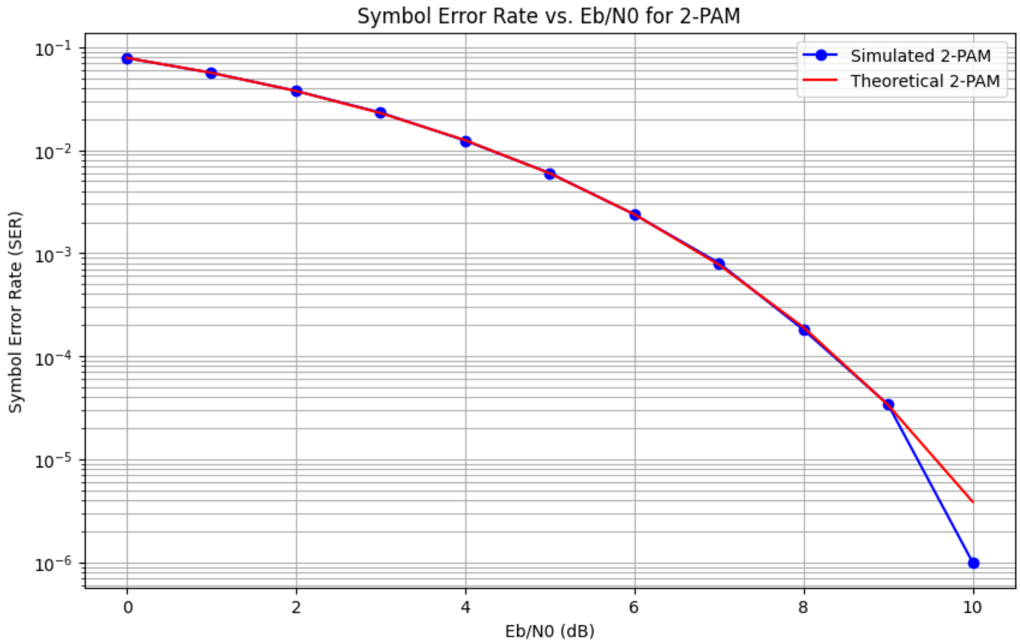

Figura 1: 2-PAM.

(b) 4-PAM

Analogamente ao caso anterior, temos a seguinte rotina para a simulação:

```
symbols_constellation = np.array([-3, -1, 1, 3])
M = 4
k = np.log2(M)
snr_db_range = np.arange(0, 11, 1)
symbol_sequence_length = int(1e7)

def simulate_ser_4pam(snr_db_range, symbol_sequence_length):
    ser = []
    for snr_db in snr_db_range:
        snr_linear = 10**(snr_db / 10.0)
        snr_symbol = snr_linear * k
        energy_symbol = np.mean(symbols_constellation**2)
        sigma = np.sqrt(energy_symbol/(2*snr_symbol))
        noise = sigma * np.random.randn(symbol_sequence_length)
        signal = np.random.choice(symbols_constellation, symbol_sequence_length)
        signal_with_noise = signal + noise
        signal_demod = [-3 if symbol < -2 else -1 if symbol < 0 else 1 if symbol < 2 else 3 for symbol in signal_with_noise]
```

```
        signal_demod = np.array(signal_demod)
        num_errors = np.sum(signal != signal_demod)
        ser.append(num_errors / symbol_sequence_length)
    return ser

simulated_ser = simulate_ser_4pam(snr_db_range, symbol_sequence_length)
```

A parte teórica:

```
def theoretical_ser(M, snr_db):
    snr_linear = 10**(snr_db / 10.0)
    Q_arg = np.sqrt(((6*np.log2(M))/(M**2 - 1)) * snr_linear)
    return 2 * (1 - 1/M) * Q(Q_arg)

theoretical_ser = theoretical_ser(M, snr_db_range)
```

O plot:

```
plt.figure(figsize=(10, 6))
plt.semilogy(snr_db_range, simulated_ser, 'bo-', label='Simulated 4-PAM')
plt.semilogy(snr_db_range, theoretical_ser, 'r-', label='Theoretical 4-PAM')
plt.grid(True, which='both')
plt.xlabel('Eb/N0 (dB)')
plt.ylabel('Symbol Error Rate (SER)')
plt.title('Symbol Error Rate vs. Eb/N0 for 4-PAM')
plt.legend()
plt.show()
```

E o seguinte resultado:

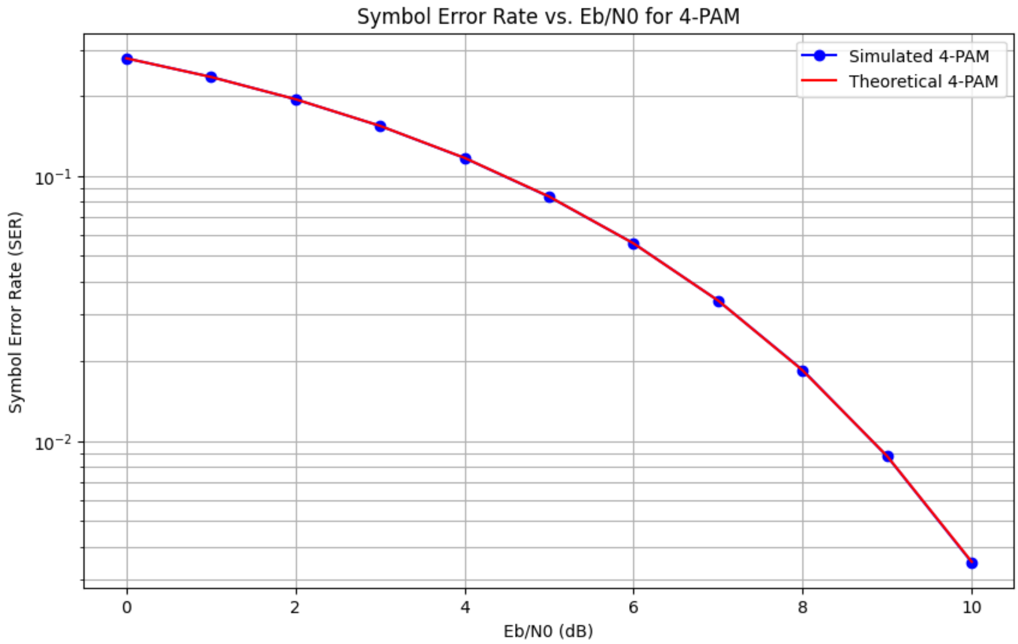

Figura 2: 4-PAM.

(c) 4-QAM

Novamente, para a simulação:

```
symbols_constellation = np.array([1+1j, 1-1j, -1+1j, -1-1j])
M = 4
k = np.log2(M)
snr_db_range = np.arange(0, 11, 1)
symbol_sequence_length = int(1e5)

def simulate_ser_4qam(snr_db_range, symbol_sequence_length):
    ser = []
    for snr_db in snr_db_range:
        snr_linear = 10**(snr_db / 10.0)
        snr_symbol = snr_linear * k
        energy_symbol = np.mean(np.abs(symbols_constellation)**2)
        sigma = np.sqrt(energy_symbol/(2*snr_symbol))
        noise = sigma * (np.random.randn(symbol_sequence_length) + 1j * np.random.randn(symbol_sequence_length))
        signal = np.random.choice(symbols_constellation, symbol_sequence_length)
        signal_with_noise = signal + noise
        signal_demod = []
```

```python
        for symbol in signal_with_noise:
            distances = np.abs(symbol - symbols_constellation)
            demod_symbol = symbols_constellation[np.argmin(distances)]
            signal_demod.append(demod_symbol)
        signal_demod = np.array(signal_demod)
        num_errors = np.sum(signal != signal_demod)
        ser.append(num_errors / symbol_sequence_length)
    return ser

simulated_ser = simulate_ser_4qam(snr_db_range, symbol_sequence_length)
```

Para o resultado teórico:

```python
def P_sqrt_M(M, snr_db):
    snr_linear = 10**(snr_db / 10.0)
    Q_arg = np.sqrt((6/(M - 1)) * snr_linear)
    return 2 * (1 - 1/np.sqrt(M)) * Q(Q_arg)

def theoretical_ser(M, snr_db):
    return 1 - (1 - P_sqrt_M(M, snr_db))**2

theoretical_ser = theoretical_ser(M, snr_db_range)
```

O código de plot:

```python
plt.figure(figsize=(10, 6))
plt.semilogy(snr_db_range, simulated_ser, 'bo-', label='Simulated 4-QAM')
plt.semilogy(snr_db_range, theoretical_ser, 'r-', label='Theoretical 4-QAM')
plt.grid(True, which='both')
plt.xlabel('Eb/N0 (dB)')
plt.ylabel('Symbol Error Rate (SER)')
plt.title('Symbol Error Rate vs. Eb/N0 for 4-QAM')
plt.legend()
plt.show()
```

E o resultado:

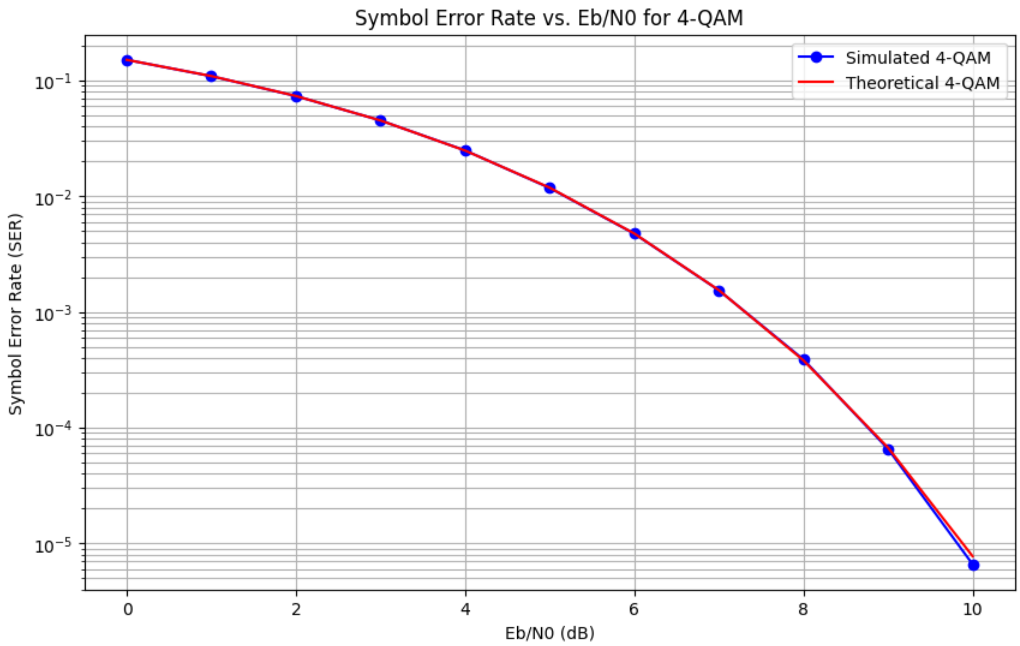

Figura 3: 4-QAM.

Exercício 2

Repita o item (c) do problema anterior para estimar a probabilidade de erro de bit (P_b ou BER $\times$ E_b/N_0) utilizando os seguintes mapeamentos: (a) mapeamento natural (00, 01, 10, 11); (b) mapeamento Gray (00, 01, 11, 10)

Para se poder simular a modulação com diversos mapeamentos distintos vamos alterar a rotina do ítem (c) anterior como se segue:

```
M = 4
k = np.log2(M)
snr_db_range = np.arange(0, 11, 0.5)
symbol_sequence_length = int(1e6)

def bits_to_signal(bits, bits_to_symbols):
    symbols = [bits_to_symbols[tuple(bits[i:i+2])] for i in range(0, len(bits), 2)]
    return np.array(symbols)

def signal_to_bits(signal, symbols_to_bits):
    bits = []
    for symbol in signal:
        bits.extend(symbols_to_bits[symbol])
```

```python
    return np.array(bits)

def simulate_ber_4qam(snr_db_range, symbol_sequence_length, bits_to_symbols, symbols_to_bits):
    ser = []
    for snr_db in snr_db_range:
        snr_linear = 10**(snr_db / 10.0)
        snr_symbol = snr_linear * k
        energy_symbol = np.mean(np.abs(list(bits_to_symbols.values()))**2)
        sigma = np.sqrt(energy_symbol/(2*snr_symbol))
        noise = sigma * (np.random.randn(symbol_sequence_length) + 1j * np.random.randn(symbol_sequence_length))
        bits = np.random.randint(0, 2, int(symbol_sequence_length * k))
        signal = bits_to_signal(bits, bits_to_symbols)
        signal_with_noise = signal + noise
        signal_demod = []
        for symbol in signal_with_noise:
            distances = np.abs(symbol - np.array(list(bits_to_symbols.values())))
            demod_symbol = list(bits_to_symbols.values())[np.argmin(distances)]
            signal_demod.append(demod_symbol)
        signal_demod = np.array(signal_demod)
        bits_demod = signal_to_bits(signal_demod, symbols_to_bits)
        num_errors = np.sum(bits != bits_demod)
        ser.append(num_errors / len(bits))
    return ser
```

Seja agora o mapeamento padrão:

```python
default_symbols_constellation = np.array([1+1j, 1-1j, -1+1j, -1-1j])
default_bits_to_symbols = {
    (0, 0): 1+1j,
    (0, 1): 1-1j,
    (1, 0): -1-1j,
    (1, 1): -1+1j
}
default_symbols_to_bits = {v: k for k, v in default_bits_to_symbols.items()}

simulated_ber_default = simulate_ber_4qam(snr_db_range, symbol_sequence_length, default_bits_to_symbols, default_symbols_to_bits)
```

E o mapeamento de Gray:

```
new_bits_to_symbols = {
    (0, 0): 1+1j,
    (0, 1): 1-1j,
    (1, 0): -1+1j,
    (1, 1): -1-1j
}
new_symbols_to_bits = {v: k for k, v in new_bits_to_symbols.items()}

simulated_ber_new = simulate_ber_4qam(snr_db_range, symbol_sequence_length,
    new_bits_to_symbols, new_symbols_to_bits)
```

Plotando os resultados, obtemos:

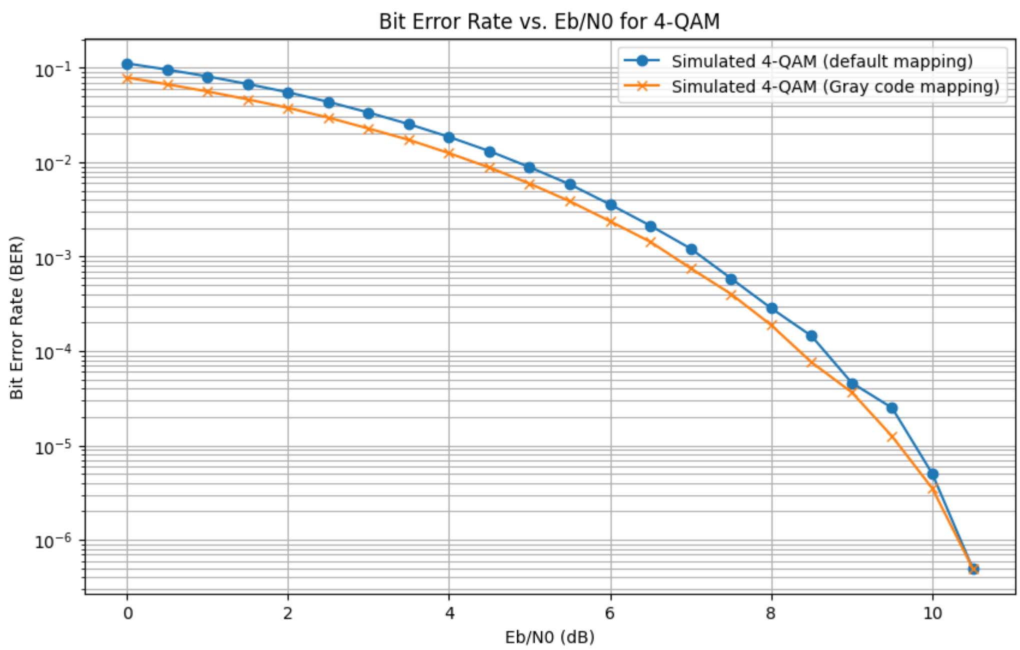

Figura 4: Mapeamento natural e de Gray.

O resultado é condizendo com o esperado, dado que o mapeamento de Gray minimiza a distância entre símbolos quando contraposto ao mapeamento natural, diminuindo levemente o erro de bit.

Projeto

Para a realização das atividades a seguir será necessário o desenvolvimento de uma rotina baseada no algoritmo de Monte Carlo, considerando o uso de protocolo de transmissão adaptativo para diferentes modulações. Considere um modelo de simulação em banda base. Para essa avaliação considere as seguintes modulações: BPSK, QPSK, 8-PSK, 16-QAM e 64-QAM. Para esse cenário, pode-se utilizar funções de modulação/demodulação já existentes no Matlab. O critério de seleção da modulação a ser utilizada é a maximização da eficiência espectral para um desempenho de probabilidade de erro limitante P_e ou P_b , o qual deve ser um parâmetro de entrada da rotina. Considere que a transmissão do sinal é corrompida por um ruído AWGN de média zero e PSD $\mathrm{N}_0/2$. A rotina deve ser comentada e documentada. Com os resultados obtidos, responda os itens a seguir.

Para essa etapa do Exame nós vamos lançar mão da facilidade apresentada pelas bibliotecas de (de)modulação prontas do Matlab. Vamos apresentar a ideia do algoritmo e posteriormente sua rotina, cujas partes serão apresentadas e explicadas em seguida.

O critério de escolha da modulação a ser empregada para uma dada taxa de erro (P_e) e razão sinal ruído (E_b/N_0) é a eficiência espectral. Ora, sabemos que maiores eficiências são atingidas para valores crescentes de M. Entretanto, o erro também aumenta na mesma medida. A solução será então partir da modulação com maior valor de M (mais eficiência) e diminuir até achar o primeiro algoritmo que respeite a probabilidade de erro dada (P_e). Para calcular os valores de erro de cada modulação vamos lançar mão das expressões teóricas dos erros.

Seguindo para o código em si, o cálculo dos valores de erro teórico são feitos com as duas funções abaixos, que cobrem as famílias de modulações *QAM* e *PSK*:

```
% Calculo dos valores teoricos de erro
function er = er_qam_theory(M, EbN0, error_type)
    if nargin < 3
        error_type = 'ser';
    end

    EbN0_linear = 10.^(EbN0 / 10);
    ser = 4 * (1 - 1/sqrt(M)) * qfunc(sqrt(3 * (log2(M)/(M - 1)) * EbN0_linear)
        );
    ber = ser / log2(M);

    if error_type == 'ser'
        er = ser;
    elseif error_type == 'ber'
        er = ber;
    end
```

```
end

function er = er_psk_theory(M, EbN0, error_type)
    EbN0_linear = 10.^(EbN0 / 10);

    if M == 2
        er = qfunc(sqrt(2*EbN0_linear)); % BPSK
        return
    end

    if nargin < 3
        error_type = 'ser';
    end

    ser = 2 * qfunc(sin(pi/M) * sqrt(2 * log2(M) * EbN0_linear));
    ber = ser / log2(M);

    if error_type == 'ser'
        er = ser;
    elseif error_type == 'ber'
        er = ber;
    end
end
```

A próxima parte importante da rotina é a seleção da modulação e aplicação da mesma, que utilizando as funções acima definidas para auxiliar nessa escolha. Ela aplica diretamente a ideia do algoritmo explicada pouco acima.

```
% Funcao que escolhe a melhor modulacao e aplica a mesma
function er = metrics_adaptative(data_bits, pe, EbN0, error_type)
    if nargin < 4
        error_type = 'ser';
    end

    if er_qam_theory(64, EbN0, error_type) < pe
        er = metrics_qam(64, data_bits, EbN0, error_type);

    elseif er_qam_theory(16, EbN0, error_type) < pe
       er = metrics_qam(16, data_bits, EbN0, error_type);

```

```matlab
    elseif er_psk_theory(8, EbN0, error_type) < pe
        er = metrics_psk(8, data_bits, EbN0, error_type);

    elseif er_psk_theory(4, EbN0, error_type) < pe
        er = metrics_psk(4, data_bits, EbN0, error_type);

    else
        er = metrics_psk(2, data_bits, EbN0, error_type);
    end
end
```

Estabelecida a lógica que escolhe o modulador, resta modular o sinal em si e calcular a taxa de erro:

```matlab
% Funcoes responsaveis pela modulacao em si
function er = metrics_psk(M, data_bits, EbN0, error_type)
    if nargin < 4
        error_type = 'ser';
    end

    k = log2(M);
    num_pad_bits = k - mod(length(data_bits), k);
    data_bits = [data_bits; zeros(num_pad_bits, 1)];
    data_symbols = bit2int(data_bits, k);
    tx_symbols = pskmod(data_symbols, M);
    snr = EbN0 + 10*log10(k);
    rx_symbols = awgn(tx_symbols, snr, 'measured');
    data_out_symbols = pskdemod(rx_symbols, M);
    data_out_bits = int2bit(data_out_symbols, k);
    [numErrors, ser] = symerr(data_symbols, data_out_symbols);
    [numErrors, ber] = biterr(data_bits, data_out_bits);

    if error_type == 'ser'
        er = ser;
    elseif error_type == 'ber'
        er = ber;
    end
end

function er = metrics_qam(M, data_bits, EbN0, error_type)
```

```
    if nargin < 4
        error_type = 'ser';
    end

    k = log2(M);
    num_pad_bits = k - mod(length(data_bits), k);
    data_bits = [data_bits; zeros(num_pad_bits, 1)];
    data_symbols = bit2int(data_bits, k);
    tx_symbols = qammod(data_symbols, M, 'UnitAveragePower', true);
    snr = EbN0 + 10*log10(k);
    rx_symbols = awgn(tx_symbols, snr, 'measured');
    data_out_symbols = qamdemod(rx_symbols, M, 'UnitAveragePower', true);
    data_out_bits = int2bit(data_out_symbols, k);
    [numErrors, ser] = symerr(data_symbols, data_out_symbols);
    [numErrors, ber] = biterr(data_bits, data_out_bits);

    if error_type == 'ser'
        er = ser;
    elseif error_type == 'ber'
        er = ber;
    end
end
```

A função acima simplesmente modula o sinal utilizando uma das duas famílias de modulação. Repare que além dos parâmetros usuais (*M*, *data* e *SNR*) ela também recebe o *tipo de erro* que se quer calcular, podendo ser o *SER* (*Symbol Error Rate*) ou o *BER* (*Bit Error Rate*).

Por fim, segue a rotina responsável por plotar uma simulação completa do comunicador adaptativo:

```
% Plota a simulacao adaptativa
function plot_adaptative(data, EbN0, pe, error_type, vlim)
    if nargin < 5
        vlim = [10.^(-6) 10.^(0)];
    end

    er = zeros(1, length(EbN0));

    for i = 1:length(EbN0)
        er(i) = metrics_adaptative(data, pe, EbN0(i), error_type);
    end
```

```
    er_64qam_theory = er_qam_theory(64, EbN0, error_type); % 64-QAM
    er_16qam_theory = er_qam_theory(16, EbN0, error_type); % 16-QAM
    er_8psk_theory = er_psk_theory(8, EbN0, error_type); % 8-PSK
    er_qpsk_theory = er_psk_theory(4, EbN0, error_type); % QPSK
    er_bpsk_theory = er_psk_theory(2, EbN0); % BPSK

    figure;
    semilogy(EbN0, er_64qam_theory, 'o-', 'DisplayName', 'Teorico 64-QAM');
    hold on;
    semilogy(EbN0, er_16qam_theory, 'o-', 'DisplayName', 'Teorico 16-QAM');
    hold on;
    semilogy(EbN0, er_8psk_theory, 'o-', 'DisplayName', 'Teorico 8-PSK');
    hold on;
    semilogy(EbN0, er_qpsk_theory, 'o-', 'DisplayName', 'Teorico QPSK');
    hold on;
    semilogy(EbN0, er_bpsk_theory, 'o-', 'DisplayName', 'Teorico BPSK');
    hold on;
    semilogy(EbN0, er, 'o-', 'DisplayName', 'Adaptativo');
    hold on;
    yline(pe, 'r', 'LineWidth', 1.2, 'DisplayName', 'Erro limite');
    ylim(vlim);
    xlabel('E_b/N_0 (dB)');

    if error_type == 'ser'
        text1 = Taxa de Erro de Simbolo (SER);
        text2 = Simbolo;
    else
        text1 = Taxa de Erro de Bit (BER);
        text2 = Bit;
    end

    ylabel(text1);
    legend show;
    title(Taxa de Erro de  + text2);
    grid on;
end
```

Exercício 3

Apresente o gráfico de SER x E_b/N_0 para os seguintes desempenhos de probabilidade de erro de símbolo limitante $P_e = 10^{-2}$, $Pe = 10^{-3}$ e $Pe = 10^{-4}$.

Para calcular os valores pedidos vamos lançar mão das funções já apresentadas e do seguinte trecho de código:

```
% Par metros
num_bits = 1000000; % N mero de s mbolos a serem transmitidos
data = randi([0 1], num_bits, 1);
EbN0 = 0:0.2:20;  % Valores de Eb/N0 em dB
Pe = [10.^(-2), 10.^(-3), 10.^(-4)];

for i = 1:length(Pe)
    plot_adaptative(data, EbN0, Pe(i), 'ser');
end
```

Os gráficos gerados foram:

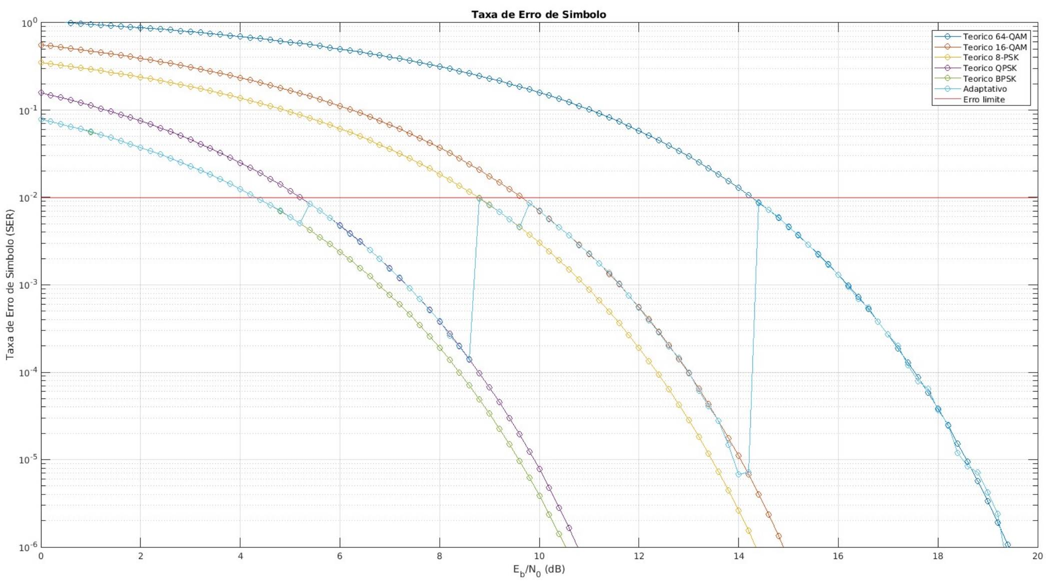

Figura 5: SER com $P_e = 10^{-2}$.

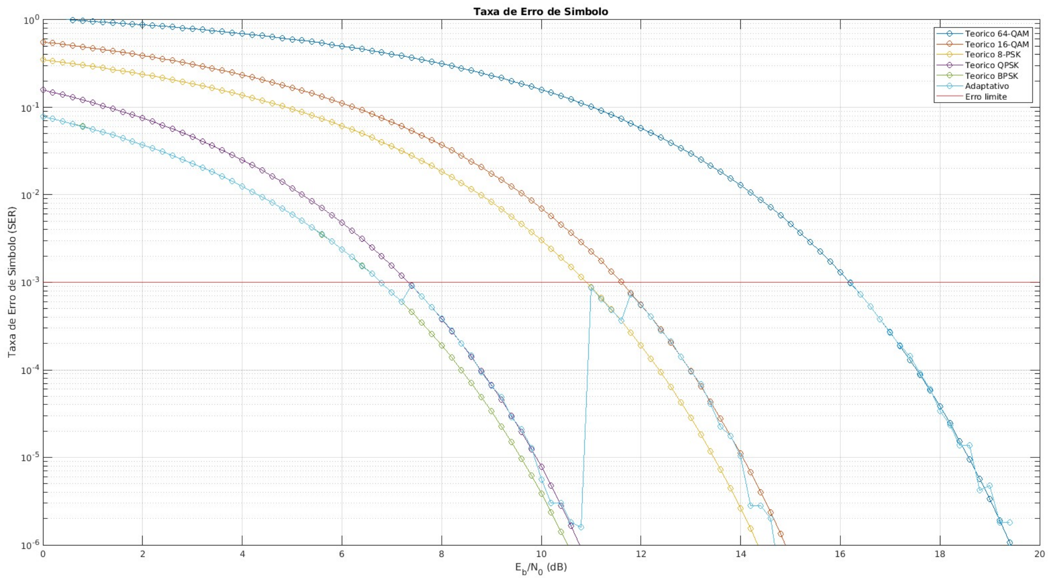

Figura 6: SER com $P_e = 10^{-3}$.

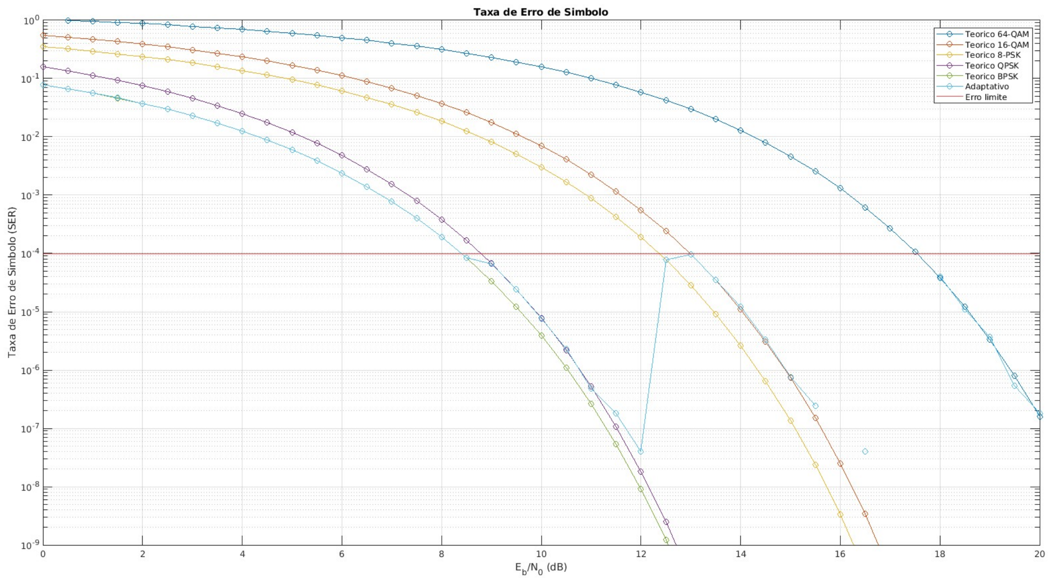

Figura 7: SER com $P_e = 10^{-4}$.

O último gráfico (Figura 7) não apresentou o melhor resultado, com quebras entre os pontos do transmissor adaptativo pois não se atingiu a precisão suficiente. Para melhorá-lo deveríamos simular com um número maior de iterações no Método de Monte Carlo. Apesar disso é possível perceber que seu aspecto esta correto.

Exercício 4

Apresente o gráfico de SER x E_b/N_0 para os seguintes desempenhos de probabilidade de erro de símbolo limitante $P_e = 10^{-2}$, $Pe = 10^{-3}$ e $Pe = 10^{-4}$.

De forma análoga ao exercício anterior e lançando mão da modularidade das funções implementadas, temos o seguinte código para o cálculo do *BER*:

```
% Par metros
num_bits = 1000000; % N mero de s mbolos a serem transmitidos
data = randi([0 1], num_bits, 1);
EbN0 = 0:0.2:20;  % Valores de Eb/N0 em dB
Pe = [10.^(-2), 10.^(-3), 10.^(-4)];

for i = 1:length(Pe)
    plot_adaptative(data, EbN0, Pe(i), 'ber');
end
```

Os gráficos obtidos foram então:

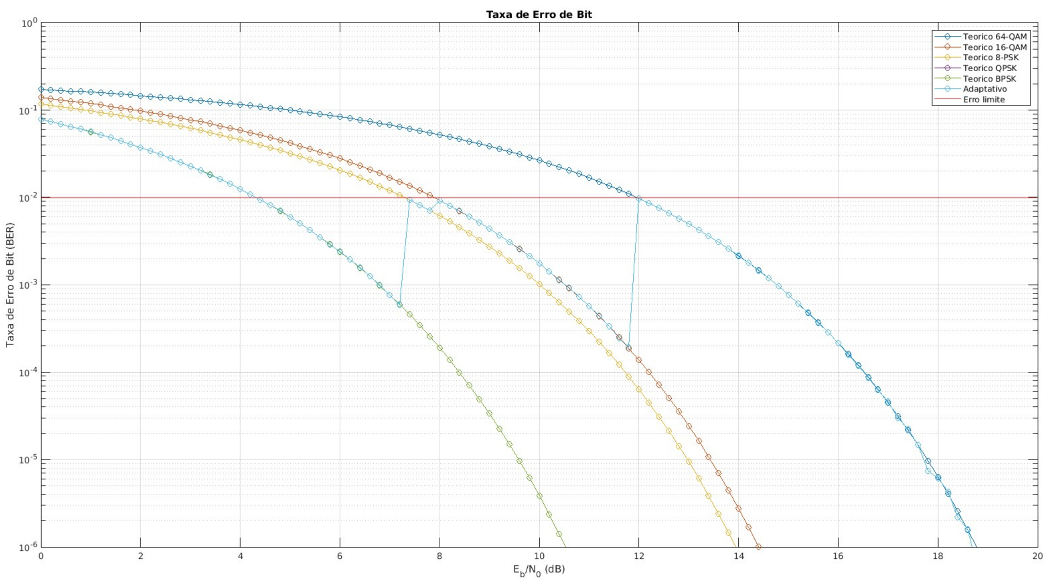

Figura 8: BER com $P_e = 10^{-2}$.

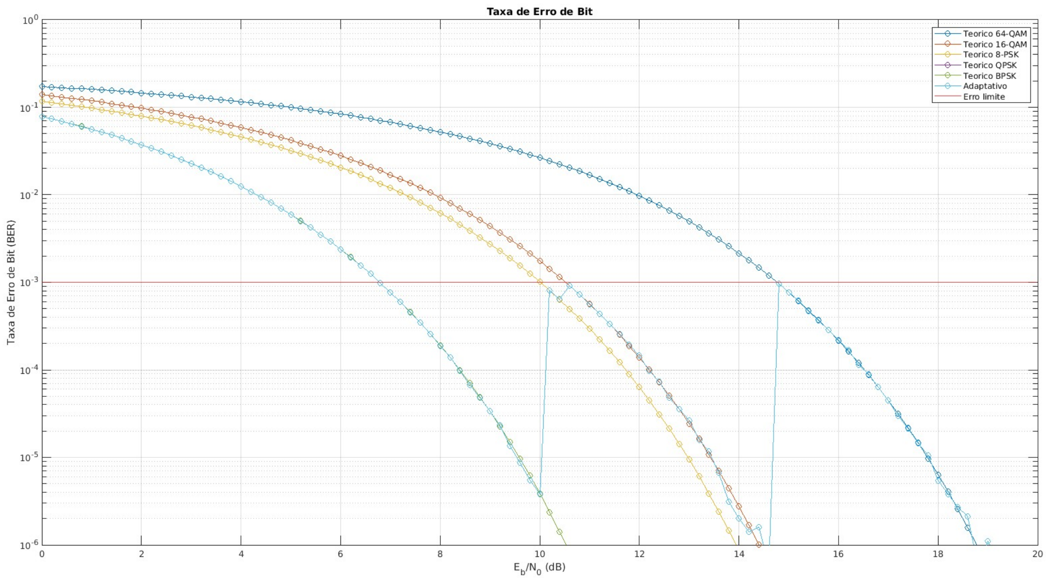

Figura 9: BER com $P_e = 10^{-3}$.

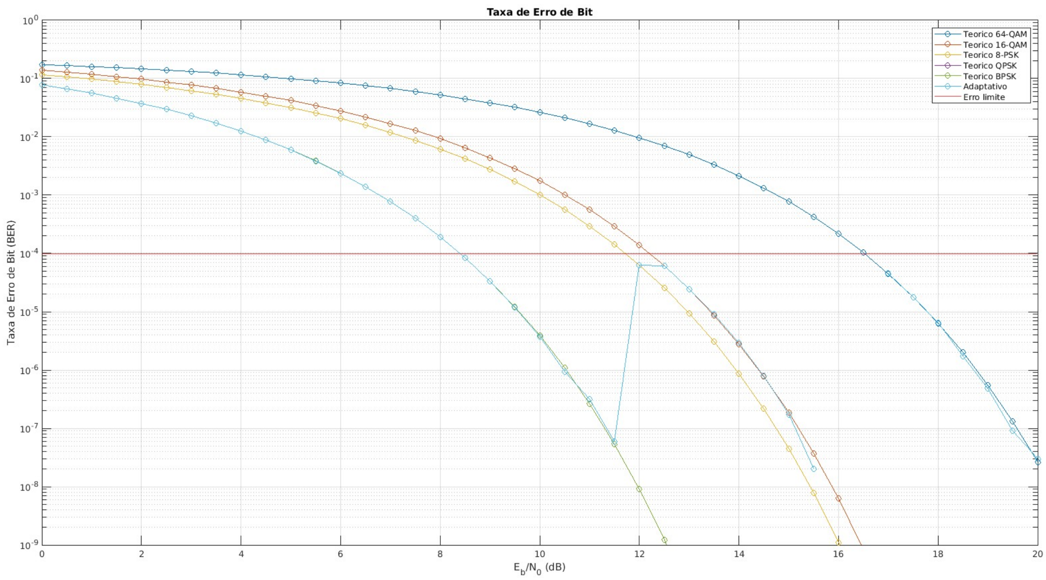

Figura 10: BER com $P_e = 10^{-4}$.

Analogamente à questão anterior, o último gráfico (figura 10) não apresentou uma precisão satisfatória para o número de iterações empregadas. Podemos notar também que as modulações *QPSK* e *BPSK* possuem a mesma taxa de erro de bit teóricas. Além disso, é possível perceber que em geral para um mesmo nível de sinal ruído (E_b/N_0) a taxa de erro simbólico é maior do que a taxa de erro de bit.

Exercício 5

A partir do resultado do exercício 3, considerando o caso em que $P_e = 10^{-4}$, quais são as faixas de operação, em termos de E_b/N_0, para cada uma das modulações consideradas.

Considerando o Exercício 3 podemos ver que para o erro dado a modulação *BPSK* perdura de $E_b/N_0 = 0$ até 8,5, a modulação *QPSK* vai de 8,5 até 12, a modulação *8-PSK* vai de 12 até 12,5, a modulação *16-QAM* vai de 12,5 até 17,5, e a modulação *64-QAM* atua daí em diante.

Exercício 6

A partir do resultado do Exercício 4 e considerando $P_b = 10^{-3}$ e as razões E_b/N_0 = [4 dB, 8 dB, 12 dB, 16 dB], quais modulações são utilizadas para cada um desses casos.

Para 4 dB e 8 dB a modulação utilizada deve ser a *QPSK* (maior eficiência espectral), para 12 dB deve-se utilizar a modulação *16-QAM* e para 24 dB a *64-QAM*.

Referências

HAYKIN SIMON MOHER, M. **Introduction to Analog and Digital Communications**. [S.l.]: Wiley New York, 2006.