



**INSTITUTO
FEDERAL**

Santa Catarina

Câmpus
São José

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	3
2. Desenvolvimento	3
2.1. Etapa 1: Comparação em QAM-OFDM	3
2.1.1. Definição dos parâmetros	3
2.1.2. Modulação / Demodulação QAM	4
2.1.3. Modulação / Demodulação OFDM	4
2.1.4. Adição do AWGN	5
2.1.5. Cálculo de BER	5
2.1.6. Simulação	6
2.1.7. Plotagem dos resultados	7
2.2. Etapa 2: Aplicação da técnica de Alamouti	7
2.2.1. Definição dos parâmetros	7
2.2.2. Modulação / Demodulação QAM	8
2.2.3. Modulação / Demodulação OFDM	9
2.2.4. Adição do AWGN	10
2.2.5. Cálculo de BER	10
2.2.6. Modulação / Demodulação Alamouti	11
2.2.7. Simulação	12
2.2.8. Plotagem dos resultados	13
3. Conclusão	14

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.

```
1 # Import das bibliotecas necessárias
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from scipy.fft import fft, ifft
5
6 # Plot da versão do numpy
7 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.

```
1 # Parâmetros do sistema
2 num_subcarriers = 128
3
4 # Número de subportadoras piloto
5 cyclic_prefix_length = 32
6
7 # Vetor de SNR para o laço do calculo (menor, maior, passo)
8 # snr_range = np.arange(-5, 40, 5)
9
10 # Nota: Aqui utilizei um valor maior para melhorar a visualização do gráfico
11 snr_range = np.arange(-5, 41, 1)
12
13 # Número de símbolos OFDM utilizados para a transmissão/recepção
14 num_symbols = 1000
15
16 # Imprime os valores configurados (apenas para debug)
17 print("Número de subportadoras: ", num_subcarriers)
18 print("Número de subportadoras piloto: ", cyclic_prefix_length)
```

```

19 print("SNR: ", snr_range)
20 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.

```

1 # Função para modular/demodular em QAM
2 # Função para modular/demodular em QAM
3
4 # Parâmetros:
5 # M: número de símbolos da constelação
6 # data: vetor de dados a serem transmitidos
7 # Retorno: vetor de símbolos QAM
8 def qam_modulate(data, M):
9     return np.sqrt(1/2) * (2*(data % np.sqrt(M)) - np.sqrt(M) + 1 +
10     1j*(2*(data // np.sqrt(M)) - np.sqrt(M) + 1))
11
12 def qam_demodulate(signal, M):
13     # Calcula a parte real e imaginária do sinal
14     real_part = np.real(signal)
15     imag_part = np.imag(signal)
16
17     # Calcula o índice do símbolo com base na parte real e imaginária
18     real_part = np.round((real_part + np.sqrt(M) - 1) / 2).astype(int)
19     imag_part = np.round((imag_part + np.sqrt(M) - 1) / 2).astype(int)
20     return real_part + np.sqrt(M) * 1j * 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.

```

1 # Função para modular/demodular em OFDM
2
3 # Parâmetros:
4 # data: vetor de dados a serem transmitidos
5 # num_subcarriers: número de subportadoras
6 # cyclic_prefix_length: tamanho do prefixo cíclico
7
8 # Retorno: vetor de símbolos OFDM
9 def ofdm_modulate(data, num_subcarriers, cyclic_prefix_length):
10
11     # Calcula a transformada de Fourier inversa
12     ofdm_symbols = ifft(data, num_subcarriers)
13

```

```

14     # Adiciona o prefixo cíclico ao sinal
15     cyclic_prefix = ofdm_symbols[:, -cyclic_prefix_length:]
16
17     # Retorna o sinal OFDM
18     return np.hstack([cyclic_prefix, ofdm_symbols])
19
20 def ofdm_demodulate(ofdm_signal, num_subcarriers, cyclic_prefix_length):
21
22     # Remove o prefixo cíclico do sinal
23     ofdm_signal = ofdm_signal[:, cyclic_prefix_length:]
24
25     # Calcula a transformada de Fourier do sinal e retorna
26     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.

```

1  # Função para adicionar ruído AWGN
2
3  # Parâmetros:
4  # signal: sinal a ser corrompido
5  # snr_db: relação
6
7  # Retorno: sinal corrompido
8  def add_awgn_noise(signal, snr_db):
9
10     # Calcula o valor da SNR em escala linear
11     snr_linear = 10**((snr_db / 10))
12
13     # Calcula a potência do sinal através da média do módulo ao quadrado
14     signal_power = np.mean(np.abs(signal)**2)
15
16     # Calcula a potência do ruído
17     noise_power = signal_power / snr_linear
18
19     # Gera o ruído AWGN com a potência calculada
20     noise = np.sqrt(noise_power / 2) * (np.random.randn(*signal.shape) + 1j
21     * np.random.randn(*signal.shape))
22     return signal + noise

```

2.1.5. Cálculo 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.

```

1  # Função para calcular a BER
2
3  # Parâmetros:
4  # original_data: dados originais
5  # demodulated_data: dados demodulados
6
7  # Retorno: razão da BER

```

```

8 def calculate_ber(original_data, demodulated_data):
9
10     # Calcula a média dos bits errados entre os dados originais e demodulados
11     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.

```

1 # Criando um vetor para armazenar os valores de BER
2 ber_16qam = []
3
4 # Loop para cada valor de SNR
5 for snr in snr_range:
6
7     # Transmissão:
8
9     # Criando dados aleatórios para a modulação 16-QAM
10    data_16qam = np.random.randint(0, 16, (num_symbols, num_subcarriers))
11
12    # Aplicando a modulação 16-QAM aos dados gerados
13    mod_data_16qam = qam_modulate(data_16qam, 16)
14
15    # Aplicando a modulação OFDM aos dados modulados
16    ofdm_signal_16qam = ofdm_modulate(mod_data_16qam, num_subcarriers,
cyclic_prefix_length)
17
18    # Canal:
19    # Adicionando ruído AWGN ao sinal OFDM modulado
20    rx_signal_16qam = add_awgn_noise(ofdm_signal_16qam, snr)
21
22
23    # Recepção:
24
25    # Demodulando o sinal OFDM recebido
26    demod_data_16qam = ofdm_demodulate(rx_signal_16qam, num_subcarriers,
cyclic_prefix_length)
27
28    # Demodulando o sinal QAM recebido
29    demod_data_16qam = qam_demodulate(demod_data_16qam, 16)
30
31    # Calculando a BER e adicionando ao vetor de BER
32    ber_16qam.append(calculate_ber(data_16qam, demod_data_16qam))

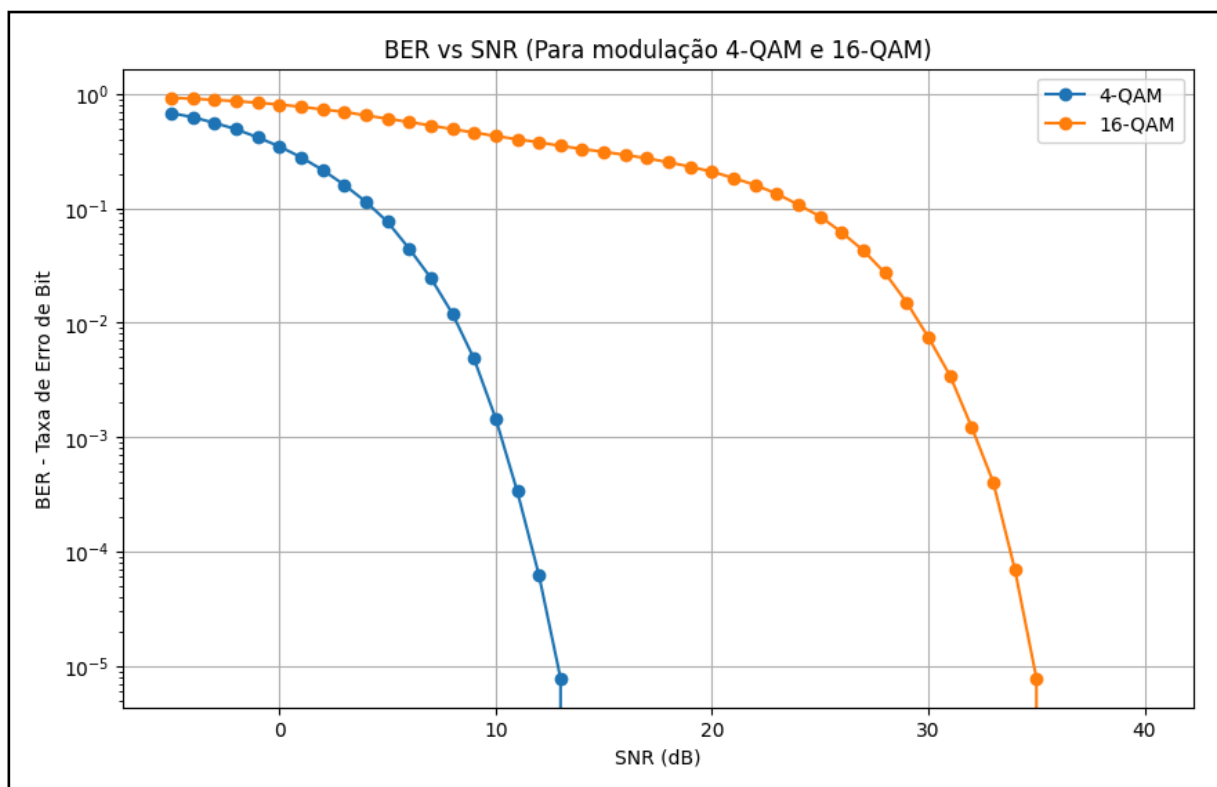
```

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.

```
1 # Plot dos resultados
2 plt.figure(figsize=(10, 6))
3
4 # Adicionando os dois vetores de dados ao gráfico
5 plt.semilogy(snr_range, ber_4qam, '-o', label='4-QAM')
6 plt.semilogy(snr_range, ber_16qam, '-o', label='16-QAM')
7
8 # Configurações do gráfico
9 plt.xlabel('SNR (dB)')
10 plt.ylabel('BER - Taxa de Erro de Bit')
11 plt.title('BER vs SNR (Para modulação 4-QAM e 16-QAM)')
12 plt.legend()
13 plt.grid(True)
14 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.

```

1 # Import das bibliotecas necessárias
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from scipy.fft import fft, ifft
5
6 # Plot da versão do numpy
7 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.

```

1 # Parâmetros do sistema
2 num_subcarriers = 128
3
4 # Número de subportadoras piloto
5 cyclic_prefix_length = 32
6
7 # Vetor de SNR para o laço do calculo (menor, maior, passo)
8 # snr_range = np.arange(-5, 40, 5)
9
10 # Nota: Aqui utilizei um valor maior para melhorar a visualização do gráfico
11 snr_range = np.arange(-5, 41, 1)
12
13 # Número de símbolos OFDM utilizados para a transmissão/recepção
14 num_symbols = 1000
15
16 # Imprime os valores configurados (apenas para debug)
17 print("Número de subportadoras: ", num_subcarriers)
18 print("Número de subportadoras piloto: ", cyclic_prefix_length)
19 print("SNR: ", snr_range)
20 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.

```

1 # Função para modular/demodular em QAM
2 # Função para modular/demodular em QAM
3
4 # Parâmetros:
5 # M: número de símbolos da constelação
6 # data: vetor de dados a serem transmitidos
7 # Retorno: vetor de símbolos QAM
8 def qam_modulate(data, M):

```



```

9         return np.sqrt(1/2) * (2*(data % np.sqrt(M)) - np.sqrt(M) + 1 +
10        1j*(2*(data // np.sqrt(M)) - np.sqrt(M) + 1))
11
12 def qam_demodulate(signal, M):
13     # Calcula a parte real e imaginária do sinal
14     real_part = np.real(signal)
15     imag_part = np.imag(signal)
16
17     # Calcula o índice do símbolo com base na parte real e imaginária
18     real_part = np.round((real_part + np.sqrt(M) - 1) / 2).astype(int)
19     imag_part = np.round((imag_part + np.sqrt(M) - 1) / 2).astype(int)
20     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.

```

1  # Função para modular/demodular em OFDM
2
3  # Parâmetros:
4  # data: vetor de dados a serem transmitidos
5  # num_subcarriers: número de subportadoras
6  # cyclic_prefix_length: tamanho do prefixo cíclico
7
8  # Retorno: vetor de símbolos OFDM
9  def ofdm_modulate(data, num_subcarriers, cyclic_prefix_length):
10
11     # Calcula a transformada de Fourier inversa
12     ofdm_symbols = ifft(data, num_subcarriers)
13     # Plot dos resultados
14     plt.figure(figsize=(12, 8))
15
16     # Plotando os valores de BER para 4-QAM, 16-QAM e Alamouti
17     plt.semilogy(snr_range, ber_4qam, '-o', label='4-QAM')
18     plt.semilogy(snr_range, ber_4qam_alamouti, '-o', label='4-QAM Alamouti')
19     plt.semilogy(snr_range, ber_16qam, '-o', label='16-QAM')
20     plt.semilogy(snr_range, ber_16qam_alamouti, '-o', label='16-QAM Alamouti')
21
22
23     # Configurações do gráfico
24     plt.xlim([-5, 40])
25     plt.ylim([1e-4, 1])
26     plt.xlabel('SNR (dB)')
27     plt.ylabel('BER')
28     plt.title('BER vs SNR (Para modulação 4-QAM e 16-QAM e Alamouti)')
29     plt.legend()
30     plt.grid(True)
31     plt.show()
32
33     # Adiciona o prefixo cíclico ao sinal
34     cyclic_prefix = ofdm_symbols[:, -cyclic_prefix_length:]

```

```

35     # Retorna o sinal OFDM
36     return np.hstack([cyclic_prefix, ofdm_symbols])
37
38 def ofdm_demodulate(ofdm_signal, num_subcarriers, cyclic_prefix_length):
39
40     # Remove o prefixo cíclico do sinal
41     ofdm_signal = ofdm_signal[:, cyclic_prefix_length:]
42
43     # Calcula a transformada de Fourier do sinal e retorna
44     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.

```

1  # Função para adicionar ruído AWGN
2
3  # Parâmetros:
4  # signal: sinal a ser corrompido
5  # snr_db: relação
6
7  # Retorno: sinal corrompido
8  def add_awgn_noise(signal, snr_db):
9
10     # Calcula o valor da SNR em escala linear
11     snr_linear = 10**(snr_db / 10)
12
13     # Calcula a potência do sinal através da média do módulo ao quadrado
14     signal_power = np.mean(np.abs(signal)**2)
15
16     # Calcula a potência do ruído
17     noise_power = signal_power / snr_linear
18
19     # Gera o ruído AWGN com a potência calculada
20     noise = np.sqrt(noise_power / 2) * (np.random.randn(*signal.shape) + 1j
21 * np.random.randn(*signal.shape))
22     return signal + noise

```

2.2.5. Cálculo 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.

```

1  # Função para calcular a BER
2
3  # Parâmetros:
4  # original_data: dados originais
5  # demodulated_data: dados demodulados
6
7  # Retorno: razão da BER
8  def calculate_ber(original_data, demodulated_data):
9
10     # Calcula a média dos bits errados entre os dados originais e demodulados

```

```
11         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.

```
1  # Função para modular/demodular em Alamouti
2
3  # Parâmetros:
4  # symbols: vetor de símbolos a serem transmitidos
5  # Retorno: vetor de símbolos Alamouti
6
7  def alamouti_encode(symbols):
8
9      # Separa os símbolos em dois vetores
10     s1, s2 = symbols[:, 0], symbols[:, 1]
11
12     # Cria a matriz de símbolos Alamouti
13     encoded = np.zeros((symbols.shape[0], 2, 2), dtype=complex)
14     encoded[:, 0, 0] = s1
15     encoded[:, 0, 1] = s2
16     encoded[:, 1, 0] = -np.conj(s2)
17     encoded[:, 1, 1] = np.conj(s1)
18
19     # Retorna a matriz de símbolos Alamouti
20     return encoded
21
22 # Parâmetros:
23 # received: vetor de símbolos recebidos
24 # channel: vetor de canais
25
26 # Retorno: vetor de símbolos demodulados
27 def alamouti_decode(received, channel):
28
29     # Separa os canais em dois vetores
30     h1, h2 = channel[:, 0], channel[:, 1]
31
32     # Separa os símbolos recebidos em dois vetores
33     r1, r2 = received[:, 0], received[:, 1]
34
35     # Calcula os símbolos demodulados
36     s1_hat = np.conj(h1) * r1 + h2 * np.conj(r2)
37     s2_hat = np.conj(h2) * r1 - h1 * np.conj(r2)
38
39     # Calcula a potência do canal combinado
40     combined_channel = np.abs(h1)**2 + np.abs(h2)**2
41
42     # Normaliza os símbolos demodulados
43     s1_hat /= combined_channel
44     s2_hat /= combined_channel
45
46     # Retorna os símbolos demodulados
47     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.

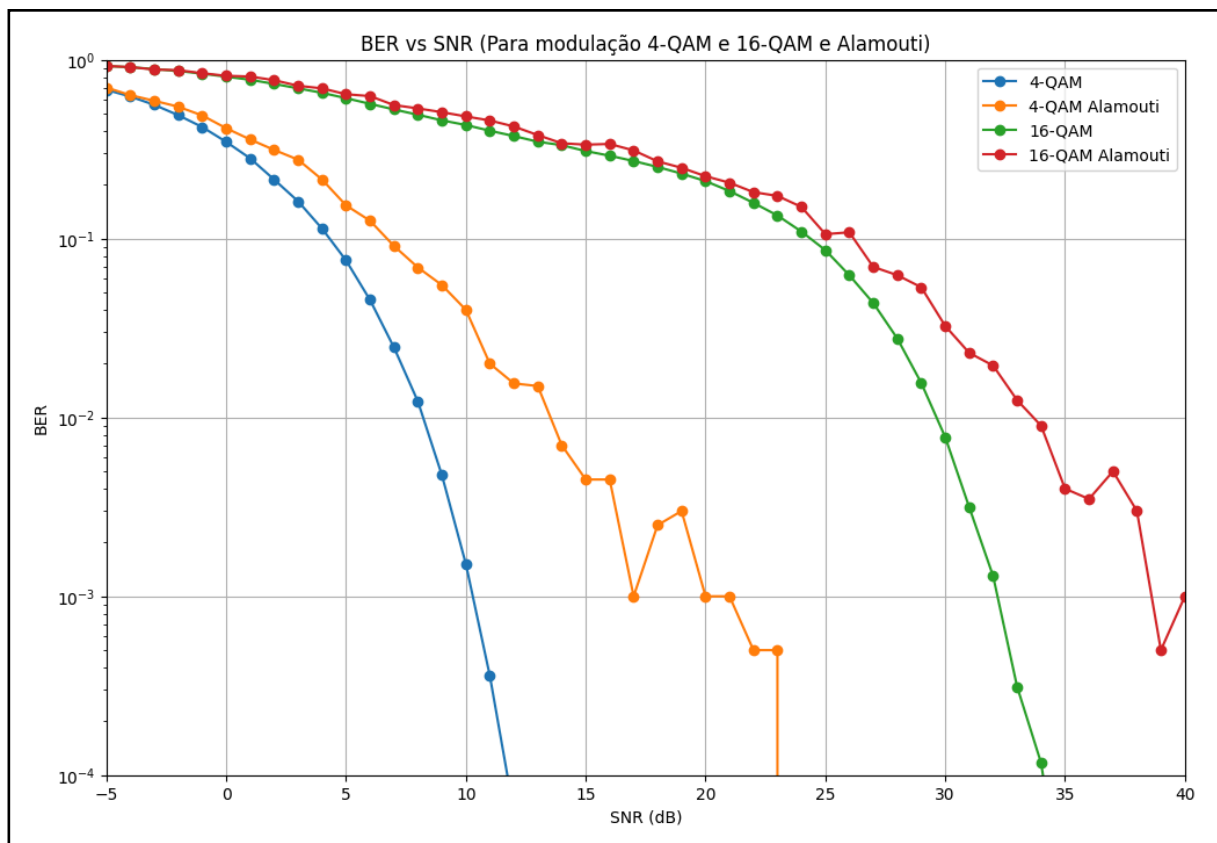
```
1  # Criando um vetor para armazenar os valores de BER para 16-QAM e 16-QAM
   com Alamouti
2  ber_4qam = []
3  ber_4qam_alamouti = []
4
5  # Loop para cada valor de SNR
6  for snr in snr_range:
7  # Modulação 4-QAM (mesma operação comentada no código anterior)
8      data_4qam = np.random.randint(0, 4, (num_symbols, num_subcarriers))
9      mod_data_4qam = qam_modulate(data_4qam, 4)
10     ofdm_signal_4qam = ofdm_modulate(mod_data_4qam, num_subcarriers,
cyclic_prefix_length)
11     rx_signal_4qam = add_awgn_noise(ofdm_signal_4qam, snr)
12     demod_data_4qam = ofdm_demodulate(rx_signal_4qam, num_subcarriers,
cyclic_prefix_length)
13     demod_data_4qam = qam_demodulate(demod_data_4qam, 4)
14     ber_4qam.append(calculate_ber(data_4qam, demod_data_4qam))
15
16 # Modulação 4-QAM com Alamouti
17
18     # Gerando os dados aleatórios para a modulação
19     data_alamouti_4qam = np.random.randint(0, 4, (num_symbols, 2))
20
21     # Modulando os dados com 4-QAM
22     mod_data_alamouti_4qam = qam_modulate(data_alamouti_4qam, 4)
23
24     # Modulando os dados em Alamouti
25     encoded_alamouti = alamouti_encode(mod_data_alamouti_4qam)
26
27     # Modulando os dados OFDM
28     channel = np.random.randn(num_symbols, 2) + 1j *
np.random.randn(num_symbols, 2)
29
30     # Mapeando os dados recebidos com o canal e adicionando ruído
31     received_alamouti = np.zeros_like(encoded_alamouti, dtype=complex)
32     for i in range(2):
33         received_alamouti[:, :, i] = encoded_alamouti[:, :, i] * channel[:,
i][:, np.newaxis]
34     received_alamouti = np.sum(received_alamouti, axis=2)
35     received_alamouti = add_awgn_noise(received_alamouti, snr)
36
37
38     # Decodificando os dados recebidos em Alamouti
39     decoded_alamouti = alamouti_decode(received_alamouti, channel)
40
41     # Demodulando os dados decodificados em 4-QAM
42     demod_data_alamouti = qam_demodulate(decoded_alamouti, 4)
43
44     # Calculando a BER
45     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.

```
1 # Plot dos resultados
2 plt.figure(figsize=(12, 8))
3
4 # Plotando os valores de BER para 4-QAM, 16-QAM e Alamouti
5 plt.semilogy(snr_range, ber_4qam, '-o', label='4-QAM')
6 plt.semilogy(snr_range, ber_4qam_alamouti, '-o', label='4-QAM Alamouti')
7 plt.semilogy(snr_range, ber_16qam, '-o', label='16-QAM')
8 plt.semilogy(snr_range, ber_16qam_alamouti, '-o', label='16-QAM Alamouti')
9
10
11 # Configurações do gráfico
12 plt.xlim([-5, 40])
13 plt.ylim([1e-4, 1])
14 plt.xlabel('SNR (dB)')
15 plt.ylabel('BER')
16 plt.title('BER vs SNR (Para modulação 4-QAM e 16-QAM e Alamouti)')
17 plt.legend()
18 plt.grid(True)
19 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.