Modulação e Demodulação QAM-OFDM

Sistemas de Comunicação II

Arthur Cadore Matuella Barcella

01 de Dezembro de 2024

Engenharia de Telecomunicações - IFSC-SJ

Sumário

1. Introdução

Neste trabalho, foi realizada a simulação de um sistema de comunicação digital utilizando modulação e demodulação QAM-OFDM. O objetivo foi comparar a eficiência da modulação QAM-OFDM com diferentes constelações de símbolos, bem como a aplicação da técnica de Alamouti para redução da taxa de erro de bit (BER).

2. Desenvolvimento

2.1. Etapa 1: Comparação em QAM-OFDM

2.1.1. Definição dos parâmetros

Inicialmente definimos as bibliotecas necessárias para a execução do código, bem como os parâmetros do sistema, como o número de subportadoras, o número de subportadoras piloto, o vetor de SNR e o número de símbolos OFDM utilizados para a transmissão/recepção.

```
# Import das bibliotecas necessárias
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, ifft

# Plot da versão do numpy
print("numpy: ", np.__version__)
```

Abaixo temos a definição dos parâmetros do sistema, como o número de subportadoras, o número de subportadoras piloto, o vetor de SNR e o número de símbolos OFDM utilizados para a transmissão/recepção.

Nota: Nessa questão foi utilizado um valor maior para o vetor de SNR para melhorar a visualização/Precisão do gráfico. Também ao final da célula, foi adicionado um print para exibir os valores configurados.

```
# Parâmetros do sistema
num_subcarriers = 128

# Número de subportadoras piloto
cyclic_prefix_length = 32

# Vetor de SNR para o laço do calculo (menor, maior, passo)
# snr_range = np.arange(-5, 40, 5)

# Nota: Aqui utilizei um valor maior para melhorar a visualização do gráfico
snr_range = np.arange(-5, 41, 1)

# Número de símbolos OFDM utilizados para a transmissão/recepção
num_symbols = 1000

# Imprime os valores configurados (apenas para debug)
print("Número de subportadoras: ", num_subcarriers)
print("Número de subportadoras piloto: ", cyclic_prefix_length)
```

```
print("SNR: ", snr_range)
print("Número de símbolos OFDM: ", num_symbols)
```

2.1.2. Modulação / Demodulação QAM

Em seguida, definimos as funções para modular e demodular em QAM. A função `qam_modulate` recebe um vetor de dados e o número de símbolos da constelação e retorna um vetor de símbolos QAM. A função `qam_demodulate` recebe um vetor de símbolos QAM e o número de símbolos da constelação e retorna um vetor de dados.

```
# Função para modular/demodular em QAM
# Função para modular/demodular em QAM

# Parâmetros:
# M: número de símbolos da constelação
# data: vetor de dados a serem transmitidos
# Retorno: vetor de símbolos QAM
def qam_modulate(data, M):
      return np.sqrt(1/2) * (2*(data % np.sqrt(M)) - np.sqrt(M) + 1 + 1j*(2*(data // np.sqrt(M)) - np.sqrt(M) + 1))

def qam_demodulate(signal, M):

    # Calcula a parte real e imaginária do sinal
    real_part = np.real(signal)
    imag_part = np.imag(signal)

    # Calcula o índice do símbolo com base na parte real e imaginária
    real_part = np.round((real_part + np.sqrt(M) - 1) / 2).astype(int)
    imag_part = np.round((imag_part + np.sqrt(M) - 1) / 2).astype(int)
    return real_part + np.sqrt(M) * imag_part
```

2.1.3. Modulação / Demodulação OFDM

Em seguida, definimos as funções para modular e demodular em OFDM. A função `ofdm_modulate` recebe um vetor de dados, o número de subportadoras e o tamanho do prefixo cíclico e retorna um vetor de símbolos OFDM. A função `ofdm_demodulate` recebe um vetor de símbolos OFDM, o número de subportadoras e o tamanho do prefixo cíclico e retorna um vetor de dados.

```
# Função para modular/demodular em OFDM

# Parâmetros:
# data: vetor de dados a serem transmitidos
# num_subcarriers: número de subportadoras
# cyclic_prefix_length: tamanho do prefixo cíclico

# Retorno: vetor de símbolos OFDM
def ofdm_modulate(data, num_subcarriers, cyclic_prefix_length):

    # Calcula a transformada de Fourier inversa
    ofdm_symbols = ifft(data, num_subcarriers)

```

```
    # Adiciona o prefixo cíclico ao sinal
    cyclic_prefix = ofdm_symbols[:, -cyclic_prefix_length:]

    # Retorna o sinal OFDM
    return np.hstack([cyclic_prefix, ofdm_symbols])

def ofdm_demodulate(ofdm_signal, num_subcarriers, cyclic_prefix_length):

    # Remove o prefixo cíclico do sinal
    ofdm_signal = ofdm_signal[:, cyclic_prefix_length:]

    # Calcula a transformada de Fourier do sinal e retorna
    return fft(ofdm_signal, num_subcarriers)
```

2.1.4. Adição do AWGN

A função `add_awgn_noise` recebe um sinal e uma relação sinal-ruído em dB e retorna o sinal corrompido com ruído AWGN. O ruído é gerado com base na potência do sinal e na relação sinal-ruído.

```
# Função para adicionar ruído AWGN

# Parâmetros:
# signal: sinal a ser corrompido
# snr_db: relação

# Retorno: sinal corrompido
def add_awgn_noise(signal, snr_db):

    # Calcula o valor da SNR em escala linear
    snr_linear = 10**(snr_db / 10)

    # Calcula a potência do sinal através da média do módulo ao quadrado
    signal_power = np.mean(np.abs(signal)**2)

    # Calcula a potência do ruído
    noise_power = signal_power / snr_linear

    # Gera o ruído AWGN com a potência calculada
    noise = np.sqrt(noise_power / 2) * (np.random.randn(*signal.shape) + 1j * np.random.randn(*signal.shape))
    return signal + noise
```

2.1.5. Calculo de BER

Para calcular a BER deve-se simplesmente comparar o vetor de dados original (antes da transmissão) com o vetor de dados demodulados e calcular a média dos bits errados.

```
# Função para calcular a BER

# Parâmetros:
# original_data: dados originais
# demodulated_data: dados demodulados

# Retorno: razão da BER
```

```python
def calculate_ber(original_data, demodulated_data):

    # Calcula a média dos bits errados entre os dados originais e demodulados
    return np.mean(original_data != demodulated_data)
```

2.1.6. Simulação

A partir das funções apresentadas, podemos realizar a simulação do sistema de comunicação. Para isso, criamos um vetor para armazenar os valores de BER, e em seguida, realizamos um loop para cada valor de SNR.

Dentro do loop, realizamos as etapas de transmissão, canal e recepção, e calculamos a BER para cada valor de SNR. Todos os passos estão comentados no código abaixo, onde cada um chama as funções definidas anteriormente. O mesmo processo feito para a modulação 4--QAM é repetido para a modulação 16-QAM.

```python
# Criando um vetor para armazenar os valores de BER
ber_16qam = []

# Loop para cada valor de SNR
for snr in snr_range:

# Transmissão:

    # Criando dados aleatórios para a modulação 16-QAM
    data_16qam = np.random.randint(0, 16, (num_symbols, num_subcarriers))

    # Aplicando a modulação 16-QAM aos dados gerados
    mod_data_16qam = qam_modulate(data_16qam, 16)

    # Aplicando a modulação OFDM aos dados modulados
      ofdm_signal_16qam = ofdm_modulate(mod_data_16qam, num_subcarriers, cyclic_prefix_length)

# Canal:
    # Adicionando ruído AWGN ao sinal OFDM modulado
    rx_signal_16qam = add_awgn_noise(ofdm_signal_16qam, snr)


# Recepção:

    # Demodulando o sinal OFDM recebido
     demod_data_16qam = ofdm_demodulate(rx_signal_16qam, num_subcarriers, cyclic_prefix_length)

    # Demodulando o sinal QAM recebido
    demod_data_16qam = qam_demodulate(demod_data_16qam, 16)

    # Calculando a BER e adicionando ao vetor de BER
    ber_16qam.append(calculate_ber(data_16qam, demod_data_16qam))
```

2.1.7. Plotagem dos resultados

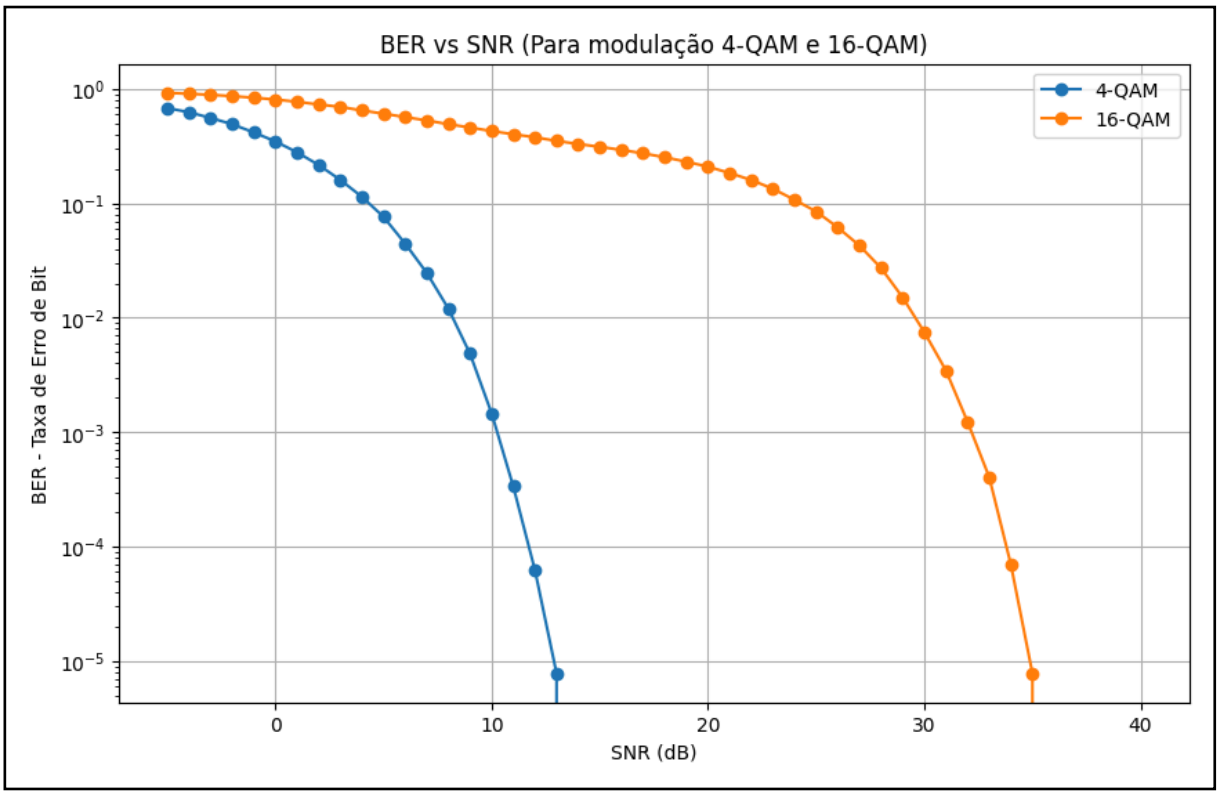
Por fim, plotamos os resultados obtidos (contidos nos vetores de BER gerados a partir da função apresentada anteriormente) para a modulação 4-QAM e 16-QAM em um gráfico de BER vs SNR.

```
# Plot dos resultados
plt.figure(figsize=(10, 6))

# Adicionando os dois vetores de dados ao gráfico
plt.semilogy(snr_range, ber_4qam, '-o', label='4-QAM')
plt.semilogy(snr_range, ber_16qam, '-o', label='16-QAM')

# Configurações do gráfico
plt.xlabel('SNR (dB)')
plt.ylabel('BER - Taxa de Erro de Bit')
plt.title('BER vs SNR (Para modulação 4-QAM e 16-QAM)')
plt.legend()
plt.grid(True)
plt.show()
```

Figura 1: Elaborada pelo Autor



2.2. Etapa 2: Aplicação da técnica de alamouti

2.2.1. Definição dos parâmetros

Inicialmente definimos as bibliotecas necessárias para a execução do código, bem como os parâmetros do sistema, como o número de subportadoras, o número de subportadoras piloto, o vetor de SNR e o número de símbolos OFDM utilizados para a transmissão/recepção.

```python
# Import das bibliotecas necessárias
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, ifft

# Plot da versão do numpy
print("numpy: ", np.__version__)
```

Abaixo temos a definição dos parâmetros do sistema, como o número de subportadoras, o número de subportadoras piloto, o vetor de SNR e o número de símbolos OFDM utilizados para a transmissão/recepção.

Nota: Nessa questão foi utilizado um valor maior para o vetor de SNR para melhorar a visualização/Precisão do gráfico. Também ao final da célula, foi adicionado um print para exibir os valores configurados.

```python
# Parâmetros do sistema
num_subcarriers = 128

# Número de subportadoras piloto
cyclic_prefix_length = 32

# Vetor de SNR para o laço do calculo (menor, maior, passo)
# snr_range = np.arange(-5, 40, 5)

# Nota: Aqui utilizei um valor maior para melhorar a visualização do gráfico
snr_range = np.arange(-5, 41, 1)

# Número de símbolos OFDM utilizados para a transmissão/recepção
num_symbols = 1000

# Imprime os valores configurados (apenas para debug)
print("Número de subportadoras: ", num_subcarriers)
print("Número de subportadoras piloto: ", cyclic_prefix_length)
print("SNR: ", snr_range)
print("Número de símbolos OFDM: ", num_symbols)
```

2.2.2. Modulação / Demodulação QAM

Em seguida, definimos as funções para modular e demodular em QAM. A função `qam_modulate` recebe um vetor de dados e o número de símbolos da constelação e retorna um vetor de símbolos QAM. A função `qam_demodulate` recebe um vetor de símbolos QAM e o número de símbolos da constelação e retorna um vetor de dados.

```python
# Função para modular/demodular em QAM
# Função para modular/demodular em QAM

# Parâmetros:
# M: número de símbolos da constelação
# data: vetor de dados a serem transmitidos
# Retorno: vetor de símbolos QAM
def qam_modulate(data, M):
```

```
        return np.sqrt(1/2) * (2*(data % np.sqrt(M)) - np.sqrt(M) + 1 + 1j*(2*(data // np.sqrt(M)) - np.sqrt(M) + 1))

def qam_demodulate(signal, M):

    # Calcula a parte real e imaginária do sinal
    real_part = np.real(signal)
    imag_part = np.imag(signal)

    # Calcula o índice do símbolo com base na parte real e imaginária
    real_part = np.round((real_part + np.sqrt(M) - 1) / 2).astype(int)
    imag_part = np.round((imag_part + np.sqrt(M) - 1) / 2).astype(int)
    return real_part + np.sqrt(M) * imag_part
```

2.2.3. Modulação / Demodulação OFDM

Em seguida, definimos as funções para modular e demodular em OFDM. A função `ofdm_modulate` recebe um vetor de dados, o número de subportadoras e o tamanho do prefixo cíclico e retorna um vetor de símbolos OFDM. A função `ofdm_demodulate` recebe um vetor de símbolos OFDM, o número de subportadoras e o tamanho do prefixo cíclico e retorna um vetor de dados.

```
# Função para modular/demodular em OFDM

# Parâmetros:
# data: vetor de dados a serem transmitidos
# num_subcarriers: número de subportadoras
# cyclic_prefix_length: tamanho do prefixo cíclico

# Retorno: vetor de símbolos OFDM
def ofdm_modulate(data, num_subcarriers, cyclic_prefix_length):

    # Calcula a transformada de Fourier inversa
    ofdm_symbols = ifft(data, num_subcarriers)
# Plot dos resultados
plt.figure(figsize=(12, 8))

# Plotando os valores de BER para 4-QAM, 16-QAM e Alamouti
plt.semilogy(snr_range, ber_4qam, '-o', label='4-QAM')
plt.semilogy(snr_range, ber_4qam_alamouti, '-o', label='4-QAM Alamouti')
plt.semilogy(snr_range, ber_16qam, '-o', label='16-QAM')
plt.semilogy(snr_range, ber_16qam_alamouti, '-o', label='16-QAM Alamouti')


# Configurações do gráfico
plt.xlim([-5, 40])
plt.ylim([1e-4, 1])
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('BER vs SNR (Para modulação 4-QAM e 16-QAM e Alamouti)')
plt.legend()
plt.grid(True)
plt.show()
    # Adiciona o prefixo cíclico ao sinal
    cyclic_prefix = ofdm_symbols[:, -cyclic_prefix_length:]

```

```
    # Retorna o sinal OFDM
    return np.hstack([cyclic_prefix, ofdm_symbols])

def ofdm_demodulate(ofdm_signal, num_subcarriers, cyclic_prefix_length):

    # Remove o prefixo cíclico do sinal
    ofdm_signal = ofdm_signal[:, cyclic_prefix_length:]

    # Calcula a transformada de Fourier do sinal e retorna
    return fft(ofdm_signal, num_subcarriers)
```

2.2.4. Adição do AWGN

A função `add_awgn_noise` recebe um sinal e uma relação sinal-ruído em dB e retorna o sinal corrompido com ruído AWGN. O ruído é gerado com base na potência do sinal e na relação sinal-ruído.

```
# Função para adicionar ruído AWGN

# Parâmetros:
# signal: sinal a ser corrompido
# snr_db: relação

# Retorno: sinal corrompido
def add_awgn_noise(signal, snr_db):

    # Calcula o valor da SNR em escala linear
    snr_linear = 10**(snr_db / 10)

    # Calcula a potência do sinal através da média do módulo ao quadrado
    signal_power = np.mean(np.abs(signal)**2)

    # Calcula a potência do ruído
    noise_power = signal_power / snr_linear

    # Gera o ruído AWGN com a potência calculada
    noise = np.sqrt(noise_power / 2) * (np.random.randn(*signal.shape) + 1j * np.random.randn(*signal.shape))
    return signal + noise
```

2.2.5. Calculo de BER

Para calcular a BER deve-se simplesmente comparar o vetor de dados original (antes da transmissão) com o vetor de dados demodulados e calcular a média dos bits errados.

```
# Função para calcular a BER

# Parâmetros:
# original_data: dados originais
# demodulated_data: dados demodulados

# Retorno: razão da BER
def calculate_ber(original_data, demodulated_data):

    # Calcula a média dos bits errados entre os dados originais e demodulados
```

```
    return np.mean(original_data != demodulated_data)
```

2.2.6. Modulação / Demodulação Alamouti

Em seguida, definimos as funções para modular e demodular em Alamouti. A função `alamouti_encode` recebe um vetor de símbolos e retorna um vetor de símbolos Alamouti. A função `alamouti_decode` recebe um vetor de símbolos recebidos e um vetor de canais e retorna um vetor de símbolos demodulados.

```
# Função para modular/demodular em Alamouti

# Parâmetros:
# symbols: vetor de símbolos a serem transmitidos
# Retorno: vetor de símbolos Alamouti

def alamouti_encode(symbols):

    # Separa os símbolos em dois vetores
    s1, s2 = symbols[:, 0], symbols[:, 1]

    # Cria a matriz de símbolos Alamouti
    encoded = np.zeros((symbols.shape[0], 2, 2), dtype=complex)
    encoded[:, 0, 0] = s1
    encoded[:, 0, 1] = s2
    encoded[:, 1, 0] = -np.conj(s2)
    encoded[:, 1, 1] = np.conj(s1)

    # Retorna a matriz de símbolos Alamouti
    return encoded

# Parâmetros:
# received: vetor de símbolos recebidos
# channel: vetor de canais

# Retorno: vetor de símbolos demodulados
def alamouti_decode(received, channel):

    # Separa os canais em dois vetores
    h1, h2 = channel[:, 0], channel[:, 1]

    # Separa os símbolos recebidos em dois vetores
    r1, r2 = received[:, 0], received[:, 1]

    # Calcula os símbolos demodulados
    s1_hat = np.conj(h1) * r1 + h2 * np.conj(r2)
    s2_hat = np.conj(h2) * r1 - h1 * np.conj(r2)

    # Calcula a potência do canal combinado
    combined_channel = np.abs(h1)**2 + np.abs(h2)**2

    # Normaliza os símbolos demodulados
    s1_hat /= combined_channel
    s2_hat /= combined_channel

    # Retorna os símbolos demodulados
    return np.stack([s1_hat, s2_hat], axis=-1)
```

2.2.7. Simulação

A partir das funções apresentadas, podemos realizar a simulação do sistema de comunicação. Para isso, criamos um vetor para armazenar os valores de BER, e em seguida, realizamos um loop para cada valor de SNR.

```
# Criando um vetor para armazenar os valores de BER para 16-QAM e 16-QAM com Alamouti
ber_4qam = []
ber_4qam_alamouti = []

# Loop para cada valor de SNR
for snr in snr_range:
# Modulação 4-QAM (mesma operação comentada no código anterior)
    data_4qam = np.random.randint(0, 4, (num_symbols, num_subcarriers))
    mod_data_4qam = qam_modulate(data_4qam, 4)
       ofdm_signal_4qam = ofdm_modulate(mod_data_4qam, num_subcarriers, cyclic_prefix_length)
    rx_signal_4qam = add_awgn_noise(ofdm_signal_4qam, snr)
      demod_data_4qam = ofdm_demodulate(rx_signal_4qam, num_subcarriers, cyclic_prefix_length)
    demod_data_4qam = qam_demodulate(demod_data_4qam, 4)
    ber_4qam.append(calculate_ber(data_4qam, demod_data_4qam))

# Modulação 4-QAM com Alamouti

    # Gerando os dados aleatórios para a modulação
    data_alamouti_4qam = np.random.randint(0, 4, (num_symbols, 2))

    # Modulando os dados com 4-QAM
    mod_data_alamouti_4qam = qam_modulate(data_alamouti_4qam, 4)

    # Modulando os dados em Alamouti
    encoded_alamouti = alamouti_encode(mod_data_alamouti_4qam)

    # Modulando os dados OFDM
            channel = np.random.randn(num_symbols, 2) + 1j * np.random.randn(num_symbols, 2)

    # Mapeando os dados recebidos com o canal e adicionando ruído
    received_alamouti = np.zeros_like(encoded_alamouti, dtype=complex)
    for i in range(2):
        received_alamouti[:, :, i] = encoded_alamouti[:, :, i] * channel[:, i][:, np.newaxis]
    received_alamouti = np.sum(received_alamouti, axis=2)
    received_alamouti = add_awgn_noise(received_alamouti, snr)


    # Decodificando os dados recebidos em Alamouti
    decoded_alamouti = alamouti_decode(received_alamouti, channel)

    # Demodulando os dados decodificados em 4-QAM
    demod_data_alamouti = qam_demodulate(decoded_alamouti, 4)

    # Calculando a BER
      ber_4qam_alamouti.append(calculate_ber(data_alamouti_4qam.flatten(), demod_data_alamouti.flatten()))
```

2.2.8. Plotagem dos resultados

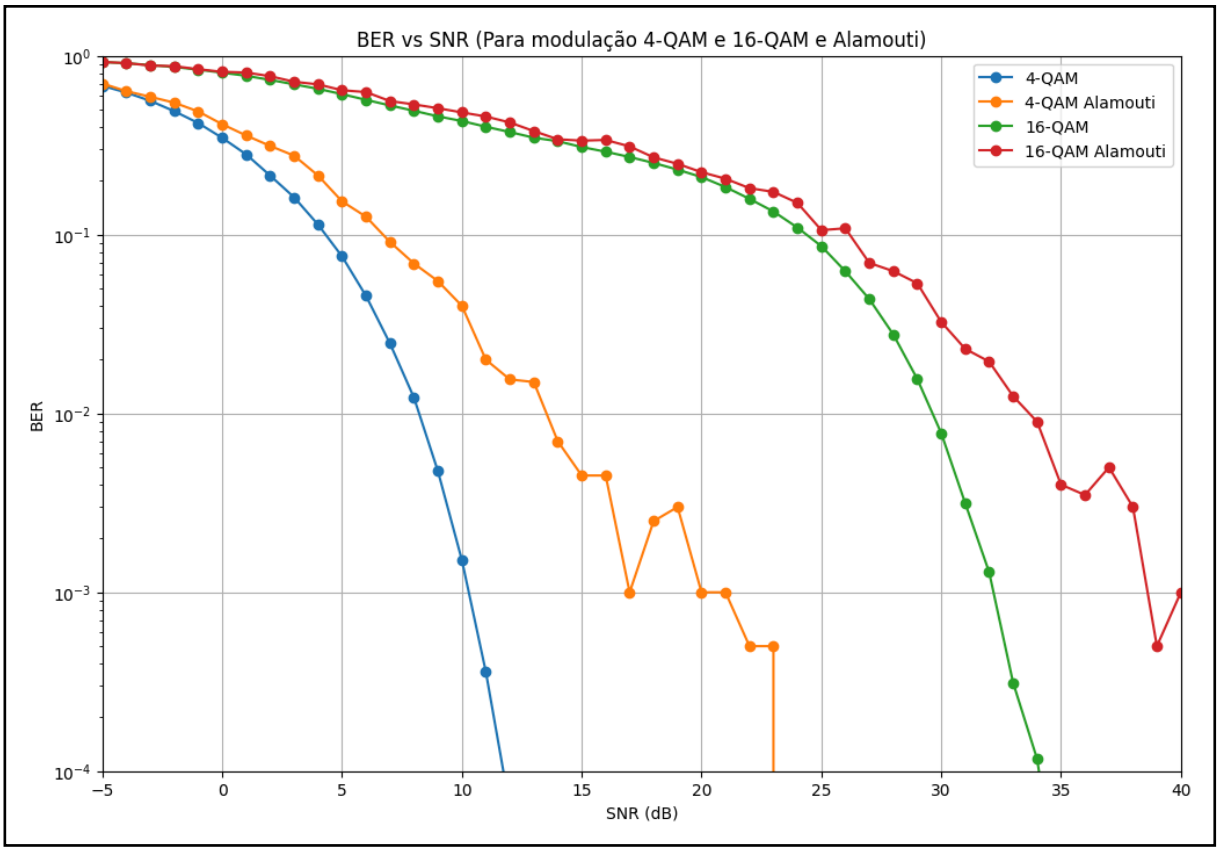
Por fim, plotamos os resultados obtidos (contidos nos vetores de BER gerados a partir da função apresentada anteriormente) para a modulação 4-QAM e 4-QAM com Alamouti em um gráfico de BER vs SNR.

```
# Plot dos resultados
plt.figure(figsize=(12, 8))

# Plotando os valores de BER para 4-QAM, 16-QAM e Alamouti
plt.semilogy(snr_range, ber_4qam, '-o', label='4-QAM')
plt.semilogy(snr_range, ber_4qam_alamouti, '-o', label='4-QAM Alamouti')
plt.semilogy(snr_range, ber_16qam, '-o', label='16-QAM')
plt.semilogy(snr_range, ber_16qam_alamouti, '-o', label='16-QAM Alamouti')


# Configurações do gráfico
plt.xlim([-5, 40])
plt.ylim([1e-4, 1])
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('BER vs SNR (Para modulação 4-QAM e 16-QAM e Alamouti)')
plt.legend()
plt.grid(True)
plt.show()
```

Figura 2: Elaborada pelo Autor



3. Conclusão

A partir dos conceitos apresentados, e resultados obtidos na simulação, foi possível observar a eficácia da técnica de Alamouti na redução da BER em sistemas de comunicação. A técnica de Alamouti é capaz de melhorar a qualidade do sinal recebido, reduzindo a taxa de erro de bit (BER) em comparação com a modulação convencional.