Relatório de Simulação

Karen dos Anjos Arcoverde

Introdução

Trata-se da implementação e análise de um sistema de multiplexação por divisão de frequência ortogonal (OFDM) aplicado a diferentes esquemas de modulação (QPSK, 16-QAM e 64-QAM) e submetido a variados cenários de canal — ruído aditivo gaussiano (AWGN), desvanecimento Rayleigh e Rayleigh com AWGN. O principal objetivo da análise é avaliar o desempenho em termos de taxa de erro de bit (BER) em função da relação sinal-ruído (SNR), investigando também o impacto do comprimento do prefixo cíclico (CP) e a eficácia de um código de repetição (taxa 1/3) como técnica de correção de erros. Com isso, pretende-se compreender como cada parâmetro afeta o sistema OFDM diante de diferentes condições de propagação.

Resultados Obtidos

Desafios

Neste bloco definem-se as variáveis fundamentais para a simulação:

- `N = 64`: número de subportadoras ortogonais.
- `CP = 16`: comprimento do prefixo cíclico (em amostras), que é adicionado para mitigar ISI.
- `S = 10000`: total de símbolos OFDM transmitidos.
- SNRs_dB $= \{-10, -5, 0, 5, \ldots, 20\}$: faixa de relações sinal-ruído (em dB) usada para varrer a BER.
- total $= N \times S = 64 \times 10000 = 640000$: número total de amostras transmitidas após a serial-para-paralelo.
- $L = 20$: número de coeficientes (taps) do canal de desvanecimento Rayleigh.

```
# Parametros OFDM
N, CP, S = 64, 16, 10000 # Numero de subportadoras, comprimento
   do prefixo ciclico,
simbolos OFDM
SNRs_dB  = np.arange(-10, 21, 5) # Faixa de SNR em dB
total    = N * S # Total de simbolos transmitidos
L = 20 # numero de taps do canal Rayleigh
```

O canal Rayleigh puro é modelado por um vetor complexo $\mathbf{h} = [h[0], h[1], \ldots, h[L-1]]$ de dimensão L, cujas componentes são amostras de um processo gaussiano circularmente simétrico, normalizadas para que a energia total do vetor seja unitária. Em Python, isso é implementado com a função `np.random.randn`, que gera amostras da distribuição Normal padrão $\mathcal{N}(0, 1)$:

```
# --- Canal Rayleigh puro ---
# h[n] = (hR + j hI)/sqrt(2*L), tamanho L
h = (np.random.randn(L) + 1j*np.random.randn(L)) / np.sqrt(2*L)
```

- `np.random.randn(L)` retorna um vetor de L amostras independentes de uma distribuição gaussiana padrão $\mathcal{N}(0, 1)$.
- A expressão `np.random.randn(L)` $+ 1j$ `np.random.randn(L)` cria vetores para as partes real ($h_R[n]$) e imaginária ($h_I[n]$) de cada coeficiente $h[n]$.

Nesta etapa, cada dois bits são agrupados em um símbolo QPSK e, em seguida, transformados para o domínio do tempo via IFFT:

```
k = 2  # bits por símbolo QPSK

# Passo 2: gera bits aleatórios
# np.random.randint produz valores em {0,1}, criando uma matriz
# de dimensão (2    total), onde cada coluna corresponde a um par
    de bits.
bits = np.random.randint(0, 2, size=(k, total))

# Passo 3: mapeamento QPSK
# Cada par (d1,d2) é mapeado em um símbolo complexo:
#   real = (1 - 2*d1)      {+1, -1}
#   imag = (1 - 2*d2)      {+1, -1}
sym = (1 - 2*bits[0]) + 1j*(1 - 2*bits[1])

# Serial-to-parallel: reorganiza o vetor de símbolos em uma
   matriz N S
# 'order="F"' (Fortran order) preenche a matriz coluna a coluna.
sym = sym.reshape((N, S), order='F')

# Passo 4: IFFT para converter cada coluna do domínio da frequê
   ncia
# (símbolos modulados em subportadoras) no domínio do tempo.
tx = np.fft.ifft(sym, axis=0)
```

- `np.random.randint(0, 2, size=(k, total))` gera uma matriz de bits com duas linhas—uma para o bit de parte real e outra para o bit de parte imaginária.
- O mapeamento $(1 - 2d)$ converte $0 \to +1$ e $1 \to -1$, produzindo pontos QPSK nos quadrantes.
- A operação `reshape((N,S), order='F')` realiza a conversão serial-para-paralelo, onde cada coluna de `sym` será um símbolo OFDM com N subportadoras.

- A IFFT em cada coluna (`axis=0`) sintetiza o sinal de tempo correspondente para cada símbolo OFDM, pronto para inserção do prefixo cíclico.

Após gerar o sinal OFDM no domínio do tempo **tx** (matriz $N \times S$), é preciso:

1. Preencher o prefixo cíclico, copiando as últimas `CP` linhas de cada símbolo.
2. Serializar novamente para obter o vetor transmitido **tx_cp**.
3. Calcular a média complexa e a variância de energia do sinal, que serão usadas para dimensionar o ruído AWGN.

```
# Passo 5: adiciona prefixo cíclico (últimas CP linhas) e
    serializa
tx_cp = np.vstack([tx[-CP:], tx]).reshape(-1, order='F')

# Estatísticas do sinal transmitido
mi = np.mean(tx_cp)                                    # média
    complexa de tx_cp
signal_var = np.mean(np.abs(tx_cp - mi)**2)            # var(tx_cp)
    = E[|x-mi|^2]
```

- `tx[-CP:]` seleciona as `CP` últimas linhas de cada coluna de **tx**, que são usadas como prefixo cíclico para evitar interferência entre símbolos.
- `np.vstack([tx[-CP:], tx])` empilha verticalmente o prefixo sobre o sinal original, produzindo uma matriz $(N + \texttt{CP}) \times S$.
- `reshape(-1, order='F')` achata essa matriz de volta a um vetor coluna, em ordem Fortran (coluna a coluna), para simular a transmissão serial.
- mi = np.mean(tx_cp) é a média complexa do sinal transmitido.
- signal_var = np.mean(np.abs(tx_cp - mi)**2) representa a potência média do sinal em torno de sua média, usada para calcular a variância de ruído adequada ao SNR desejado.

Primeiro, criamos três listas vazias para armazenar os valores de BER (*Bit Error Rate*) em cada canal:

```
ber_awgn     = []   # BER no canal AWGN puro
ber_ray      = []   # BER no canal Rayleigh (sem ruído adicional)
ber_ray_awgn = []   # BER no canal Rayleigh seguido de AWGN
```

Em seguida, iteramos sobre cada valor de SNR em decibéis:

```
for snr_db in SNRs_dB:
    # 1) Converte SNR de dB para razão linear:
    snr       = 10**(snr_db/10)
    # 2) Define a variância do ruído para obter o SNR desejado:
    noise_var = signal_var / snr
```

- $\text{snr} = 10^{\frac{\text{snr\_db}}{10}}$ converte dB para escala linear.
- $\text{noise\_var} = \frac{\text{signal\_var}}{\text{snr}}$ ajusta a potência do ruído em função da potência do sinal e do SNR.

```
# Geração de ruído AWGN complexo
noise    = np.sqrt(noise_var/2) * (
             np.random.randn(*tx_cp.shape)
           + 1j * np.random.randn(*tx_cp.shape)
           )
# Sinal recebido
rx_awgn = tx_cp + noise
```

- `np.random.randn(*tx_cp.shape)` produz amostras i.i.d. de $\mathcal{N}(0, 1)$.
- Multiplicar por $\sqrt{\frac{\text{noise\_var}}{2}}$ garante que cada componente (real e imaginária) do ruído tenha variância $\frac{\text{noise\_var}}{2}$.
- O sinal recebido `rx_awgn` é a soma do sinal transmitido `tx_cp` com o ruído AWGN.

```
# Aplicação do canal Rayleigh via convolução com h
rx_ray = np.convolve(tx_cp, h, mode='full')[:tx_cp.size]
```

- A operação `np.convolve(tx_cp, h, mode='full')` simula o desvanecimento Rayleigh, espalhando o sinal conforme a resposta ao impulso $h[n]$.
- O resultado é truncado a `tx_cp.size` para manter o mesmo comprimento do vetor de transmissão.

```
# Ruído AWGN sobre o sinal com desvanecimento Rayleigh
noise2      = np.sqrt(noise_var/2) * (
                np.random.randn(*rx_ray.shape)
              + 1j * np.random.randn(*rx_ray.shape)
              )
rx_ray_awgn = rx_ray + noise2
```

- Primeiro aplica-se o desvanecimento Rayleigh (`rx_ray`).
- Em seguida, gera-se ruído AWGN da mesma forma que no canal puro e soma-se ao sinal desvanecido.
- O resultado `rx_ray_awgn` incorpora ambos os efeitos: fading e ruído.

Esta função interna realiza as etapas de recepção de um vetor de amostras s, remoção do prefixo cíclico, FFT, equalização (quando necessário) e demodulação hard QPSK:

```
def ofdm_recv(s, channel_type):
    # Passo 7: serial-to-parallel e remoção do CP
    # Reconstrói matriz (N+CP) S  e descarta as CP primeiras
       linhas
    mat = s.reshape((N+CP, S), order='F')[CP:, :]

    # Passo 8: FFT de cada coluna para voltar ao domínio da frequ
       ência
    Y = np.fft.fft(mat, axis=0)  # resultado é N S

    # Equalização somente no canal Rayleigh+AWGN
    if channel_type == 'ray_awgn':
        # FFT do vetor h de comprimento N
        H_fft = np.fft.fft(h, N)
        # Evita divisões por valores muito pequenos
        H_fft[np.abs(H_fft) < 1e-3] = 1e-3
        # Equaliza dividindo Y por H_fft em cada subportadora
        Y = Y / H_fft[:, None]

    # Serializa de volta para um vetor de dimensão 1  ( N S )
    y = Y.reshape(-1, order='F')

    # Passo 9: demodulação QPSK (decisão hard)
    b1 = (y.real < 0).astype(int)  # bit a partir do sinal real
    b2 = (y.imag < 0).astype(int)  # bit a partir do sinal imagin
       ário

    return b1, b2
```

- **Remoção do CP (Passo 7):** `s.reshape((N+CP, S), order='F')` reconstrói a matriz de símbolos OFDM com prefixo; o slicing [`CP` :, :] descarta as amostras do prefixo cíclico.
- **FFT (Passo 8):** A função `np.fft.fft(mat, axis=0)` transforma cada coluna (símbolo OFDM) de volta ao domínio da frequência para recuperar os símbolos modulados.
- **Equalização:** Apenas no caso `ray_awgn` é aplicada equalização. Calcula-se `H_fft` = `fft`(h) para obter a resposta do canal em cada subportadora, evita-se divisão por zero e normaliza-se Y para compensar o fading.
- **Serialização:** `Y.reshape(-1, order='F')` converte a matriz de volta em vetor para demodulação bit a bit.
- **Demodulação hard QPSK (Passo 9):** Usa-se o sinal real/imaginário de y para decidir bits: valores negativos correspondem a 1, positivos a 0.

```
errs = np.sum(bits != np.vstack([b1, b2]))
```

- `bits` é a matriz original de bits transmitidos, de forma $(2 \times$ `total`$)$.

- `b1` e `b2` são vetores (cada um com `total` elementos) contendo os bits demodulados das partes real e imaginária, respectivamente.
- `np.vstack([b1, b2])` empilha verticalmente esses dois vetores, formando uma matriz $2 \times$ `total`.
- A comparação `bits != np.vstack([b1, b2])` gera uma matriz booleana onde cada entrada é `True` se o bit estimado for diferente do original (ou seja, erro) e `False` caso contrário.
- Por fim, `np.sum(...)` soma todos os `True` (que valem 1), produzindo `errs`, o total de bits recebidos incorretamente.

```
return errs / bits.size
```

- `errs` é o número total de bits incorretos calculado previamente.
- `bits.size` retorna o número total de bits transmitidos.
- A expressão $\frac{\texttt{errs}}{\texttt{bits.size}}$ calcula a fração de bits recebidos incorretamente em relação ao total, ou seja, a BER (Bit Error Rate).

Agora para o caso M-QAM:

1. **Inicialização do modem**

```
modem = QAMModem(M)                # escolhe uma constelação M-QAM
k     = modem.num_bits_symbol      # bits por símbolo (log M)
```

   - Em QPSK, k era fixo em 2. Aqui, o `QAMModem` calcula internamente $k = \log_2(M)$.

2. **Geração e organização dos bits**

```
bits = np.random.randint(0, 2, size=(k, total)).reshape(-1, order='F')
```

   - Em QPSK, criávamos dois vetores separados (`bits[0]` e `bits[1]`).
   - Aqui, geramos um vetor serial de $k \times$ `total` bits e o reorganizamos em paralelo (coluna a coluna) para formar a matriz $N \times S$.

3. **Mapeamento de símbolos**

```
sym = modem.modulate(bits).reshape((N, S), order='F')
```

   - Em QPSK, fazíamos manualmente $(1 - 2d_1) + j(1 - 2d_2)$.
   - Em M-QAM, delegamos ao `modem.modulate` todo o mapeamento conforme a constelação de M pontos.

4. **Demodulação e cálculo de erros**

```
bits_hat = modem.demodulate(y, 'hard')
errs     = np.sum(bits != bits_hat)
return errs / bits.size
```

- No QPSK, extraíamos `b1` e `b2` diretamente dos sinais real e imaginário.
- Aqui, usamos `modem.demodulate` para converter o vetor `y` em bits estimados de uma só vez, simplificando o código e suportando qualquer M.

A seguir, nas Figuras 1 a 3, apresentam-se as curvas de BER em função do SNR para o sistema OFDM sob três condições de canal distintas: (i) AWGN puro, (ii) Rayleigh puro sem AWGN adicional e (iii) Rayleigh com AWGN para as modulações QPSK, 16-QAM e 64-QAM.

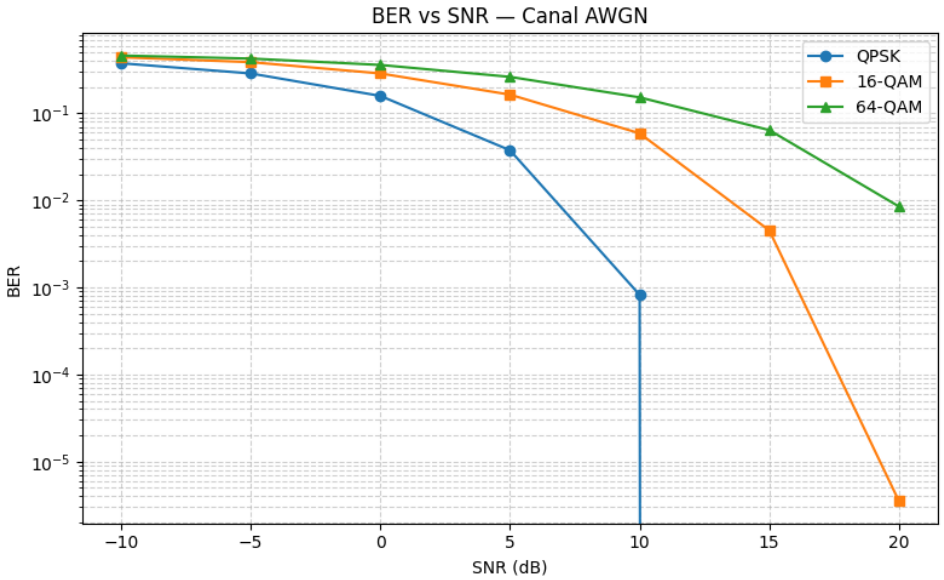

Figura 1: BER vs. SNR para um sistema OFDM em canal AWGN puro.

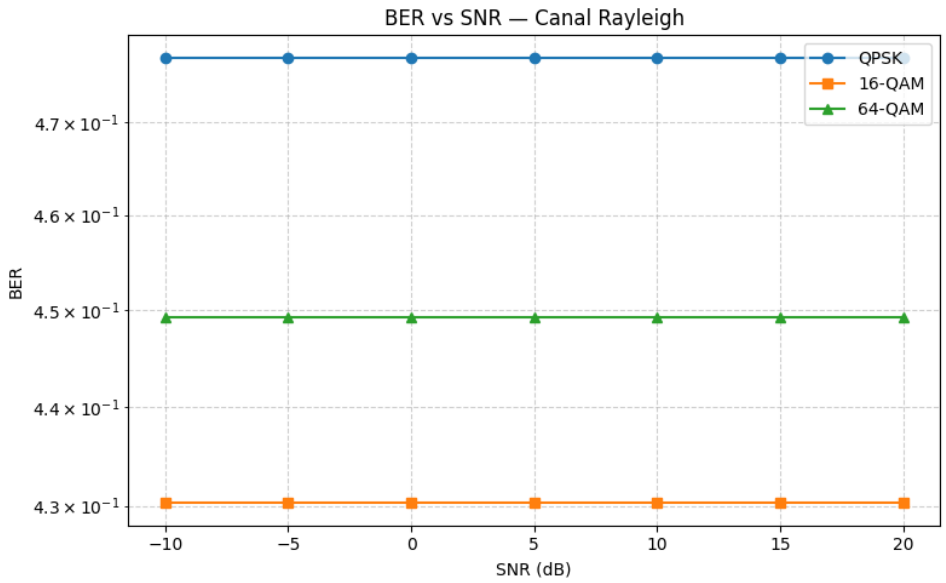

Figura 2: BER vs. SNR para um sistema OFDM em canal Rayleigh puro.

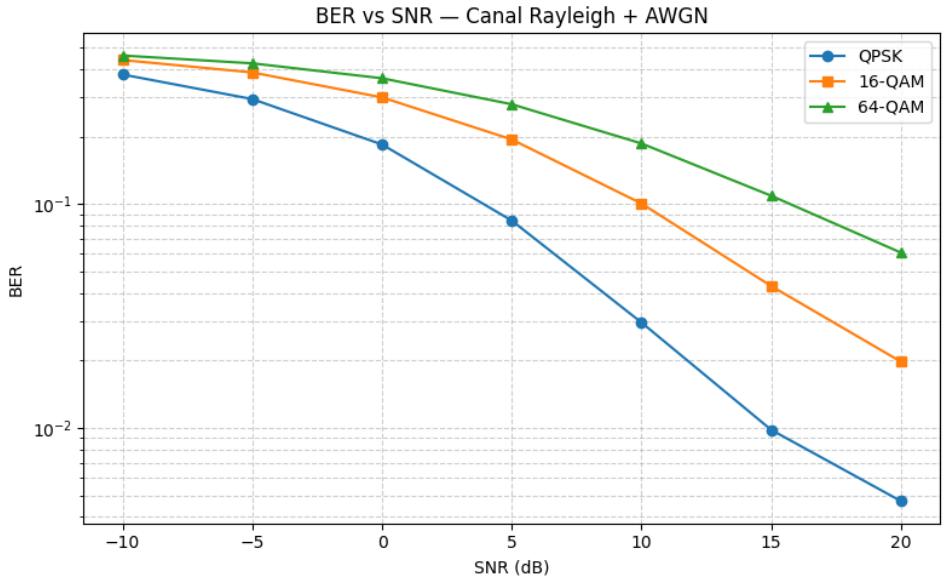

Figura 3: BER vs. SNR para um sistema OFDM em canal Rayleigh com AWGN.

No canal AWGN puro, a taxa de erro de bit (BER) decresce de forma acentuada à medida que o SNR aumenta, evidenciando a influência exclusiva do ruído térmico; no canal Rayleigh puro, a BER permanece praticamente inalterada com o SNR, indicando que o desvanecimento multipercurso impõe um limite de desempenho independente da potência de ruído; e no canal Rayleigh com AWGN, observa-se uma combinação dos dois efeitos: a BER cai com o SNR, mas de modo retardado (curva se desloca para a direita) em comparação ao AWGN puro, refletindo simultaneamente o impacto do fading e do ruído.

As modulações QPSK, 16-QAM e 64-QAM diferem principalmente na densidade de símbolos: na curva de QPSK vemos a queda mais rápida da BER com o aumento do SNR, pois cada símbolo transporta apenas 2 bits e há maior separação entre pontos de

constelação; já a 16-QAM, que carrega 4 bits por símbolo, exige SNR mais alto para atingir o mesmo nível de erro, fazendo sua curva se deslocar para a direita e ter declive mais suave; e a 64-QAM, com 6 bits por símbolo e constelação ainda mais densa, aparece como a mais vulnerável, mantendo BER maiores em níveis de SNR onde QPSK e 16-QAM já apresentam desempenho muito melhor.

A seguir, são apresentados os espectros de magnitude em função da frequência de um único símbolo OFDM, modulados em QPSK, antes e após a adição de ruído AWGN (–5 dB e 20 dB) nos cenários de canal AWGN puro, Rayleigh e Rayleigh + AWGN.

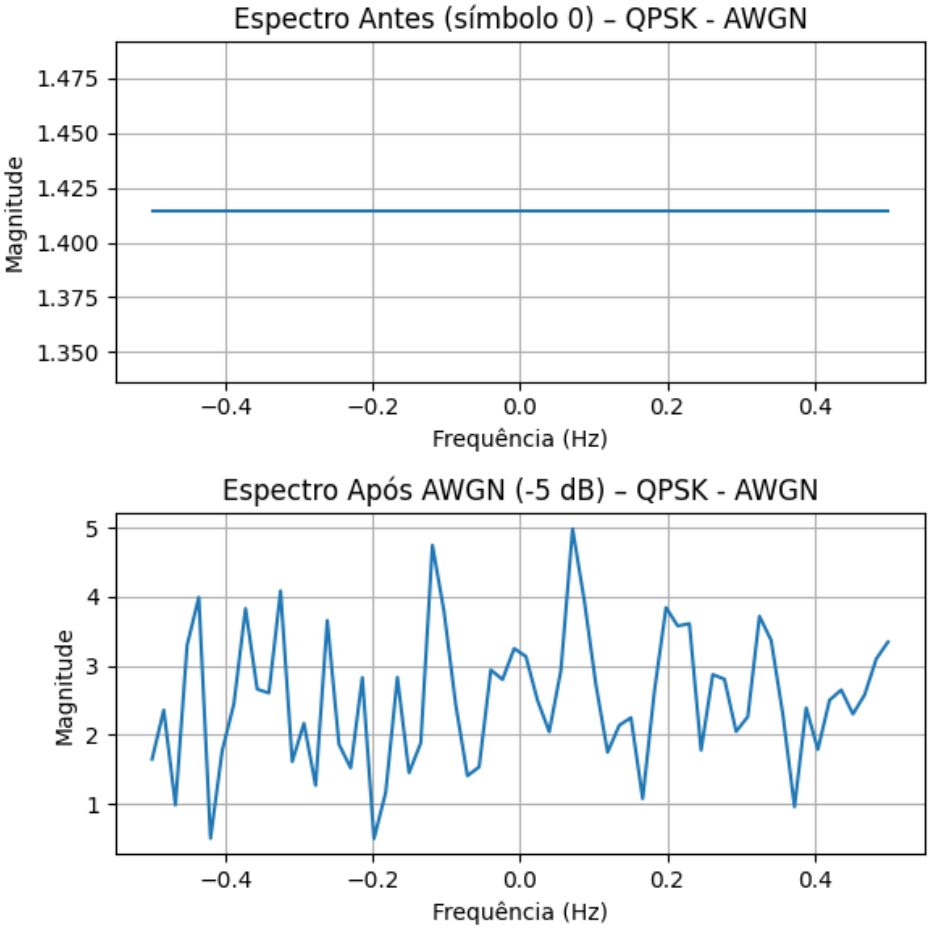

Figura 4: Espectro Antes (símbolo 0) e Após AWGN (SNR -5 dB) — QPSK, Canal AWGN.

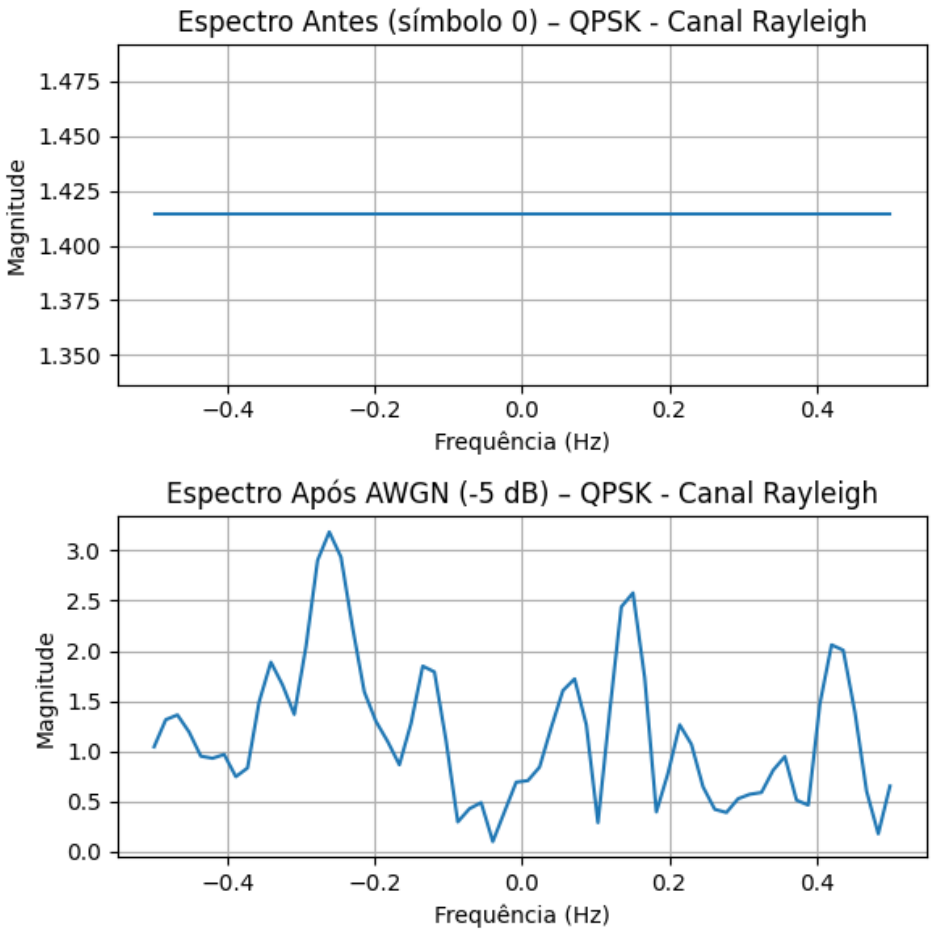

Figura 5: Espectro Antes (símbolo 0) e Pós-Equalização (SNR -5 dB) — QPSK, Canal Rayleigh.

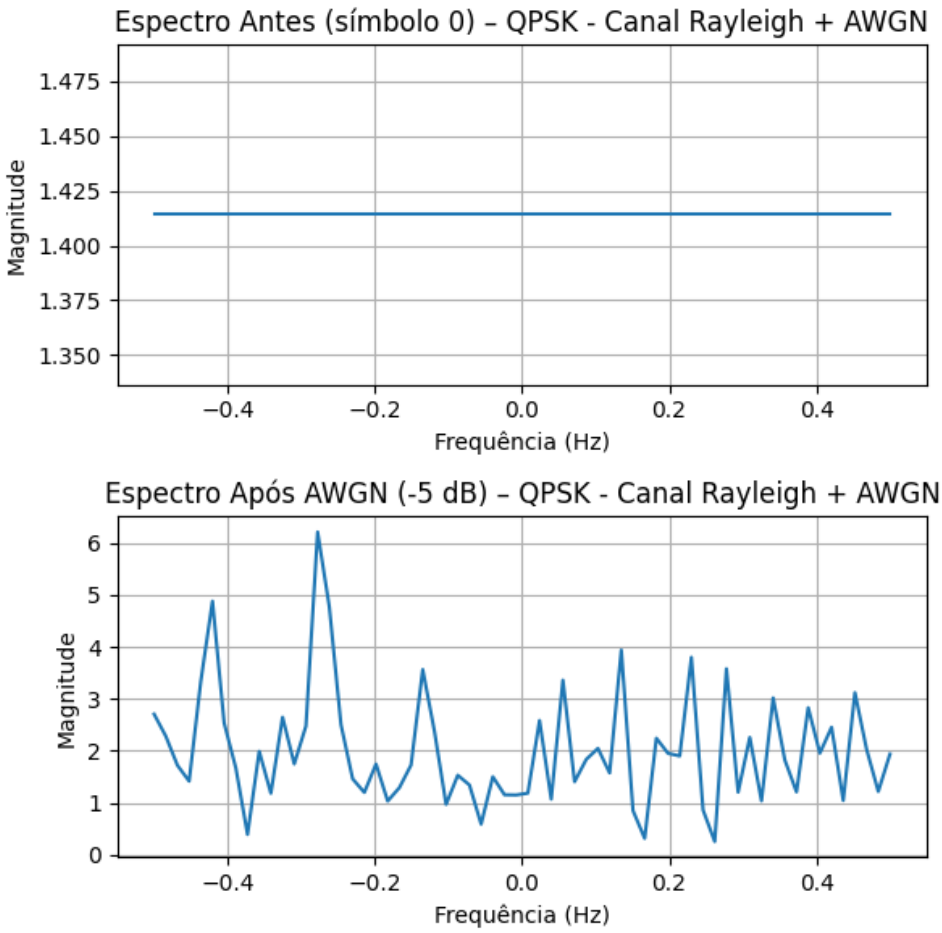

Figura 6: Espectro Antes (símbolo 0) e Após AWGN (SNR -5 dB) — QPSK, Canal Rayleigh + AWGN.

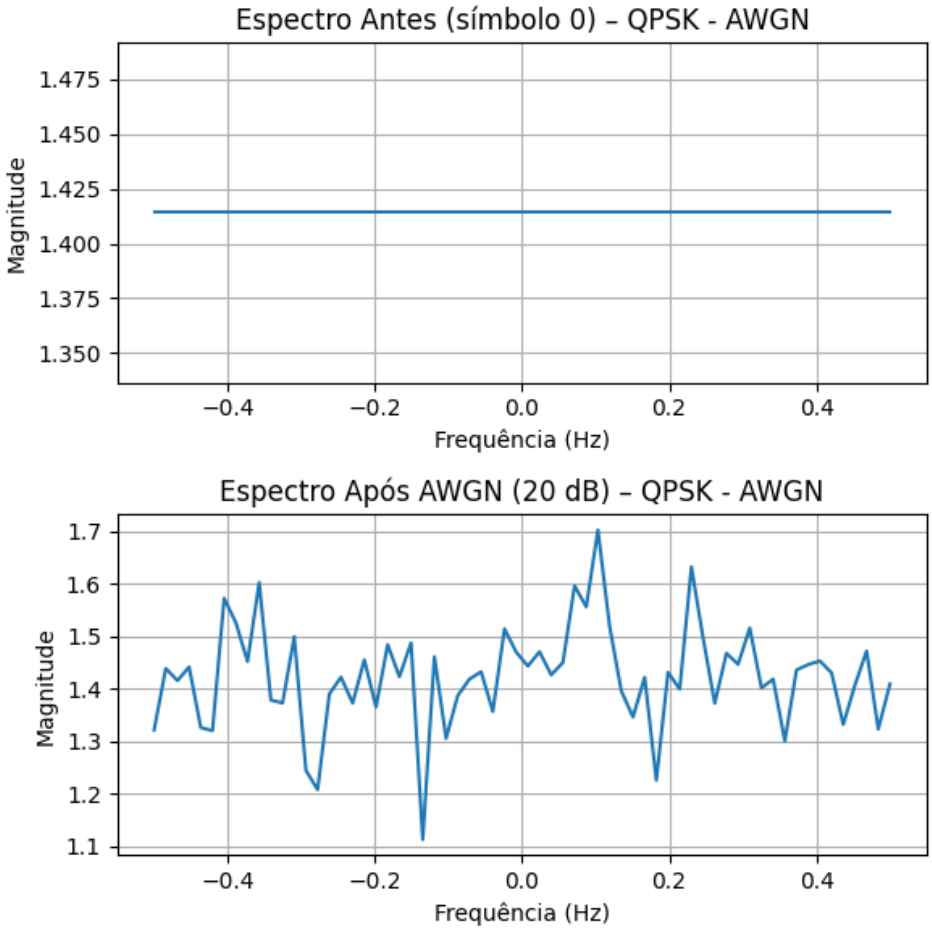

Figura 7: Espectro Antes (símbolo 0) e Após AWGN (SNR 20 dB) — QPSK, Canal AWGN.

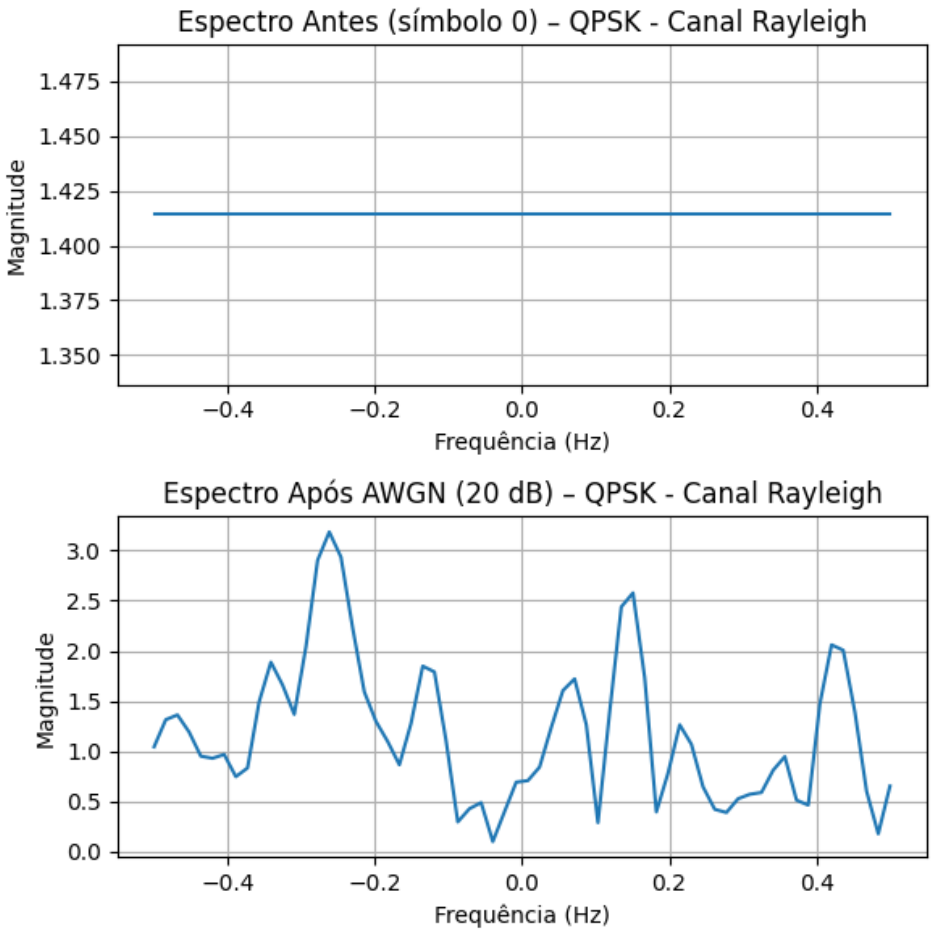

Figura 8: Espectro Antes (símbolo 0) e Pós-Equalização (SNR 20 dB) — QPSK, Canal Rayleigh.

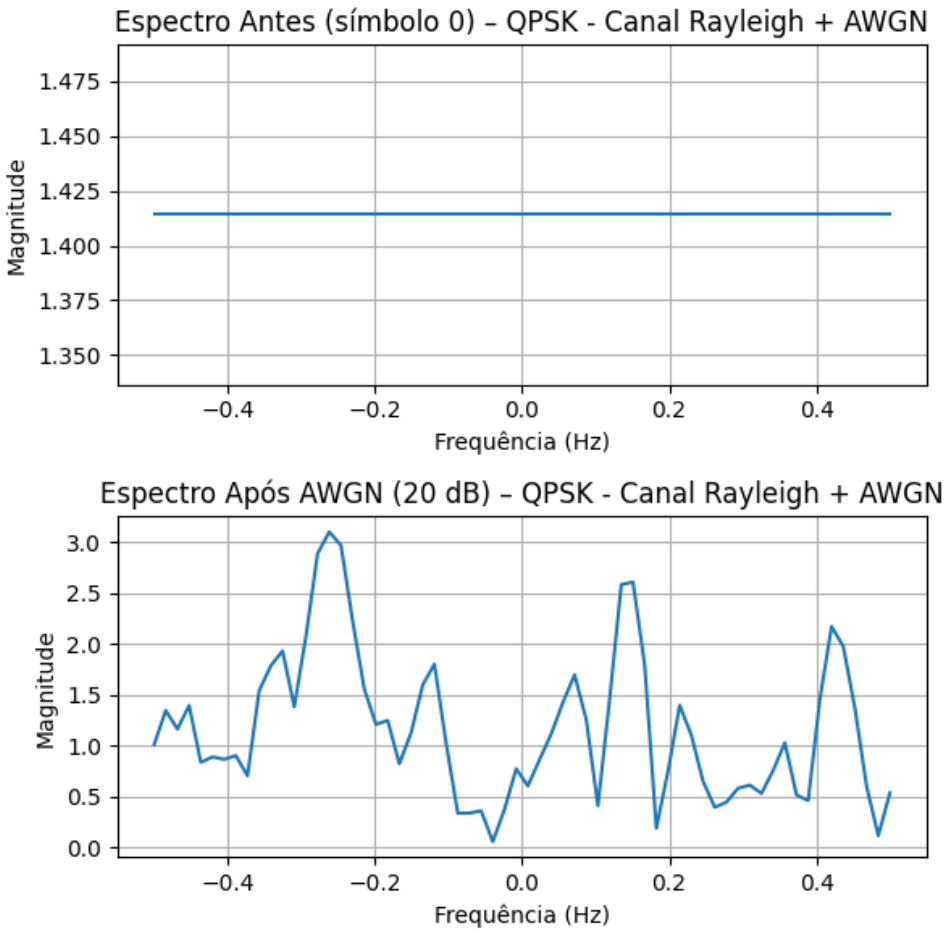

Figura 9: Espectro Antes (símbolo 0) e Após AWGN (SNR 20 dB) — QPSK, Canal Rayleigh + AWGN.

A seguir, apresentam-se os espectros de magnitude em função da frequência de um único símbolo OFDM, modulados em 64-QAM, nos canais AWGN puro, Rayleigh e Rayleigh + AWGN, antes e após a adição de ruído AWGN com SNR de –5 dB e 20 dB.

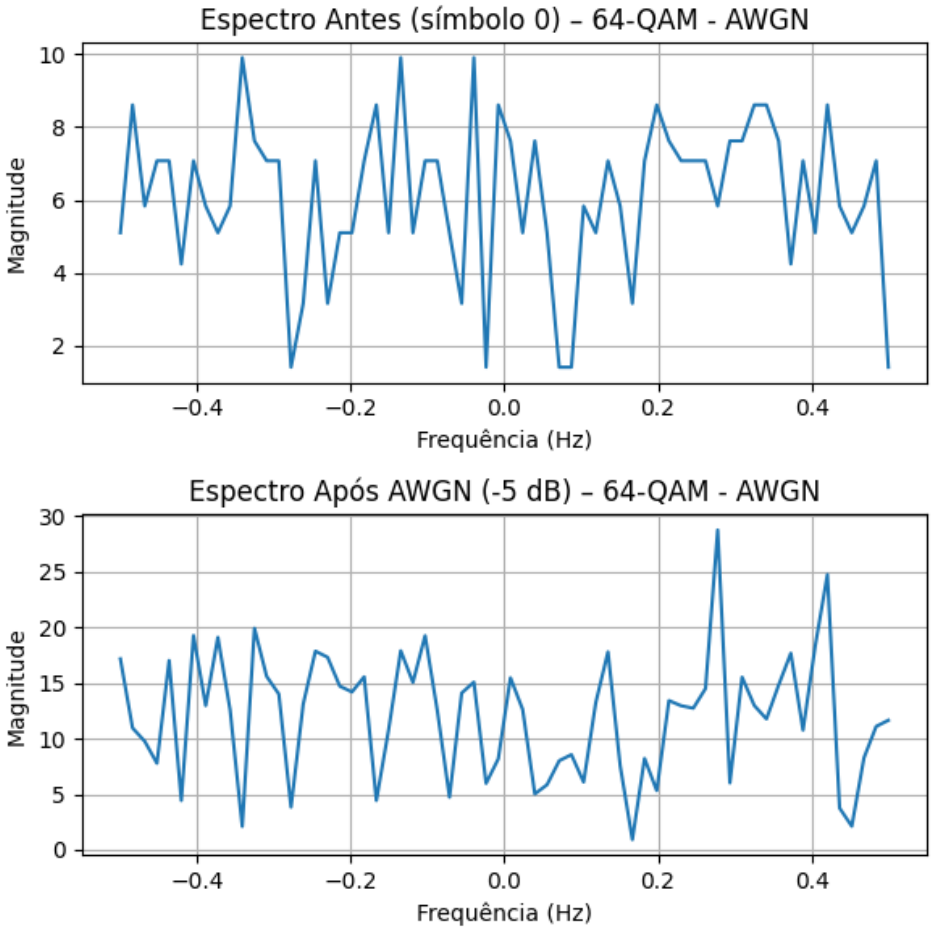

Figura 10: Espectro Antes (símbolo 0) e Após AWGN (SNR -5 dB) — 64-QAM, Canal AWGN.

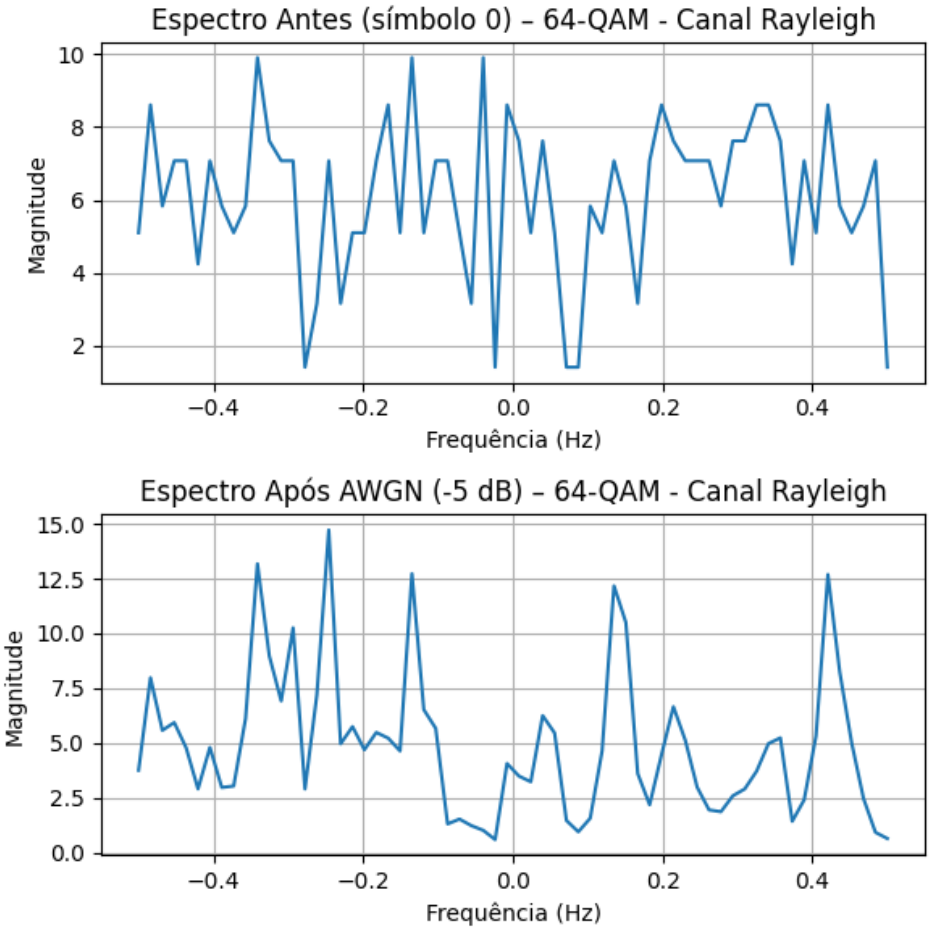

Figura 11: Espectro Antes (símbolo 0) e Pós-Equalização (SNR -5 dB) — 64-QAM, Canal Rayleigh.

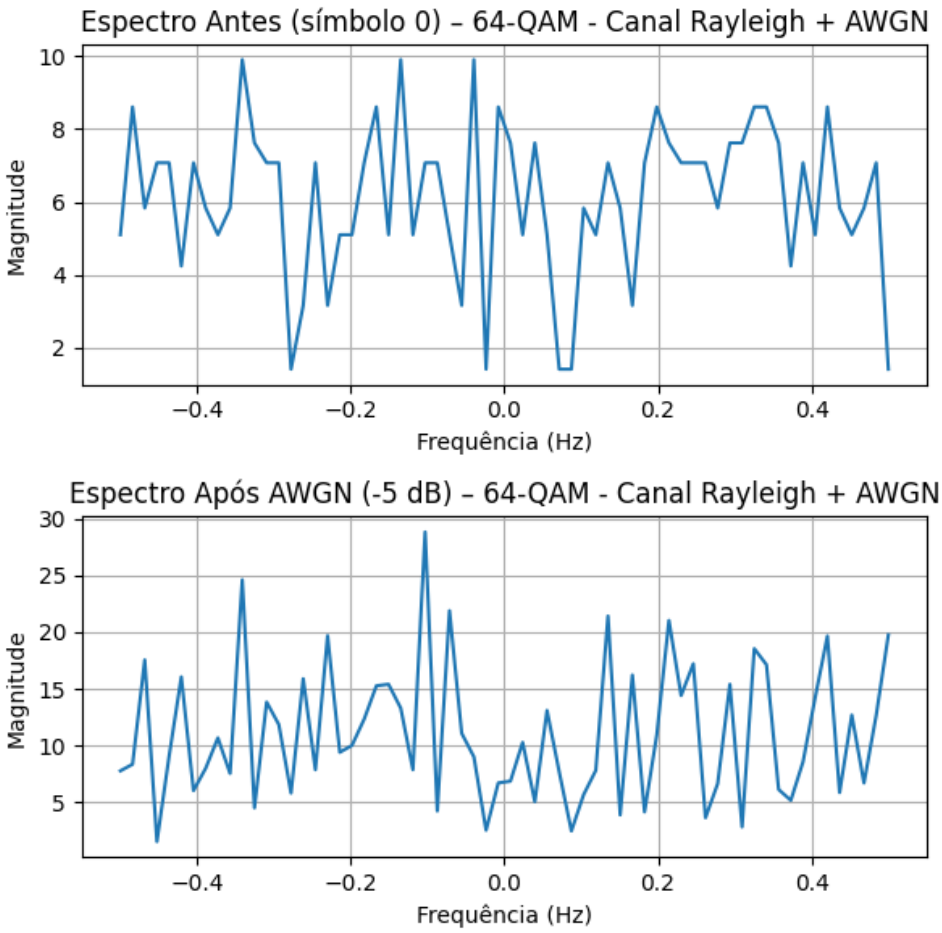

Figura 12: Espectro Antes (símbolo 0) e Após AWGN (SNR -5 dB) — 64-QAM, Canal Rayleigh + AWGN.

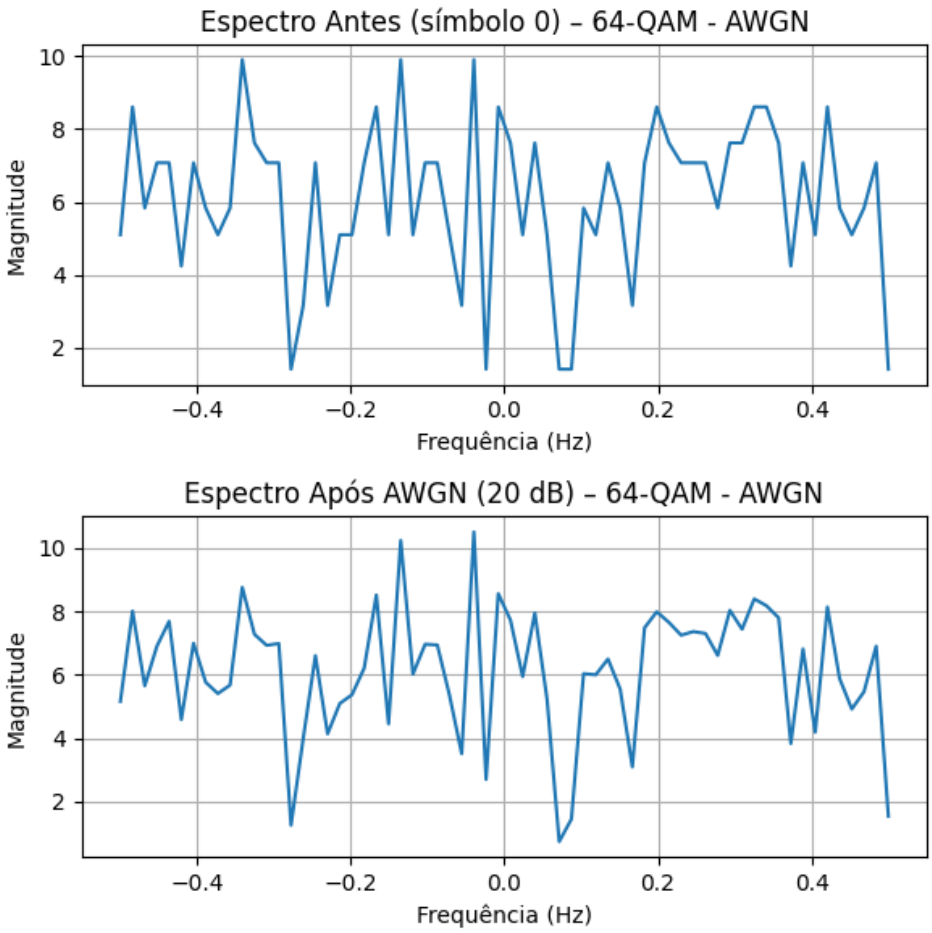

Figura 13: Espectro Antes (símbolo 0) e Após AWGN (SNR 20 dB) — 64-QAM, Canal AWGN.

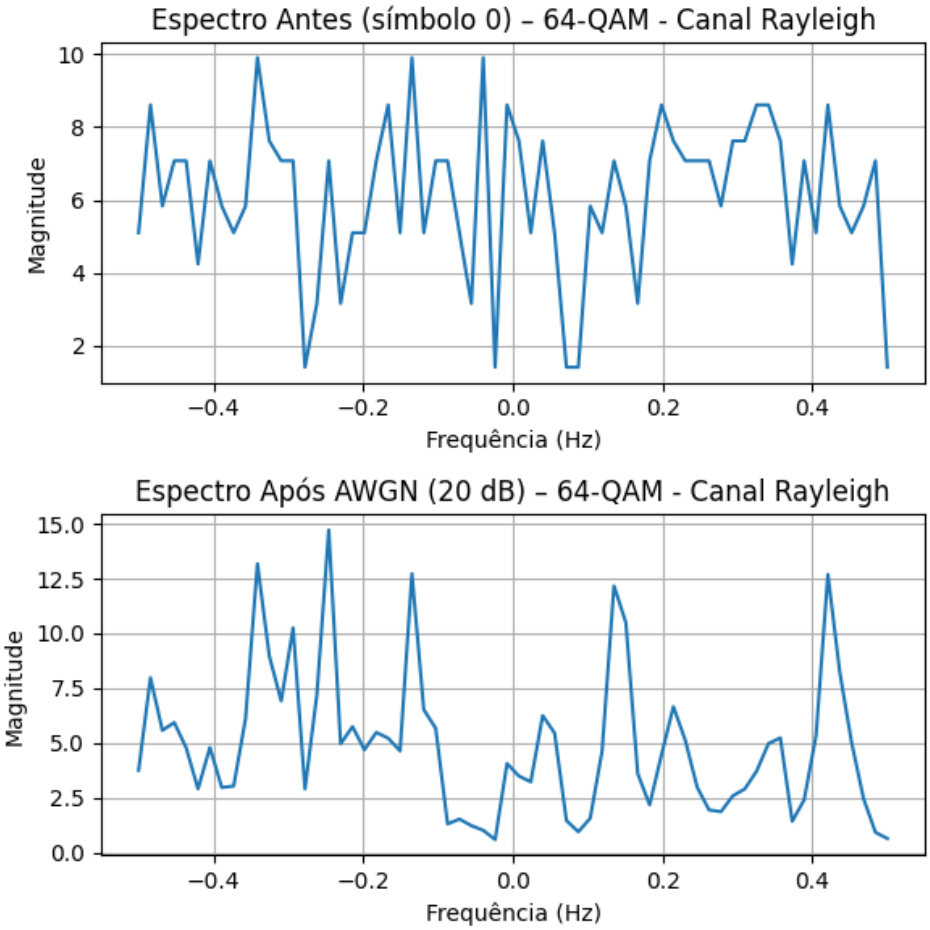

Figura 14: Espectro Antes (símbolo 0) e Pós-Equalização (SNR 20 dB) — 64-QAM, Canal Rayleigh.

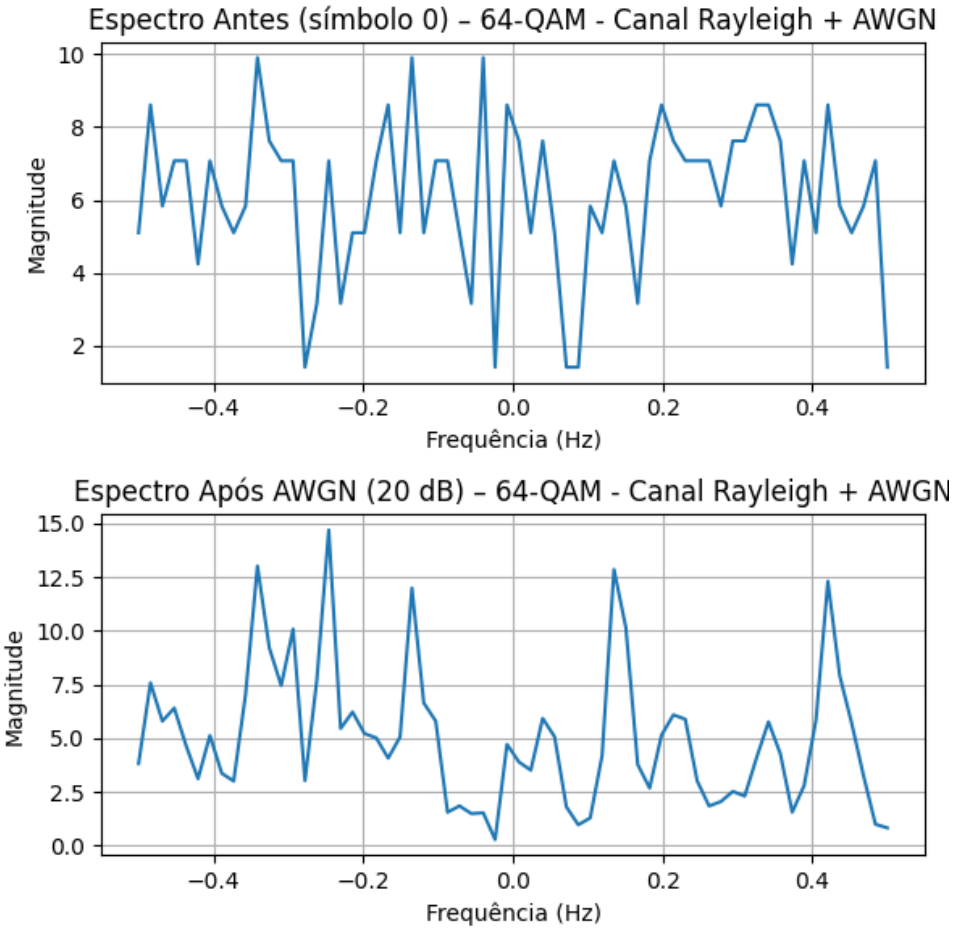

Figura 15: Espectro Antes (símbolo 0) e Após AWGN (SNR 20 dB) — 64-QAM, Canal Rayleigh + AWGN.

Ao acrescentar AWGN com boa relação sinal-ruído (20 dB), aparece um leve aumento da variabilidade entre as amostras. Com SNR ruim (–5 dB), o ruído eleva fortemente a linha de base do espectro e gera picos irregulares em várias frequências, degradando a coerência do sinal. No canal Rayleigh puro surgem quedas acentuadas no sinal e picos devido ao desvanecimento seletivo em frequência, e quando se combina Rayleigh + AWGN essas distorções se somam: a linha de base do espectro fica ainda mais elevado, as variações de magnitude tornam-se muito mais evidentes, e o ruído espalha energia por todo o espectro. Essas observações mostram como ruído e fading prejudicam a eficiência espectral do OFDM.

Os espectros QPSK são mais uniformes antes e após o ruído, porque todos os símbolos têm a mesma amplitude, então mesmo com ruído ou fading o nível de ruído do espectro só sobe um pouco e mantém o formato. Já nos gráficos de 64-QAM é possível visualizar variações de amplitude bem maiores desde o início (o traçado oscila mais) e, quando há ruído fraco (–5 dB) ou fading, a linha de base do espectro sobe muito e surgem picos irregulares. Dessa forma, QPSK gera espectros mais estáveis, enquanto o 64-QAM já começa com um perfil mais desigual e piora muito mais com ruído e desvanecimento.

A seguir, apresentam-se os gráficos de taxa de erro de bit (BER) em função do SNR para as modulações QPSK, 16-QAM e 64-QAM nos canais AWGN, Rayleigh e Rayleigh+AWGN, comparando os diferentes comprimentos de prefixo cíclico.

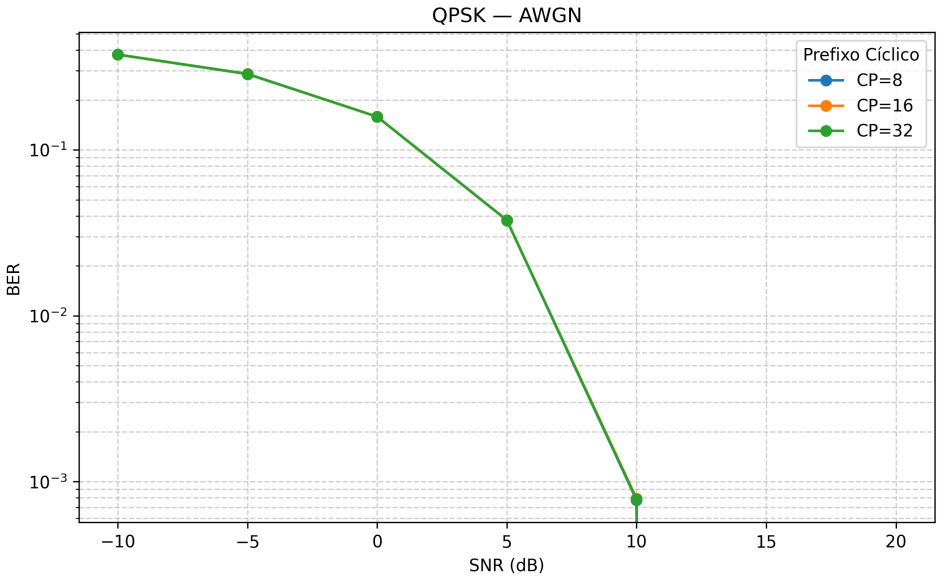

Figura 16: QPSK — Canal AWGN, comparação de BER para diferentes comprimentos de prefixo cíclico (CP).

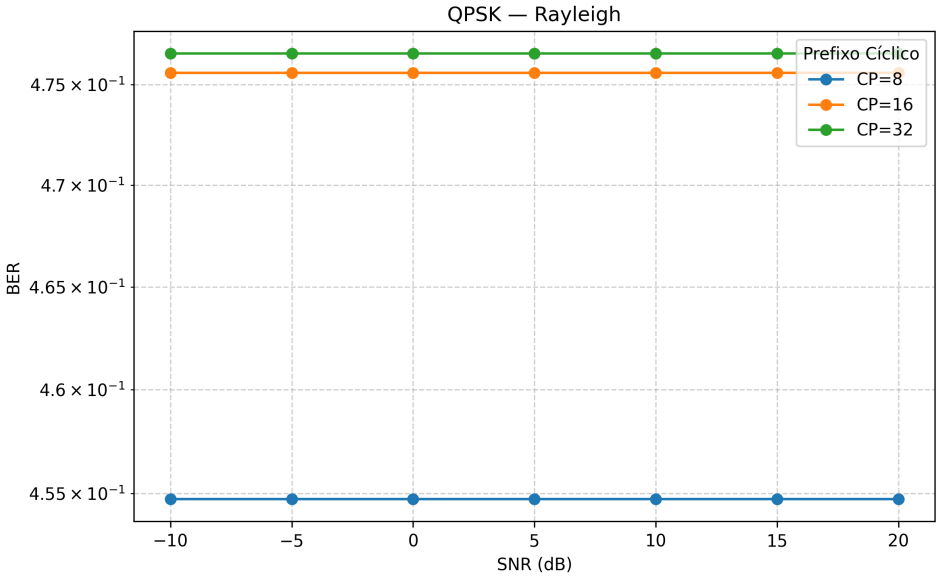

Figura 17: QPSK — Canal Rayleigh puro, comparação de BER para diferentes prefixos cíclicos.

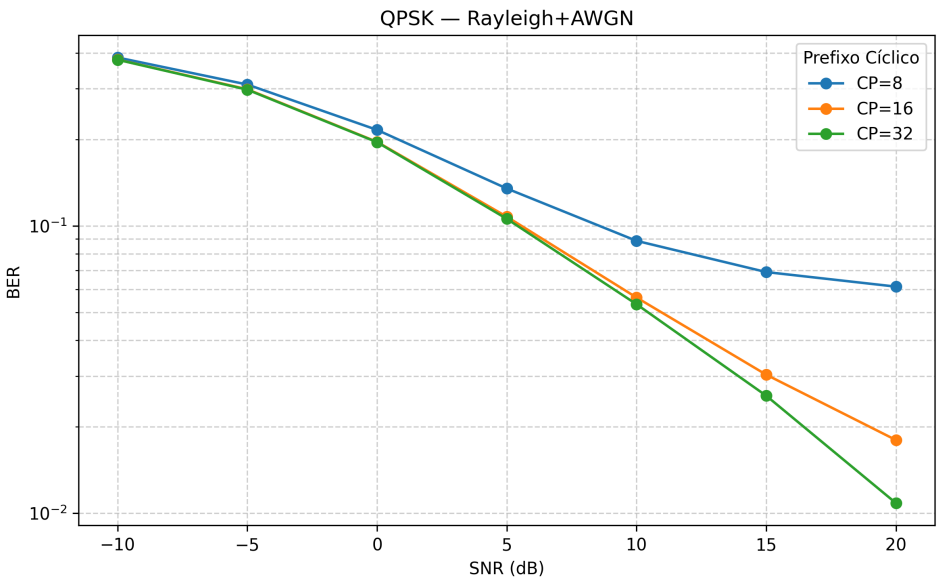

Figura 18: QPSK — Canal Rayleigh + AWGN, comparação de BER para diferentes prefixos cíclicos.

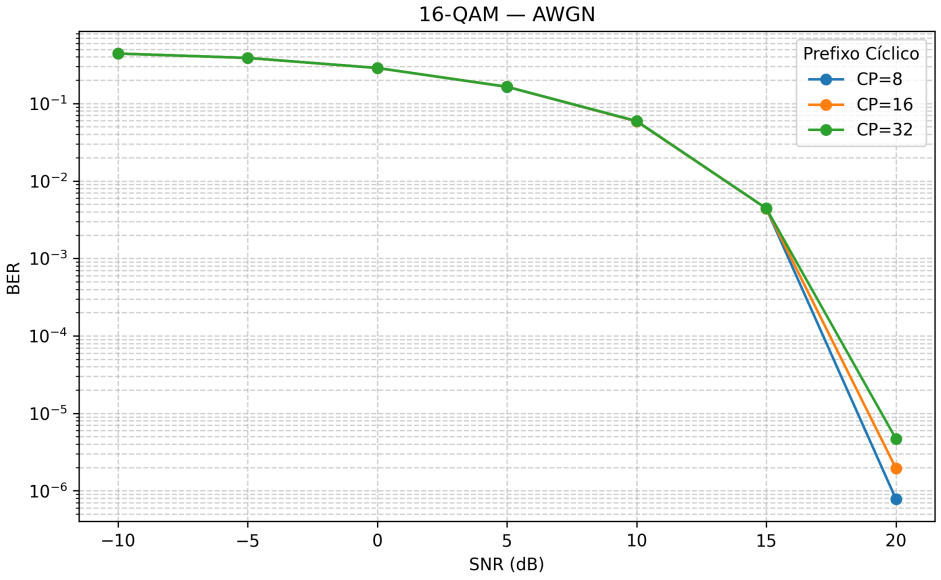

Figura 19: 16-QAM — Canal AWGN, comparação de BER para diferentes prefixos cíclicos.

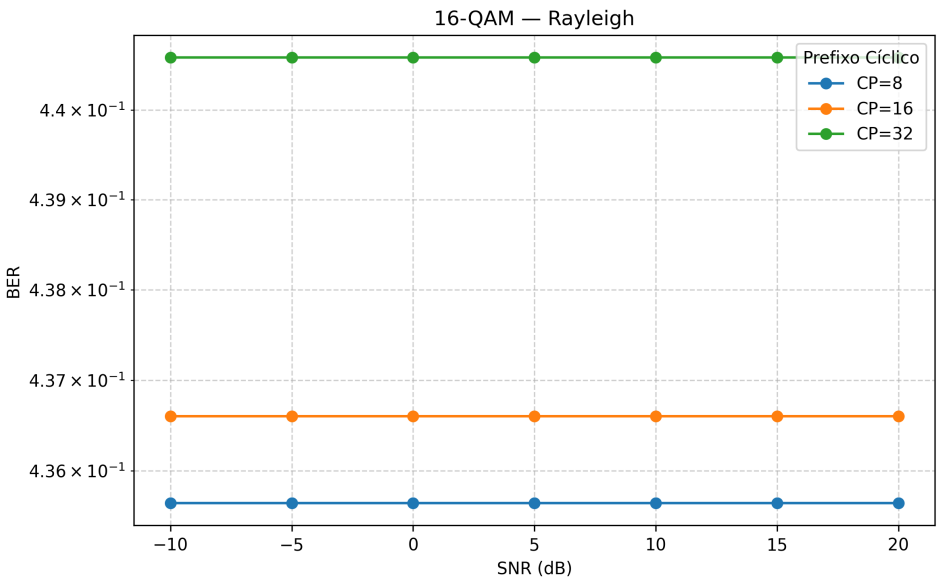

Figura 20: 16-QAM — Canal Rayleigh puro, comparação de BER para diferentes prefixos cíclicos.

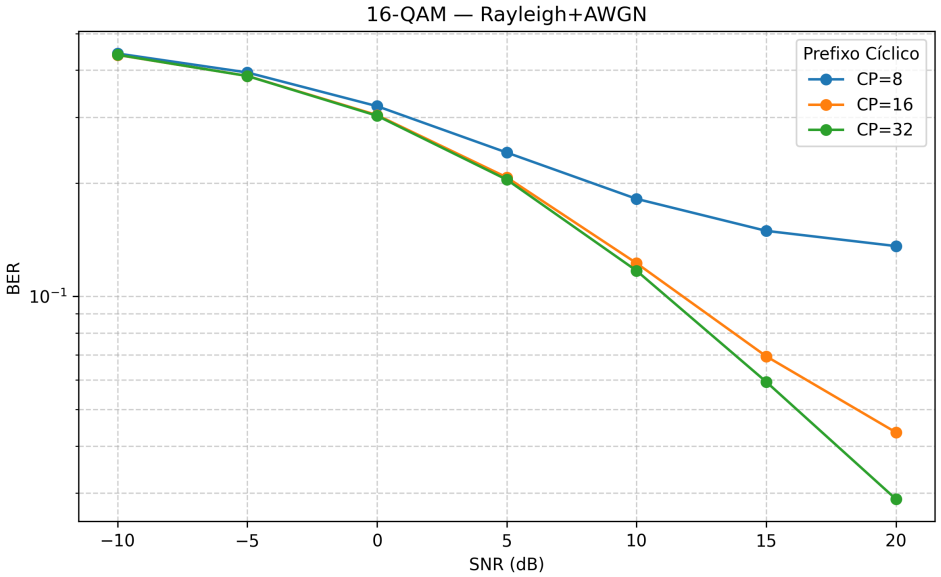

Figura 21: 16-QAM — Canal Rayleigh + AWGN, comparação de BER para diferentes prefixos cíclicos.

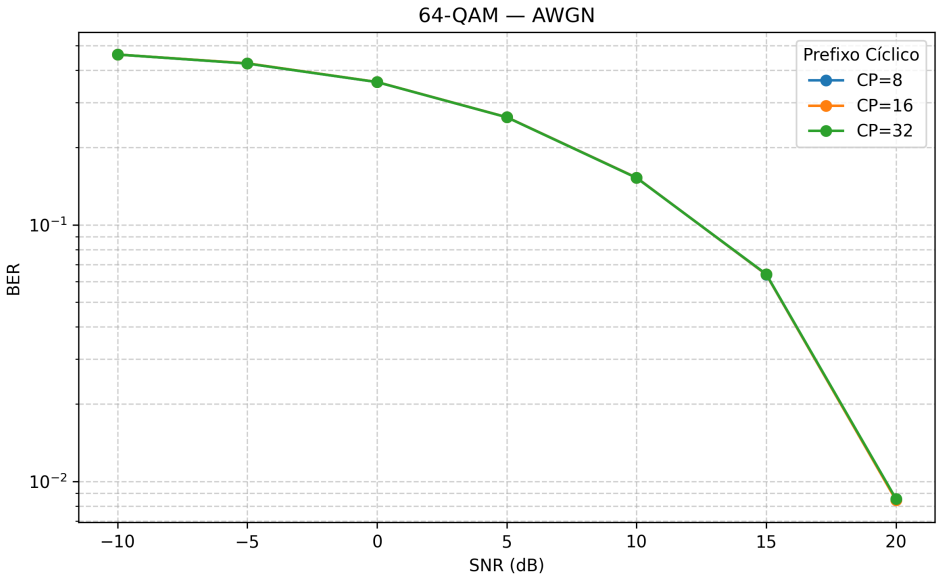

Figura 22: 64-QAM — Canal AWGN, comparação de BER para diferentes prefixos cíclicos.

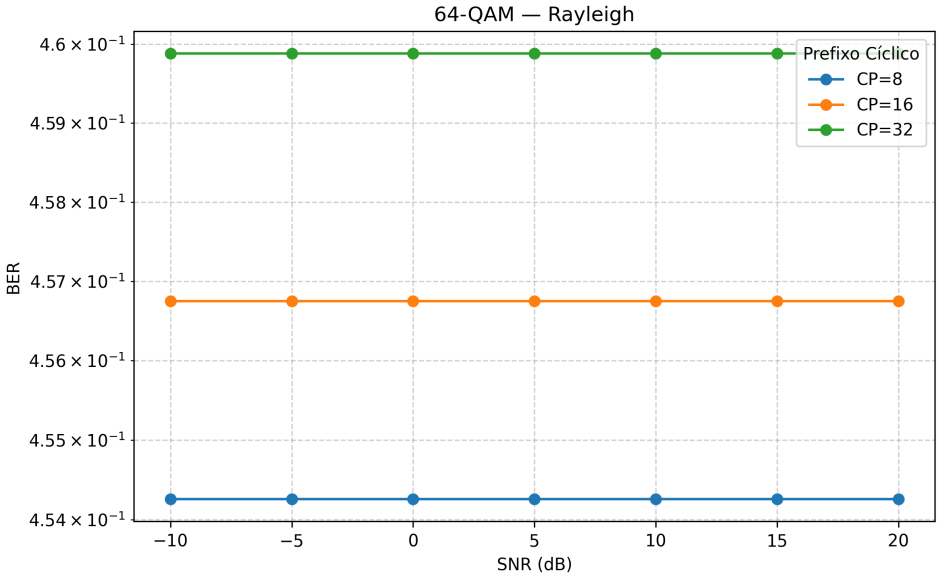

Figura 23: 64-QAM — Canal Rayleigh puro, comparação de BER para diferentes prefixos cíclicos.

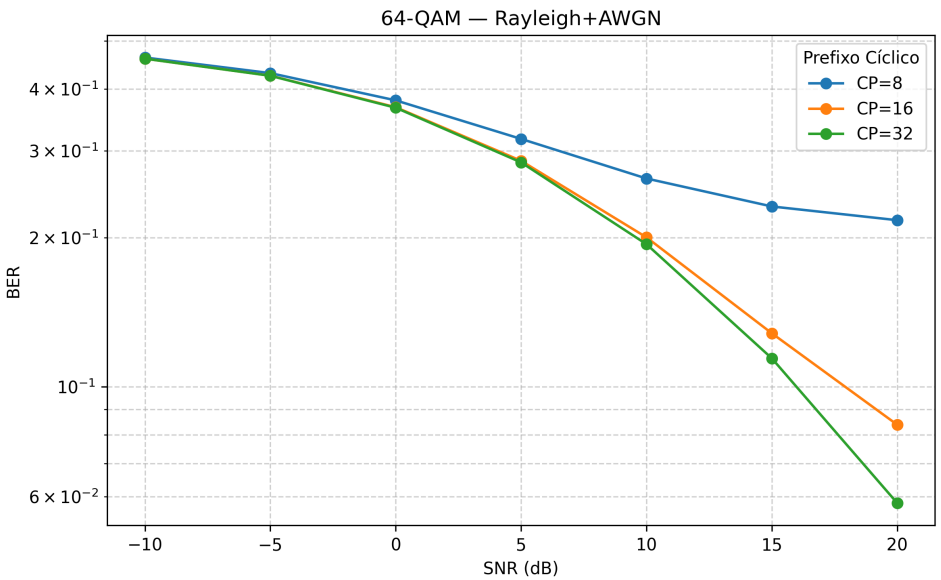

Figura 24: 64-QAM — Canal Rayleigh + AWGN, comparação de BER para diferentes prefixos cíclicos.

Em um ambiente com apenas ruído branco, a taxa de erros decai rapidamente conforme se aumenta a relação sinal-ruído, mas modulações com constelações mais densas (como 16-QAM ou 64-QAM) sofrem mais cedo a perda de robustez em comparação à QPSK. O comprimento do prefixo cíclico não altera esse comportamento, pois não há dispersão temporal a ser combatida.

No canal Rayleigh puro, sem ruído branco, a potência instantânea do sinal sofre flutuações significativas, com fortes atenuações aleatórias que podem praticamente zerar o símbolo recebido; por isso, a BER permanece quase inalterada mesmo ao aumentar a potência média, já que não há ruído a ser reduzido.

No cenário combinado Rayleigh+AWGN, além do ruído branco há dispersão temporal causada por múltiplos caminhos que chegam com atrasos diferentes, gerando interferência entre símbolos (ISI) quando o prefixo cíclico não é longo o bastante; por isso, observa-se que CP=8 apresenta BER mais alta em todos os SNRs, enquanto CP=16 e CP=32 reduzem significativamente os erros e chegam a quase se sobrepor, indicando que um prefixo cíclico moderado já é suficiente para acomodar o atraso de multipercurso simulado e restaurar a ortogonalidade dos subportadores.

Nessa próxima etapa, um código de repetição com taxa R=1/3 será incluído. O passo a passo do código com as diferenças para o código sem repetição é mostrado abaixo. O mesmo raciocínio é aplicado tanto no QPSK quanto M-QAM.

1. Cada bit é repetido r=3 vezes antes da modulação:

```
bits       = np.random.randint(0, 2, size=(k, total))
bits_rep   = np.repeat(bits, r, axis=1)
```

2. Após repetição, a matriz de símbolos tem dimensão $N \times \frac{\texttt{total} \times r}{N}$:

```
sym = sym.reshape((N, -1), order='F')
```

3. Após a demodulação hard, cada bit original foi repetido r vezes. O vetor `b1_rep` contém essas repetições em sequência, por exemplo:

$$\underbrace{b, b, \ldots, b}_{r \text{ vezes}}, \underbrace{b', b', \ldots, b'}_{r \text{ vezes}}, \ldots$$

```
# Ajusta tamanho para múltiplos de r e reshape para blocos de
    r repetições
valid_len    = (b1_rep.size // r) * r
b1_rep       = b1_rep[:valid_len].reshape(-1, r)
b2_rep       = b2_rep[:valid_len].reshape(-1, r)

# Decisão por maioria: cada linha soma as r repetições e
   compara a r/2
b1_dec       = (np.sum(b1_rep, axis=1) > r/2).astype(int)
b2_dec       = (np.sum(b2_rep, axis=1) > r/2).astype(int)

# Bits de referência originais (sem repetição)
b1_ref       = bits[0, :len(b1_dec)]
b2_ref       = bits[1, :len(b2_dec)]

# Conta erros e calcula BER
err          = np.sum(b1_dec != b1_ref) + np.sum(b2_dec !=
   b2_ref)
total_bits   = b1_ref.size + b2_ref.size
ber_rep.append(err / total_bits)
```

- **Ajuste de comprimento:** `valid_len` $= \lfloor \frac{\texttt{b1\_rep.size}}{r} \rfloor \times r$ garante que o vetor de bits repetidos tenha tamanho múltiplo de r, evitando sobras ao fazer o `reshape`.
- **Agrupamento em blocos:** O comando `reshape(-1, r)` transforma o vetor em uma matriz com $\frac{\texttt{valid\_len}}{r}$ linhas e r colunas, onde cada linha contém as r repetições de um bit original.
- **Decisão por maioria:** `np.sum(b1_rep, axis=1)` conta quantos dos r bits são 1; se essa soma exceder $r/2$, decide-se 1, caso contrário 0. O mesmo vale para `b2_rep`.
- **Seleção dos bits de referência:** `b1_ref` e `b2_ref` extraem dos bits originais (antes da repetição) as primeiras `len(b1_dec)` e `len(b2_dec)` posições, garantindo correspondência exata.
- **Cálculo da BER:** Soma-se os erros em ambas as sequências e divide-se pelo total de bits comparados, armazenando o resultado na lista `ber_rep`.

As Figuras a seguir apresentam, para cada esquema de modulação (QPSK, 16-QAM e 64-QAM) com e sem código de repetição $r = 3$, as curvas de BER em função do SNR sob três cenários de canal distintos: AWGN puro, Rayleigh puro e Rayleigh+AWGN.

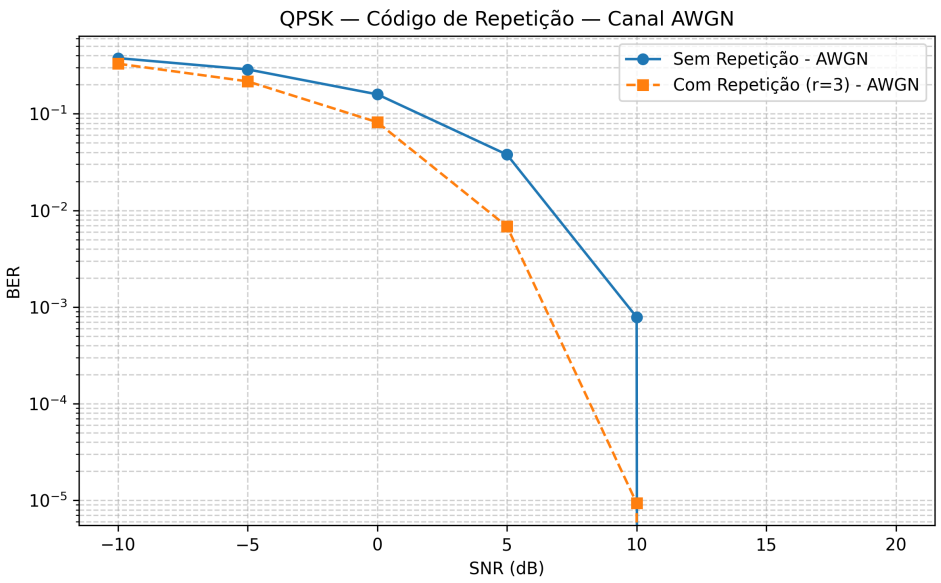

Figura 25: QPSK — Código de Repetição (r=3) — Canal AWGN.

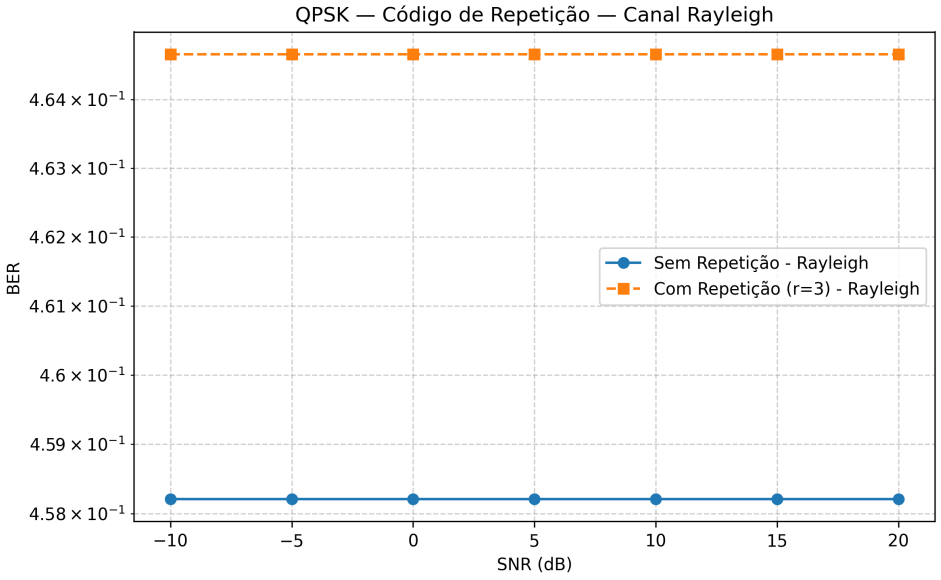

Figura 26: QPSK — Código de Repetição (r=3) — Canal Rayleigh puro.

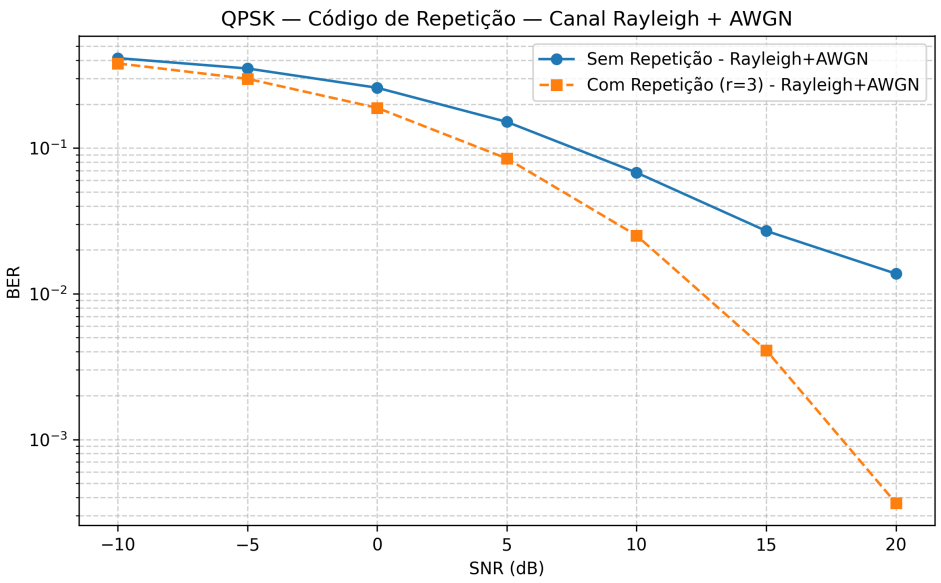

Figura 27: QPSK — Código de Repetição (r=3) — Canal Rayleigh + AWGN.

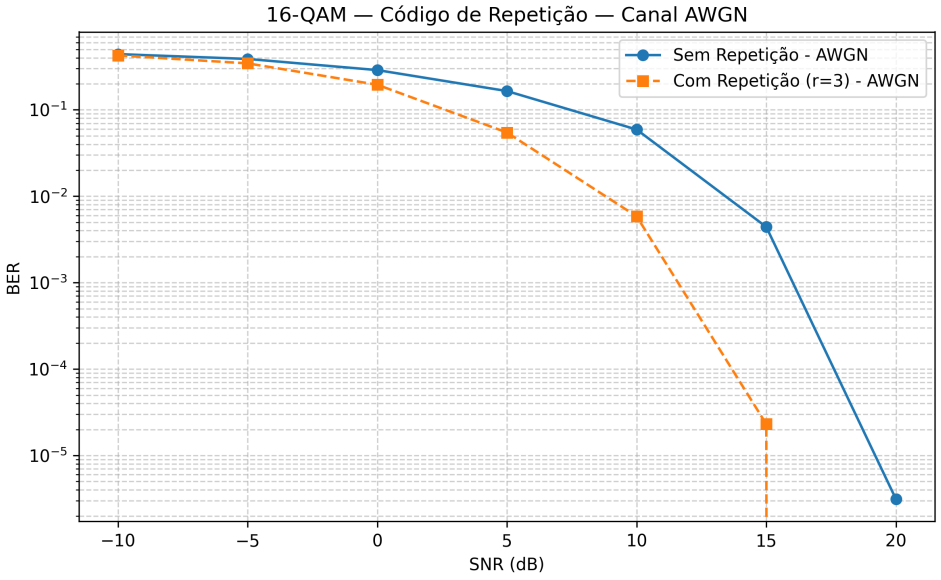

Figura 28: 16-QAM — Código de Repetição (r=3) — Canal AWGN.

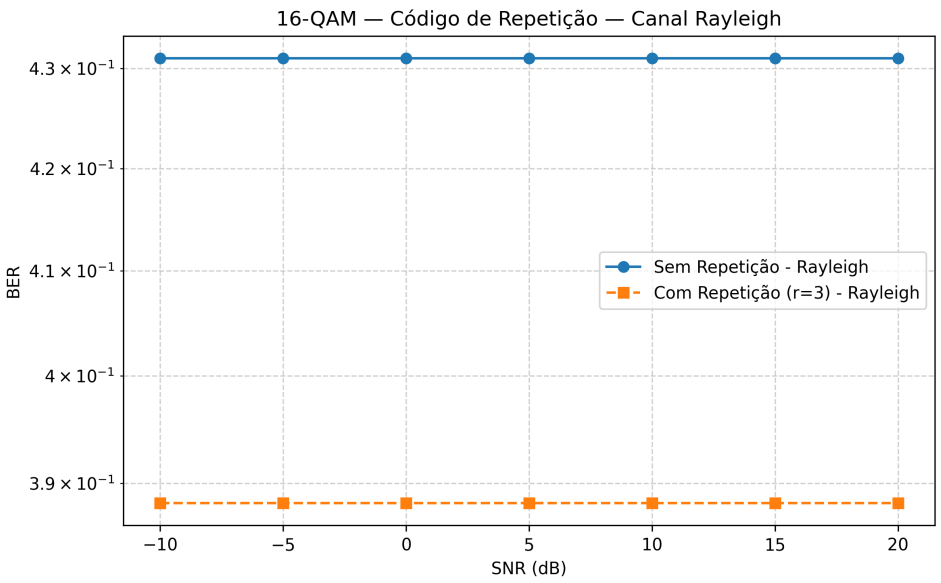

Figura 29: 16-QAM — Código de Repetição (r=3) — Canal Rayleigh puro.

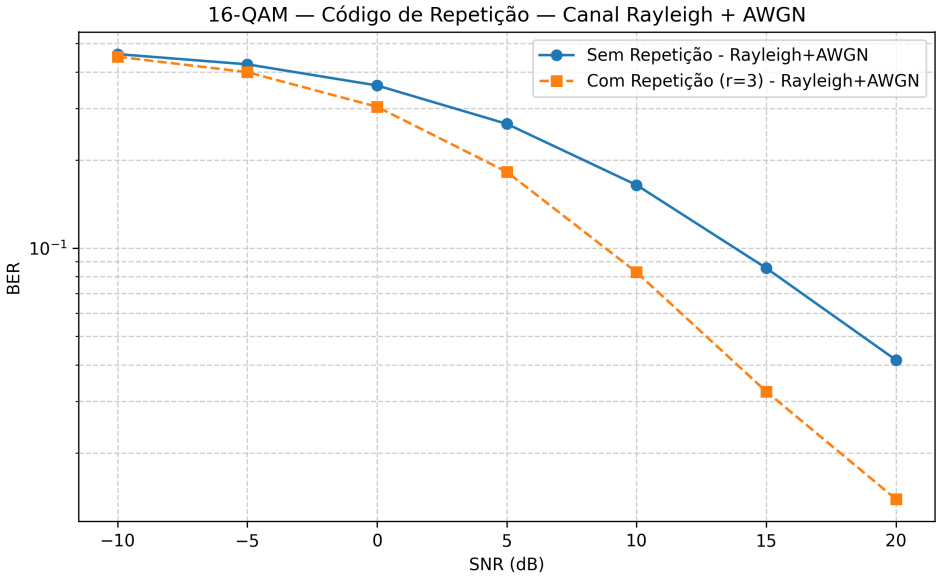

Figura 30: 16-QAM — Código de Repetição (r=3) — Canal Rayleigh + AWGN.

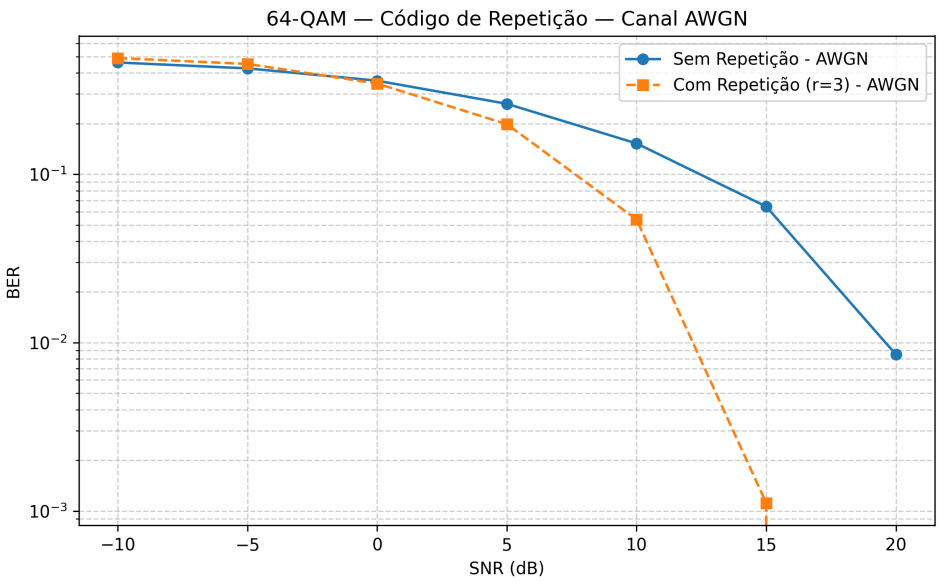

Figura 31: 64-QAM — Código de Repetição (r=3) — Canal AWGN.

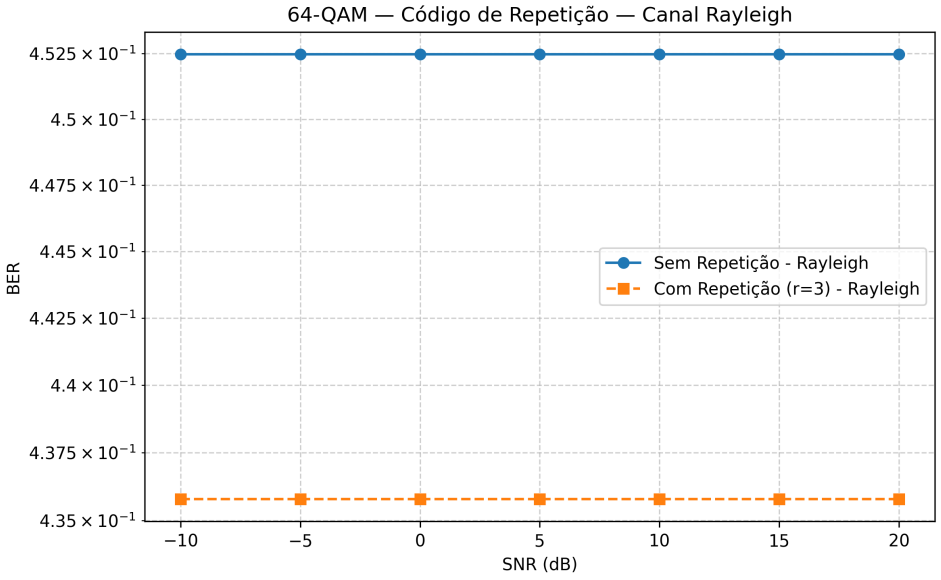

Figura 32: 64-QAM — Código de Repetição (r=3) — Canal Rayleigh puro.

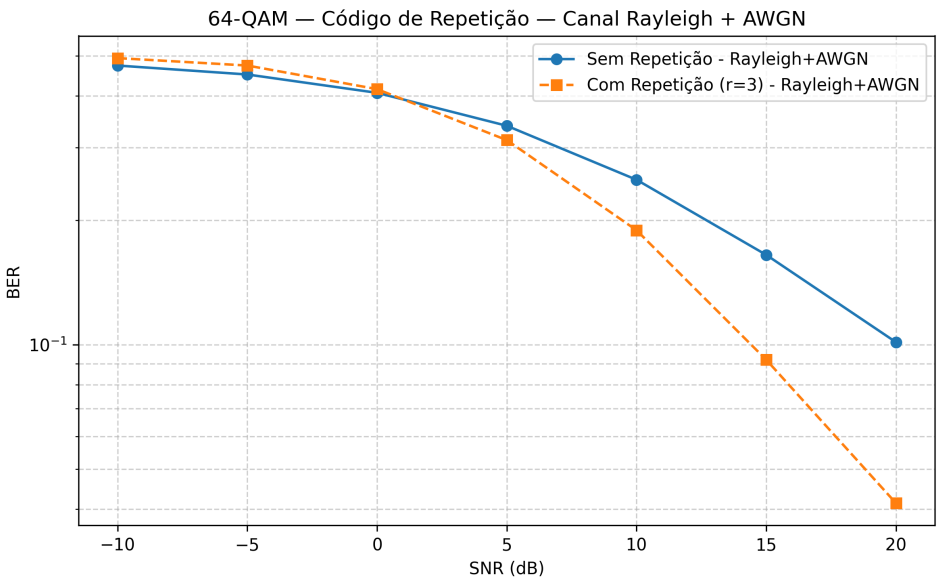

Figura 33: 64-QAM — Código de Repetição (r=3) — Canal Rayleigh + AWGN.

O código de repetição (r=3) oferece um ganho de confiabilidade maior em canais onde o ruído domina (AWGN) ou onde há tanto desvanecimento quanto ruído (Rayleigh+AWGN): para um mesmo SNR, a curva com repetição está consistentemente abaixo da sem repetição. Em contraste, no canal Rayleigh puro — dominado por perdas e sem componente de ruído — a repetição sozinha não melhora a BER, pois o erro é imposto pelo fading e não pelo ruído aleatório. Além disso, como a curva de Rayleigh+AWGN fica sempre acima da do AWGN puro para o mesmo SNR, seu desempenho é inferior. Isso já é esperado, pois além do ruído, é preciso superar as atenuações aleatórias do fading.

Ao comparar QPSK, 16-QAM e 64-QAM com código de repetição, é possível notar que quanto maior a ordem da modulação maior é o SNR necessário para obter o mesmo benefício de repetição: em AWGN puro, a vantagem da repetição desloca a curva de QPSK algumas décadas de BER para baixo em SNRs moderados, mas em 16-QAM essa melhoria só emerge em SNRs mais altos, e em 64-QAM exige ainda mais potência para aparecer; de forma semelhante, em Rayleigh+AWGN a repetição corrige erros de QPSK já em SNRs médios, mas para 16-QAM e 64-QAM a melhoria só ocorre em SNRs crescentemente maiores, refletindo que constelações mais densas são mais vulneráveis ao ruído e ao fading e demandam maior SNR para para uma BER menor.

Códigos Utilizados

A seguir, apresentam-se os códigos em Python utilizados no relatório.

Desafios

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
```

```
matplotlib.use('TkAgg')
from commpy.modulation import QAMModem

# ------------------ Configuração de Parâmetros
    ------------------
# Parâmetros OFDM
N, CP, S = 64, 16, 10000            # Número de subportadoras
    , comprimento do prefixo cíclico, símbolos OFDM
SNRs_dB  = np.arange(-10, 21, 5) # Faixa de SNR em dB
total    = N * S                  # Total de símbolos
    transmitidos
L =20                             # número de taps do canal
    Rayleigh
# --- Canal Rayleigh puro ---
# h[n] = (hR + j hI)/sqrt(2), tamanho L
h = (np.random.randn(L) + 1j*np.random.randn(L)) / np.sqrt(2*
    L)

# 1) Simulação OFDM com QPSK
# ------------------ 1) Simulação OFDM com QPSK
    ------------------
def simulate_ofdm_qpsk():
    k = 2  # bits por símbolo QPSK
    # Passo 2: gera bits aleatórios (2 linhas    total
       colunas)
    bits = np.random.randint(0, 2, size=(k, total))

    # Passo 3: mapeamento QPSK: s = (1 - 2*d1) + j*(1 - 2*d2)
    sym = (1 - 2*bits[0]) + 1j*(1 - 2*bits[1])
    # Aqui faz o S/P: converte esse vetor serial em uma
       matriz N S ,
    # onde cada coluna é um símbolo OFDM paralelo de N sub-
       portadoras.
    # Serial-to-parallel: reorganiza em matriz N
       subportadoras x S símbolos
    sym = sym.reshape((N, S), order='F')  # reorganiza em
       matriz N S

    # Passo 4: IFFT para domínio do tempo
    tx = np.fft.ifft(sym, axis=0)

    # Passo 5: adiciona prefixo cíclico (últimas CP linhas) e
        serializa
    tx_cp = np.vstack([tx[-CP:], tx]).reshape(-1, order='F')
    # Estatísticas do sinal transmitido
       = np.mean(tx_cp)                                  # mé
       dia complexa de tx_cp
    signal_var = np.mean(np.abs(tx_cp -   )**2)          # var
       (tx_cp) = E[|  x    |  ]

    ber_awgn = [] # Lista para BER no canal AWGN
```

```
    ber_ray  = [] # Lista para BER no canal Rayleigh puro
    ber_ray_awgn = [] # Lista para BER no canal Rayleigh +
       AWGN

    for snr_db in SNRs_dB:
        # Passo 6: canal AWGN - adiciona ruído com variância
           adequada
        # Converte SNR de dB para razão linear e calcula vari
           ância de ruído
        snr = 10**(snr_db/10)
        noise_var = signal_var / snr
        # --- AWGN puro ---
        # Canal AWGN puro: adiciona ruído gaussiano complexo
        noise = np.sqrt(noise_var/2)*(np.random.randn(*tx_cp.
           shape) + 1j*np.random.randn(*tx_cp.shape))
        rx_awgn = tx_cp + noise


        # convolução tx_cp * h
        # Canal Rayleigh puro: convolução do sinal com
           resposta ao impulso do canal
        rx_ray = np.convolve(tx_cp, h, mode='full')[:tx_cp.
           size]

        # --- Rayleigh + AWGN ---
        # Canal Rayleigh + AWGN: aplica ruído ao sinal
           Rayleigh
        noise2 = np.sqrt(noise_var/2)*(np.random.randn(*
           rx_ray.shape)
                                       +1j*np.random.randn(*
                                          rx_ray.shape))
        rx_ray_awgn = rx_ray + noise2

        # função auxiliar de recebimento, FFT, demod e BER
        # Função interna para recepção, equalização e cálculo
            de BER
        def ofdm_recv(s,type):
            # Passo 7: paraleliza e remove prefixo cíclico
            # Remove prefixo cíclico e converte para matriz N
                x S
            mat = s.reshape((N+CP, S), order='F')[CP:,:]
            # Passo 8: FFT de volta para o domínio da freguê
               ncia e serializa
            # sempre faz FFT
            # FFT para voltar ao domínio da frequência
            Y = np.fft.fft(mat, axis=0)   # N S

            # or type == 'ray'
            # Equalização se canal Rayleigh
            if type == 'awgn' or type == 'ray':
```

```
            # AWGN puro      não tem fading, basta
              serializar
            y = Y.reshape(-1, order='F') # Sem equalizaçã
              o para AWGN ou Rayleigh sem ruído
        else:
            # Rayleigh+AWGN      equaliza
            H_fft = np.fft.fft(h, N) # FFT do canal
            # evita divisões por valores muito pequenos
            H_fft[np.abs(H_fft) < 1e-3] = 1e-3 # Evita
              divisão por zero
            Y_eq = Y / H_fft[:,None] # Equalização no dom
              ínio da frequência
            y      = Y_eq.reshape(-1, order='F')


        # Passo 9: demodulação QPSK (decisão hard no
          sinal real/imaginário)
        b1  = (y.real<0).astype(int)
        b2  = (y.imag<0).astype(int)

        #      bloco de espectro símbolo 0
        # Cálculo e plotagem do espectro do primeiro sí
          mbolo
        freq_axis = np.linspace(-0.5, 0.5, N)

        t0  = tx[:, 0] # Símbolo OFDM transmitido
        r0  = mat[:, 0] # Símbolo OFDM recebido
        T0  = np.fft.fft(t0, N)
        R0  = np.fft.fft(r0, N)

        if type == "awgn":
            text = "AWGN"
        if type == "ray":
            text = "Canal Rayleigh"
        if type == "ray_awgn":
            text = "Canal Rayleigh + AWGN"

        plt.figure(figsize=(6,6))
        plt.subplot(2,1,1)
        plt.plot(freq_axis, np.abs(np.fft.fftshift(T0)))
        plt.title(f'Espectro Antes (símbolo 0)     QPSK -
            {text}')
        plt.xlabel('Frequência (Hz)')
        plt.ylabel('Magnitude')
        plt.grid(True)

        plt.subplot(2,1,2)
        plt.plot(freq_axis, np.abs(np.fft.fftshift(R0)))
        plt.title(f'Espectro Após AWGN ({snr_db} dB)
          QPSK - {text}')
        plt.xlabel('Frequência (Hz)')
```

```
            plt.ylabel('Magnitude')
            plt.grid(True)

            plt.tight_layout()
            plt.savefig(f'espectro_qpsk_{snr_db}dB_{type}.png
               ')
            plt.close()
            #     fim bloco

            # Passo 10: calcula BER comparando com bits
               transmitidos
            # Compara bits estimados com originais para
               calcular BER
            errs = np.sum(bits != np.vstack([b1,b2]))
            return errs/(bits.size)

        # Executa recepção para cada tipo de canal
        ber_awgn.append(ofdm_recv(rx_awgn,'awgn'))
        ber_ray.append(ofdm_recv(rx_ray,'ray'))
        ber_ray_awgn.append(ofdm_recv(rx_ray_awgn,'ray_awgn')
           )


    return ber_awgn, ber_ray, ber_ray_awgn


# 2) Simulação genérica M-QAM usando CommPy QAMModem
def simulate_ofdm_qam(M):
    modem = QAMModem(M)                  # inicializa modem
       QAM
    k     = modem.num_bits_symbol        # bits por símbolo
       QAM

    # Passo 2: gera vetor de bits aleatório de comprimento k*
       total
    # Gera um vetor serial de bits aleatórios e converte em
       paralelo (serial-to-parallel)
    bits = np.random.randint(0, 2, size=(k, total)).reshape
       (-1, order='F')

    # Passo 3: modula bits em símbolos QAM e paraleliza
    # Modula bits em símbolos QAM e reorganiza em matriz N
       subportadoras    S símbolos
    sym = modem.modulate(bits).reshape((N, S), order='F')

    # Passo 4: IFFT
    # IFFT: converte os símbolos modulados para o domínio do
       tempo
    tx = np.fft.ifft(sym, axis=0)
```

```
    # Passo 5: adiciona prefixo cíclico (as últimas CP
       amostras) e serializa
    tx_cp = np.vstack([tx[-CP:], tx]).reshape(-1, order='F')
    # Estatísticas do sinal transmitido: média e variância
       = np.mean(tx_cp)                                    # mé
       dia complexa de tx_cp
    signal_var = np.mean(np.abs(tx_cp -   )**2)           # var
       (tx_cp) = E[|  x   |  ]  # variância do sinal

    # Listas para armazenar BER (Bit Error Rate) em cada tipo
        de canal
    ber_awgn = [] # para canal AWGN puro
    ber_ray  = [] # para canal Rayleigh puro
    ber_ray_awgn = [] # para canal Rayleigh + AWGN

    # Varre cada nível de SNR em dB
    for snr_db in SNRs_dB:
        # Passo 6: canal AWGN
        # Converte SNR de dB para escala linear e define vari
           ância do ruído
        snr = 10**(snr_db/10)
        noise_var = signal_var / snr
        # --- AWGN puro --- ---: adiciona ruído gaussiano
           complexo
        noise = np.sqrt(noise_var/2)*(np.random.randn(*tx_cp.
           shape) + 1j*np.random.randn(*tx_cp.shape))
        rx_awgn = tx_cp + noise

        # convolução tx_cp * h ---: convolução com resposta
           ao impulso h
        rx_ray = np.convolve(tx_cp, h, mode='full')[:tx_cp.
           size]

        # --- Rayleigh + AWGN --- adiciona ruído ao sinal
           Rayleigh
        noise2 = np.sqrt(noise_var/2)*(np.random.randn(*
           rx_ray.shape)
                                       +1j*np.random.randn(*
                                          rx_ray.shape))
        rx_ray_awgn = rx_ray + noise2

        # função auxiliar de recebimento, FFT, demod e BER
        # Função interna para recepção, equalização e cálculo
            do BER
        def ofdm_recv(s,type):
            # Passo 7: paraleliza e remove prefixo cíclico e
               retorna matriz N S
            mat = s.reshape((N+CP, S), order='F')[CP:,:]
            # Passo 8: FFT de volta para o domínio da frequê
               ncia e serializa
            # sempre faz FFT
```

```
            # FFT para converter de volta ao domínio da frequ
                ência
            Y = np.fft.fft(mat, axis=0)    # N S

            # or type == 'ray'
            # Equalização apenas nos casos de Rayleigh+AWGN
            if type == 'awgn' or type == 'ray':
                # AWGN puro     não tem fading, basta
                    serializar
                y = Y.reshape(-1, order='F') # sem equalizaçã
                    o
            else:
                # Rayleigh+AWGN     equaliza
                H_fft = np.fft.fft(h, N) # FFT do canal
                # evita divisões por valores muito pequenos
                H_fft[np.abs(H_fft) < 1e-3] = 1e-3 # evita
                    divisão por zero
                Y_eq = Y / H_fft[:,None]  # equaliza em cada
                    subportadora
                y     = Y_eq.reshape(-1, order='F')


            # Passo 9: demodulação por decisão hard
            bits_hat = modem.demodulate(y, 'hard')

            #     bloco de espectro símbolo 0
            freq_axis = np.linspace(-0.5, 0.5, N)

            t0  = tx[:, 0]
            r0  = mat[:, 0]
            T0  = np.fft.fft(t0, N)
            R0  = np.fft.fft(r0, N)

            if type == "awgn":
                text = "AWGN"
            if type == "ray":
                text = "Canal Rayleigh"
            if type == "ray_awgn":
                text = "Canal Rayleigh + AWGN"

            plt.figure(figsize=(6,6))
            plt.subplot(2,1,1)
            plt.plot(freq_axis, np.abs(np.fft.fftshift(T0)))
            plt.title(f'Espectro Antes (símbolo 0)     {M}-
                QAM - {text}')
            plt.xlabel('Frequência (Hz)'); plt.ylabel('
                Magnitude'); plt.grid(True)

            plt.subplot(2,1,2)
            plt.plot(freq_axis, np.abs(np.fft.fftshift(R0)))
```

```
            plt.title(f'Espectro Após AWGN ({snr_db} dB)
               {M}-QAM - {text}')
            plt.xlabel('Frequência (Hz)'); plt.ylabel('
               Magnitude'); plt.grid(True)

            plt.tight_layout()
            plt.savefig(f'espectro_{M}qam_{snr_db}dB_{type}.
               png')
            plt.close()
            #     fim bloco

            # Passo 10: calcula BER comparando com vetor de
               bits transmitidos
            # Cálculo do BER comparando bits estimados vs.
               originais
            errs = np.sum(bits != bits_hat)
            return errs / bits.size

        # Aplica ofdm_recv a cada canal e armazena o BER
           resultante
        ber_awgn.append(ofdm_recv(rx_awgn,'awgn'))
        ber_ray.append(ofdm_recv(rx_ray,'ray'))
        ber_ray_awgn.append(ofdm_recv(rx_ray_awgn,'ray_awgn')
           )

    # Retorna as três curvas de BER para os diferentes
       canais
    return ber_awgn, ber_ray, ber_ray_awgn

#Executa simulações para QPSK, 16-QAM e 64-QAM
# ------------------ Execução das simulações iniciais
   ------------------
BER_AWGN_QPSK, BER_RAY_QPSK, BER_RAY_AWGN_QPSK =
   simulate_ofdm_qpsk()
BER_AWGN_16QAM, BER_RAY_16QAM, BER_RAY_AWGN_16QAM =
   simulate_ofdm_qam(16)
BER_AWGN_64QAM, BER_RAY_64QAM, BER_RAY_AWGN_64QAM =
   simulate_ofdm_qam(64)

# ------------------ Geração de gráficos BER vs SNR
   ------------------
# 1) Canal AWGN
plt.figure(figsize=(8,5))
plt.semilogy(SNRs_dB, BER_AWGN_QPSK,  'o-', label='QPSK')
plt.semilogy(SNRs_dB, BER_AWGN_16QAM, 's-', label='16-QAM')
plt.semilogy(SNRs_dB, BER_AWGN_64QAM, '^-', label='64-QAM')
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('BER vs SNR     Canal AWGN')
# fixa os ticks em potências de 10
# yticks = [1, 1e-1, 1e-2, 1e-3, 1e-4,1e-5,1e-6]
```

```
# ylabels = [r'$10^0$', r'$10^{-1}$', r'$10^{-2}$', r'$10
    ^{-3}$', r'$10^{-4}$', r'$10^{-5}$',r'$10^{-6}$']
# plt.yticks(yticks, ylabels)
plt.grid(which='both', ls='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.savefig('ber_awgn.png')
plt.close()

# 2) Canal Rayleigh puro
plt.figure(figsize=(8,5))
plt.semilogy(SNRs_dB, BER_RAY_QPSK,  'o-', label='QPSK')
plt.semilogy(SNRs_dB, BER_RAY_16QAM, 's-', label='16-QAM')
plt.semilogy(SNRs_dB, BER_RAY_64QAM, '^-', label='64-QAM')
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('BER vs SNR     Canal Rayleigh')
# fixa os ticks em potências de 10
# yticks = [1, 1e-1, 1e-2, 1e-3]
# ylabels = [r'$10^0$', r'$10^{-1}$', r'$10^{-2}$', r'$10
    ^{-3}$']
# plt.yticks(yticks, ylabels)
plt.grid(which='both', ls='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.savefig('ber_ray.png')
plt.close()


# 2) Canal Rayleigh + AWGN
plt.figure(figsize=(8,5))
plt.semilogy(SNRs_dB, BER_RAY_AWGN_QPSK,  'o-', label='QPSK')
plt.semilogy(SNRs_dB, BER_RAY_AWGN_16QAM, 's-', label='16-QAM
    ')
plt.semilogy(SNRs_dB, BER_RAY_AWGN_64QAM, '^-', label='64-QAM
    ')
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('BER vs SNR     Canal Rayleigh + AWGN')
# fixa os ticks em potências de 10
# yticks = [1, 1e-1, 1e-2, 1e-3]
# ylabels = [r'$10^0$', r'$10^{-1}$', r'$10^{-2}$', r'$10
    ^{-3}$']
# plt.yticks(yticks, ylabels)
plt.grid(which='both', ls='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.savefig('ber_ray_awgn.png')
plt.close()
```

```
# ------------------ Varredura de prefixo cíclico
    ------------------
# Lista de diferentes comprimentos de prefixo cíclico a
    testar
CP_list = [8,16,32]
# Definição das modulações a simular: função de simulação + r
    ótulo para legenda
mods = [
    (simulate_ofdm_qpsk, 'QPSK'),
    (lambda: simulate_ofdm_qam(16), '16-QAM'),
    (lambda: simulate_ofdm_qam(64), '64-QAM')
]
# Definição dos canais: chave + função que seleciona a curva
    BER correspondente
canals = [
    ('AWGN',     lambda awgn, ray, rawn: awgn), # apenas BER
        no canal AWGN
    ('Rayleigh',lambda awgn, ray, rawn: ray), # apenas BER no
         canal Rayleigh
    ('Rayleigh+AWGN', lambda awgn, ray, rawn: rawn) # apenas
        BER no canal Rayleigh+AWGN
]

# Loop principal: para cada modulação e cada tipo de canal...
for mod_func, mod_label in mods:
    for canal_key, select_ber in canals:
        # Cria nova figura para este par (modulação, canal)
        plt.figure(figsize=(8,5))
        # Para cada valor de CP, atualiza global e recalcula
            BER
        for CP in CP_list:
            globals()['CP'] = CP # atualiza o prefixo cíclico
                usado pelas funções de simulação
            ber_awgn, ber_ray, ber_rawn = mod_func() #
                executa simulação completa
            ber = select_ber(ber_awgn, ber_ray, ber_rawn) #
                escolhe a curva de BER do canal atual
            # Plota BER vs SNR com marcador e legenda
                indicando o CP usado
            plt.semilogy(
                SNRs_dB, ber, '-o',
                label=f'CP={CP}'
            )
        # Ajusta títulos e rótulos do gráfico
        plt.title(f'{mod_label}     {canal_key}')
        plt.xlabel('SNR (dB)')
        plt.ylabel('BER')
        # yticks = [1, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5]
        # ylabels = [r'$10^0$', r'$10^{-1}$', r'$10^{-2}$', r
            '$10^{-3}$', r'$10^{-4}$', r'$10^{-5}$']
```

```
        # plt.yticks(yticks, ylabels)
        plt.grid(which='both', ls='--', alpha=0.6)
        plt.legend(title='Prefixo Cíclico', loc='upper right'
            )
        plt.tight_layout()
        # Salva o gráfico em arquivo em vez de exibir
        # em vez de plt.show(), salve o arquivo:
        filename = f'BER_{mod_label}_{canal_key}.png'.replace
            ('+','p').replace(' ','_')
        plt.savefig(filename, dpi=300)
        plt.close() # fecha a figura para liberar memória
        print(f'Salvo: {filename}')
        # plt.show()
        # plt.close()

# ------------------ 3) Simulação com Código de Repetição
    ------------------
def simulate_ofdm_qpsk_rep(r=3, canal="rayleigh_awgn"):
    k = 2  # bits por símbolo QPSK
    # Gera matriz de bits aleatórios e repete cada bit r
       vezes (código de repetição)
    bits = np.random.randint(0, 2, size=(k, total))
    bits_rep = np.repeat(bits, r, axis=1)
    # Mapeamento QPSK nos bits repetidos
    sym = (1 - 2*bits_rep[0]) + 1j*(1 - 2*bits_rep[1])
    # Serial-to-parallel: organiza em matriz N subportadoras
          (total*r / N) símbolos
    sym = sym.reshape((N, -1), order='F')

    # IFFT
    tx = np.fft.ifft(sym, axis=0)
    # Adição de prefixo cíclico e serialização
    tx_cp = np.vstack([tx[-CP:], tx]).reshape(-1, order='F')
     # Estatísticas do sinal transmitido
       = np.mean(tx_cp)
    signal_var = np.mean(np.abs(tx_cp -   )**2)

    ber_rep = [] # lista de BER com repetição
    # Varre SNRs
    for snr_db in SNRs_dB:
        # Converte SNR dB      linear e define variância do ru
           ído
        snr = 10**(snr_db/10)
        noise_var = signal_var / snr
        noise = np.sqrt(noise_var/2)*(np.random.randn(*tx_cp.
           shape) + 1j*np.random.randn(*tx_cp.shape))

        # Seleção do canal
        if canal == "awgn":
            rx = tx_cp + noise # AWGN puro
        elif canal == "rayleigh_awgn":
```

```
            rx = np.convolve(tx_cp, h, mode='full')[:tx_cp.
                size] + noise # Rayleigh + AWGN
        elif canal == "rayleigh":
            rx = np.convolve(tx_cp, h, mode='full')[:tx_cp.
                size] # Rayleigh puro
        else:
            raise ValueError("Canal deve ser 'awgn', '
                rayleigh_awgn' ou 'rayleigh'.")

        # Remove o prefixo cíclico e reconstrói matriz N ?
        mat = rx.reshape((N+CP, -1), order='F')[CP:, :]
        # FFT para domínio da frequência
        Y = np.fft.fft(mat, axis=0)

        # Equalização no caso Rayleigh+AWGN
        if canal in ["rayleigh_awgn"]:
            H_fft = np.fft.fft(h, N)
            H_fft[np.abs(H_fft) < 1e-3] = 1e-3 # evita divisõ
                es problemáticas
            Y = Y / H_fft[:, None]

        # Serializa de volta para vetor
        y = Y.reshape(-1, order='F')
        # Decisão dura QPSK nos rads real e imaginário
        b1_rep = (y.real < 0).astype(int)
        b2_rep = (y.imag < 0).astype(int)
        # Ajusta tamanho para múltiplos de r e reshape para
            blocos de r repetições
        valid_len = (b1_rep.size // r) * r
        b1_rep = b1_rep[:valid_len].reshape(-1, r)
        b2_rep = b2_rep[:valid_len].reshape(-1, r)

        # Decisão por maioria: soma de cada bloco comparada a
            r/2
        b1_dec = (np.sum(b1_rep, axis=1) > r / 2).astype(int)
        b2_dec = (np.sum(b2_rep, axis=1) > r / 2).astype(int)
        # Referência: bits originais correspondentes
        b1_ref = bits[0, :len(b1_dec)]
        b2_ref = bits[1, :len(b2_dec)]
        # Conta erros e calcula BER
        err = np.sum(b1_dec != b1_ref) + np.sum(b2_dec !=
            b2_ref)
        total_bits = b1_ref.size + b2_ref.size
        ber_rep.append(err / total_bits)

    return ber_rep # retorna curva de BER para o código de
        repetição

# Executa repetição para QPSK no canal AWGN e Rayleigh+AWGN
# Executa simulações de repetição (r=3) para QPSK em dois
    canais
```

```
BER_AWGN_QPSK_REP = simulate_ofdm_qpsk_rep(r=3, canal="awgn")
    # AWGN puro
BER_RAY_QPSK_REP = simulate_ofdm_qpsk_rep(r=3, canal="
   rayleigh") # Rayleigh puro
BER_RAY_AWGN_QPSK_REP = simulate_ofdm_qpsk_rep(r=3, canal="
   rayleigh_awgn") # Rayleigh + AWGN

# Gera gráficos comparativos de repetição vs sem repetição
# Canal AWGN
plt.figure(figsize=(8,5))
# Plota curva sem repetição (resultado de simulate_ofdm_qpsk
   em AWGN)
plt.semilogy(SNRs_dB, BER_AWGN_QPSK, 'o-', label='Sem Repetiç
   ão - AWGN')
# Plota curva com repetição (resultado de
   simulate_ofdm_qpsk_rep em AWGN)
plt.semilogy(SNRs_dB, BER_AWGN_QPSK_REP, 's--', label='Com
   Repetição (r=3) - AWGN')

plt.xlabel('SNR (dB)') # rótulo eixo x
plt.ylabel('BER') # rótulo eixo y
plt.title('QPSK     Código de Repetição     Canal AWGN') # tí
   tulo do gráfico
plt.grid(which='both', ls='--', alpha=0.6) # grade pontilhada
plt.legend() # legenda
plt.tight_layout() # ajusta layout para evitar cortes
plt.savefig('ber_qpsk_rep_awgn.png', dpi=300) # salva figura
   em arquivo
plt.close() # fecha figura para liberar memória

# QPSK     Rayleigh puro
plt.figure(figsize=(8,5))
plt.semilogy(SNRs_dB, BER_RAY_QPSK, 'o-', label='Sem Repetiçã
   o - Rayleigh')
plt.semilogy(SNRs_dB, BER_RAY_QPSK_REP, 's--', label='Com
   Repetição (r=3) - Rayleigh')
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('QPSK     Código de Repetição     Canal Rayleigh')
plt.grid(which='both', ls='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.savefig('ber_qpsk_rep_rayleigh.png', dpi=300)
plt.close()


# Canal Rayleigh + AWGN
plt.figure(figsize=(8,5))
# Plota curva sem repetição em Rayleigh+AWGN
plt.semilogy(SNRs_dB, BER_RAY_AWGN_QPSK, 'o-', label='Sem
   Repetição - Rayleigh+AWGN')
```

```
# Plota curva com repetição em Rayleigh+AWGN
plt.semilogy(SNRs_dB, BER_RAY_AWGN_QPSK_REP, 's--', label='
    Com Repetição (r=3) - Rayleigh+AWGN')
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('QPSK     Código de Repetição     Canal Rayleigh +
    AWGN')
plt.grid(which='both', ls='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.savefig('ber_qpsk_rep_rayleigh_awgn.png', dpi=300)
plt.close()


def simulate_ofdm_qam_rep(M, r=3, canal="rayleigh_awgn"):
    # Inicializa o modem QAM de ordem M
    modem = QAMModem(M)
    # Número de bits por símbolo para essa modulação
    k = modem.num_bits_symbol
    # Gera vetor serial de bits aleatórios e converte para
        paralelo
    bits = np.random.randint(0, 2, size=(k, total)).reshape
        (-1, order='F')
    # Aplica repetição de cada bit r vezes (código de repetiç
        ão)
    bits_rep = np.repeat(bits, r)
    # Mapeia bits repetidos em símbolos QAM e organiza em
        matriz N x ?
    sym = modem.modulate(bits_rep).reshape((N, -1), order='F'
        )
    # IFFT: converte símbolos para domínio do tempo
    tx = np.fft.ifft(sym, axis=0)
    # Adiciona prefixo cíclico e serializa sinal transmitido
    tx_cp = np.vstack([tx[-CP:], tx]).reshape(-1, order='F')
    # Estatísticas do sinal transmitido
        = np.mean(tx_cp) # média complexa de tx_cp
    signal_var = np.mean(np.abs(tx_cp -   )**2) # variância
        do sinal

    ber_rep = [] # lista para armazenar BER após repetição
    # Varre cada valor de SNR
    for snr_db in SNRs_dB:
        # Converte SNR de dB para escala linear e calcula
            variância de ruído
        snr = 10**(snr_db/10)
        noise_var = signal_var / snr
        # Gera ruído AWGN
        noise = np.sqrt(noise_var/2)*(np.random.randn(*tx_cp.
            shape) + 1j*np.random.randn(*tx_cp.shape))
        # Seleciona comportamento do canal
        if canal == "awgn":
```

```
            rx = tx_cp + noise # AWGN puro
        elif canal == "rayleigh_awgn":
            # Convolução com canal Rayleigh + AWGN
            rx = np.convolve(tx_cp, h, mode='full')[:tx_cp.
               size] + noise
        elif canal == "rayleigh":
            # Apenas canal Rayleigh (sem ruído adicional)
            rx = np.convolve(tx_cp, h, mode='full')[:tx_cp.
               size]
        else:
            raise ValueError("Canal deve ser 'awgn', '
               rayleigh_awgn' ou 'rayleigh'.")
        # Remove prefixo cíclico e reconstrói matriz N x ?
        mat = rx.reshape((N+CP, -1), order='F')[CP:, :]
        # FFT para voltar ao domínio da frequência
        Y = np.fft.fft(mat, axis=0)
        # Equalização para caso Rayleigh+AWGN
        if canal in ["rayleigh_awgn"]:
            H_fft = np.fft.fft(h, N)
            H_fft[np.abs(H_fft) < 1e-3] = 1e-3 # evita divisõ
               es por zero
            Y = Y / H_fft[:, None]
        # Serializa de volta para vetor de símbolos
        y = Y.reshape(-1, order='F')
        # Demodula símbolos repetidos (hard decision)
        bits_hat_rep = modem.demodulate(y, 'hard')
        # Ajusta tamanho para múltiplos de r e organiza em
           blocos de r
        valid_len = (len(bits_hat_rep) // r) * r
        bits_hat_rep = bits_hat_rep[:valid_len]
        bits_hat_reshape = bits_hat_rep.reshape(-1, r)
        # Decisão por maioria em cada bloco
        bits_hat_dec = (np.sum(bits_hat_reshape, axis=1) > r
           / 2).astype(int)
        # Define bits de referência (originais)
        bits_ref = bits[:len(bits_hat_dec)]
        # Conta número de erros
        err = np.sum(bits_hat_dec != bits_ref)
        # Calcula e armazena BER para este SNR
        ber_rep.append(err / bits_ref.size)

    return ber_rep # retorna curva de BER com repetição para
       cada SNR

# Simulações de repetição em M-QAM com r=3 para distintos cen
   ários de canal
# 64-QAM     Código de Repetição em AWGN e Rayleigh+AWGN
BER_AWGN_64QAM_REP = simulate_ofdm_qam_rep(64, r=3, canal="
   awgn") # 64-QAM no canal AWGN
BER_RAY_AWGN_64QAM_REP = simulate_ofdm_qam_rep(64, r=3, canal
   ="rayleigh_awgn")  # 64-QAM no canal Rayleigh+AWGN
```

```
# Gráfico comparativo para AWGN puro
plt.figure(figsize=(8,5))
# Curva sem repetição (resultado de simulate_ofdm_qam)
plt.semilogy(SNRs_dB, BER_AWGN_64QAM, 'o-', label='Sem Repeti
    ção - AWGN')
# Curva com repetição (resultado de simulate_ofdm_qam_rep)
plt.semilogy(SNRs_dB, BER_AWGN_64QAM_REP, 's--', label='Com
    Repetição (r=3) - AWGN')
plt.xlabel('SNR (dB)') # Rótulo do eixo x
plt.ylabel('BER') # Rótulo do eixo y
plt.title('64-QAM     Código de Repetição     Canal AWGN') #
    Título do gráfico
plt.grid(which='both', ls='--', alpha=0.6) # Grade de fundo
plt.legend() # Exibe legenda
plt.tight_layout() # Ajusta layout
plt.savefig('ber_64qam_rep_awgn.png', dpi=300) # Salva figura
plt.close() # Fecha figura

# Gráfico comparativo para Rayleigh + AWGN
plt.figure(figsize=(8,5))
# Curva sem repetição
plt.semilogy(SNRs_dB, BER_RAY_AWGN_64QAM, 'o-', label='Sem
    Repetição - Rayleigh+AWGN')
# Curva com repetição
plt.semilogy(SNRs_dB, BER_RAY_AWGN_64QAM_REP, 's--', label='
    Com Repetição (r=3) - Rayleigh+AWGN')
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('64-QAM     Código de Repetição     Canal Rayleigh
    + AWGN')
plt.grid(which='both', ls='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.savefig('ber_64qam_rep_rayleigh_awgn.png', dpi=300)
plt.close()

# 16-QAM     Código de Repetição em AWGN e Rayleigh+AWGN
BER_AWGN_16QAM_REP = simulate_ofdm_qam_rep(16, r=3, canal="
    awgn")
BER_RAY_AWGN_16QAM_REP = simulate_ofdm_qam_rep(16, r=3, canal
    ="rayleigh_awgn")
# AWGN puro para 16-QAM
plt.figure(figsize=(8,5))
plt.semilogy(SNRs_dB, BER_AWGN_16QAM, 'o-', label='Sem Repeti
    ção - AWGN')
plt.semilogy(SNRs_dB, BER_AWGN_16QAM_REP, 's--', label='Com
    Repetição (r=3) - AWGN')
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('16-QAM     Código de Repetição     Canal AWGN')
plt.grid(which='both', ls='--', alpha=0.6)
```

```
plt.legend()
plt.tight_layout()
plt.savefig('ber_16qam_rep_awgn.png', dpi=300)
plt.close()
# Rayleigh + AWGN para 16-QAM
plt.figure(figsize=(8,5))
plt.semilogy(SNRs_dB, BER_RAY_AWGN_16QAM, 'o-', label='Sem
    Repetição - Rayleigh+AWGN')
plt.semilogy(SNRs_dB, BER_RAY_AWGN_16QAM_REP, 's--', label='
    Com Repetição (r=3) - Rayleigh+AWGN')
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('16-QAM     Código de Repetição     Canal Rayleigh
    + AWGN')
plt.grid(which='both', ls='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.savefig('ber_16qam_rep_rayleigh_awgn.png', dpi=300)
plt.close()

# 16-QAM     Código de Repetição no canal Rayleigh puro
# 16-QAM     Rayleigh puro
BER_RAY_16QAM_REP = simulate_ofdm_qam_rep(16, r=3, canal="
    rayleigh")
plt.figure(figsize=(8,5))
plt.semilogy(SNRs_dB, BER_RAY_16QAM, 'o-', label='Sem Repetiç
    ão - Rayleigh')
plt.semilogy(SNRs_dB, BER_RAY_16QAM_REP, 's--', label='Com
    Repetição (r=3) - Rayleigh')
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
plt.title('16-QAM     Código de Repetição     Canal Rayleigh'
    )
plt.grid(which='both', ls='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.savefig('ber_16qam_rep_rayleigh.png', dpi=300)
plt.close()

# 64-QAM     Código de Repetição no canal Rayleigh puro
# 64-QAM     Rayleigh puro
BER_RAY_64QAM_REP = simulate_ofdm_qam_rep(64, r=3, canal="
    rayleigh")
plt.figure(figsize=(8,5))
plt.semilogy(SNRs_dB, BER_RAY_64QAM, 'o-', label='Sem Repetiç
    ão - Rayleigh')
plt.semilogy(SNRs_dB, BER_RAY_64QAM_REP, 's--', label='Com
    Repetição (r=3) - Rayleigh')
plt.xlabel('SNR (dB)')
plt.ylabel('BER')
```

```
plt.title('64-QAM     Código de Repetição     Canal Rayleigh'
    )
plt.grid(which='both', ls='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.savefig('ber_64qam_rep_rayleigh.png', dpi=300)
plt.close()
```