Cairo University

Cairo University, Faculty of Engineering,
Electronics and Electrical Communications
Department (EECE)
Fourth Year, First Term
2025/2026

Advanced Digital Communication
Information Theory

Team : 46

Report1_ELC4020

Name	ID	SEC
احمد ابراهيم حسن ابراهيم	9220014	1
احمد خلف محمد على	9220036	1
احمد محمد إبراهيم	9220078	1

Submitted to: Dr. Mohamed Khairy

& Eng. Mohamed Khaled

Submission date: 20-12-2025

Table of Contents

Table of Figures

Introduction

Modern digital communication systems rely heavily on efficient signal processing algorithms and robust modulation techniques to ensure reliable data transmission over challenging wireless channels. This report, presents a comprehensive simulation study using MATLAB to analyze fundamental and advanced concepts in digital communications. The scope of this work is divided into three primary objectives: the computational efficiency of frequency transforms, the performance of modulation schemes under fading conditions, and the implementation of Orthogonal Frequency Division Multiplexing (OFDM).

Execution time of DFT and FFT

The objective of this experiment is to compare the execution time of the Discrete Fourier Transform (DFT) and the Fast Fourier Transform (FFT). The comparison is performed by implementing the DFT directly according to its mathematical definition and comparing its execution time with MATLAB's built-in FFT function for the same input signal.

1. Theoretical Background

The Discrete Fourier Transform (DFT) of a discrete-time signal $x(n)$of length N is defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n)\ e^{-j\frac{2\pi nk}{N}}, 0 \leq k \leq N-1$$

The direct computation of the DFT requires $O(N^2)$complex multiplications and additions.
The Fast Fourier Transform (FFT) is an efficient algorithm for computing the DFT that reduces the computational complexity to $O(N\log_2 N)$, which significantly improves execution time for large values of N.

2. Methodology

2.1 DFT Implementation

A MATLAB function was written to compute the DFT directly using two nested loops, following the mathematical definition of the DFT.

```
function X_k = DFT_function(X_n)
N=length(X_n);
X_k=zeros(1,N);
for k = 1:N-1
    for n = 1:N-1
        X_k(k+1) = X_k(k+1) + X_n(n+1) * exp((-1i*2*pi*n)/N);
    end
end
end
```

2.2 Test Signal Generation

A random test signal of length $L = 4096$was generated using MATLAB's rand function:

```
L= 4096;
X_N = rand(1,L);
```

2.3 Execution Time Measurement

The execution time of both the DFT and FFT implementations was measured using MATLAB's tic and toc commands.

```
% count time taken for DFT
tic;
X_K_DFT= DFT_function(X_N);
taken_time_DFT=toc;
% count timw taken for fft
tic;
X_K_fft= fft(X_N);
taken_time_fft=toc;
```

2.4 Results

The execution times obtained for both transforms were printed and visualized using a bar graph.

```
% print time taken for DFT and fft
fprintf("DFT excution time : %.6f seconds\n",taken_time_DFT);
fprintf("fft excution time : %.6f seconds\n",taken_time_fft);
% ---------------- Bar Graph ----------------
times = [taken_time_DFT, taken_time_fft];
x=["DFT" "FFT"];
figure;
b = bar(x, times);   % <-- store bar handle
grid on;
ylabel('Execution Time (seconds)');
xlabel('type of transformer');
title('Execution Time Comparison: DFT vs FFT');
```

```
xtips = b.XEndPoints;
ytips = b.YEndPoints;
labels = string(b.YData);

text(xtips, ytips, labels, ...
    'HorizontalAlignment','center', ...
    'VerticalAlignment','bottom');
```

3. Discussion

The results demonstrate a substantial difference in execution time between the two implementations. The direct DFT computation is much slower due to its quadratic computational complexity $O(N^2)$. For $N = 4096$, this results in a large number of arithmetic operations.
In contrast, the FFT implementation uses an optimized algorithm that reduces the number of required operations to $O(N\log_2 N)$, leading to a dramatically faster execution time. This performance advantage becomes more pronounced as the signal length increases.

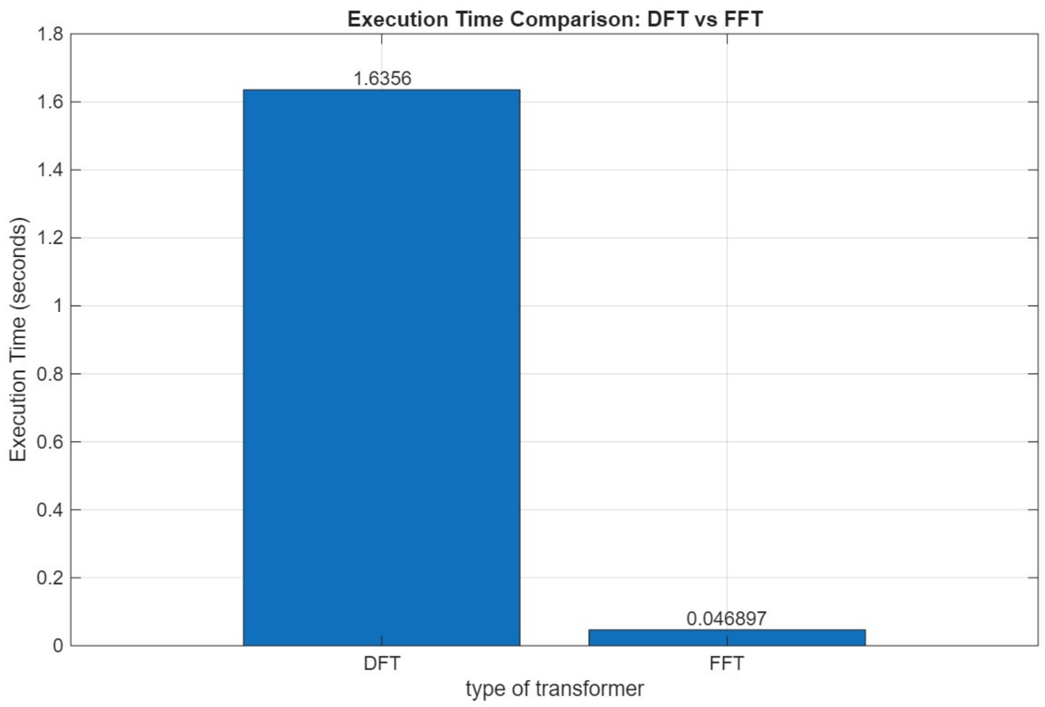

Figure 1:Execution time comparison of DFT and FFT for a signal length of L=4096

From the experimental results, it can be concluded that:
The FFT provides significantly superior performance compared to the direct DFT implementation in terms of execution time.
The FFT is the preferred method for computing the Fourier transform of large signals

Bit-error rate performance for BPSK and 16-QAM over Rayleigh flat fading channel

Simulate the Bit-Error Rate (BER) performance of BPSK, QPSK, and 16-QAM over a Rayleigh flat fading channel. Compare the performance with and without a rate 1/5 repetition code.

1. Theoretical Background

For a Rayleigh flat fading channel, the received signal is modeled as:

$$y_k = (h_r + jh_i)_k \cdot x_k + (n_c + jn_s)_k, k = 0,1,2, \ldots$$

Where:

x_k is the transmitted symbol. For BPSK:

$$x_k \in \{-\sqrt{E_b}, +\sqrt{E_b}\}.$$

$h_r, h_i \sim \mathcal{N}(0,1/2)$ are the real and imaginary parts of the Rayleigh channel.
$n_c, n_s \sim \mathcal{N}(0, N_0/2)$ are the components of additive white Gaussian noise (AWGN).
The channel amplitude $| h |$ is Rayleigh distributed.

Modulation Schemes:

- BPSK: 1 bit per symbol.
- QPSK: 2 bits per symbol.
- 16-QAM: 4 bits per symbol.

Repetition Coding:
A rate 1/5 repetition code repeats each bit 5 times. The receiver applies majority voting to decode.

2. Methodology

2.1 Bitstream Generation

Random binary data is generated according to the selected modulation type. The number of bits and bits per symbol are set as follows:

```
% Set number of bits based on modulation
switch upper(modulation_type)
    case 'BPSK'
        Num_bits = 5e6;
        bits_per_symbol = 1;
    case 'QPSK'
        Num_bits = 5e5;
        bits_per_symbol = 2;
    case '16QAM'
        Num_bits = 1.2e6;
        bits_per_symbol = 4;
    otherwise
        error('Unsupported modulation type');
end

fprintf('Running %s simulation with repetition factor = %d...\n',
modulation_type, repetition_factor);

% Generate original bits (column vector)
Tx_bits = randi([0 1], Num_bits, 1);
```

2.2 Repetition Encoding

For improved reliability over fading channels, a rate 1/5 repetition code is applied:
Each bit is repeated repetition_factor times, and the receiver applies majority voting during decoding.

```
function bits_rep = encode_bits(bits, repetition_factor)
    % Repeat each bit 'repetition_factor' times
    bits_rep = repelem(bits, repetition_factor);
 end
```

2.3 Modulation

Bits are mapped to symbols depending on the modulation scheme:

```
function symbols = map_bits(bits, modulation)
    % Maps bits to symbols based on modulation
    switch upper(modulation)
        case 'BPSK'
            symbols = 2*bits - 1; % 0->-1, 1->1
        case 'QPSK'
            b1 = bits(1:2:end-1);
            b2 = bits(2:2:end);
            symbols = complex(2*b1-1, 2*b2-1);
        case '16QAM'
            % 16-QAM Gray Mapping
            bits = bits(:);
            b = reshape(bits,4,[]).';
            gray = zeros(size(b));
            gray(:,1) = b(:,1);
            gray(:,2) = xor(b(:,1),b(:,2));
            gray(:,3) = xor(b(:,2),b(:,3));
            gray(:,4) = xor(b(:,3),b(:,4));
            idx = bi2de(gray,'left-msb')+1;
            % 16-QAM constellation (normalized)
            I_levels = [-3 -1 3 1]; Q_levels = [-3 -1 3 1];
            const = zeros(16,1);
            for k=0:15
                bin_k = de2bi(k,4,'left-msb');
                bin_nat = zeros(size(bin_k));
                bin_nat(1) = bin_k(1);
                bin_nat(2) = xor(bin_nat(1), bin_k(2));
                bin_nat(3) = xor(bin_nat(2), bin_k(3));
                bin_nat(4) = xor(bin_nat(3), bin_k(4));
                i_idx = bi2de(bin_nat(1:2),'left-msb')+1;
                q_idx = bi2de(bin_nat(3:4),'left-msb')+1;
                const(k+1) = I_levels(i_idx)+1j*Q_levels(q_idx);
            end
            const = const/sqrt(mean(abs(const).^2));
            symbols = const(idx).';
        otherwise
            error('Unsupported modulation type');
    end
end
```

2.4 Rayleigh Channel Implementation

The transmitted symbols pass through a Rayleigh flat fading channel with additive white Gaussian noise (AWGN):
Here, $h = h_r + jh_i$represents the complex channel gain, and $n = n_c + jn_s$is the complex noise.

```
function [rx, h] = rayleigh_channel(tx, N0)
    tx = tx(:); % ensure column vector
    h = (randn(length(tx),1) + 1j*randn(length(tx),1))/sqrt(2); % Rayleigh
    n = sqrt(N0/2)*(randn(length(tx),1) + 1j*randn(length(tx),1)); % AWGN
    rx = h .* tx + n;
end
```

2.5 Receiver Processing

Assuming perfect channel knowledge, equalization is applied to remove the channel effect:

```
function z = equalize(rx,h)
    % Channel equalization
    z = rx ./ h;
end
```

2.6 Demapping

The equalized symbols are converted back to bits:

```
function bits_dec = decode_bits(bits_rx, repetition_factor)
    % Majority vote decoding
    bits_matrix = reshape(bits_rx, repetition_factor, []);
    bits_dec = sum(bits_matrix,1) >= ceil(repetition_factor/2);
    bits_dec = bits_dec.'; % column vector
  end
```

2.7 Decoding

For repeated bits, majority voting restores the original bitstream:

```
function bits = demap_symbols(symbols, modulation)
    % Demap symbols back to bits
    switch upper(modulation)
        case 'BPSK'
            bits = real(symbols) >= 0;
        case 'QPSK'
            bits = zeros(2*length(symbols),1);
            bits(1:2:end-1) = real(symbols)>=0;
            bits(2:2:end) = imag(symbols)>=0;
       case '16QAM'
    symbols = symbols(:); % ensure column vector
    % Define constellation
    I_levels = [-3 -1 3 1]; Q_levels = [-3 -1 3 1];
    const = zeros(16,1);
    for k=0:15
        bin_k = de2bi(k,4,'left-msb');
        bin_nat = zeros(size(bin_k));
        bin_nat(1) = bin_k(1);
        bin_nat(2) = xor(bin_nat(1), bin_k(2));
        bin_nat(3) = xor(bin_nat(2), bin_k(3));
        bin_nat(4) = xor(bin_nat(3), bin_k(4));
        i_idx = bi2de(bin_nat(1:2),'left-msb')+1;
        q_idx = bi2de(bin_nat(3:4),'left-msb')+1;
        const(k+1) = I_levels(i_idx)+1j*Q_levels(q_idx);
    end
    const = const/sqrt(mean(abs(const).^2));
    % Nearest neighbor detection
    [~, idx] = min(abs(symbols - const.'),[],2); % now safe
    gray = de2bi(idx-1,4,'left-msb');
    % Reverse Gray
    bits_mat = zeros(size(gray));
    bits_mat(:,1) = gray(:,1);
    bits_mat(:,2) = xor(bits_mat(:,1),gray(:,2));
    bits_mat(:,3) = xor(bits_mat(:,2),gray(:,3));
    bits_mat(:,4) = xor(bits_mat(:,3),gray(:,4));
    bits = reshape(bits_mat.',[],1);
        otherwise
            error('Unsupported modulation type');
    end
end
```

2.8 BER Computation

Finally, the Bit Error Rate (BER) is computed for each SNR value:

```
function BER = compute_ber(tx,rx)
    % Compute Bit Error Rate
    BER = sum(tx~=rx)/length(tx);
end
```

2.9 BER Simulation over SNR

The bit-error rate (BER) is evaluated across a range of SNR values. For each SNR:

1. The noise power $N_0$is computed from the given E_b/N_0:

$$N_0 = \frac{E_b}{(E_b/N_0) \cdot \text{bits per symbol}}$$

BER without repetition (Normal):

- Bits are mapped to symbols using the chosen modulation.
- Symbols are transmitted through a Rayleigh flat fading channel with AWGN.
- The receiver performs equalization using the known channel.
- Symbols are demapped back to bits and compared with the transmitted bits to compute BER.

BER with repetition (factor 5):

- The original bitstream is encoded using rate 1/5 repetition coding.
- Encoded bits are mapped, transmitted through the Rayleigh channel, and equalized at the receiver.
- Demapping and majority-vote decoding are applied to recover the original bits.
- BER is computed by comparing the decoded bits with the original transmitted bits.

```
for idx = 1:length(SNR_db)

    %% ----- Noise power -----
    EbNo_linear = 10^(SNR_db(idx)/10);
    N0 = Eb / (EbNo_linear * bits_per_symbol);

    %% ----- BER without repetition (Normal) -----
    tx_symbols_normal = map_bits(Tx_bits, modulation_type);
    [rx_symbols_normal, h_normal] = rayleigh_channel(tx_symbols_normal, N0);
    z_normal = equalize(rx_symbols_normal, h_normal);
    rx_bits_normal = demap_symbols(z_normal, modulation_type);
    BER_Normal(idx) = compute_ber(Tx_bits, rx_bits_normal);

    %% ----- BER with repetition (factor 5) -----
    Tx_bits_rep = encode_bits(Tx_bits, repetition_factor);
    tx_symbols_rep = map_bits(Tx_bits_rep, modulation_type);
    [rx_symbols_rep, h_rep] = rayleigh_channel(tx_symbols_rep, N0);
    z_rep = equalize(rx_symbols_rep, h_rep);
    rx_bits_rep = demap_symbols(z_rep, modulation_type);
    Rx_bits = decode_bits(rx_bits_rep, repetition_factor);
    BER_Rep5(idx) = compute_ber(Tx_bits, Rx_bits);

end
```

2.10 Result Visualization

The BER performance for normal transmission and repetition-coded transmission is visualized over the specified SNR range using a semilogarithmic plot. This highlights the improvement offered by the rate 1/5 repetition code in Rayleigh fading conditions.

```
% PART 3: Plot Results
figure;
semilogy(SNR_db, BER_Normal, 'b-o','LineWidth',1.5); hold on;
semilogy(SNR_db, BER_Rep5, 'r-s','LineWidth',1.5); grid on;
xlabel('Eb/No (dB)');
ylabel('Bit Error Rate (BER)');
legend('Normal','Repetition 5');
title(sprintf('%s Performance over Rayleigh Channel', modulation_type));
```

3. Results

The simulation was performed for BPSK, QPSK, and 16-QAM over a Rayleigh flat fading channel with and without repetition coding (rate 1/5). The Bit Error Rate (BER) was computed over an SNR range of -3 to 10 dB.

BPSK BER over Rayleigh Channel

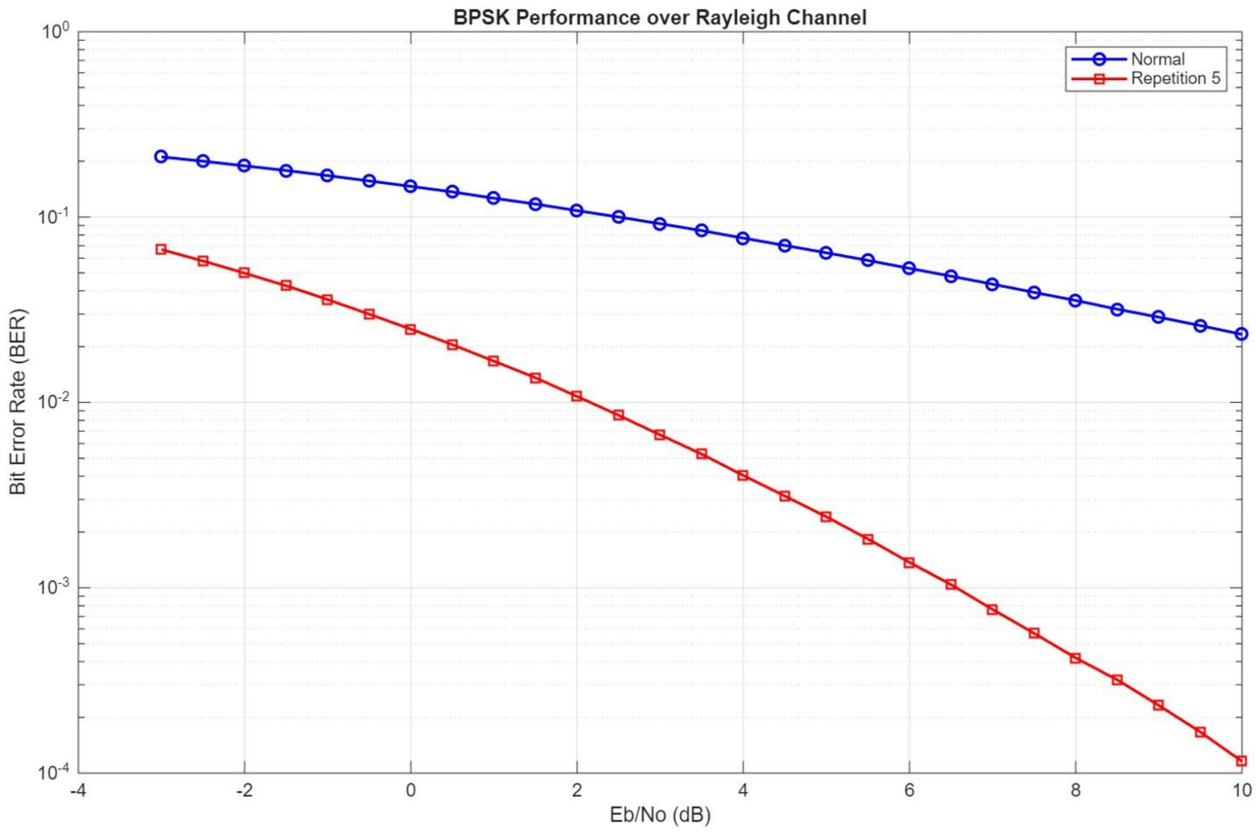

Figure 2:BPSK BER over Rayleigh Channel

As seen in Fig. 2, the BER for repetition-5 is significantly better than that of normal BPSK. In this case, an error occurs only if three or more bits are changed, which is much less likely than a single-bit error in normal BPSK. However, this improvement comes at a cost: the transmission rate is reduced because each bit is repeated five times, resulting in an effective rate that is one-fifth of normal BPSK. Additionally, achieving this lower BER requires more energy, with the total transmitted energy equal to five times that of normal BPSK.

QPSK BER over Rayleigh Channel

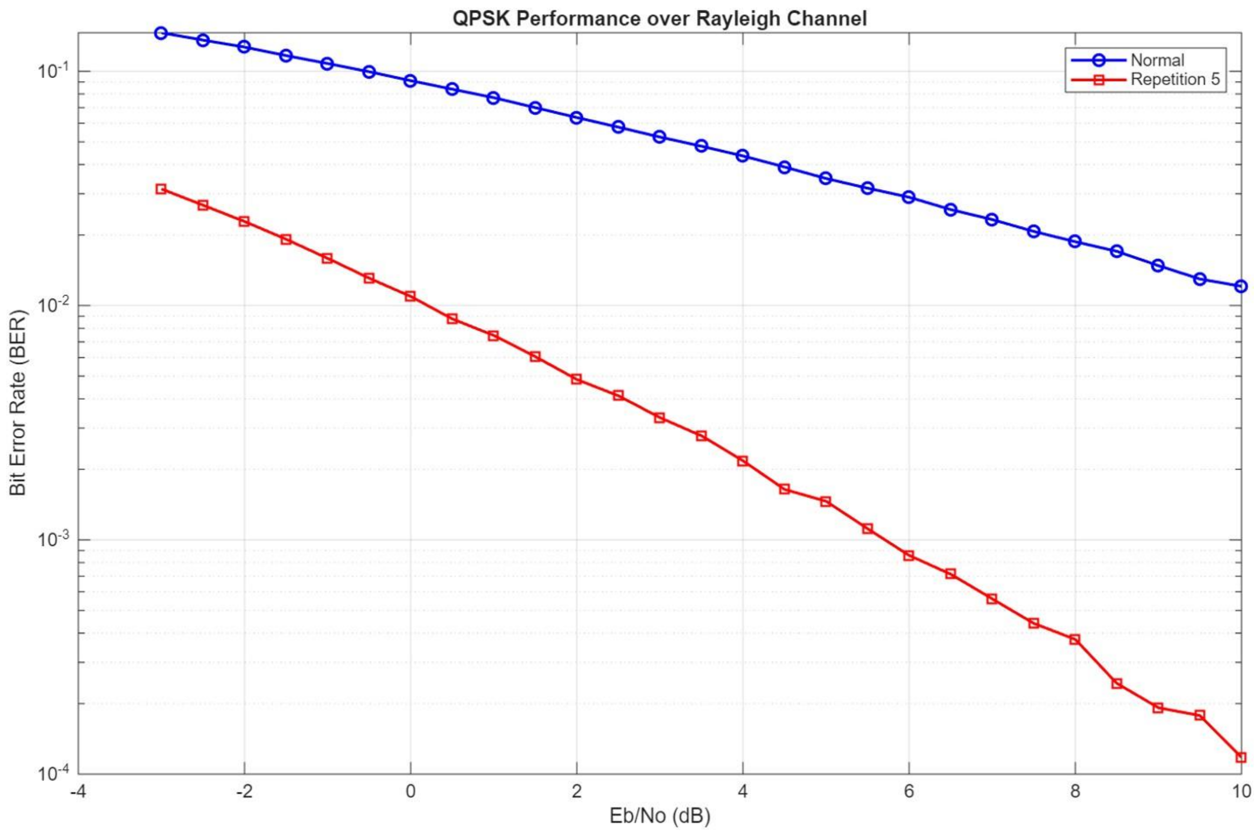

Figure 3:QPSK BER over Rayleigh Channel

- Fig. 3 shows the performance comparison between normal QPSK and repetition-5 QPSK. As expected, QPSK with repetition-5 demonstrates better performance.
- Comparing Fig. 2 and Fig. 3, we can see that the difference in BER between the two cases is relatively small. However, QPSK transmits at twice the rate of BPSK, which makes its efficiency higher despite the slightly higher BER.

16QAM BER over Rayleigh Channel

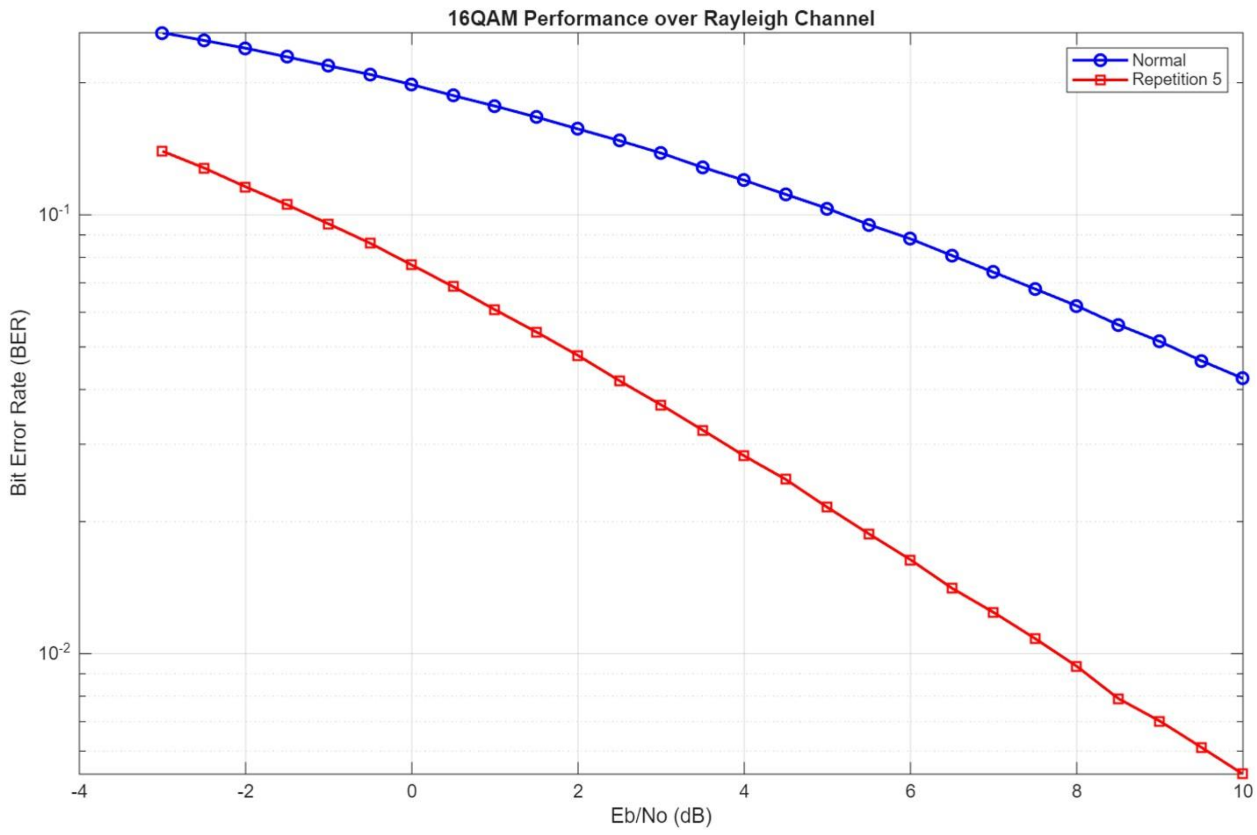

Figure 4:16QAM BER over Rayleigh Channel

- As the modulation order increases, the BER also increases. This occurs because higher-order constellations have more symbols, which reduces the distance between points and tightens the decision boundaries, making the system more sensitive to noise. This can be observed by comparing Figs. 2, 3, and 4, where the BER for 16-QAM is worse than that of both BPSK and QPSK.
- On the other hand, higher-order modulation provides a higher data rate. For example, 16-QAM transmits at four times the rate of BPSK and twice the rate of QPSK.

BPSK, QPSK, 16-QAM BER over Rayleigh (Normal vs Rep 5)

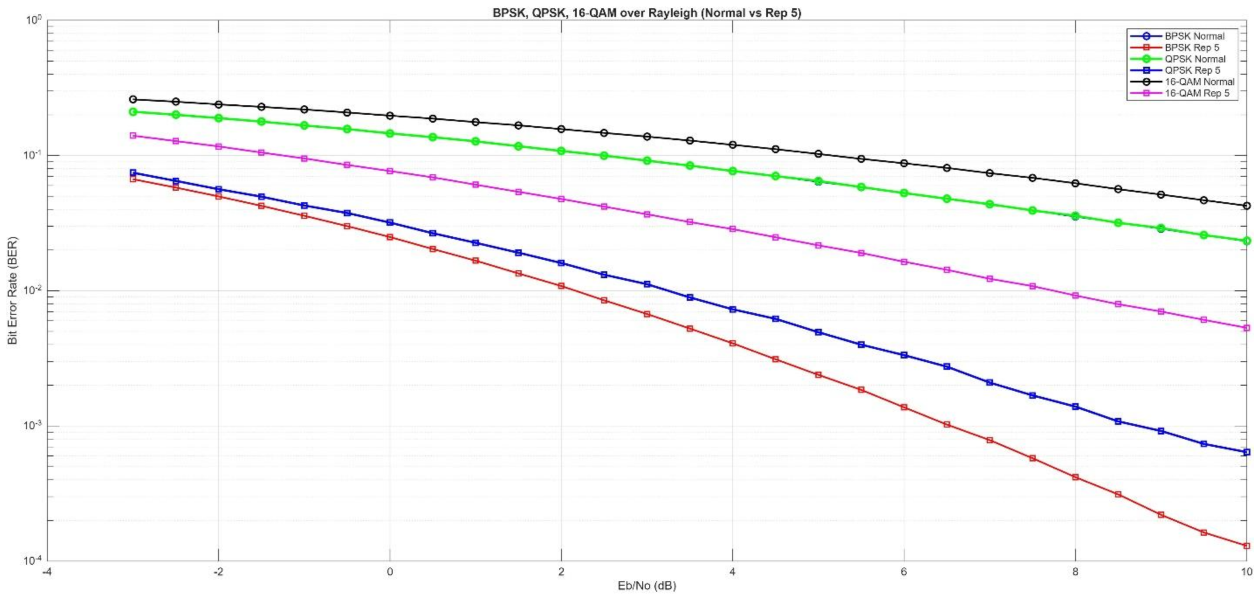

Figure 5:BPSK, QPSK, 16-QAM BER over Rayleigh (Normal vs Rep 5)

Fig. 5 shows a compilation of all the required curves. As observed, the BER performance of BPSK and QPSK is almost identical; however, QPSK is preferred due to its higher data rate. On the other hand, 16-QAM exhibits the worst BER performance, but it provides a higher data rate than both BPSK and QPSK, as discussed earlier.

We can notice that the BPSK and QPSK repetition 5 curves are not the same, however the are the exactly the same in the normal BPSK and QPSK as the two curves are identical as seen above and that is because the nature of QPSK, if we imagine we send a bit let's say 1 after repetition it will be 11111,in BPSK each symbol carries only one bit so there will be an error if three symbols or more changed,but in QPSK case each symbol carries 2 bits so if one symbol changed it means that there is an error in 2 bits not one like BPSK and so there will be an error for detecting the bit if two or more symbols changed and so it is expected to have more BER in QPSK repetition 5 than BPSK as shown above.

OFDM System Simulation

This section details the simulation of an Orthogonal Frequency Division Multiplexing (OFDM) system. The objective is to evaluate the Bit Error Rate (BER) performance of BPSK and 16-QAM modulation schemes over both Rayleigh flat fading and frequency-selective fading channels, with and without rate 1/5 repetition coding.

1. Theoretical Background

OFDM is a multi-carrier modulation technique that divides the available spectrum into multiple orthogonal sub-carriers. This approach effectively combats inter-symbol interference (ISI) caused by multipath channels.

- **IFFT/FFT:** The modulation and demodulation are efficiently performed using the Inverse Fast Fourier Transform (IFFT) and Fast Fourier Transform (FFT).
- **Cyclic Prefix (CP):** A cyclic extension is added to the start of each OFDM symbol to preserve orthogonality and eliminate ISI, provided the CP length exceeds the channel delay spread.
- **Interleaving:** Interleaving is used to randomize burst errors that may occur in fading channels, making error correction codes more effective.

2. Methodology

2.1 Parameters

```
% ===================== System Parameters =====================
cp = 16;                        % Cyclic prefix length
rate = 1/5;                     % Coding rate
N_fft = 256;                    % FFT size
Eb_N0_dB = 0:2:20;              % Eb/N0 range in dB
N_OFDM = 10000;                 % Number of OFDM symbols
modulation ='QPSK';             % Modulation type: (BPSK, QPSK, 16QAM)
energy_type = 'Same_Per_Trans'; % Energy normalization type
channel_type = 'FreqSel';       % Channel type: Flat or Frequency Selective
```

2.2 Bitstream Generation

Random binary data is generated according to the selected modulation type, coded/uncoded, and fft size as follows:

```
function R = generate_data(m, N_fft, N_code)
    R = randi([0 1], 1, floor(N_fft/N_code)*m);
end
```

2.3 Encoder

The encoder used to apply the repetition code on the generated data, in case of nocode NCode=1, it also pad the data to fit into a ofdm symbol.

```
function R = Encoder(data, m, N_fft, N_code)
    encoded_data = repelem(data, N_code);
    R = [encoded_data, zeros(1, (m*N_fft)-length(encoded_data))];
end
```

2.4 Interleaving

An interleaver is utilized to shuffle the bits before mapping. The dimensions of the interleaver matrix depend on the modulation scheme:

- **BPSK:** 32 × 8
- **QPSK:** 32 × 16
- **16-QAM:** 32 × 32

```
function R = Interleaver(mod, data)
    if isequal(mod, 'BPSK')
       reshape_tx_data = reshape(data, 8, 32)';
    elseif isequal(mod, 'QPSK')
       reshape_tx_data = reshape(data, 16, 32)';
    elseif isequal(mod, '16QAM')
       reshape_tx_data = reshape(data, 32, 32)';
    end
    R = reshape_tx_data(:)';
end
```

2.5 Mapping and OFDM Modulation

The interleaved bits are mapped to BPSK, QPSK, or 16-QAM symbols. An IFFT of size 256 is applied to the symbols, followed by the addition of the cyclic prefix.

```
% Mapper
mapper_output = Mapper(interleaver_output, modulation);

% ifft
ifft_output = ifft(mapper_output, N_fft);

% cyclic extension
cyclic_output = [ifft_output(end-cp+1:end), ifft_output];
```

2.6 Channel Models

Two channel models are simulated:

1. **Rayleigh Flat Fading:** The channel is modelled as a single complex value, constant across all sub-carriers for a given symbol.
2. **Frequency Selective Fading:** Modelled in the frequency domain, where every sub-channel is independently faded by a different Rayleigh fading coefficient.

```matlab
function [Channel_Output, H] = Channel(data, m, Eb_N0_dB, N_code, N_fft, cp, 
channeltype, energy_type)
    % Noise
    segma = sqrt((1/2)./10^(Eb_N0_dB/10));
    Eb_actual = (sum(abs(data).^2)/length(data))/m;
    noise = sqrt(Eb_actual) * segma  * (randn(size(data)) + 
1j*randn(size(data)));

    if isequal(energy_type, 'Same_Per_Info')
        noise = sqrt(N_code) * noise;
    end

    if isequal(channeltype, 'Flat')
        % Flat fading channel
        H = sqrt(1/2)*(randn(1) + 1j*randn(1));
        Channel_Output = H*data + noise;
    else
        % Frequency selective channel
        H = sqrt(1/2)*(randn(1, N_fft) + 1j*randn(1, N_fft));

        % Remove cyclic extension
        data_cp_removed = data(cp+1:end);

        % Frequency domain
        data_freq = fft(data_cp_removed, N_fft);

        % Applay Freqency Selctive Channel
        ch_out_freq = data_freq .* H;

        % Time domain
        ch_out = ifft(ch_out_freq, N_fft);

        % Add cyclic extension
        ch_out = [ch_out(end-cp+1:end), ch_out];

        %Add Noise
        Channel_Output = ch_out + noise;
    end
end
```

2.7 Receiver

The receiver performs the reverse operations: removing the cyclic prefix, applying the FFT, performing) equalization using the known channel state, de-mapping, de-interleaving, and decoding.

```
% Remove cyclic extension
remove_cyclic_output = channel_output(cp+1:end);

% fft
fft_output = fft(remove_cyclic_output, N_fft);

% channel equalization
channel_equ = fft_output ./ H;

% De-Mapper
demapper_output = Demapper(channel_equ, modulation);

% De-Interleaver
deinterleaver_output = DeInterleaver(modulation, demapper_output);

% Decoder
decoder_output = Decoder(deinterleaver_output, m, N_fft, N_code);
```

3. Result

The system was simulated for BPSK and 16-QAM modulations over both channel types. The BER vs. Eb/N0 performance is analyzed below.

3.1 QPSK vs 16QAM over Flat Fading Channel (same energy per transmission bit)

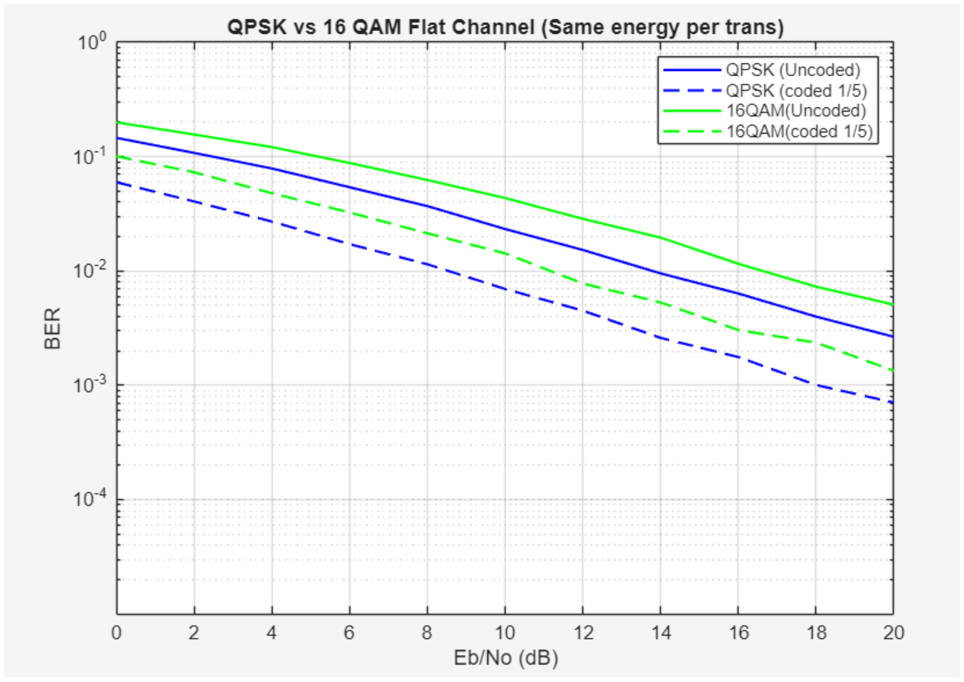

Figure 6: QPSK vs 16QAM over Flat Fading Channel (same energy per transmission bit)

QPSK vs 16-QAM:

- QPSK achieves significantly lower BER than 16-QAM due to its larger constellation distance, even when channel coding is applied.

Effect of Coding (Repetition 1/5):

- when a deep fade occurs, it affects all subcarriers simultaneously. Even though the repetition code sends 5 copies of the bit, all 5 copies suffer the exact same fade. This results in the inability to exploit the multi carrier advantages. The improvement seen in Fig. 6 is purely due to energy of the repetition code.

3.2 QPSK vs 16QAM over Frequency Selective Fading Channel (same energy per transmission bit)

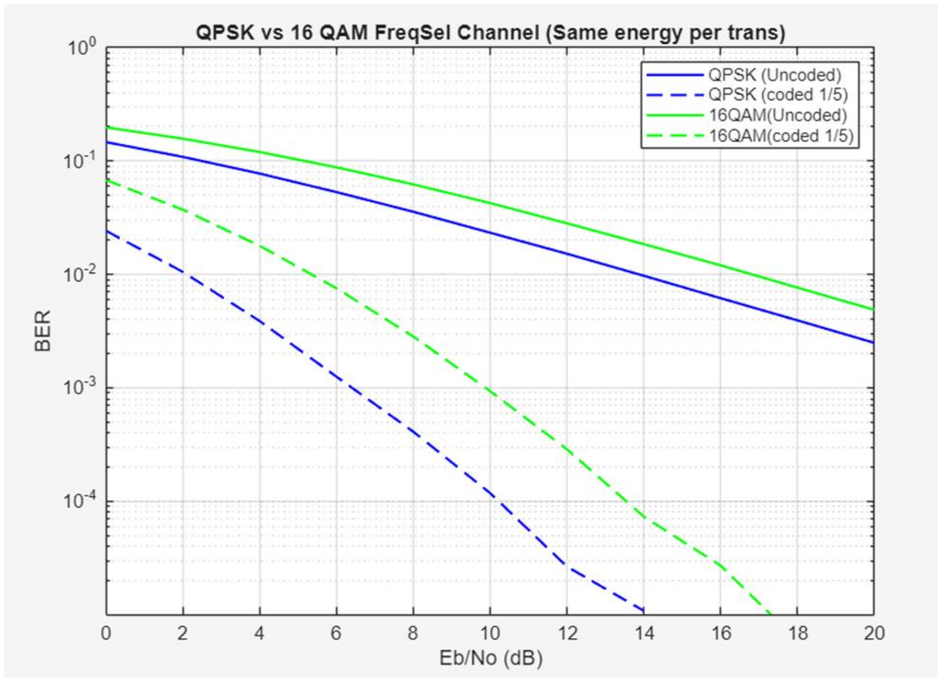

Figure 7: QPSK vs 16QAM over Frequency Selective Fading Channel (same energy per transmission bit)

QPSK vs 16-QAM:

- QPSK achieves significantly lower BER than 16-QAM due to its larger constellation distance, even when channel coding is applied.

Effect of Coding (Repetition 1/5):

- Fig. 7 indicates that the system is exploiting the multi carrier advantages. Because the channel is frequency selective, the fading on different subcarriers is independent
- The repetition code spreads the 5 copies of a bit across different subcarriers. The probability of *3 of 5* subcarriers being in a deep fade simultaneously is low. Therefore, the decoder can successfully recover the bit using the copies received on the good subcarriers. Resulting in better performance.

3.3 QPSK vs 16QAM over Flat Fading Channel (same energy per information bit)

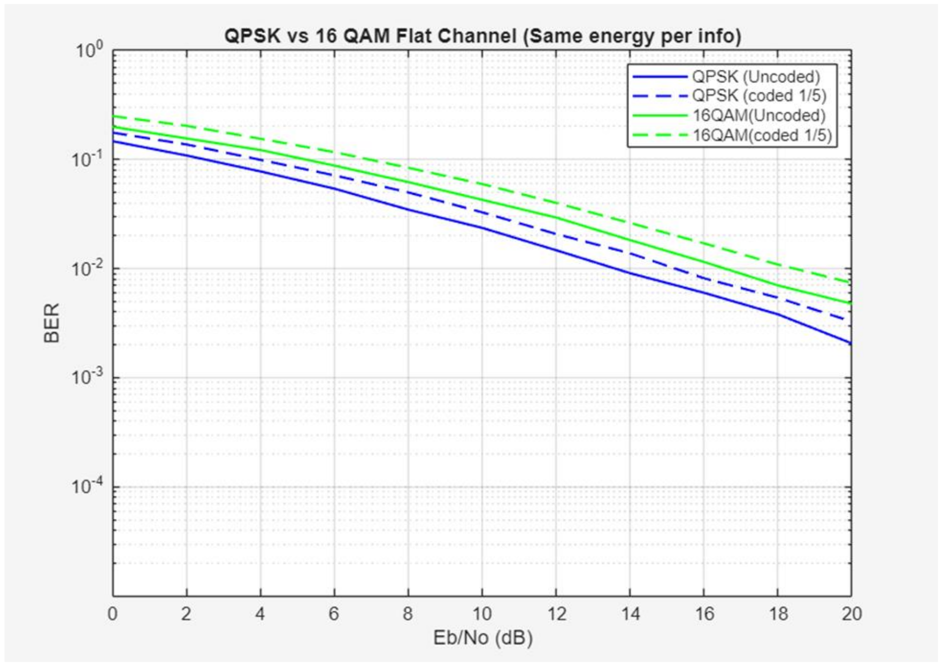

Figure 8: QPSK vs 16QAM over Flat Fading Channel (same energy per information bit)

QPSK vs 16-QAM:

- QPSK achieves significantly lower BER than 16-QAM due to its larger constellation distance, even when channel coding is applied.

Effect of Coding (Repetition 1/5):

- Similar result as in section 3.1. However, because energy averaging of the repetition code is not applied anymore the improvement seen in Fig.6 is no longer applied as seen in Fig. 8

3.4 QPSK vs 16QAM over Flat Fading Channel (same energy per information bit)

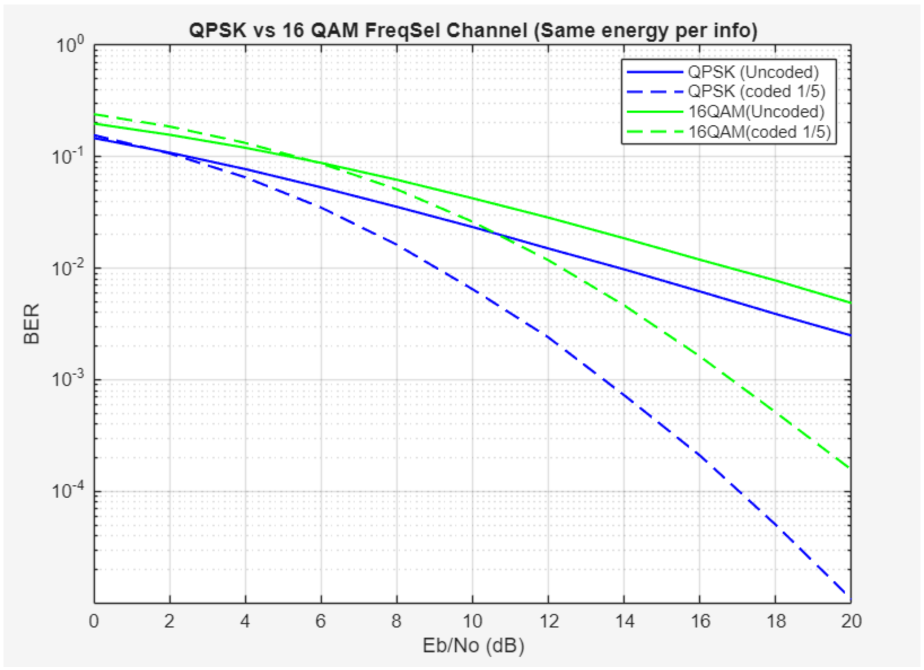

Figure 9: QPSK vs 16QAM over Flat Fading Channel (same energy per information bit)

QPSK vs 16-QAM:

- QPSK achieves significantly lower BER than 16-QAM due to its larger constellation distance, even when channel coding is applied.

Effect of Coding (Repetition 1/5):

- Similar result as in section 3.2. However, because now the energy is the same per information bit the improvement that was the result of energy averaging of the repetition code is not applied anymore as seen in Fig. 9

.

APPENDIX

```
clear;clc;
% =========================================================================
%              Question 1 : comaparing between FFT and DFT
% =========================================================================

fprintf('Running Part 2.1: FFT/DFT Simulation...\n');
L= 4096;
X_N = rand(1,L);
% count time taken for DFT
tic;
X_K_DFT= DFT_function(X_N);
taken_time_DFT=toc;
% count timw taken for fft
tic;
X_K_fft= fft(X_N);
taken_time_fft=toc;
% print time taken for DFT and fft
fprintf("DFT excution time : %.6f seconds\n",taken_time_DFT);
fprintf("fft excution time : %.6f seconds\n",taken_time_fft);

% ---------------- Bar Graph ----------------
times = [taken_time_DFT, taken_time_fft];
x=["DFT" "FFT"];
figure;
b = bar(x, times);   % <-- store bar handle
grid on;
ylabel('Execution Time (seconds)');
xlabel('type of transformer');
title('Execution Time Comparison: DFT vs FFT');
xtips = b.XEndPoints;
ytips = b.YEndPoints;
labels = string(b.YData);

text(xtips, ytips, labels, ...
    'HorizontalAlignment','center', ...
    'VerticalAlignment','bottom');

function X_k = DFT_function(X_n)
N=length(X_n);
X_k=zeros(1,N);
for k = 1:N-1
    for n = 1:N-1
        X_k(k+1) = X_k(k+1) + X_n(n+1) * exp((-1i*2*pi*n)/N);
    end
end
end
```

```matlab
%% ===============================================================================================
%  Question2 : Bit-error rate performance for BPSK and 16-QAM over Rayleigh flat fading channel
% ================================================================================================

% ==============================
% Single-File Rayleigh Fading Simulator
% Supports: BPSK, QPSK, 16-QAM
% Repetition factor fixed at 5
% Computes BER: normal vs repetition
% ==============================

clear; clc; rng(0);

% ------------------------------
% PART 1: User Input / Parameters
% ------------------------------

modulation_type = input('Select modulation type (BPSK / QPSK / 16QAM): ','s');
repetition_factor = 5;          % Fixed repetition factor
SNR_db = -3:0.5:10;             % SNR range (dB)
Eb = 1;                         % Energy per bit

% Set number of bits based on modulation
switch upper(modulation_type)
    case 'BPSK'
        Num_bits = 5e6;
        bits_per_symbol = 1;
    case 'QPSK'
        Num_bits = 5e5;
        bits_per_symbol = 2;
    case '16QAM'
        Num_bits = 1.2e6;
        bits_per_symbol = 4;
    otherwise
        error('Unsupported modulation type');
end

fprintf('Running %s simulation with repetition factor = %d...\n', modulation_type,
repetition_factor);

% Generate original bits (column vector)
Tx_bits = randi([0 1], Num_bits, 1);

% Preallocate BER results
BER_Normal = zeros(size(SNR_db));
BER_Rep5   = zeros(size(SNR_db));
```

```
% ------------------------------
% PART 2: Main SNR Loop
% ------------------------------

for idx = 1:length(SNR_db)

    % ----- Noise power -----
    EbNo_linear = 10^(SNR_db(idx)/10);
    N0 = Eb / (EbNo_linear * bits_per_symbol);

    % ----- BER without repetition (Normal) -----
    tx_symbols_normal = map_bits(Tx_bits, modulation_type);
    [rx_symbols_normal, h_normal] = rayleigh_channel(tx_symbols_normal, N0);
    z_normal = equalize(rx_symbols_normal, h_normal);
    rx_bits_normal = demap_symbols(z_normal, modulation_type);
    BER_Normal(idx) = compute_ber(Tx_bits, rx_bits_normal);

    % ----- BER with repetition (factor 5) -----
    Tx_bits_rep = encode_bits(Tx_bits, repetition_factor);
    tx_symbols_rep = map_bits(Tx_bits_rep, modulation_type);
    [rx_symbols_rep, h_rep] = rayleigh_channel(tx_symbols_rep, N0);
    z_rep = equalize(rx_symbols_rep, h_rep);
    rx_bits_rep = demap_symbols(z_rep, modulation_type);
    Rx_bits = decode_bits(rx_bits_rep, repetition_factor);
    BER_Rep5(idx) = compute_ber(Tx_bits, Rx_bits);

end

% ------------------------------
% PART 3: Plot Results
% ------------------------------
figure;
semilogy(SNR_db, BER_Normal, 'b-o','LineWidth',1.5); hold on;
semilogy(SNR_db, BER_Rep5, 'r-s','LineWidth',1.5); grid on;
xlabel('Eb/No (dB)');
ylabel('Bit Error Rate (BER)');
legend('Normal','Repetition 5');
title(sprintf('%s Performance over Rayleigh Channel', modulation_type));

% PART 4: Local Functions
function bits_rep = encode_bits(bits, repetition_factor)
    % Repeat each bit 'repetition_factor' times
    bits_rep = repelem(bits, repetition_factor);
 end

function bits_dec = decode_bits(bits_rx, repetition_factor)
    % Majority vote decoding
    bits_matrix = reshape(bits_rx, repetition_factor, []);
    bits_dec = sum(bits_matrix,1) >= ceil(repetition_factor/2);
    bits_dec = bits_dec.'; % column vector
  end

function symbols = map_bits(bits, modulation)
    % Maps bits to symbols based on modulation
    switch upper(modulation)
```

```
        case 'BPSK'
            symbols = 2*bits - 1; % 0->-1, 1->1
        case 'QPSK'
            b1 = bits(1:2:end-1);
            b2 = bits(2:2:end);
            symbols = complex(2*b1-1, 2*b2-1);
        case '16QAM'
            % 16-QAM Gray Mapping
            bits = bits(:);
            b = reshape(bits,4,[]).';
            gray = zeros(size(b));
            gray(:,1) = b(:,1);
            gray(:,2) = xor(b(:,1),b(:,2));
            gray(:,3) = xor(b(:,2),b(:,3));
            gray(:,4) = xor(b(:,3),b(:,4));
            idx = bi2de(gray,'left-msb')+1;
            % 16-QAM constellation (normalized)
            I_levels = [-3 -1 3 1]; Q_levels = [-3 -1 3 1];
            const = zeros(16,1);
            for k=0:15
                bin_k = de2bi(k,4,'left-msb');
                bin_nat = zeros(size(bin_k));
                bin_nat(1) = bin_k(1);
                bin_nat(2) = xor(bin_nat(1), bin_k(2));
                bin_nat(3) = xor(bin_nat(2), bin_k(3));
                bin_nat(4) = xor(bin_nat(3), bin_k(4));
                i_idx = bi2de(bin_nat(1:2),'left-msb')+1;
                q_idx = bi2de(bin_nat(3:4),'left-msb')+1;
                const(k+1) = I_levels(i_idx)+1j*Q_levels(q_idx);
            end
            const = const/sqrt(mean(abs(const).^2));
            symbols = const(idx).';
        otherwise
            error('Unsupported modulation type');
    end
end

function bits = demap_symbols(symbols, modulation)
    % Demap symbols back to bits
    switch upper(modulation)
        case 'BPSK'
            bits = real(symbols) >= 0;
        case 'QPSK'
            bits = zeros(2*length(symbols),1);
            bits(1:2:end-1) = real(symbols)>=0;
            bits(2:2:end) = imag(symbols)>=0;
       case '16QAM'
    symbols = symbols(:); % ensure column vector
    % Define constellation
    I_levels = [-3 -1 3 1]; Q_levels = [-3 -1 3 1];
    const = zeros(16,1);
    for k=0:15
        bin_k = de2bi(k,4,'left-msb');
        bin_nat = zeros(size(bin_k));
        bin_nat(1) = bin_k(1);
        bin_nat(2) = xor(bin_nat(1), bin_k(2));
        bin_nat(3) = xor(bin_nat(2), bin_k(3));
        bin_nat(4) = xor(bin_nat(3), bin_k(4));
        i_idx = bi2de(bin_nat(1:2),'left-msb')+1;
        q_idx = bi2de(bin_nat(3:4),'left-msb')+1;
        const(k+1) = I_levels(i_idx)+1j*Q_levels(q_idx);
    end
    const = const/sqrt(mean(abs(const).^2));
    % Nearest neighbor detection
    [~, idx] = min(abs(symbols - const.'),[],2); % now safe
    gray = de2bi(idx-1,4,'left-msb');
    % Reverse Gray
    bits_mat = zeros(size(gray));
    bits_mat(:,1) = gray(:,1);
    bits_mat(:,2) = xor(bits_mat(:,1),gray(:,2));
    bits_mat(:,3) = xor(bits_mat(:,2),gray(:,3));
    bits_mat(:,4) = xor(bits_mat(:,3),gray(:,4));
    bits = reshape(bits_mat.',[],1);

        otherwise
            error('Unsupported modulation type');
    end
end

function [rx, h] = rayleigh_channel(tx, N0)
    tx = tx(:); % ensure column vector
    h = (randn(length(tx),1) + 1j*randn(length(tx),1))/sqrt(2); % Rayleigh
    n = sqrt(N0/2)*(randn(length(tx),1) + 1j*randn(length(tx),1)); % AWGN
    rx = h .* tx + n;
end

function z = equalize(rx,h)
    % Channel equalization
    z = rx ./ h;
end

function BER = compute_ber(tx,rx)
    % Compute Bit Error Rate
    BER = sum(tx~=rx)/length(tx);
end
```

```
%%
% ========================================================================
%                Question 3 : OFDM System Simulation
% ========================================================================
clear; clc;
% ===================== System Parameters =====================
cp = 16;                        % Cyclic prefix length
rate = 1/5;                     % Coding rate
N_fft = 256;                    % FFT size
Eb_N0_dB = 0:2:20;              % Eb/N0 range in dB
N_OFDM = 10000;                 % Number of OFDM symbols
modulation ='QPSK';             % Modulation type: (BPSK, QPSK, 16QAM)
energy_type = 'Same_Per_Trans'; % Energy normalization type : (Same_Per_Trans, Same_Per_Info)
channel_type = 'Flat';          % Channel type: Flat or Frequency Selective (Flat, FreqSel)

% ===================== Modulation Order =====================
if isequal(modulation, 'BPSK')
    m = 1;
elseif isequal(modulation, 'QPSK')
    m = 2;
elseif isequal(modulation, '16QAM')
    m = 4;
end

% Loop for uncoded (N_code=1) and coded (N_code=1/rate) systems
for N_code = [1 1/rate]
    for i = 1:length(Eb_N0_dB)
        symbol_error = [];
        total_bits = [];
        for j = 1:N_OFDM

        % ===================== Transmitter =====================

            % Generate random input bits
            data = generate_data(m, N_fft, N_code);

            % Encoder
            encoder_output = Encoder(data, m, N_fft, N_code);

            % Interleaver
            interleaver_output = Interleaver(modulation, encoder_output);


            % Mapper
            mapper_output = Mapper(interleaver_output, modulation);

            % ifft
            ifft_output = ifft(mapper_output, N_fft);

            % cyclic extension
            cyclic_output = [ifft_output(end-cp+1:end), ifft_output];

            % channel
            [channel_output, H] = Channel(cyclic_output, m, Eb_N0_dB(i), N_code, N_fft, cp, 
channel_type, energy_type);


        % ===================== Receiver =====================

            % Remove cyclic extension
            remove_cyclic_output = channel_output(cp+1:end);

            % fft
            fft_output = fft(remove_cyclic_output, N_fft);

            % channel equalization
            channel_equ = fft_output ./ H;

            % De-Mapper
            demapper_output = Demapper(channel_equ, modulation);

            % De-Interleaver
            deinterleaver_output = DeInterleaver(modulation, demapper_output);

            % Decoder
            decoder_output = Decoder(deinterleaver_output, m, N_fft, N_code);

            % Compute BER for curren ofdm symbol
            symbol_error(j) = sum(decoder_output ~= data)/length(data);

        end
        % Average BER for current Eb/N0
        BER(i) = sum(symbol_error) / N_OFDM;
    end

    if N_code == 1
        BER_uncoded = BER;
    else
        BER_coded = BER;
    end
end
%
% ============================== Plot ==============================
figure('Name', [modulation ', ' channel_type ' Channel']);
semilogy(Eb_N0_dB , BER_uncoded ,'b-', 'LineWidth', 1.25) ;
hold on
semilogy(Eb_N0_dB , BER_coded ,'b--', 'LineWidth', 1.25);
hold off
grid on
xlabel('Eb/No (dB)');
ylabel('BER');
ylim([1e-5 1])
legend([modulation,'(Uncoded)'], [modulation,'(coded 1/5)']) ;

title([modulation ', ' channel_type ' Channel'])
```

```matlab
% ==================== Functions =====================
function R = generate_data(m, N_fft, N_code)
    R = randi([0 1], 1, floor(N_fft/N_code)*m);
end


function R = Encoder(data, m, N_fft, N_code)
    encoded_data = repelem(data, N_code);
    R = [encoded_data, zeros(1, (m*N_fft)-length(encoded_data))];
end

function R = Decoder(data, m, N_fft, N_code)
    padding_size = length(data) - floor(N_fft/N_code)*m*N_code;
    data_noPad = data(1:end-padding_size);

    % Majority vote decoder
    R = sum(reshape(data_noPad, N_code, []), 1) >= ceil(N_code/2);
end

function [Channel_Output, H] = Channel(data, m, Eb_N0_dB, N_code, N_fft, cp, channeltype,
energy_type)
    % Noise
    segma = sqrt((1/2)./10^(Eb_N0_dB/10));
    Eb_actual = (sum(abs(data).^2)/length(data))/m;
    noise = sqrt(Eb_actual) * segma  * (randn(size(data)) + 1j*randn(size(data)));

    if isequal(energy_type, 'Same_Per_Info')
        noise = sqrt(N_code) * noise;
    end

    if isequal(channeltype, 'Flat')
        % Flat fading channel
        H = sqrt(1/2)*(randn(1) + 1j*randn(1));
        Channel_Output = H*data + noise;
    else
        % Frequency selective channel
        H = sqrt(1/2)*(randn(1, N_fft) + 1j*randn(1, N_fft));

        % Remove cyclic extension
        data_cp_removed = data(cp+1:end);

        % Frequency domain
        data_freq = fft(data_cp_removed, N_fft);

        % Applay Freqency Selctive Channel
        ch_out_freq = data_freq .* H;

        % Time domain
        ch_out = ifft(ch_out_freq, N_fft);

        % Add cyclic extension
        ch_out = [ch_out(end-cp+1:end), ch_out];

        %Add Noise
        Channel_Output = ch_out + noise;
    end
end



function R = Interleaver(mod, data)
    if isequal(mod, 'BPSK')
       reshape_tx_data = reshape(data, 8, 32)';
    elseif isequal(mod, 'QPSK')
       reshape_tx_data = reshape(data, 16, 32)';
    elseif isequal(mod, '16QAM')
       reshape_tx_data = reshape(data, 32, 32)';
    end
    R = reshape_tx_data(:)';
end

function R = DeInterleaver(mod, data)
    if isequal(mod, 'BPSK')
       reshape_rx_data = reshape(data, 32, 8);
    elseif isequal(mod, 'QPSK')
       reshape_rx_data = reshape(data, 32, 16);
    elseif isequal(mod, '16QAM')
       reshape_rx_data = reshape(data, 32, 32);
    end
    reshape_rx_data = reshape_rx_data';
    R = reshape_rx_data(:)';
end



function R = Mapper(data, mod)
    if isequal(mod, 'BPSK')
            R = data*2-1;
    elseif isequal(mod, 'QPSK')
        data_reshaped = reshape(data, 2, []);
        % 00 -> -1 -1 => -1 - j
        % 01 -> -1 +1 => -1 + j
        % 10 -> +1 -1 => +1 - j
        % 11 -> +1 +1 => +1 + j
        I = 2*data_reshaped(1,:) - 1;      % MSB
        Q = 2*data_reshaped(2,:) - 1;      % LSB
        data_mapped = I + 1j * Q;
        R = data_mapped;
    elseif isequal(mod, '16QAM')
        R = mod16(data);
    end
end

function R = Demapper(data, mod)
    if isequal(mod, 'BPSK')
        % if data > 0 it represents 1 else represents 0
        R = data > 0;
    elseif isequal(mod, 'QPSK')
        demapped_I = real(data) > 0;
        demapped_Q = imag(data) > 0;
        R= reshape([demapped_I; demapped_Q], 1, []);
    elseif isequal(mod, '16QAM')
        R = demod16(data);
    end
end



function [rxsig] = mod16(txbits)
    psk16mod=[1+j*1 3+j*1 1+j*3 3+j*3 1-j*1 3-j*1 1-j*3 3-j*3 -1+j*1 -3+j*1 -1+j*3 -3+j*3 -1-j*1
-3-j*1 -1-j*3 -3-j*3];
    sigham=txbits;
    m=4;
    sigqam16=reshape(sigham,m,length(sigham)/m);
    rxsig=(psk16mod(bi2de(sigqam16')+1));
end


function [rxbits]=demod16(rxsig)
    m=4;
    psk16demod=[15 14 6 7 13 12 4 5 9 8 0 1 11 10 2 3];
    rxsig(find(real(rxsig)>3))=3+j*imag(rxsig(find(real(rxsig)>3)));
    rxsig(find(imag(rxsig)>3))=real(rxsig(find(imag(rxsig)>3)))+j*3;
    rxsig(find(real(rxsig)<-3))=-3+j*imag(rxsig(find(real(rxsig)<-3)));
    rxsig(find(imag(rxsig)<-3))=real(rxsig(find(imag(rxsig)<-3)))-j*3;
    rxdemod=round(real((rxsig+3+j*3)/2))+j*round(imag((rxsig+3+j*3)/2));
    rxdebi=real(rxdemod)+4*(imag(rxdemod));
    sigbits=de2bi(psk16demod(rxdebi+1));
    rxbits= reshape(sigbits',1,size(sigbits, 1)*m);
end
```