Faculty of Engineering – Cairo University

Electronics And Electrical Communication Department

Third year-Mainstream

Digital Communications Project #3

Submitted to: Dr. Mohamed Nafie

Submitted to: Dr. Mohamed Khairy

Submitted to: TA. Mohamed Khaled

Team:33

Names	Sec	ID	Role
HossamEldin Abdelnasser Ali Mady	2	9220261	BFSK
Ahmed Mohamed Soltan	1	9220080	Report
Yousef Khaled Omar Mahmoud	4	9220984	BPSK, QPSK, 16-QAM

Table of Contents

Table of figures:

1 Introduction

Modern communication systems heavily rely on digital modulation methods to ensure accurate and efficient data transfer across different transmission media. These modulation schemes vary in terms of design complexity, bandwidth efficiency, and resilience to noise, making it essential to choose the most suitable one based on specific system needs. This report and experiment conduct a comparative analysis of key digital modulation formats—including BPSK, QPSK, 8-PSK, 16-QAM, and BFSK—by evaluating their Bit Error Rate (BER) performance in the presence of Additive White Gaussian Noise (AWGN), utilizing MATLAB-based simulation tools.

1.1 System Description

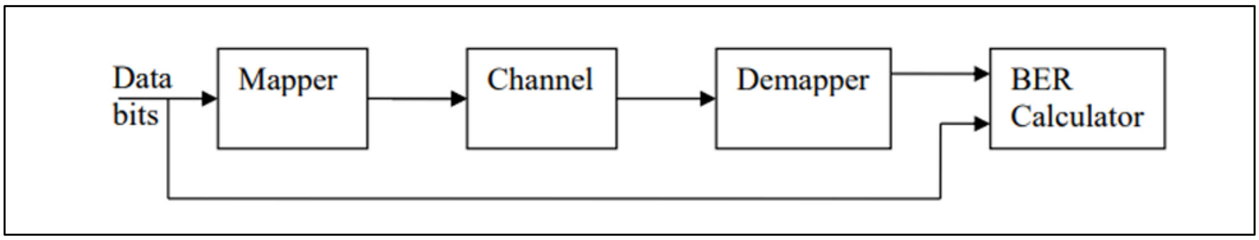

Figure 1 Communication System Blocks

As shown in figure 1, We're going to simulate a simple digital modulation communication system using MATLAB which consists of:

1.1.1 Data Bits Generator:

```
%--------Part 1----------
% ========================
% Simulation Parameters
% ========================
bits_Num = 6 * 2^15;                                  % Number of bits to transmit
mod_types = {'BPSK', 'QPSK', 'QPSKNG', '8PSK', '16-QAM', 'BFSK'}; % Cell array of modulation types
SNR_db_range = -4:1:16;

% Generate random bits (same for all modulations for fair comparison)
Tx_bits = randi([0 1], 1, bits_Num);
```

The data is generated randomly in binary representation that will be transmitted using the mapper.

1.1.2 TX Mapper:

```
function [Tx_Vector, Table, Eavg, Eb] = mapper(bits, mod_type)
    % MAPPER Digital modulation mapper with explicit symbol table and energy calculation
    % Inputs:
    %   bits     - Binary input array (row vector)
    %   mod_type - 'BPSK', 'QPSK', 'QPSKNG', '8PSK', 'BFSK', '16-QAM'
    % Outputs:
    %   Tx_Vector - Complex modulated symbols
    %   Table     - Constellation points (M-ary symbols)
    %   Eavg      - Average symbol energy (normalized)
    %   Eb        - Energy per bit

    % Ensure bits are row vector
    bits = bits(:)';

    % Define modulation parameters
    switch upper(mod_type)
        case 'BPSK'
            n = 1;  % bits per symbol
            M = 2;   % constellation size
            Table = [-1, 1];  % BPSK symbols (real)

        case 'QPSK'
            n = 2;
            M = 4;
            Table = [-1-1j, -1+1j, 1-1j, 1+1j];  % QPSK symbols

        case 'QPSKNG'
            n = 2;
            M = 4;
            Table = [-1-1j, -1+1j, 1+1j, 1-1j];  % QPSKNG symbols

        case '8PSK'
            n = 3;
            M = 8;
            angles =[0, 1, 3, 2, 7, 6, 4, 5]*pi/4;  % Gray-coded 8PSK
            Table = exp(1j*angles);

        case 'BFSK'
            n=1;
            M=2;
            Table = [ 1, 1j];

        case '16-QAM'
            n = 4;
            M = 16;
            % 16-QAM with unit average power (normalized)
            Table = [-3-3j, -3-1j, -3+3j, -3+1j, ...
                     -1-3j, -1-1j, -1+3j, -1+1j, ...
                      3-3j,  3-1j,  3+3j,  3+1j, ...
                      1-3j,  1-1j,  1+3j,  1+1j];

        otherwise
            error('Unsupported modulation type: %s', mod_type);
    enda

    % Pad bits if not multiple of n
    if mod(length(bits), n) ~= 0
        bits = [bits zeros(1, n - mod(length(bits), n))];
    end

    % Calculate average symbol energy
    Eavg = mean(abs(Table).^2);

    % Calculate average bit energy
    Eb =  Eavg / n;

    % Reshape into n-bit groups
    bit_groups = reshape(bits, n, [])';

    % Convert to decimal symbols (0 to M-1)
    Array_symbol = bi2de(bit_groups, 'left-msb') + 1;  % MATLAB uses 1-based indexing

    % Map to constellation points
    Tx_Vector = Table(Array_symbol);
end
```

The Tx mapper encodes the input binary data into symbols by using table for each modulation, using the equation: **XBB = XI + j XQ**

the tables are:

Modulation Tabels:

Modulation	Bits	Decimal	Symbol
BPSK	0	0	-1
	1	1	1
QPSK	00	0	-1-j
	01	1	1+j
	10	2	1-j
	11	3	1+j
QPSKNG (not Grey)	00	0	-1-j
	01	1	-1+j
	10	2	1+j
	11	3	1-j
8PSK	000	0	1
	001	1	$\frac{1}{\sqrt{2}}(1+j)$
	010	2	$\frac{1}{\sqrt{2}}(-1+j)$
	011	3	j
	100	4	$\frac{1}{\sqrt{2}}(1-j)$
	101	5	-j
	110	6	-1
	111	7	$\frac{1}{\sqrt{2}}(-1-j)$
16QAM	0000	0	-3-3j
	0001	1	-3-j
	0010	2	-3+3j
	0011	3	-3+j
	0100	4	-1-3j
	0101	5	-1-j
	0110	6	-1+3j
	0111	7	-1+j
	1000	8	3-3j
	1001	9	3-j
	1010	10	3+3j
	1011	11	3+j
	1100	12	1-3j
	1101	13	1-j
	1110	14	1+3j
	1111	15	1+j
BFSK	0	0	1
	1	1	j

1.1.3 Channel

```
function noisy_signals = addAWGNChannel(SNR_range_db, clean_signal, Eb)
    % ADDAGWNCHANNEL General AWGN channel noise adder
    % Inputs:
    %   SNR_range_db - Array of SNR values in dB
    %   clean_signal - Input signal (vector or matrix)
    %   Eb - Energy per bit
    % Output:
    %   noisy_signals - Cell array of noisy signals for each SNR

    % Initialize output cell array
    noisy_signals = cell(length(SNR_range_db), 1);

    % Get size of input signal
    signal_size = size(clean_signal);

    % Process each SNR point
    for i = 1:length(SNR_range_db)
        % Convert SNR from dB to linear scale
        SNR_linear = 10^(SNR_range_db(i)/10);

        % Calculate noise power (N0)
        N0 = 1 / SNR_linear;

        % Generate proper noise
        if isreal(clean_signal)
            % Real noise for real signals
            noise = sqrt(Eb*N0/2) * randn(signal_size);
        else
            % Complex noise for complex signals
            noise = sqrt(Eb*N0/2) * (randn(signal_size) + 1j*randn(signal_size));
        end

        % Add noise to the signal
        noisy_signals{i} = clean_signal + noise;
    end

    % If only one SNR point was requested, return array instead of cell
    if length(SNR_range_db) == 1
        noisy_signals = noisy_signals{1};
    end
end
```

The channel represents a real communication medium which is AWGN (Additive White Gaussian Noise) channel. But it’s not in base band so we made an equivalent noise using the equation for a given energy-per-bit to noise ratio (Eb/N0):

$$\mu + \sigma * randn()$$

Where Noise Standrad deviation = $\sqrt{\frac{Eb}{No*2}}$ and mean =0.

1.1.4 RX Demapper:

```
function [received_bits] = demapper(received_symbols, mod_type)
    % DEMAPPER Digital demodulation demapper
    % Inputs:
    %   received_symbols - Complex received symbols (array or cell array)
    %   mod_type         - Modulation type ('BPSK', 'QPSK', etc.)
    % Output:
    %   received_bits   - Demodulated bit stream (array or cell array)

    % Check if input is cell array (multiple SNR cases)
    if iscell(received_symbols)
        % Process each SNR case
        received_bits = cell(size(received_symbols));
        for i = 1:numel(received_symbols)
            received_bits{i} = demodulate_symbols(received_symbols{i}, mod_type);
        end
    else
        % Single SNR case
        received_bits = demodulate_symbols(received_symbols, mod_type);
    end
end
function bits = demodulate_symbols(symbols, mod_type)
    % Helper function for actual demodulation

    % Determine bits per symbol
    switch upper(mod_type)
        case 'BPSK'
            n = 1;
        case 'QPSK'
            n = 2;
        case 'QPSKNG'
            n = 2;
        case '8PSK'
            n = 3;
        case {'16QAM', '16-QAM'}
            n = 4;
        case 'BFSK'
            n=1;
        otherwise
            error('Unsupported modulation type');
    end

    % Initialize output bits
    bits = zeros(1, length(symbols)*n);

    % ======================
    % Special case for BFSK
    % ======================
    if strcmpi(mod_type, 'BFSK')
        for i = 1:length(symbols)
            theta = angle(symbols(i));
            if (theta > pi/4 && theta < 5*pi/4)
                bits(i) = 1;
            else
                bits(i) = 0;
            end
        end
        return;
    end

    % Get constellation table from mapper
    [~, Table] = mapper([1], mod_type);

    % Demodulate each symbol
    for i = 1:length(symbols)
        % Find nearest constellation point
        [~, idx] = min(abs(symbols(i) - Table));

        % Convert to binary (0-based index)
        bin_str = dec2bin(idx-1, n);

        % Store bits
        bits((i-1)*n+1:i*n) = bin_str - '0';
    end
end
```

For the Rx demmaper we used a reversed logic from the Tx, we check what’s the nearest symbol from the table to the received one (decision region). And assigns the binary equivalent to it.

1.1.5 BER Calculator:

```
function [BER, bit_errors] = calculateBER(original_bits, received_bits)
    % CALCULATEBER Compute Bit Error Rate for single or multiple SNR cases
    % Inputs:
    %   original_bits - Transmitted bit sequence (1D array)
    %   received_bits - Received bits (1D array or cell array for multiple SNR)
    % Outputs:
    %   BER - Bit Error Rate (scalar or array matching received_bits input)
    %   bit_errors - Number of errors (scalar or array)

    % Ensure original bits are row vector
    original_bits = original_bits(:)';

    % Handle cell array input (multiple SNR cases)
    if iscell(received_bits)
        BER = zeros(size(received_bits));
        bit_errors = zeros(size(received_bits));

        for i = 1:numel(received_bits)
            [BER(i), bit_errors(i)] = calculateSingleBER(original_bits, received_bits{i});
        end
    else
        % Single SNR case
        [BER, bit_errors] = calculateSingleBER(original_bits, received_bits);
    end
end

function [BER, bit_errors] = calculateSingleBER(original_bits, received_bits)
    % Helper function for single SNR case BER calculation

    % Ensure received bits are row vector
    received_bits = received_bits(:)';

    % Trim received bits if longer (due to padding)
    if length(received_bits) > length(original_bits)
        received_bits = received_bits(1:length(original_bits));
    end

    % Calculate errors
    bit_errors = sum(original_bits ~= received_bits);
    BER = bit_errors / length(original_bits);
end
```

It's a simple comparison between TX and RX bits

BER = $\frac{\boldsymbol{Num\ of\ error}}{\boldsymbol{Num\ of\ bits}}$

2 Modulation Schemes

2.1 BPSK

2.1.1 Description

It's a simple technique which uses two phases, 0° and 180°.

2.1.2 Basis Functions

$$\varphi_1(t) = \sqrt{2/T_b}\, cos(\omega_c t)$$

2.1.3 Symbol's Mathematical Representation

$$S_i(t) = \sqrt{2E/T_b}\, cos(\omega_c t + (i-1)\pi)$$

$$S_1(t) = \sqrt{E}\varphi_1(t), S_2(t) = -\sqrt{E}\varphi_1(t)$$

2.2 QPSK

2.2.1 Description

It's an advanced version of BPSK which uses two bits to represent a symbol. This doubles the data rate compared to BPSK for the same bandwidth.

2.2.2 Basis Functions

$$\varphi_1(t) = \sqrt{2/T_b}\cos(\omega_c t)\,, \varphi_2(t) = \sqrt{2/T_b}\sin(\omega_c t)$$

2.2.3 Symbol's Mathematical Representation

$$S_i(t) = \sqrt{2E/T_b}\cos\left(\omega_c t + \frac{(2i-1)\pi}{4}\right)$$

$$S_1(t) = \sqrt{E/2}\big(\varphi_1(t) - \varphi_2(t)\big),\ S_2(t) = \sqrt{E/2}\big(-\varphi_1(t) - \varphi_2(t)\big)$$
$$S_3(t) = \sqrt{E/2}\big(-\varphi_1(t) + \varphi_2(t)\big),\ S_4(t) = \sqrt{E/2}\big(\varphi_1(t) + \varphi_2(t)\big)$$

2.3 8PSK

2.3.1 Description

It's a technique with three bits used to represent eight distinct phases. this results in a higher data rate compared to BPSK and QPSK but requires more precise synchronization.

2.3.2 Basis Functions

$$\varphi_1(t) = \sqrt{2/T_b}\cos(\omega_c t)\,, \varphi_2(t) = \sqrt{2/T_b}\sin(\omega_c t)$$

2.3.3 Symbol's Mathematical Representation

$$S_i(t) = \sqrt{2E/T_b}\cos\left(\omega_c t + \frac{(2i-1)\pi}{8}\right)$$

2.4 16QAM

2.4.1 Description

It’s a technique that uses both amplitude and phase modulation. It can encode four bits per symbol by using 16 different signal points

2.4.2 Basis Functions

$$\varphi_1(t) = \sqrt{2/T_b}\cos(\omega_c t)\,, \varphi_2(t) = \sqrt{2/T_b}\sin(\omega_c t)$$

2.4.3 Symbol’s Mathematical Representation

$S_i(t) = \sqrt{2E/T_b}\,[\,a_i\cos(\omega_c t) - \,b_i\sin(\,\omega_c t)\,]$, where $\boldsymbol{ai}$=±**1**,±**3**,±**5**,… and $\boldsymbol{bi}$=±**1**,±**3**,±**5**,…

3 Noise Free

We’re going to test if our Tx and Rx are working correctly before adding any noise:

3.1 The TX Mapper

For the Tx mapper, we just convert the bits into decimal values to index it with symbol table,

which is grey-coded, from the complex constellations:

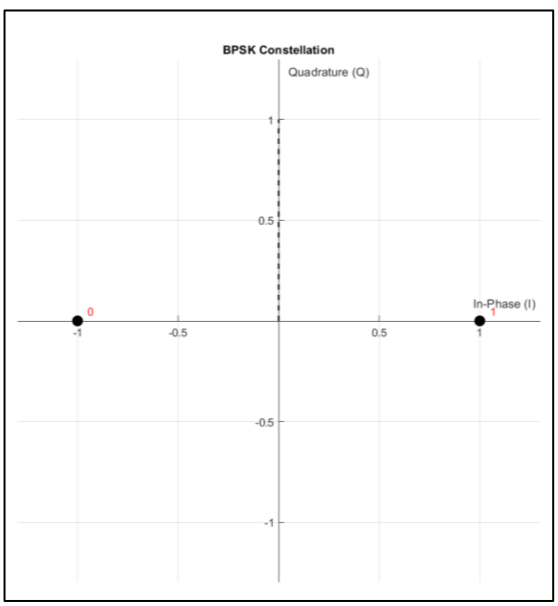

Figure 2 BPSK constellation

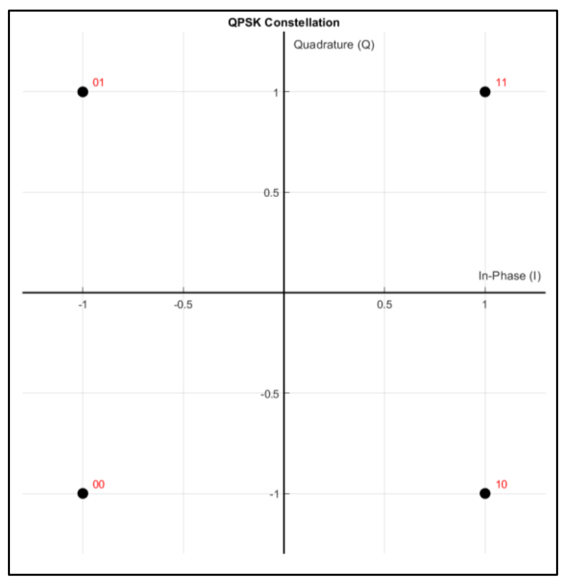

Figure 3 QPSK constellation

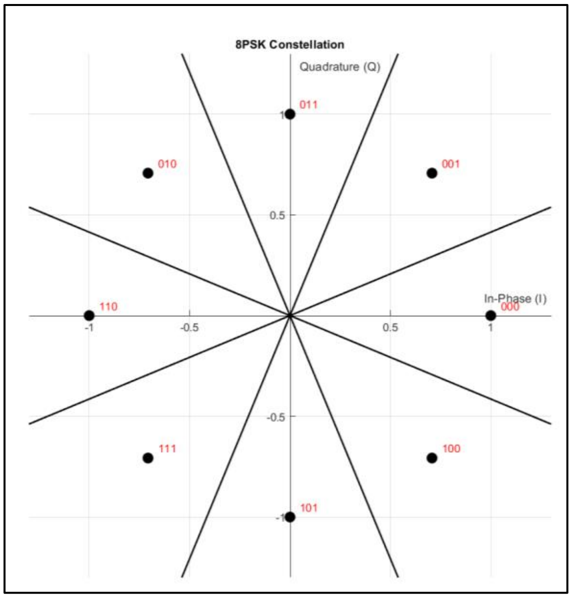

Figure 4 8PSK constellation

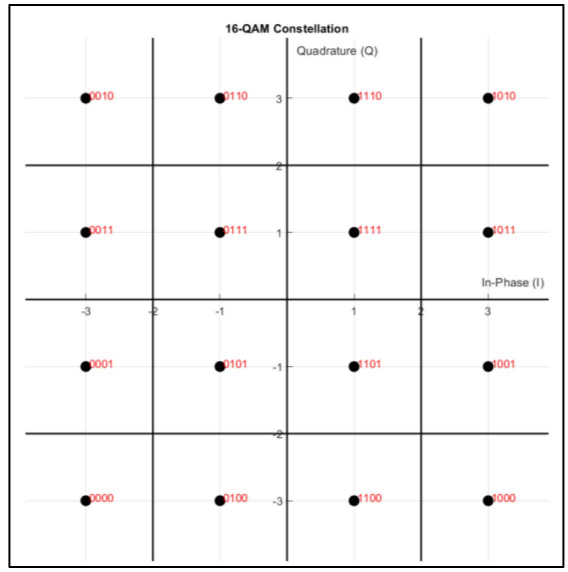

Figure 5 16-QAM constellation

As shown in the figures 2, 3, 4 and 5, we just make some linear algebra operations. As the I is the real part and Q is the imaginary part

```
function [Tx_Vector, Table] = mapper(bits, mod_type)
```

The **mapper** function converts a binary bitstream into complex symbols based on the selected modulation type (e.g., BPSK, QPSK, 8PSK, or 16-QAM). It groups the input bits according to the number of bits per symbol, converts each group to a decimal index, and maps it to a predefined constellation point. The result is a sequence of modulated symbols (**Tx_Vector**) ready for transmission. The function also returns the constellation (**Table**) used for mapping.

3.2 The RX Demapper

```
function [received_bits] = demapper(received_symbols, mod_type
```

We designed the demapper to recover transmitted bits from received symbols. For **BPSK** and **QPSK**, we used decision boundaries to determine the transmitted bit based on the region in which the symbol lies. For **8PSK** and **16-QAM**, we calculated the Euclidean distance between the received symbol and all points in the constellation, selecting the one with the smallest distance. The corresponding bit pattern was then assigned.

Finally, we computed the **Bit Error Rate (BER)** by comparing the demapped bits with the original transmitted bits and dividing the number of mismatches by the total number of bits.

3.3 Simulation

```
% Simulation Parameters
% ========================
bits_Num = 48;                                 % Number of bits to transmit
mod_types = {'BPSK', 'QPSK', '8PSK', '16-QAM'}; % Cell array of modulation types

% Generate random bits (same for all modulations for fair comparison)
Tx_bits = randi([0 1], 1, bits_Num);

% Loop through all modulation types
for mod_idx = 1:length(mod_types)
    mod_type = mod_types{mod_idx};

    fprintf('\n=== Testing %s Modulation ===\n', mod_type);

    % 1. Mapping (Modulation)
    % ========================
    [tx_symbols, constellation] = mapper(Tx_bits, mod_type);

    % 2. Display Constellation
    % ========================
    drawConstellation(constellation, mod_type);
    title(sprintf('%s Constellation', mod_type));

    % 3. Demapping (Demodulation)
    % ========================
    Rx_bits = demapper(rx_symbols, mod_type);

    % 4. Display Results
    % ========================
    % Calculate BER
    [BER, bit_errors] = calculateBER(Tx_bits, Rx_bits);

    % Display input/output comparison
    fprintf('Original bits:\n');
    disp(reshape(Tx_bits, 16, [])'); % Display in 16-bit groups

    fprintf('Received bits:\n');
    disp(reshape(Rx_bits(1:bits_Num), 16, [])'); % Display in 16-bit groups

    fprintf('Bit errors: %d\n', bit_errors);
    fprintf('BER: %.2e\n\n', BER);

end
```

Now we will try a small noise free simulation to make sure that the Rx and Tx runs properly

In the simulation we'll generate random bits and modulate it with each type and check if there's an error

3.4 Simulation Results:

```
=== Testing BPSK Modulation ===
Bit errors: 0
BER: 0.00e+00
Original bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Received bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Bit errors: 0
BER: 0.00e+00
```

Figure 6 BPSK Test

```
=== Testing QPSK Modulation ===
Bit errors: 0
BER: 0.00e+00
Original bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Received bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Bit errors: 0
BER: 0.00e+00
```

Figure 7 QPSK Test

```
=== Testing 8PSK Modulation ===
Bit errors: 0
BER: 0.00e+00
Original bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Received bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Bit errors: 0
BER: 0.00e+00
```

Figure 8 8PSK Test

```
=== Testing 16-QAM Modulation ===
Bit errors: 0
BER: 0.00e+00
Original bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Received bits:
     0     1     1     0     1     1     0     1     0     1     0     1     1     0     1     0
     1     1     1     1     1     1     1     0     1     1     1     0     1     0     0     1
     1     0     1     0     1     0     1     0     0     1     0     1     1     1     0     0

Bit errors: 0
BER: 0.00e+00
```

Figure 9 16-QAM Test

As shown in the figures 6, 7, 8 and 9, The noise free has zero error which means that the Tx and Rx are working properly.

4 AWGN channel

To simulate a realistic communication channel, we added **Additive White Gaussian Noise (AWGN)** to the transmitted signal. The function addAWGNChannel takes a clean modulated signal and adds noise based on a specified **Signal-to-Noise Ratio (SNR)** in dB.

For each SNR value:

- The SNR is converted from dB to a linear scale.
- Using the bit energy (**Eb**), the corresponding noise power (**N_0**) is calculated.
- Gaussian noise is generated with a variance proportional to the noise power:
    - If the signal is real (e.g., BPSK), real noise is used.
    - If the signal is complex (e.g., QPSK, 8PSK), complex noise is generated.
- This noise is added to the signal, simulating how signals are affected in real-world channels.

This allows us to observe how different modulation schemes perform under various noise conditions and evaluate their robustness by analyzing the resulting **Bit Error Rate (BER)**.

4.1 Code:

```
function noisy_signals = addAWGNChannel(SNR_range_db, clean_signal, Eb)
    % ADDAGWNCHANNEL General AWGN channel noise adder
    % Inputs:
    %   SNR_range_db - Array of SNR values in dB
    %   clean_signal - Input signal (vector or matrix)
    %   Eb - Energy per bit
    % Output:
    %   noisy_signals - Cell array of noisy signals for each SNR

    % Initialize output cell array
    noisy_signals = cell(length(SNR_range_db), 1);

    % Get size of input signal
    signal_size = size(clean_signal);

    % Process each SNR point
    for i = 1:length(SNR_range_db)
        % Convert SNR from dB to linear scale
        SNR_linear = 10^(SNR_range_db(i)/10);

        % Calculate noise power (N0)
        N0 = 1 / SNR_linear;

        % Generate proper noise
        if isreal(clean_signal)
            % Real noise for real signals
            noise = sqrt(Eb*N0/2) * randn(signal_size);
        else
            % Complex noise for complex signals
            noise = sqrt(Eb*N0/2) * (randn(signal_size) + 1j*randn(signal_size));
        end

        % Add noise to the signal
        noisy_signals{i} = clean_signal + noise;
    end

    % If only one SNR point was requested, return array instead of cell
    if length(SNR_range_db) == 1
        noisy_signals = noisy_signals{1};
    end
end
```

4.2 Simulation Results:

4.2.1 BPSK:

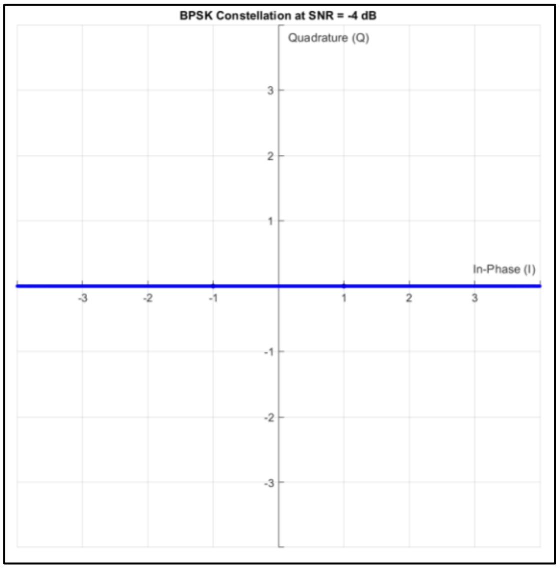

Figure 11 Noise on BPSK with SNR = -4 dB

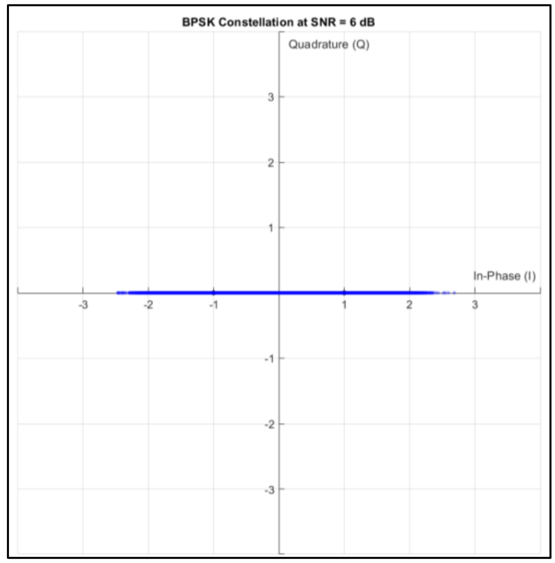

Figure 10 Noise on BPSK with SNR = 6 dB

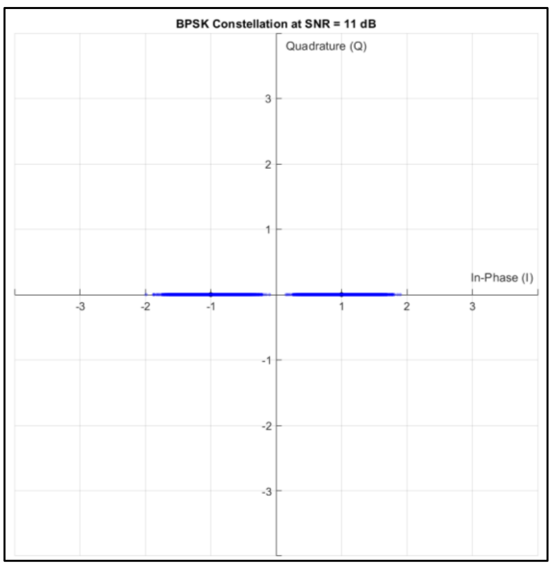

Figure 13 Noise on BPSK with SNR = 11 dB

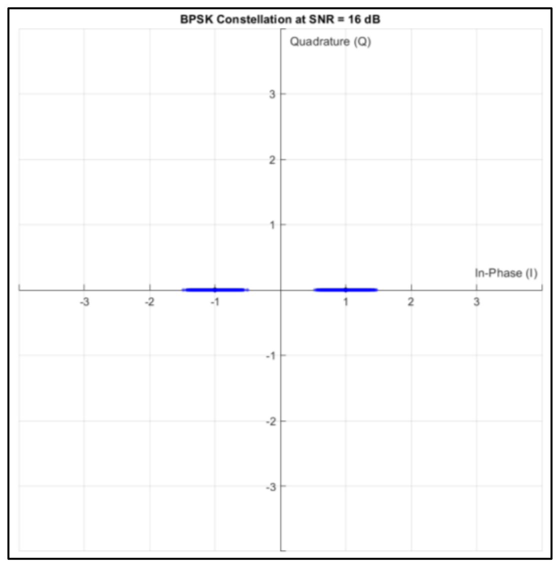

Figure 12 Noise on BPSK with SNR = 16 dB

4.2.2 QPSK:

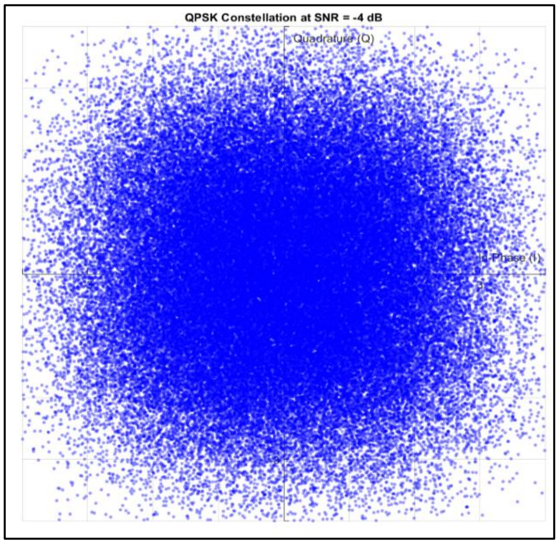

Figure 15 Noise on QPSK with SNR = -4 dB

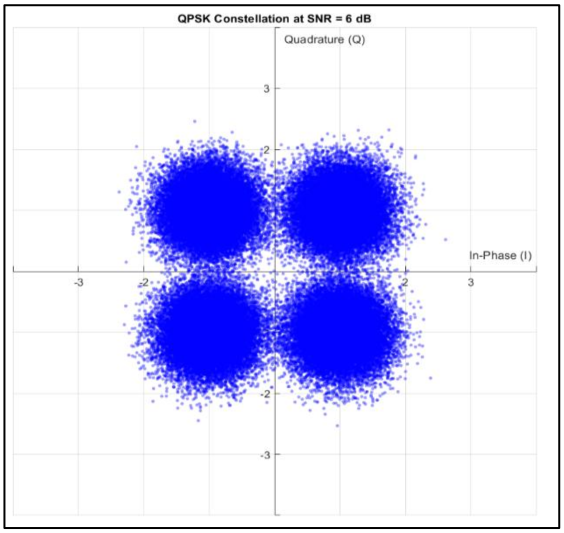

Figure 14 Noise on QPSK with SNR = 6 dB

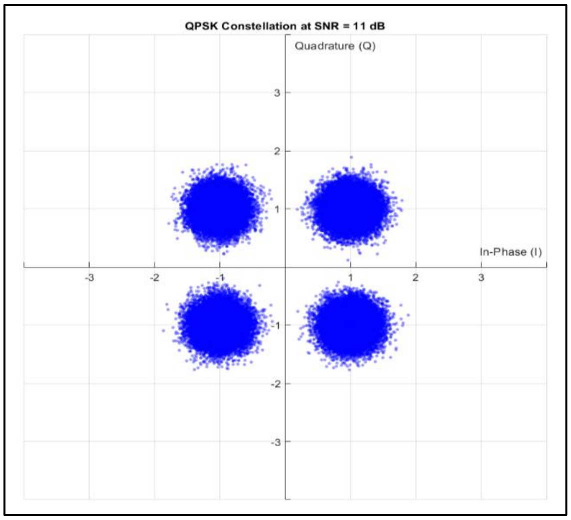

Figure 16 Noise on QPSK with SNR = 11 dB

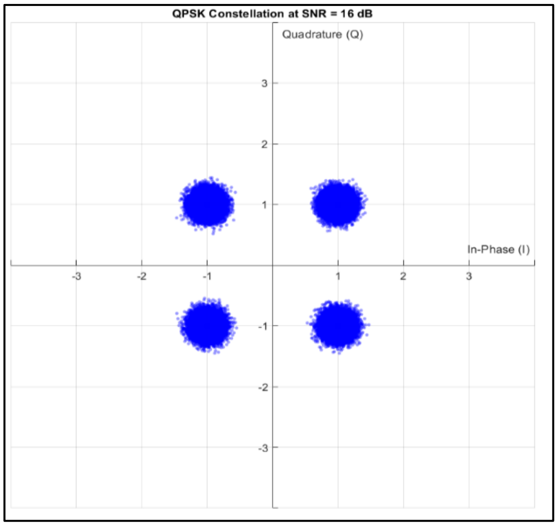

Figure 17 Noise on QPSK with SNR = 16 dB

4.2.3 8PSK:

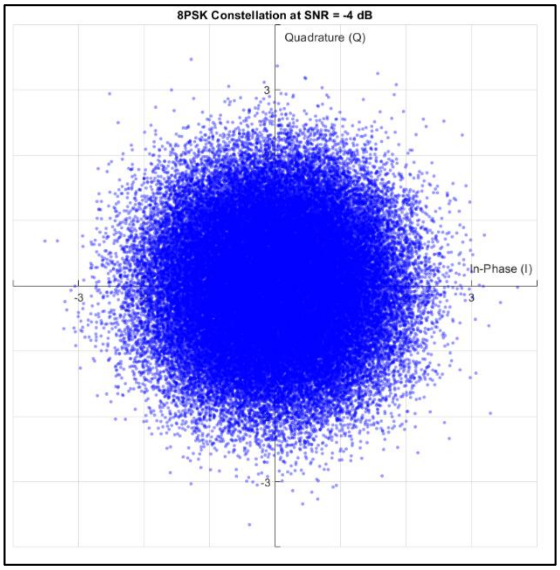

Figure 18 Noise on 8PSK with SNR = -4 dB

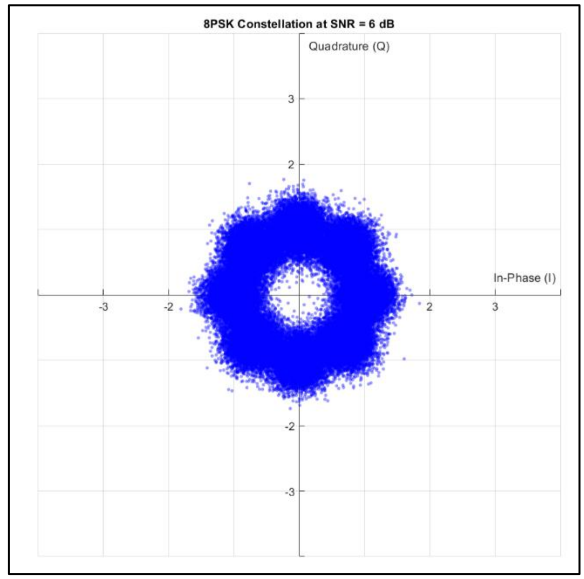

Figure 19 Noise on 8PSK with SNR = 6 dB

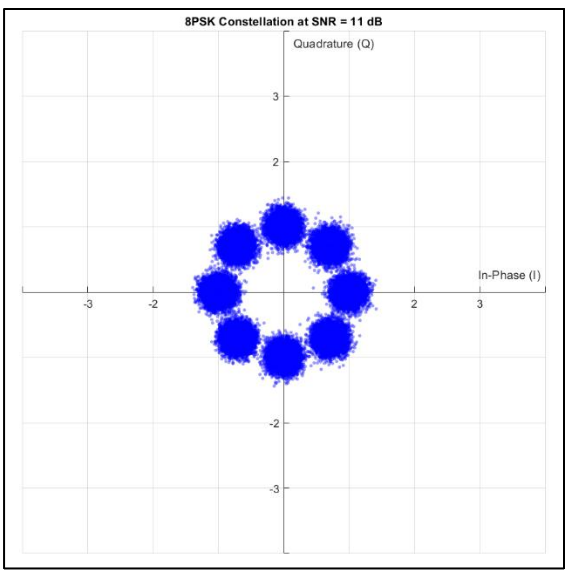

Figure 20 Noise on 8PSK with SNR = 11 dB

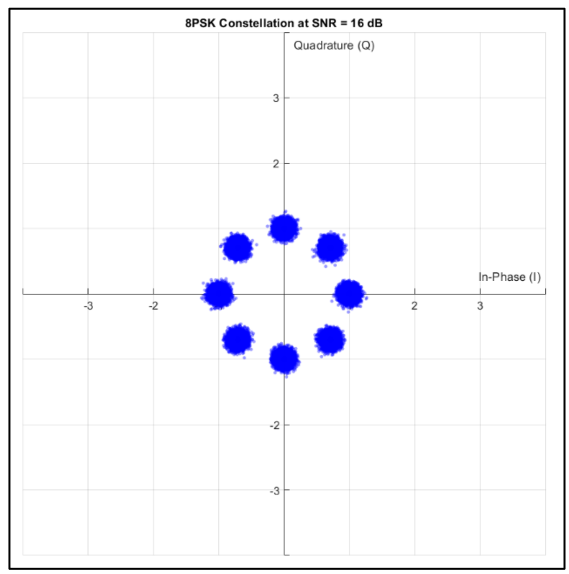

Figure 21 Noise on 8PSK with SNR = 16 dB

4.2.4 16QAM:

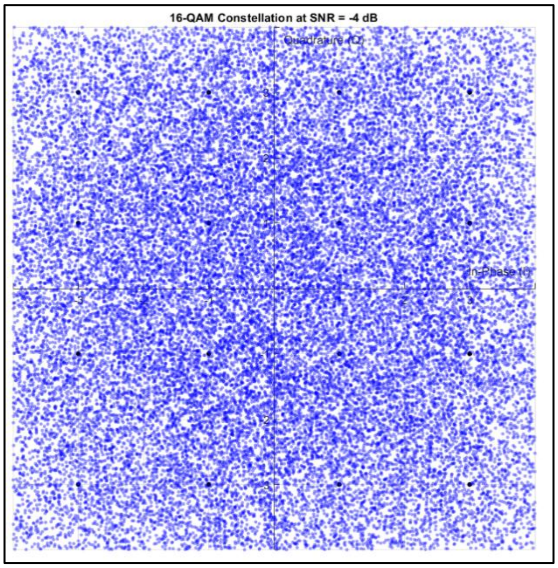

Figure 23 Noise on 16QAM with SNR = -4 dB

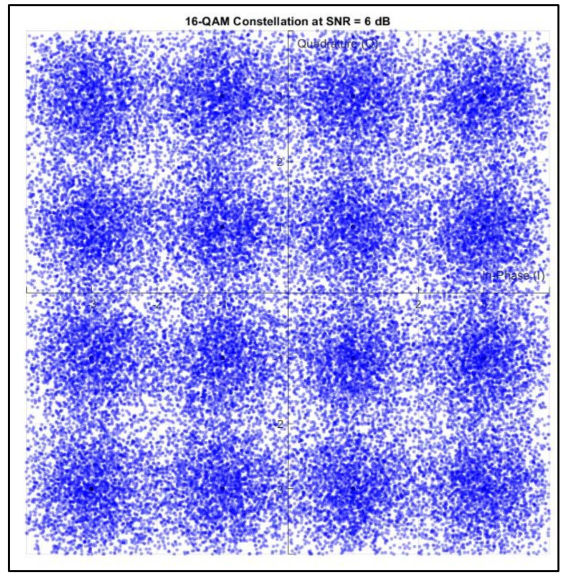

Figure 22 Noise on 16QAM with SNR = 1 dB

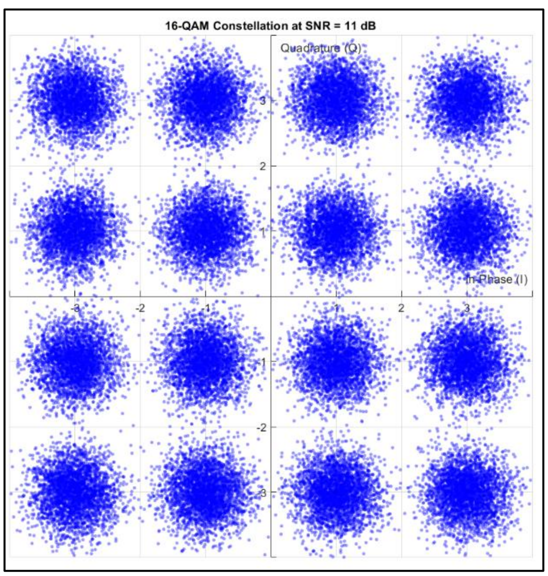

Figure 24 Noise on 16QAM with SNR = 11 dB

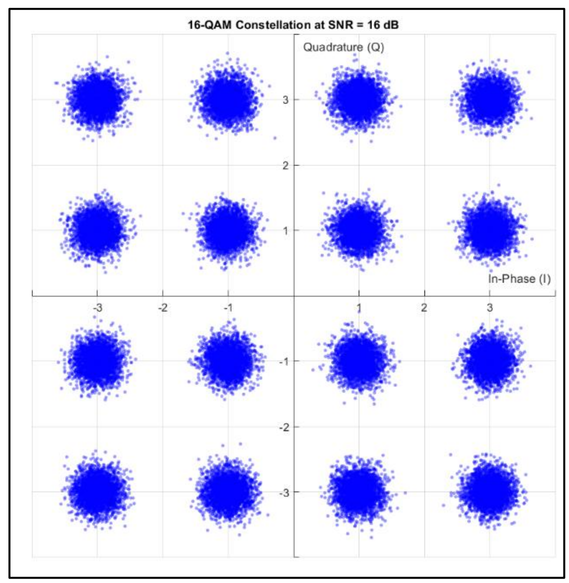

Figure 25 Noise on 16QAM with SNR = 16 dB

From the constellation plots, it is evident that noise causes the received symbols to deviate from their original positions. The extent to which the symbols remain distinguishable under noise directly impacts the Bit Error Rate (BER). Modulation schemes with greater spacing between constellation points exhibit better noise resilience. Based on the observed scatter,

we can qualitatively rank the modulation schemes in terms of their robustness to noise as follows:

BPSK > QPSK > 8PSK > 16QAM > BFSK. This ranking reflects how well the symbols are separated and how easily they can be distinguished at the receiver despite the presence of noise.

5 BER

5.1 Hand Analysis:

5.1.1 BPSK:

the theoretical BER = $\frac{1}{2}\ erfc(\sqrt{\frac{Eb}{No}})$

This can be obtained by performing integration on the decision boundaries which reduces to the closed form written in terms of the complementary error function and the SNR $\frac{Eb}{No}$

5.1.2 QPSK:

the theoretical BER = $\frac{1}{2}\ erfc(\sqrt{\frac{Eb}{No}})$

Since the QPSK is Grey-Encoded, the integration on the decision boundaries can make use of the independence obtained from the Grey Encoding, which makes the integration reduce to the same form obtained in the BFSK.

5.1.3 8PSK:

performing the integration on the decision boundaries for general MPSK modulation schemes is not an easy task and usually can not be written in a closed form. So either this integration can be evaluated using MATLAB or in better method, we are going to use the tight union bounds. For a general MPSK scheme, a tight union bound can be obtained

The tight union bound is $\frac{1}{\log_2 M}\ erfc(\sin(\frac{\pi}{M})\ \sqrt{\log_2 M * \frac{Eb}{No}})$

So we can say that for 8PSK BER ≈ $\frac{1}{3}\ erfc(\sin(\frac{\pi}{8})\ \sqrt{3 * \frac{Eb}{No}})$

5.1.4 16QAM:

An exact closed form expression for the BER is not easy to obtain. However, in a similar manner done for the general MPSK schemes, so we can say that for 16QAM BER = $\frac{3}{8}\ erfc(\sqrt{\frac{Eb}{2.5*No}})$

5.1.5 Code:

```
% Plot theoretical or tight upper bound BER
        switch Mod_Types{idx}
            case 'BPSK'
                BER_theory = 0.5 * erfc(sqrt(EbNo));
            case 'QPSK'
                BER_theory = 0.5 * erfc(sqrt(EbNo)); % same as BPSK
            case 'QPSKNG'
                BER_theory = 0.5 * erfc(sqrt(EbNo)); % same as QPSK
            case '8PSK'
                BER_theory = erfc(sin(pi/8) * sqrt(3 * EbNo)) / 3;
            case '16-QAM'
                BER_theory = (3/8)*erfc(sqrt((2/5)*EbNo));
            case '64qam'
                BER_theory = (7/24)*erfc(sqrt((7/21)*EbNo));
            case 'BFSK'
                BER_theory = 0.5*erfc(sqrt(0.5*EbNo));
         end
```

5.2 Simulation Results

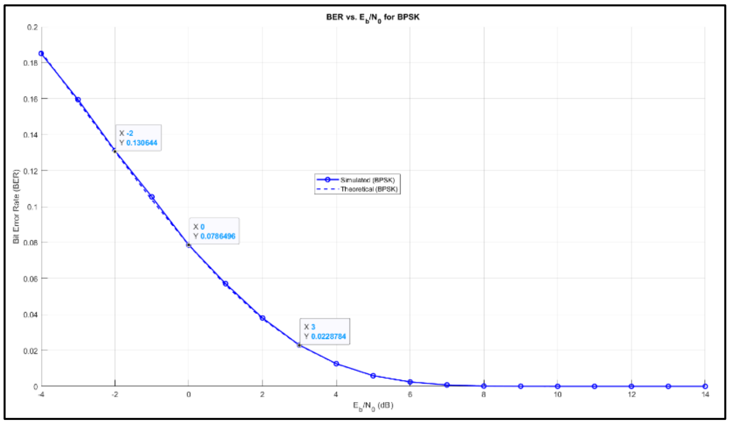

Figure 27 Simulated vs Theoretical BER for BPSK

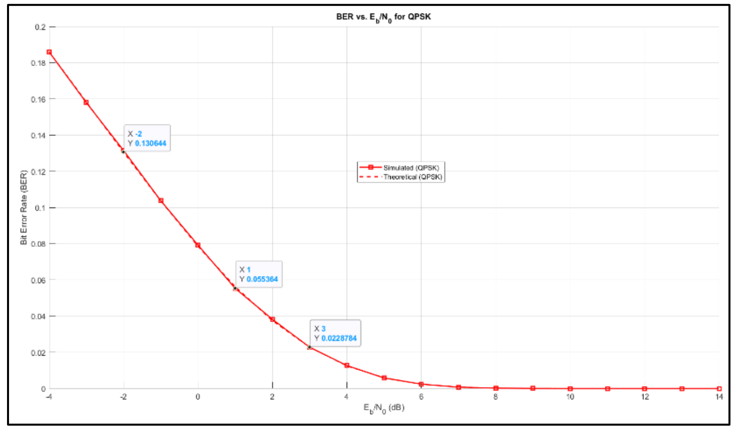

Figure 28 Simulated vs Theoretical BER for QPSK

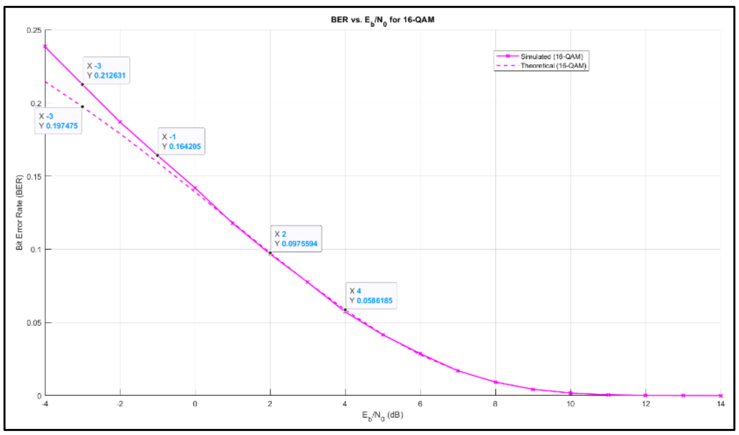

Figure 29 Simulated vs Theoretical BER for 16QAM

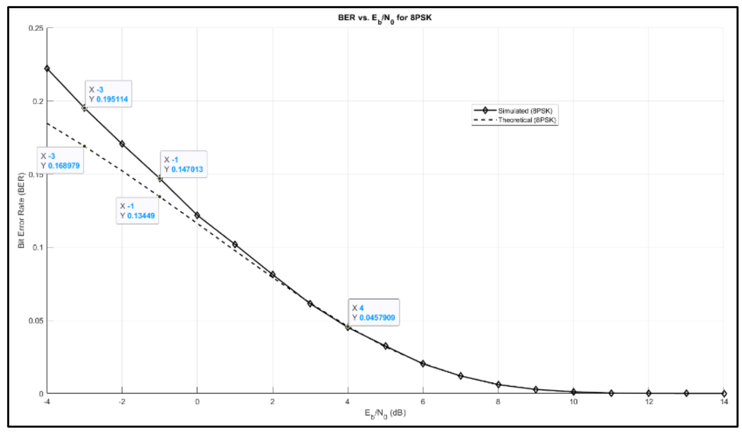

Figure 26 Simulated vs Theoretical BER for 8PSK

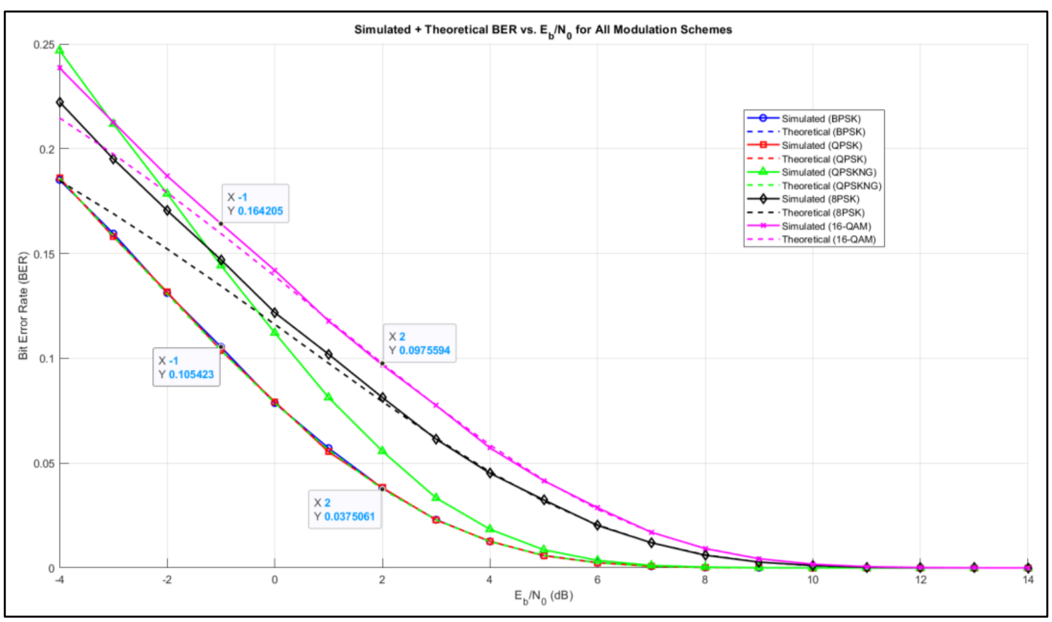

Figure 30 Simulated and Theoritical BER for BPSK, QPSK, 8PSK and 16QAM

5.3 Results Discussion

- As shown in figures 27:30, he simulated BER for the four modulation schemes almost matches the theoritical BER.

- The BER curves show that **BPSK** and **Gray-coded QPSK** perform almost identically.

- An important insight, also reflected in the theoretical BER equations discussed earlier, is that BPSK and QPSK exhibit identical BER performance. This is clearly illustrated in the simulated results (Figure 30), where the BER curves of both modulation schemes nearly overlap. The reason lies in their signal structure: BPSK transmits data using a single basis function, whereas QPSK utilizes two orthogonal basis functions. This allows QPSK to convey twice the amount of information within the same bandwidth, effectively doubling the data rate without increasing the error rate. As a result, QPSK achieves superior spectral efficiency compared to BPSK while maintaining equivalent reliability. Consequently, in systems where multiple basis functions are feasible, QPSK is generally preferred over BPSK due to its more efficient use of available bandwidth.

- As observed in the results, **8PSK** exhibits a higher BER than both BPSK and QPSK. This is expected since 8PSK encodes 3 bits per symbol, making it more susceptible to noise. Additionally, the decision regions in its constellation are smaller, increasing the probability of incorrect symbol detection under noisy conditions.

- **16-QAM** shows the highest BER among all schemes. With each symbol representing 4 bits, even small deviations due to noise can result in multiple bit errors. However, despite its lower noise immunity, 16-QAM offers better spectral efficiency, making it advantageous in bandwidth-constrained systems.

- Finally, the close agreement between the simulated and theoretical BER curves, as illustrated in the plots, confirms the accuracy of the simulation. This match is largely due to the use of a **large number of transmitted bits**, which ensures statistical reliability and convergence to theoretical expectations.

6 QPSK Not Grey

In digital modulation schemes, the arrangement of bits within the signal constellation significantly impacts the system's bit error performance.

Gray coding is commonly used across modulation techniques because it helps minimize the Bit Error Rate (BER) for a given Symbol Error Rate (SER). While SER—the likelihood of incorrectly identifying a symbol—mainly depends on the Signal-to-Noise Ratio (SNR), BER is influenced by both the SNR and the bit mapping strategy used in the constellation.

Gray coding ensures that adjacent constellation points differ by only one bit, thereby reducing the chance of multiple bit errors when a symbol is incorrectly decoded.

To illustrate the impact of bit mapping on performance, this section compares QPSK simulations using Gray coding and an alternative non-Gray mapping scheme, highlighting the difference in BER outcomes as visualized in (Figure 31).

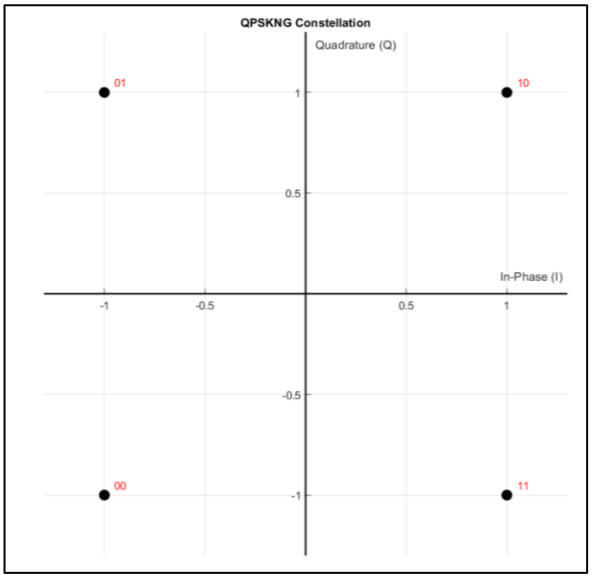

Figure 31 QPSK NG constellation

6.1 Simulation Results

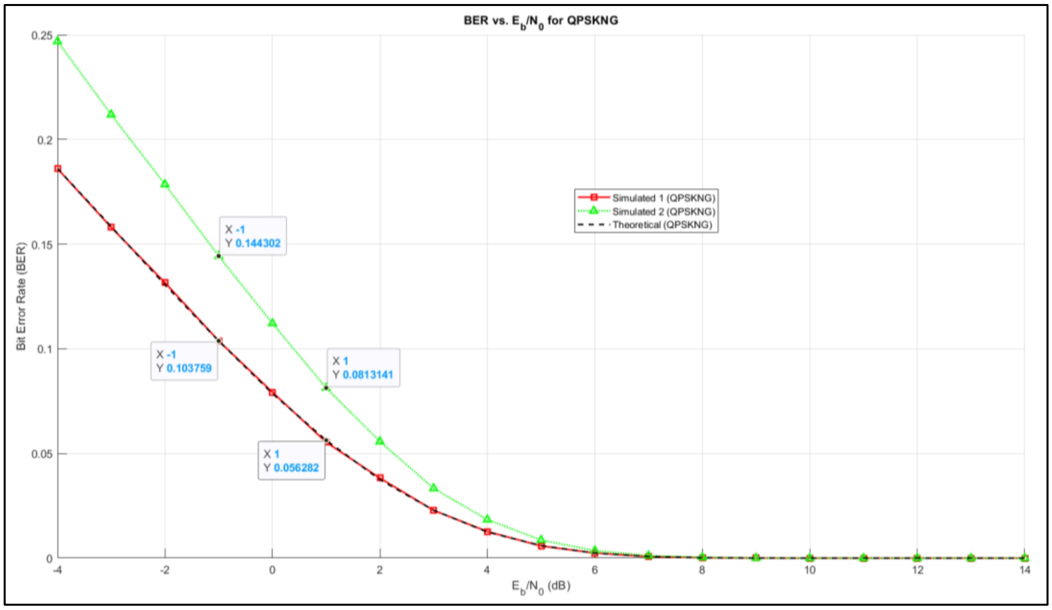

Figure 32 QPSK vs QPSK NG BER

6.2 Results Discussion

As illustrated in (Figure 32), both QPSK modulation variants operate under the same Signal-to-Noise Ratio (SNR), resulting in identical Symbol Error Rates (SER). However, the QPSK variant using Gray coding exhibits a significantly lower Bit Error Rate (BER) compared to its non-Gray coded counterpart. This advantage arises from the way Gray coding structures the constellation: each symbol differs from its nearest neighbors by just a single bit. Therefore, even when a symbol is misidentified due to noise, the resulting bit error is minimized. This property makes Gray coding highly effective in reducing BER without impacting SER, which is why it is widely adopted in practical modulation schemes.

7 BFSK

7.1 Modulation Schemes

7.1.1 Description

It's a simple technique which uses two frequencies to represent binary data.

7.1.2 Basis Function

$$\phi_1(t) = \sqrt{\frac{2}{T_b}} \cos\left(2\pi \frac{n_c + 1}{T_b} t\right), \quad 0 \le t \le T_b$$

$$\phi_2(t) = \sqrt{\frac{2}{T_b}} \cos\left(2\pi \frac{n_c + 2}{T_b} t\right), \quad 0 \le t \le T_b$$

7.1.3 Symbol's Mathematical Representation

$$s_i(t) = \begin{cases} \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_i t), & 0 \le t \le T_b \\ 0 & \text{otherwise} \end{cases}$$

The Tx frequency is $f_i = (n_c + i)/T_b \quad , i = 1,2$

As shown in figure 33, The first symbol is mapped to 0 and second is mapped to j.

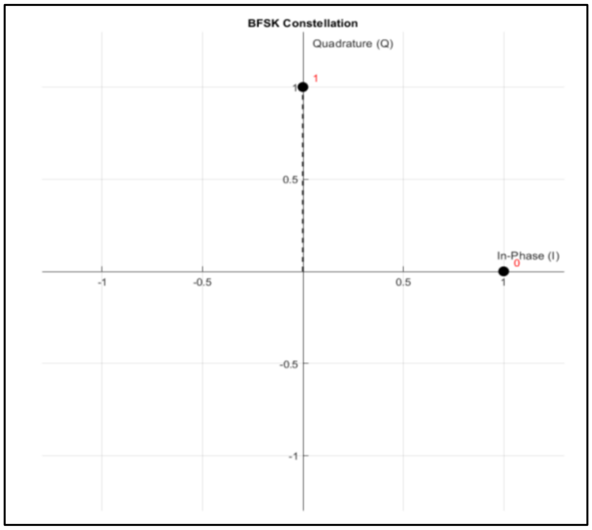

Figure 33 BFSK constellation

7.2 Simulation Results

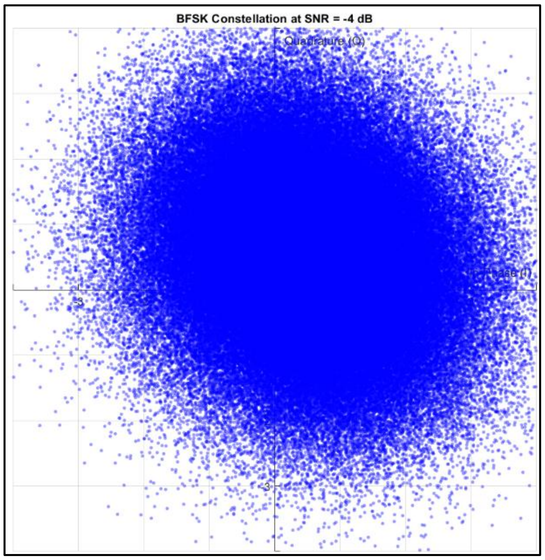

Figure 35 Noise on BFSK with SNR = -4 dB

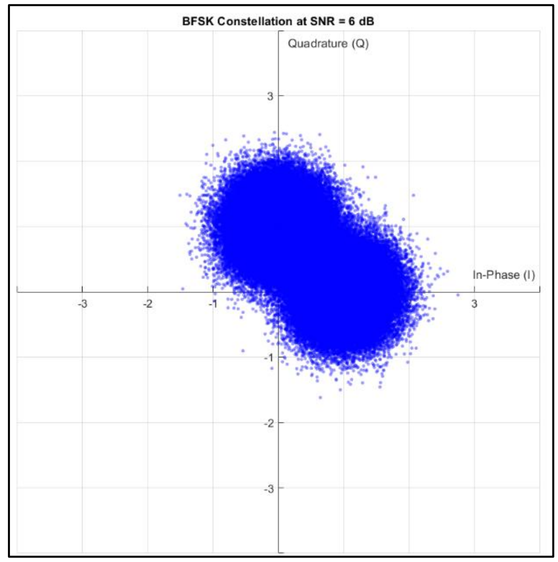

Figure 34 Noise on BFSK with SNR = 6 dB

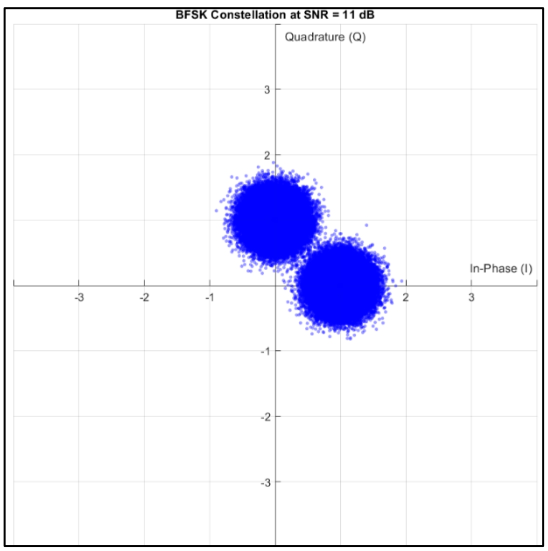

Figure 37 Noise on BFSK with SNR = 11 dB

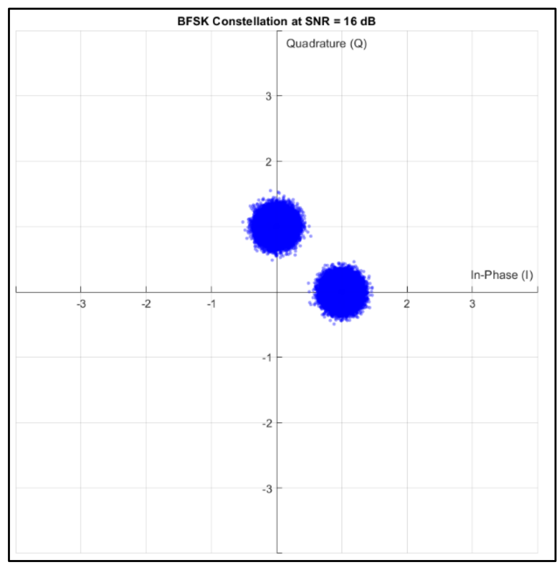

Figure 36 Noise on BFSK with SNR = 16 dB

7.3 BER

To determine the bit error rate (BER), we map bit 0 to 1 and bit 1 to the imaginary unit **j**, since each frequency corresponds to a unique orthogonal basis function.

After the signal passes through the channel, the received symbol's phase angle relative to the x-axis is used to retrieve the original bit.

If the angle lies between 45° and 225°, the bit is decoded as 1; otherwise, it is decoded as 0.

The theoretical BER is then calculated using the expression:

$$P_e = \frac{1}{2}\, erfc\left(\sqrt{\frac{2E_b}{T_b}}\right)$$

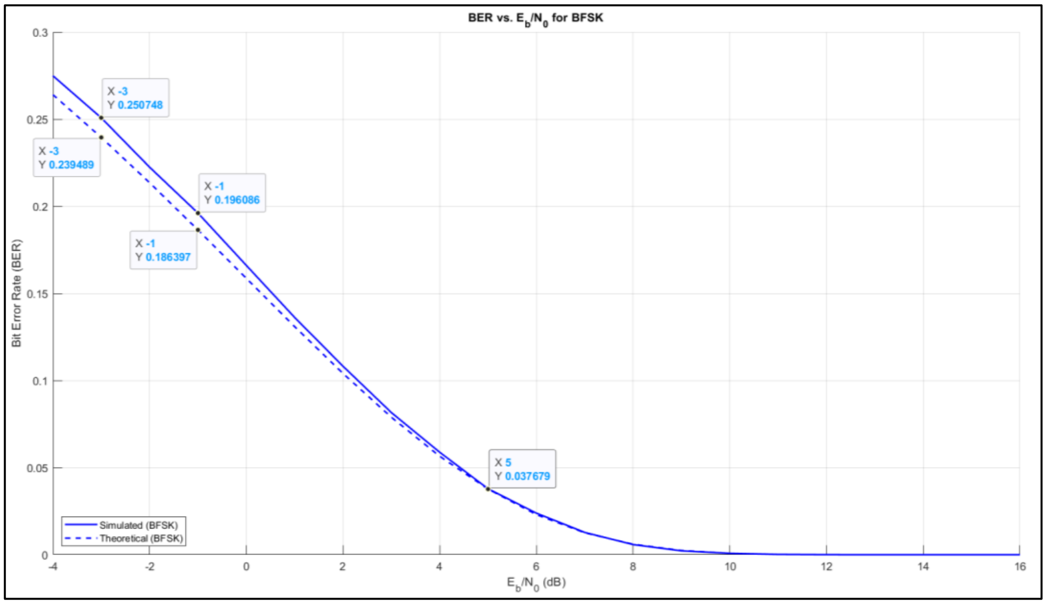

Figure 38 Simulated vs Theoretical BER for BFSK

As shown in figure 38, The Simulated BER is nearly equal and matched to the Theoritcal BER.

7.4 Base Band

The expression for the baseband equivalent signals for this set is

$$S_1(t) = \sqrt{2E/T_b}\cos(2\pi fct) \quad , S_2(t) = \sqrt{2E/T_b}\cos\left(2\pi(fc + \tfrac{1}{Tb})\, t\right)$$

$$Then, S_2(t) = \sqrt{\frac{2E}{T_b}}\left[\cos(2\pi f_c t)\cos\left(2\pi\frac{1}{T_b}t\right) - \sin(2\pi f_c t)\sin\left(2\pi\frac{1}{T_b}t\right)\right]$$

Where Carrier frequency1 = fc, Carrier frequency2 = fc + $\frac{1}{Tb}$

7.4.1 Code:

```
% ===============================
% declaring parameters (for PSD)
% ===============================
bits_Num = 100;  %less number of bits from the BER
N_realization = 10000;
data = randi([0 1], N_realization, bits_Num + 1);
samples_per_bit=7;
samples_num = samples_per_bit*bits_Num;
sampled_data = repelem(data, 1, samples_per_bit);
Tb = 0.07; % each sample takes 0.01 second

t = 0:Tb/samples_per_bit:Tb;
Fs = 100;
tx_with_delay = zeros(N_realization, 700);

% mapping to BB signals
tx_out = BFSK_BB(bits_Num, N_realization, Tb, Eb, samples_per_bit, sampled_data, t);

% random delay
for i = 1:N_realization
    r = randi([0 (samples_per_bit - 1)]);
    tx_with_delay(i,:) = tx_out(i,r+1:samples_num+r);
end


function [tx_out] = BFSK_BB(bits_Num, N_realization, Tb, Eb, samples_per_bit, sampled_data, t)
% BFSK_BB Generate baseband BFSK time-domain signal
%
% Inputs:
%   bits_Num       - Number of bits per realization
%   N_realization  - Number of realizations
%   Tb             - Bit duration in seconds
%   Eb             - Energy per bit
%
% Output:
%   tx_out         - Baseband BFSK output signal (N_realization x 7*(bits_Num+1))

    % === Derived Parameters ===
    total_samples = samples_per_bit * (bits_Num + 1); % Total samples per realization

    % === Initialize Output Signal ===
    tx_out = zeros(N_realization, total_samples);

    % === Map to Baseband BFSK Signal ===
    for i = 1:N_realization
        for j = 1:samples_per_bit:total_samples
            if sampled_data(i, j) == 0
                tx_out(i, j:j+samples_per_bit-1) = sqrt(2 * Eb / Tb);  % Non-coherent tone for 0
            else
                for k = 1:samples_per_bit
                    tx_out(i, j + k - 1) = sqrt(2 * Eb / Tb) * ...
                        (cos(2 * pi * t(k) / Tb) + 1i * sin(2 * pi * t(k) / Tb));
                end
            end
        end
    end
end
```

7.5 Auto Correlation

We generate 10,000 BFSK signals by mapping each bit to its baseband form and taking 7 samples per bit. A random delay is added to each signal. Then, we calculate the autocorrelation by comparing the center of each signal with its shifted versions and averaging the result.

7.5.1 Code

```
function BFSK_autocorr = compute_BFSK_autocorrelation(tx_with_delay)
% COMPUTE_BFSK_AUTOCORRELATION Computes autocorrelation of delayed BFSK signals
% centered at the middle sample.
%
% Input:
%   tx_with_delay   - Matrix of delayed BFSK signals (N_realization × N_samples)
%
% Output:
%   BFSK_autocorr   - Autocorrelation vector (1 × N_samples)

    [~, N_samples] = size(tx_with_delay);

    % Ensure N_samples is even for symmetric range
    if mod(N_samples, 2) ~= 0
        error('N_samples must be even for symmetric autocorrelation.');
    end

    BFSK_autocorr = zeros(1, N_samples);
    center_idx = N_samples / 2;

    for j = -center_idx+1 : center_idx
        i = j + center_idx;
        if i >= 1 && i <= N_samples
            p = conj(tx_with_delay(:, center_idx)) .* tx_with_delay(:, i);
            BFSK_autocorr(i) = sum(p) / length(p);
        end
    end
end
```

7.5.2 Simulation Result

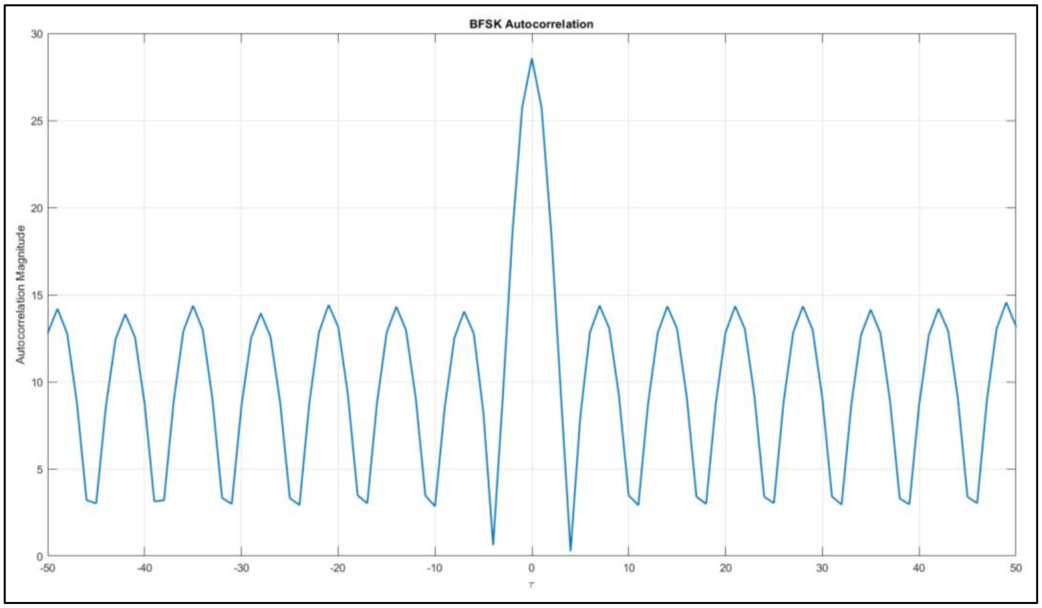

Figure 39 BFSK Auto Correlation

7.6 PSD:

To calculate the PSD, we first take the Fourier transform of the autocorrelation function. We also compute the theoretical PSD using its known formula.

$$S_{BB}(f) = \frac{2E_b}{T_b}\left(\delta\left(f - \frac{1}{T_b}\right) + \delta\left(f + \frac{1}{T_b}\right) + \frac{8E_b \cos^2(\pi f T_b)}{\pi^2(4T_b^2 f^2 - 1)^2}\right)$$

Since the simulated and theoretical signals use different baseband mappings, we shift the theoretical PSD to match the simulation results.

7.6.1 Code

```
% Practical PSD
BFSK_PSD = fftshift(fft(Rx_BFSK));     % Use fftshift to center the practical PSD
f = (-350:349) / 700 * Fs;             % Frequency vector for practical PSD
f_normalized = f * Tb;                 % Normalize frequency axis to match the theoretical PSD

% Theoretical PSD
PSD_theoritical = (8 * cos(pi * Tb * f).^2) ./ (pi^2 * (4 * Tb^2 * f.^2 - 1).^2);

% Handle Inf values in the theoretical PSD
idx = PSD_theoritical == Inf;
PSD_theoritical(idx) = 2;  % Change Inf to finite value for plotting
```

We generate a baseband BFSK (Binary Frequency Shift Keying) signal for power spectral density (PSD) analysis. we create random binary data across **multiple realizations**, then **upsamples** each bit to produce a continuous-time signal. The BFSK_BB function maps bit '0' to a constant amplitude (real) signal and bit '1' to a complex sinusoid, representing the frequency shift. A random delay is introduced to each realization to simulate timing offsets typically seen in real systems. The resulting signals can be used to compute an averaged PSD.

7.6.2 Simulation Result

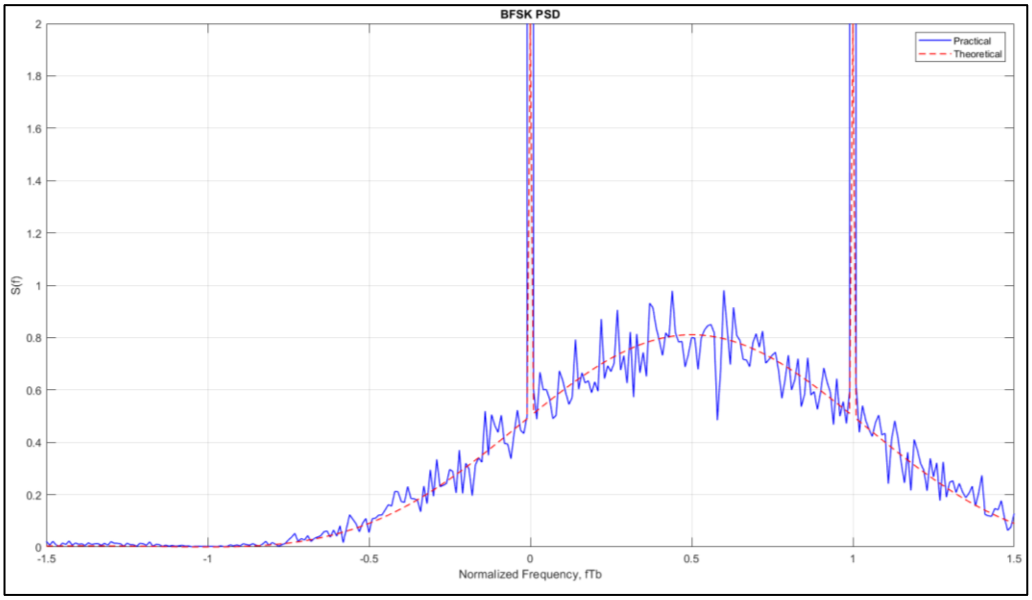

Figure 40 BFSK PSD

7.6.3 Results Discussion

The simulation results are in good agreement with the theoretical predictions for both Bit Error Rate (BER) and Power Spectral Density (PSD). Any small differences, like the slight shift in the PSD, are due to differences in how the signal is mapped in the simulation compared to the theory.

From the PSD graph, we see that the frequency separation $\Delta f = \frac{1}{T_b}$, where T_b is the bit period. This gives the bandwidth $BW = \Delta f$. The practical PSD is very close to the theoretical one, confirming that the BFSK implementation in the simulation is accurate and the analysis method is reliable.

8 Appendix

```
clear; clc; close all;

%--------Part 1----------
% ========================
% Simulation Parameters
% ========================
bits_Num = 6 * 2^15;                                  % Number of bits to transmit
mod_types = {'BPSK', 'QPSK', 'QPSKNG', '8PSK', '16-QAM', 'BFSK'}; % Cell array of modulation types
SNR_db_range = -4:1:16;

% Generate random bits (same for all modulations for fair comparison)
Tx_bits = randi([0 1], 1, bits_Num);

% ========================
% Initialize storage matrices
% ========================

% Initialize rx_symbols_all as 2D cell matrix
% Rows: modulation types, Columns: SNR values
rx_symbols_all = cell(length(mod_types), length(SNR_db_range));

% Initialize storage for Energy Bits
Eb_all = cell(1, length(mod_types));

% Initialize storage for Error
BER_all = zeros(length(mod_types), length(SNR_db_range));
error_count_all = zeros(length(mod_types), length(SNR_db_range));

%   Loop through all modulation types
for mod_idx = 1:length(mod_types)
    mod_type = mod_types{mod_idx};

    fprintf('\n=== %s Modulation ===\n', mod_type);

    % ========================
    % 1. Mapping (Modulation)
    % ========================
    [tx_symbols, constellation,~,Eb] = mapper(Tx_bits, mod_type);

    % Store Energy of bit for this modulation type
    Eb_all{mod_idx} = Eb;

    % ========================
    % 2. Display Constellation
    % ========================
    drawConstellation(constellation, mod_type, 1);
    title(sprintf('%s Constellation', mod_type));

    % ========================
    % 3. Channel Transmission
    % ========================
    % Get noisy symbols for all SNR values
    rx_noisy_symbols = addAWGNChannel(SNR_db_range, tx_symbols, Eb);

    % Store in 2D cell matrix
    rx_symbols_all(mod_idx, :) = rx_noisy_symbols;

    % ========================
    % 4. Demapping (Demodulation)
    % ========================
    Rx_bits = demapper(rx_noisy_symbols, mod_type);

    % ========================
    % 5. Calculate and Store Results
    % ========================
    fprintf('\nSNR Results:\n');
    fprintf('------------\n');

    for snr_idx = 1:1:length(SNR_db_range)
        [BER_all(mod_idx, snr_idx), error_count_all(mod_idx, snr_idx)] = ...
            calculateBER(Tx_bits, Rx_bits{snr_idx});

        % Display results for each SNR
        fprintf('SNR: %6.1f dB | BER: %8.2e | Errors: %4d/%d\n', ...
            SNR_db_range(snr_idx), ...
            BER_all(mod_idx, snr_idx), ...
            error_count_all(mod_idx, snr_idx), ...
            length(Tx_bits));
    end


end
```

```matlab
%   Display Noise
drawNoisyConstellations(rx_symbols_all, SNR_db_range, mod_types);

%   Graph BER Vs SNR (task 1)
plot_BER_vs_SNR(BER_all, SNR_db_range, mod_types);

%   Graph BER grey vs not grey QPSK (task 2)
plot_BER_vs_SNR_dual(BER_all(2, :), BER_all(3, :), SNR_db_range, mod_types(2:3));

%   Graph BER Vs SNR (task 1)
plot_BER_vs_SNR_all(BER_all, SNR_db_range, mod_types);

% ========================
%    BFSK
% ========================

% ===============================
% declaring parameters (for PSD)
% ===============================
bits_Num = 100;  %less number of bits from the BER
N_realization = 10000;
data = randi([0 1], N_realization, bits_Num + 1);
samples_per_bit=7;
samples_num = samples_per_bit*bits_Num;
sampled_data = repelem(data, 1, samples_per_bit);
Tb = 0.07; % each sample takes 0.01 second

t = 0:Tb/samples_per_bit:Tb;
Fs = 100;
tx_with_delay = zeros(N_realization, 700);

% mapping to BB signals
tx_out = BFSK_BB(bits_Num, N_realization, Tb, Eb, samples_per_bit, sampled_data, t);

% random delay
for i = 1:N_realization
    r = randi([0 (samples_per_bit - 1)]);
    tx_with_delay(i,:) = tx_out(i,r+1:samples_num+r);
end

% Autocorrelation
BFSK_autocorr = compute_BFSK_autocorrelation(tx_with_delay);
Rx_BFSK = BFSK_autocorr;

% plt auto correlation
draw_autocorr(Rx_BFSK);

Practical PSD
BFSK_PSD = fftshift(fft(Rx_BFSK));    % Use fftshift to center the practical PSD
f = (-350:349) / 700 * Fs;            % Frequency vector for practical PSD
f_normalized = f * Tb;                % Normalize frequency axis to match the theoretical PSD

% Theoretical PSD
PSD_theoritical = (8 * cos(pi * Tb * f).^2) ./ (pi^2 * (4 * Tb^2 * f.^2 - 1).^2);

% Handle Inf values in the theoretical PSD
idx = PSD_theoritical == Inf;
PSD_theoritical(idx) = 2;  % Change Inf to finite value for plotting

% Plot PSD
draw_psd(f_normalized, BFSK_PSD, PSD_theoritical);
```

8.1 Functions:

```
% ========================
% Functions
% ========================

function [Tx_Vector, Table, Eavg, Eb] = mapper(bits, mod_type)
    % MAPPER Digital modulation mapper with explicit symbol table and energy calculation
    % Inputs:
    %   bits     - Binary input array (row vector)
    %   mod_type - 'BPSK', 'QPSK', 'QPSKNG', '8PSK', 'BFSK', '16-QAM'
    % Outputs:
    %   Tx_Vector - Complex modulated symbols
    %   Table     - Constellation points (M-ary symbols)
    %   Eavg      - Average symbol energy (normalized)
    %   Eb        - Energy per bit

    % Ensure bits are row vector
    bits = bits(:)';

    % Define modulation parameters
    switch upper(mod_type)
        case 'BPSK'
            n = 1;  % bits per symbol
            M = 2;   % constellation size
            Table = [-1, 1];  % BPSK symbols (real)

        case 'QPSK'
            n = 2;
            M = 4;
            Table = [-1-1j, -1+1j, 1-1j, 1+1j];  % QPSK symbols

        case 'QPSKNG'
            n = 2;
            M = 4;
            Table = [-1-1j, -1+1j, 1+1j, 1-1j];  % QPSKNG symbols

        case '8PSK'
            n = 3;
            M = 8;
            angles =[0, 1, 3, 2, 7, 6, 4, 5]*pi/4;  % Gray-coded 8PSK
            Table = exp(1j*angles);

        case 'BFSK'
            n=1;
            M=2;
            Table = [ 1, 1j];

        case '16-QAM'
            n = 4;
            M = 16;
            % 16-QAM with unit average power (normalized)
            Table = [-3-3j, -3-1j, -3+3j, -3+1j, ...
                     -1-3j, -1-1j, -1+3j, -1+1j, ...
                      3-3j,  3-1j,  3+3j,  3+1j, ...
                      1-3j,  1-1j,  1+3j,  1+1j];

        otherwise
            error('Unsupported modulation type: %s', mod_type);
    end

    % Pad bits if not multiple of n
    if mod(length(bits), n) ~= 0
        bits = [bits zeros(1, n - mod(length(bits), n))];
    end

    % Calculate average symbol energy
    Eavg = mean(abs(Table).^2);

    % Calculate average bit energy
    Eb =  Eavg / n;

    % Reshape into n-bit groups
    bit_groups = reshape(bits, n, [])';

    % Convert to decimal symbols (0 to M-1)
    Array_symbol = bi2de(bit_groups, 'left-msb') + 1;  % MATLAB uses 1-based indexing

    % Map to constellation points
    Tx_Vector = Table(Array_symbol);
end
```

```matlab
function drawConstellation(Table, mod_type, showdetails)
    % DRAWCOnSTELLATION Enhanced constellation visualization
    % Inputs:
    %   Table - Constellation points (complex numbers)
    %   mod_type - Modulation type ('BPSK', 'QPSK', etc.)
    %   showdetails- true to show colored regions, false for boundaries only

    if nargin < 3
        show_regions = true; % Default to showing regions
    end

    figure;
    hold on;

    % Ensure Table is column vector and get points
    Table = Table(:);
    points = [real(Table), imag(Table)];

    % Create grid for visualization
    x_range = linspace(min(points(:,1))-1, max(points(:,1))+1, 200);
    y_range = linspace(min(points(:,2))-1, max(points(:,2))+1, 200);
    [x_grid, y_grid] = meshgrid(x_range, y_range);
    grid_points = x_grid(:) + 1j*y_grid(:);

    % ============================================
    % 1. Decision Visualization
    % ============================================
    if showdetails == 1
        if length(Table) > 2  % Voronoi needs at least 3 points
            [vx, vy] = voronoi(points(:,1), points(:,2));
            plot(vx, vy, 'k-', 'LineWidth', 1.5);
        else
            % For BPSK, draw simple decision boundary
            plot([0 0], ylim, 'k--', 'LineWidth', 1.5);
        end
    end

    % ============================================
    % 2. Constellation Points
    % ============================================
    if showdetails == 1
        scatter(points(:,1), points(:,2), 100, 'filled', 'k');
    else
        scatter(points(:,1), points(:,2), 20, 'filled', 'k');
    end
    % ============================================
    % 3. Binary Labels
    % ============================================
    switch upper(mod_type)
        case 'BPSK'
            n = 1;
        case 'QPSK'
            n = 2;
        case 'QPSKNG'
            n = 2;
        case '8PSK'
            n = 3;
        case {'16QAM', '16-QAM'}
            n = 4;
        case 'BFSK'
            n=1;
        otherwise
            error('Unsupported modulation type');
    end

    if showdetails == 1
        for i = 1:length(Table)
            bin_str = dec2bin(i-1, n);
            % Position text slightly offset from the point
            text(real(Table(i)) + 0.05, imag(Table(i)) + 0.05, bin_str, ...
                'FontSize', 10, 'Color', 'r');
        end
    end
    % ============================================
    % 4. Plot Formatting
    % ============================================
    title(sprintf('%s Constellation', mod_type));
    xlabel('In-Phase (I)'); ylabel('Quadrature (Q)');
    grid on;
    axis equal;

    % Center axes
    ax = gca;
    ax.XAxisLocation = 'origin';
    ax.YAxisLocation = 'origin';

    % Set axis limits
    max_val = max([abs(points(:))]) * 1.3;
    xlim([-max_val, max_val]);
    ylim([-max_val, max_val]);


    hold off;
end
```

```matlab
function drawNoisyConstellations(rx_symbols_all, SNR_db_range, mod_types)
    % DRAWNOISYCONSTELLATIONS Plot constellations with noisy received points
    % Inputs:
    %   rx_symbols_all - Cell array, rx_symbols_all{mod_idx, snr_idx}
    %   SNR_db_range   - Vector of SNR values (dB)
    %   mod_types      - Cell array of modulation type strings (e.g., {'BPSK', 'QPSK'})

    % Validate inputs
    if ~iscell(rx_symbols_all) || ~iscell(mod_types)
        error('rx_symbols_all and mod_types must be cell arrays.');
    end

    num_mods = numel(mod_types);
    num_snr = numel(SNR_db_range);

    for mod_idx = 1:num_mods
        mod_type = mod_types{mod_idx};

        % Generate constellation table for this modulation
        [~, Table] = mapper([1], mod_type);

        for snr_idx = 1:floor(num_snr/4):num_snr
            rx_symbols = rx_symbols_all{mod_idx, snr_idx};
            snr_db = SNR_db_range(snr_idx);

            % Center axes
            ax = gca;
            ax.XAxisLocation = 'origin';
            ax.YAxisLocation = 'origin';

            % Plot decision regions and ideal points
            drawConstellation(Table, mod_type, 0);
            title(sprintf('%s Constellation at SNR = %d dB', mod_type, snr_db));
            xlabel('In-Phase (I)'); ylabel('Quadrature (Q)');
            grid on;
            axis equal;
            hold on;
            % Plot noisy received symbols
            scatter(real(rx_symbols), imag(rx_symbols), 10, 'b', 'filled', 'MarkerFaceAlpha', 0.4);

            % Set axis limits a bit bigger to fit noisy points
            max_val = 4;
            xlim([-max_val, max_val]);
            ylim([-max_val, max_val]);

            hold off;
        end
    end
end

function [received_bits] = demapper(received_symbols, mod_type)
    % DEMAPPER Digital demodulation demapper
    % Inputs:
    %   received_symbols - Complex received symbols (array or cell array)
    %   mod_type         - Modulation type ('BPSK', 'QPSK', etc.)
    % Output:
    %   received_bits    - Demodulated bit stream (array or cell array)

    % Check if input is cell array (multiple SNR cases)
    if iscell(received_symbols)
        % Process each SNR case
        received_bits = cell(size(received_symbols));
        for i = 1:numel(received_symbols)
            received_bits{i} = demodulate_symbols(received_symbols{i}, mod_type);
        end
    else
        % Single SNR case
        received_bits = demodulate_symbols(received_symbols, mod_type);
    end
end
```

```matlab
function bits = demodulate_symbols(symbols, mod_type)
    % Helper function for actual demodulation

    % Determine bits per symbol
    switch upper(mod_type)
        case 'BPSK'
            n = 1;
        case 'QPSK'
            n = 2;
        case 'QPSKNG'
            n = 2;
        case '8PSK'
            n = 3;
        case {'16QAM', '16-QAM'}
            n = 4;
        case 'BFSK'
            n=1;
        otherwise
            error('Unsupported modulation type');
    end

    % Initialize output bits
    bits = zeros(1, length(symbols)*n);

    % ======================
    % Special case for BFSK
    % ======================
    if strcmpi(mod_type, 'BFSK')
        for i = 1:length(symbols)
            theta = angle(symbols(i));
            if (theta > pi/4 && theta < 5*pi/4)
                bits(i) = 1;
            else
                bits(i) = 0;
            end
        end
        return;
    end

    % ======================
    % General case
    % ======================

    % Get constellation table from mapper
    [~, Table] = mapper([1], mod_type);

    % Demodulate each symbol
    for i = 1:length(symbols)
        % Find nearest constellation point
        [~, idx] = min(abs(symbols(i) - Table));

        % Convert to binary (0-based index)
        bin_str = dec2bin(idx-1, n);

        % Store bits
        bits((i-1)*n+1:i*n) = bin_str - '0';
    end
end

function noisy_signals = addAWGNChannel(SNR_range_db, clean_signal, Eb)
    % ADDAGWNCHANNEL General AWGN channel noise adder
    % Inputs:
    %   SNR_range_db - Array of SNR values in dB
    %   clean_signal - Input signal (vector or matrix)
    %   Eb - Energy per bit
    % Output:
    %   noisy_signals - Cell array of noisy signals for each SNR

    % Initialize output cell array
    noisy_signals = cell(length(SNR_range_db), 1);

    % Get size of input signal
    signal_size = size(clean_signal);

    % Process each SNR point
    for i = 1:length(SNR_range_db)
        % Convert SNR from dB to linear scale
        SNR_linear = 10^(SNR_range_db(i)/10);

        % Calculate noise power (N0)
        N0 = 1 / SNR_linear;

        % Generate proper noise
        if isreal(clean_signal)
            % Real noise for real signals
            noise = sqrt(Eb*N0/2) * randn(signal_size);
        else
            % Complex noise for complex signals
            noise = sqrt(Eb*N0/2) * (randn(signal_size) + 1j*randn(signal_size));
        end



        % Add noise to the signal
        noisy_signals{i} = clean_signal + noise;
    end



    % If only one SNR point was requested, return array instead of cell
    if length(SNR_range_db) == 1
        noisy_signals = noisy_signals{1};
    end
end
```

```matlab
function [BER, bit_errors] = calculateBER(original_bits, received_bits)
    % CALCULATEBER Compute Bit Error Rate for single or multiple SNR cases
    % Inputs:
    %   original_bits - Transmitted bit sequence (1D array)
    %   received_bits - Received bits (1D array or cell array for multiple SNR)
    % Outputs:
    %   BER - Bit Error Rate (scalar or array matching received_bits input)
    %   bit_errors - Number of errors (scalar or array)

    % Ensure original bits are row vector
    original_bits = original_bits(:)';

    % Handle cell array input (multiple SNR cases)
    if iscell(received_bits)
        BER = zeros(size(received_bits));
        bit_errors = zeros(size(received_bits));

        for i = 1:numel(received_bits)
            [BER(i), bit_errors(i)] = calculateSingleBER(original_bits, received_bits{i});
        end
    else
        % Single SNR case
        [BER, bit_errors] = calculateSingleBER(original_bits, received_bits);
    end
end


function [BER, bit_errors] = calculateSingleBER(original_bits, received_bits)
    % Helper function for single SNR case BER calculation

    % Ensure received bits are row vector
    received_bits = received_bits(:)';

    % Trim received bits if longer (due to padding)
    if length(received_bits) > length(original_bits)
        received_bits = received_bits(1:length(original_bits));
    end

    % Calculate errors
    bit_errors = sum(original_bits ~= received_bits);
    BER = bit_errors / length(original_bits);
End


function displayBitComparison(Tx_bits, Rx_bits, bit_errors, BER, bits_per_group)
    % DISPLAYBITCOMPARISON Display input/output bit comparison and BER results
    %
    % Inputs:
    %   Tx_bits - Transmitted bit sequence
    %   Rx_bits - Received bit sequence
    %   bit_errors - Number of bit errors
    %   BER - Bit Error Rate
    %   bits_per_group - Number of bits to display per row (default: 16)

    if nargin < 5
        bits_per_group = 16; % Default to 16-bit groups
    end

    % Ensure inputs are row vectors
    Tx_bits = Tx_bits(:)';
    Rx_bits = Rx_bits(:)';

    % Display original bits
    fprintf('Original bits:\n');
    disp(reshape(Tx_bits, bits_per_group, [])');

% Display received bits (trimmed to original length) fprintf('\nReceived
bits:\n'); disp(reshape(Rx_bits(1:length(Tx_bits)), bits_per_group,
[])');

% Display error statistics fprintf('\nError
Analysis:\n'); fprintf('Bit errors: %d\n',
bit_errors); fprintf('BER: %.2e\n', BER);

end
```

```
function plot_BER_vs_SNR(BER_all, SNR_Range, Mod_Types)
% This function plots BER vs SNR for multiple modulation types
% Inputs:
%            BER_all    : matrix (SNR points × modulation types)
%            SNR_Range  : vector of SNR values in dB
%            Mod_Types  : cell array of modulation type names (strings)

% Transpose BER_all if it has the wrong dimensions if
size(BER_all, 1) ~= length(SNR_Range)
BER_all = BER_all.';
end

% Number of modulation types num_mods =
length(Mod_Types);

% Define colors and markers for different mod types colors =
['b', 'r', 'g', 'k', 'm', 'c', 'y'];
%markers = ['o', 's', '^', 'd', 'x', '+', '*'];

% Loop over each modulation type and create a new figure for each for idx =
1:num_mods
% Create a new figure for each modulation type figure;
hold on; grid on;

% Plot simulated BER
semilogy(SNR_Range, BER_all(:, idx), ... [colors(mod(idx-
1,length(colors))+1) ], ... 'LineWidth', 1.5);

EbNo = 10.^(SNR_Range/10); % Convert SNR from dB to linear

% Plot theoretical or tight upper bound BER switch
Mod_Types{idx}
case 'BPSK'
BER_theory = 0.5 * erfc(sqrt(EbNo)); case 'QPSK'
BER_theory = 0.5 * erfc(sqrt(EbNo)); % same as BPSK case 'QPSKNG'
BER_theory = 0.5 * erfc(sqrt(EbNo)); % same as QPSK case '8PSK'
BER_theory = erfc(sin(pi/8) * sqrt(3 * EbNo)) / 3; case '16-QAM'
BER_theory = (3/8)*erfc(sqrt((2/5)*EbNo)); case '64qam'
BER_theory = (7/24)*erfc(sqrt((7/21)*EbNo)); case 'BFSK'
BER_theory = 0.5*erfc(sqrt(0.5*EbNo)); otherwise
warning('No theoretical curve for %s. Skipping.', Mod_Types{idx}); BER_theory =
nan(size(EbNo));
end

% If theoretical BER is computed, plot it if
~any(isnan(BER_theory))
semilogy(SNR_Range, BER_theory, ...
[colors(mod(idx-1,length(colors))+1) '--'], ... 'LineWidth', 1.5);
end

% Labels and title xlabel('E_b/N_0 (dB)');
ylabel('Bit Error Rate (BER)');
title(['BER vs. E_b/N_0 for ' Mod_Types{idx}]);

% Add a legend
legend_entries = {['Simulated (' Mod_Types{idx} ')'], ['Theoretical (' Mod_Types{idx} ')']}; legend(legend_entries,
'Location', 'southwest');

% Set plot limits
%ylim([1e-5 1]);
xlim([min(SNR_Range) max(SNR_Range)]);

            hold off;
        end
end
```

```
function plot_BER_vs_SNR_dual(BER1, BER2, SNR_Range, Mod_Types)
% Plots BER vs SNR for two BER datasets + theoretical for multiple mod types
% Inputs:
         %  BER1        : matrix (SNR points × modulation types) - first BER dataset
         %  BER2        : matrix (SNR points × modulation types) - second BER dataset
         %  SNR_Range   : vector of SNR values in dB
         %  Mod_Types   : cell array of modulation type names (strings)

% Transpose if needed
if size(BER1, 1) ~= length(SNR_Range) BER1 =
BER1.';
end
if size(BER2, 1) ~= length(SNR_Range) BER2 =
BER2.';
end

num_mods = length(Mod_Types);
colors = ['b', 'r', 'g', 'k', 'm', 'c', 'y'];
%markers = ['o', 's', '^', 'd', 'x', '+', '*'];

for idx = 1:num_mods figure;
hold on; grid on;

EbNo = 10.^(SNR_Range/10); % Convert SNR from dB to linear

% Plot BER1 (e.g., baseline) semilogy(SNR_Range, BER1, ...
[colors(mod(idx-1,length(colors))+1) ], ... 'LineWidth', 1.5);

% Plot BER2 (e.g., improved method) semilogy(SNR_Range,
BER2, ...
[colors(mod(idx,length(colors))+1) ], ... 'LineWidth', 1.5);

% Compute theoretical BER switch
Mod_Types{idx}
case 'BPSK'
BER_theory = 0.5 * erfc(sqrt(EbNo)); case 'QPSK'
BER_theory = 0.5 * erfc(sqrt(EbNo)); case 'QPSKNG'
BER_theory = 0.5 * erfc(sqrt(EbNo)); case '8PSK'
BER_theory = erfc(sin(pi/8) * sqrt(3 * EbNo)) / 3; case '16-QAM'
BER_theory = (3/8)*erfc(sqrt((2/5)*EbNo)); case '64qam'
BER_theory = (7/24)*erfc(sqrt((7/21)*EbNo)); case 'BFSK'
BER_theory = 0.5*erfc(sqrt(0.5*EbNo)); otherwise
warning('No theoretical curve for %s. Skipping.', Mod_Types{idx}); BER_theory =
nan(size(EbNo));
end

% Plot theoretical BER if available if
~any(isnan(BER_theory))
semilogy(SNR_Range, BER_theory, ... [colors(mod(idx+1,length(colors))+1) '--'],
... 'LineWidth', 1.5);
end

% Labels and title xlabel('E_b/N_0 (dB)');
ylabel('Bit Error Rate (BER)');
title(['BER vs. E_b/N_0 for ' Mod_Types{idx}]);

% Legend
legend_entries = {['Simulated 1 (' Mod_Types{idx} ')'], ...
['Simulated 2 (' Mod_Types{idx} ')'], ...
['Theoretical (' Mod_Types{idx} ')']}; legend(legend_entries, 'Location',
'southwest');

xlim([min(SNR_Range) max(SNR_Range)]);
%ylim([1e-5 1]);

             hold off;
         end
end
```

```matlab
function plot_BER_vs_SNR_all(BER_all, SNR_Range, Mod_Types)
% This function plots:
% 1. All simulated BER curves in one figure
% 2. All simulated + theoretical BER curves in another figure
%
% Inputs:
%            BER_all    : matrix (SNR points × modulation types)
%            SNR_Range  : vector of SNR values in dB
%            Mod_Types  : cell array of modulation type names (strings)

% Transpose if needed
if size(BER_all, 1) ~= length(SNR_Range) BER_all =
BER_all.';
end

colors = ['b', 'r', 'g', 'k', 'm', 'c', 'y'];
%markers = ['o', 's', '^', 'd', 'x', '+', '*']; EbNo =
10.^(SNR_Range / 10); % Convert to linear

% 1. PLOT ONLY SIMULATED BER
figure;
hold on; grid on; legend_entries = {};

for idx = 1:length(Mod_Types)
color = colors(mod(idx-1, length(colors)) + 1);
%marker = markers(mod(idx-1, length(markers)) + 1);

semilogy(SNR_Range, BER_all(:, idx), ... [color], ...
'LineWidth', 1.5);

legend_entries{end+1} = ['Simulated (' Mod_Types{idx} ')'];
end

xlabel('E_b/N_0 (dB)'); ylabel('Bit
Error Rate (BER)');
title('Simulated BER vs. E_b/N_0 for All Modulation Schemes');
legend(legend_entries, 'Location', 'southwest'); xlim([min(SNR_Range),
max(SNR_Range)]);
hold off;

% 2. PLOT SIMULATED + THEORETICAL BER
figure;
hold on; grid on; legend_entries = {};

for idx = 1:length(Mod_Types)
color = colors(mod(idx-1, length(colors)) + 1);
%marker = markers(mod(idx-1, length(markers)) + 1);

% Simulated
semilogy(SNR_Range, BER_all(:, idx), ... [color], ...
'LineWidth', 1.5);
legend_entries{end+1} = ['Simulated (' Mod_Types{idx} ')'];

% Theoretical
switch Mod_Types{idx}
case {'BPSK', 'QPSK', 'QPSKNG'}
BER_theory = 0.5 * erfc(sqrt(EbNo)); case '8PSK'
BER_theory = erfc(sin(pi/8) * sqrt(3 * EbNo)) / 3; case '16-QAM'
BER_theory = (3/8) * erfc(sqrt((2/5)*EbNo)); case '64qam'
BER_theory = (7/24) * erfc(sqrt((7/21)*EbNo)); case 'BFSK'
BER_theory = 0.5 * erfc(sqrt(0.5*EbNo)); otherwise
BER_theory = nan(size(EbNo));
end

if ~any(isnan(BER_theory)) semilogy(SNR_Range,
BER_theory, ...
[color '--'], ... 'LineWidth', 1.5);
legend_entries{end+1} = ['Theoretical (' Mod_Types{idx} ')'];
end
end

        xlabel('E_b/N_0 (dB)');
        ylabel('Bit Error Rate (BER)');
        title('Simulated + Theoretical BER vs. E_b/N_0 for All Modulation Schemes');
        legend(legend_entries, 'Location', 'southwest');
        xlim([min(SNR_Range), max(SNR_Range)]);
        hold off;
end
```

```matlab
function [tx_out] = BFSK_BB(bits_Num, N_realization, Tb, Eb, samples_per_bit, sampled_data, t)
% BFSK_BB Generate baseband BFSK time-domain signal
%
% Inputs:
%        bits_Num       - Number of bits per realization
%        N_realization  - Number of realizations
%        Tb             - Bit duration in seconds
%        Eb             - Energy per bit
%
% Output:
%        tx_out         - Baseband BFSK output signal (N_realization x 7*(bits_Num+1))

% === Derived Parameters ===
total_samples = samples_per_bit * (bits_Num + 1); % Total samples per realization

% === Initialize Output Signal ===
tx_out = zeros(N_realization, total_samples);

% === Map to Baseband BFSK Signal === for i =
1:N_realization
for j = 1:samples_per_bit:total_samples if
sampled_data(i, j) == 0
tx_out(i, j:j+samples_per_bit-1) = sqrt(2 * Eb / Tb);  % Non-coherent tone for 0 else
for k = 1:samples_per_bit
tx_out(i, j + k - 1) = sqrt(2 * Eb / Tb) * ...
(cos(2 * pi * t(k) / Tb) + 1i * sin(2 * pi * t(k) / Tb));
end
end
end
end
end

function [tx_with_delay] = apply_random_delay(tx_out, samples_per_bit)
% APPLY_RANDOM_DELAY Applies random symbol-aligned delay to each realization
%
% Inputs:
%        tx_out           - Original signal matrix (N_realization × total_samples)
%        samples_per_bit  - Number of samples per bit (e.g., 7)
%
% Output:
%        tx_with_delay    - Delayed signals, trimmed to same size (N_realization × trimmed_samples)

[N_realization, total_samples] = size(tx_out); trimmed_samples =
total_samples - samples_per_bit; tx_with_delay =
zeros(N_realization, trimmed_samples);

for i = 1:N_realization
r = randi([0 (samples_per_bit - 1)]);  % Random delay in samples
tx_with_delay(i, :) = tx_out(i, r + 1 : r + trimmed_samples);
end
end

function BFSK_autocorr = compute_BFSK_autocorrelation(tx_with_delay)
% COMPUTE_BFSK_AUTOCORRELATION Computes autocorrelation of delayed BFSK signals
% centered at the middle sample.
%
% Input:
%        tx_with_delay   - Matrix of delayed BFSK signals (N_realization × N_samples)
%
% Output:
%        BFSK_autocorr   - Autocorrelation vector (1 × N_samples)

[~, N_samples] = size(tx_with_delay);

% Ensure N_samples is even for symmetric range if
mod(N_samples, 2) ~= 0
error('N_samples must be even for symmetric autocorrelation.');
end

BFSK_autocorr = zeros(1, N_samples); center_idx =
N_samples / 2;

for j = -center_idx+1 : center_idx i = j +
center_idx;
if i >= 1 && i <= N_samples
p = conj(tx_with_delay(:, center_idx)) .* tx_with_delay(:, i); BFSK_autocorr(i) = sum(p) /
length(p);
end
end
end
```

```
function draw_autocorr(Rx_BFSK)

% DRAW_AUTOCORR Plots the magnitude of the symmetric autocorrelation

%

% Input:

%       Rx_BFSK - 1 × N vector of autocorrelation values (only one-sided)



N = length(Rx_BFSK); tau = (-
N+1):(N-1);



% plot the graph

figure('Name', 'Autocorrelation');

plot(tau-N/2, abs(fliplr([Rx_BFSK Rx_BFSK(2:end)])), 'LineWidth', 1.5); xlabel('\tau');
%       f_normalized           - Frequency axis (normalized by bit rate)
        BFSK_PSD               - Practical PSD values (1 × N)
%       PSD_theoretical    - Theoretical PSD values (1 × N), aligned with f_normalized

        figure('Name', 'PSD');

        plot(f_normalized, abs(BFSK_PSD) / 100, 'b', 'LineWidth', 1);                    % Practical PSD hold
        on;

        plot(f_normalized + 0.5, abs(PSD_theoretical), 'r--', 'LineWidth', 1); % Shifted theoretical PSD hold off;

        xlabel('Normalized Frequency, fTb'); ylabel('S(f)');

        title('BFSK PSD'); xlim([-
        1.5 1.5]);

        ylim([0 2]);
end
```