Department of Electronics and
Electrical Communications Engineering
Faculty of Engineering - Cairo University

ELC 3070 – Spring 2024

Communications 2

Project #2

Matched Filter

Submitted to

Dr. Mohamed Khairy

Dr. Mohamed Nafea

Eng. Mohamed Khaled

Submitted by

Team: 19

Name	Code	Role
Passant Nabil Fawzy	136	Simulating Filters & Eye Diagrams
Gamal Alaa Eldin	144	BER Calculations & Report Documentation
Roaa Atef Mohamed	211	Generating Data & Simulating Filters
Shehab Eldin Tarek	224	Noise Analysis & BER Calculations
Omar Ahmed Ragab	244	ISI & Raised cosine

Table of Contents

Table of Figures

Matched filters and correlators in noise-free environment

Impulse Train Generation and Pulse Shaping

In this part, we simulate a simple binary communication system using Pulse Amplitude Modulation (PAM) under ideal, noise-free conditions. The system sends 10 random binary bits, applies pulse shaping, and then evaluates the response using both a matched filter and a correlator. We begin by defining the symbol duration, sampling rate, bitstream, and pulse shaping filter. The binary bits are converted to bipolar values, and a normalized pulse is defined to shape the transmitted signal. This setup ensures a symbol period of 1 second with each symbol represented by 5 samples (i.e., 200 ms sample spacing). The bipolar bit stream is generated by mapping logical '1' to +1 and '0' to -1. The pulse shaping filter p is a descending sequence scaled to have unit energy, which helps in minimizing inter-symbol interference (ISI) and maximizing signal-to-noise ratio (SNR) at the receiver. The impulse train is up sampled, inserting zeros between the symbols to create the desired sample spacing

Code Snippet

```
%% Parameters
Ts = 1;
samples_per_symbol = 5;     % Samples per symbol
bits = [1 0 1 1 0 0 1 0 1 1]; % random 10 bits
data = 2 * bits - 1;        % Mapping 1 → +1, 0 → -1
p = [5 4 3 2 1] / sqrt(55); % Normalized pulse shape (energy = 1)

%% Impulse Train
impulse_train = upsample(data, samples_per_symbol); % Insert zeros between symbols (4 zeroes out of 5 samples)
t_impulse = 0:Ts/samples_per_symbol:(length(impulse_train)-1)*Ts/samples_per_symbol;
```

Graphs

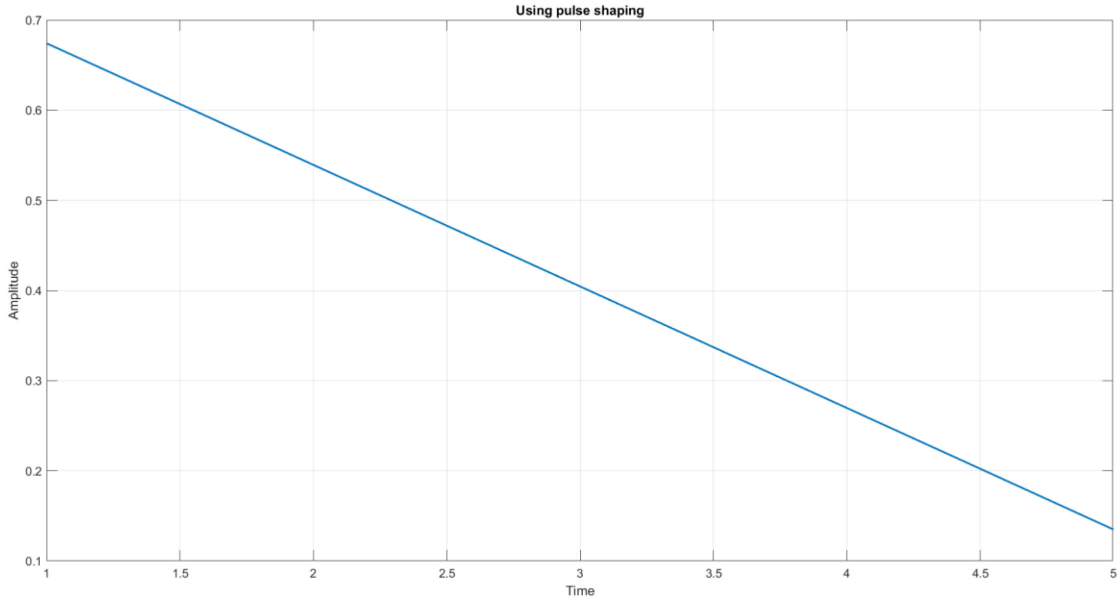

Figure 1: Pulse shaping waveform used in matched filter and correlator analysis.

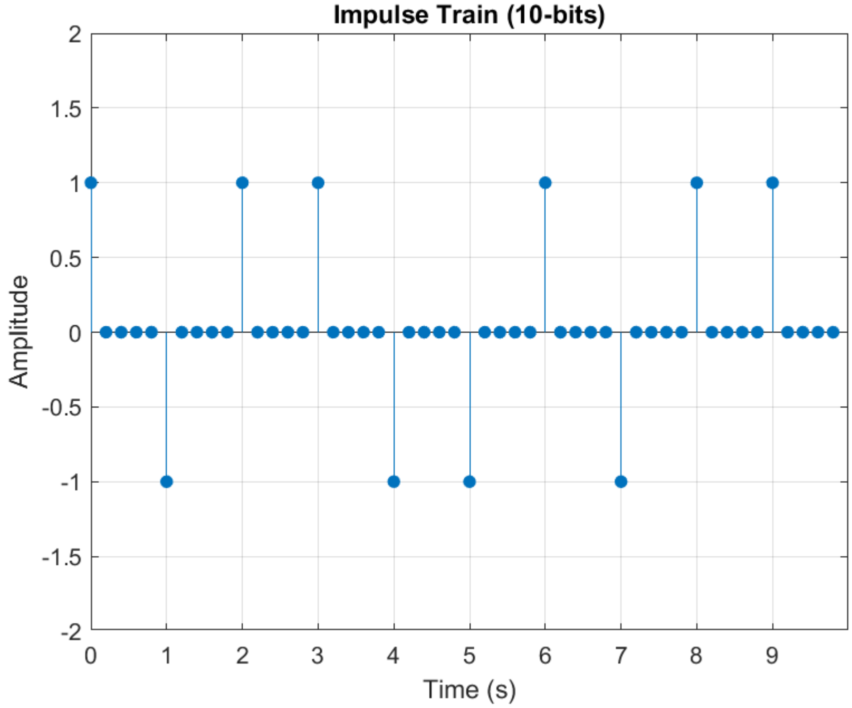

Figure 2: Impulse Train Representation of 10 Random Binary Bits with Up Sampling

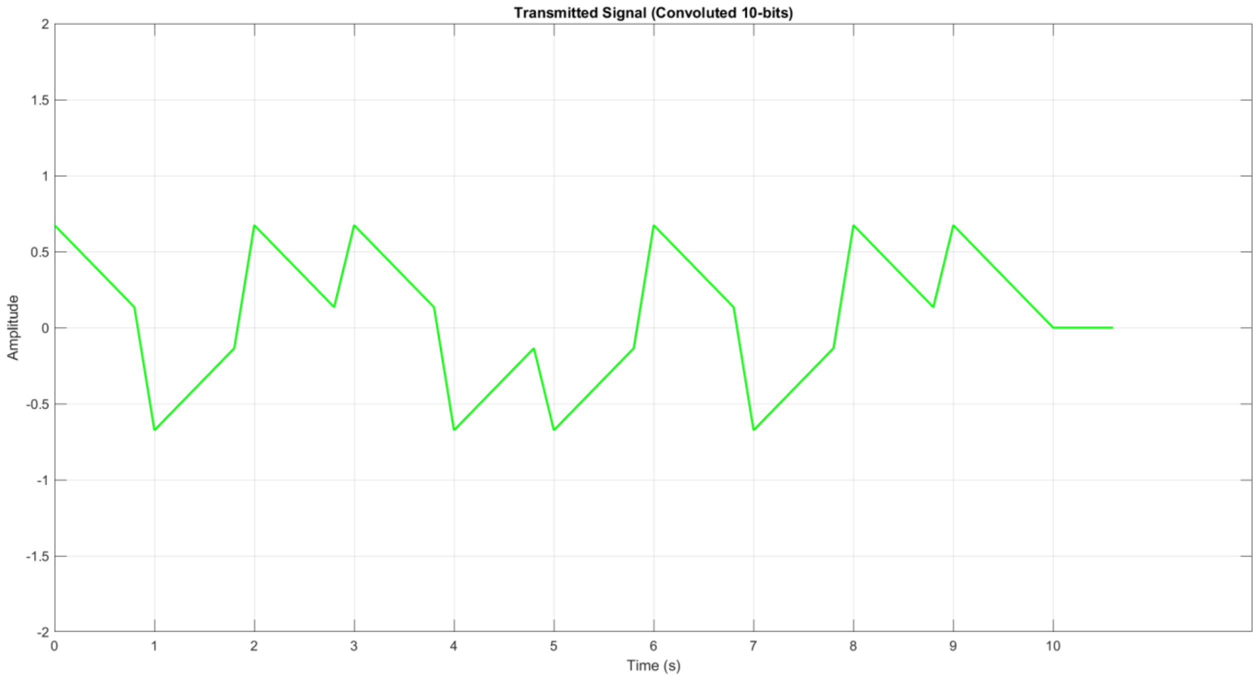

Figure 3: Output of convolution

Requirement 1, Part A: Matched and Unmatched filters Outputs

After pulse shaping, the received signal is processed using a matched filter, implemented as the time-reversed version of the pulse. This enhances detection by maximizing the SNR at sampling instants. Additionally, we apply a rectangular filter for comparison, acting as a basic correlator. Both filter outputs help evaluate the effectiveness of signal recovery at the receiver.

Code Snippet

```
%% Matched Filter Output
matched_filter = fliplr(p); % Time-reversed pulse shape
y_matched = conv(y_tx, matched_filter, 'full');
delay = length(p) - 1;  % Delay introduced by convolution
t_matched = (0:length(y_matched)-1) * (Ts/samples_per_symbol) - delay *
(Ts/samples_per_symbol); % delay advanced MF output

%% Rectangular Filter Output (for comparison)
rect_filter = ones(1, samples_per_symbol) / sqrt(samples_per_symbol); % Normalized
y_rect = conv(y_tx, rect_filter, 'full');
t_rect = 0:Ts/samples_per_symbol:(length(y_rect)-1)*Ts/samples_per_symbol;
```

The Comparison Graph

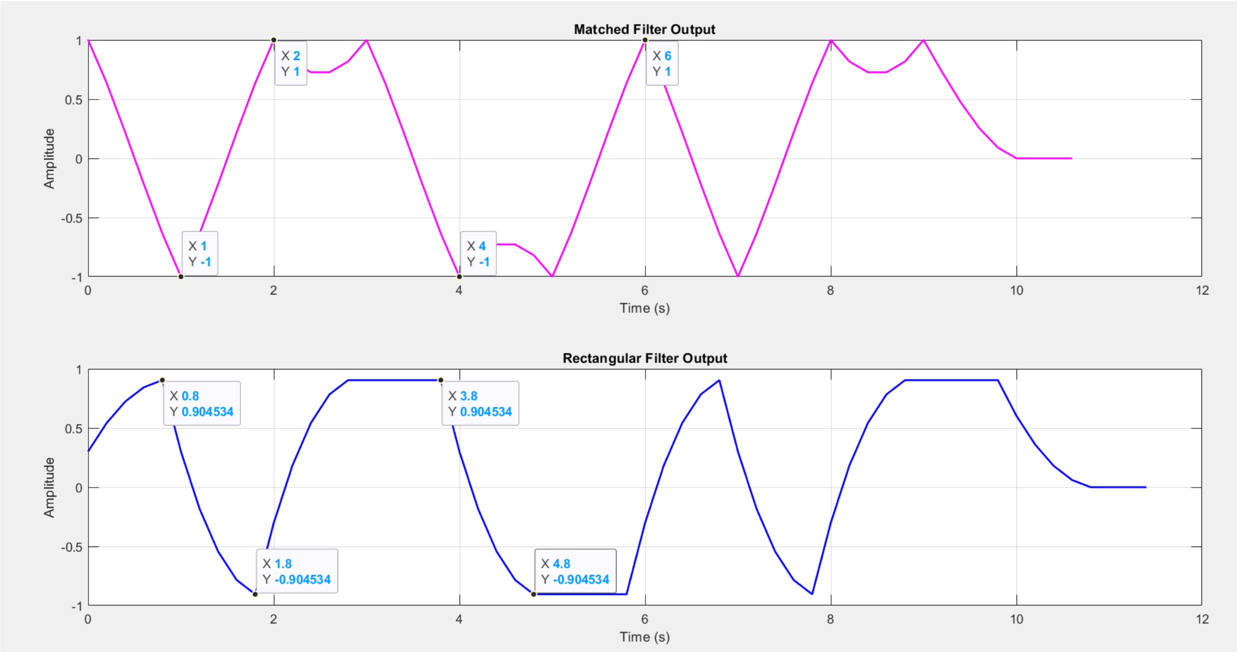

Figure 4: Matched and Unmatched Filter Output

Comment

This graph compares the outputs of the matched filter and the rectangular filter at the receiver. At the sampling instants, the matched filter output reaches an amplitude of exactly ±1, indicating perfect alignment with the transmitted symbols. In contrast, the rectangular filter produces slightly lower peak values (e.g., 0.904), reflecting suboptimal detection. The matched filter maximizes the signal amplitude at the decision points, which enhances detection accuracy and improves the SNR.

Requirement 1, Part B: Correlator

In this part, we implement the correlator output by performing an "Integrate and Dump" operation. For each bit, we extract the corresponding symbol segment from the transmitted signal y_tx and compute the correlation with the pulse shape p. This results in a correlation output (corr_output) for each symbol, which helps recover the transmitted data by matching the received signal to the expected pulse shape.

Next, we sample the matched filter output at the symbol rate by selecting every samples_per_symbol-th sample from the y_matched signal. This gives the sampled matched filter output, which corresponds to the symbol-spaced time instances, enabling us to recover the data at the symbol rate. The time vectors for both outputs (t_corr and t_samples) are defined to reflect symbol-spaced timing.

Code Snippet

```
%% Correlator Output (Integrate & Dump)
num_bits = length(bits);
corr_output = zeros(1, num_bits);
for i = 1:num_bits
    start_idx = (i-1)*samples_per_symbol + 1;
    end_idx = i*samples_per_symbol;
    segment = y_tx(start_idx:end_idx);
    corr_output(i) = sum(segment .* p); % Correlation with pulse shape
end
t_corr = (0:num_bits-1) * Ts;
%t_corr = (0:length(corr_output)-1) * Ts;% Symbol-spaced time vector

%% Sampled Matched Filter (Symbol Rate Ts)
matched_sampled = y_matched(samples_per_symbol:samples_per_symbol:end);
t_samples = (0:length(matched_sampled)-1) * Ts;  % Symbol-spaced times
```

The Comparison Graph

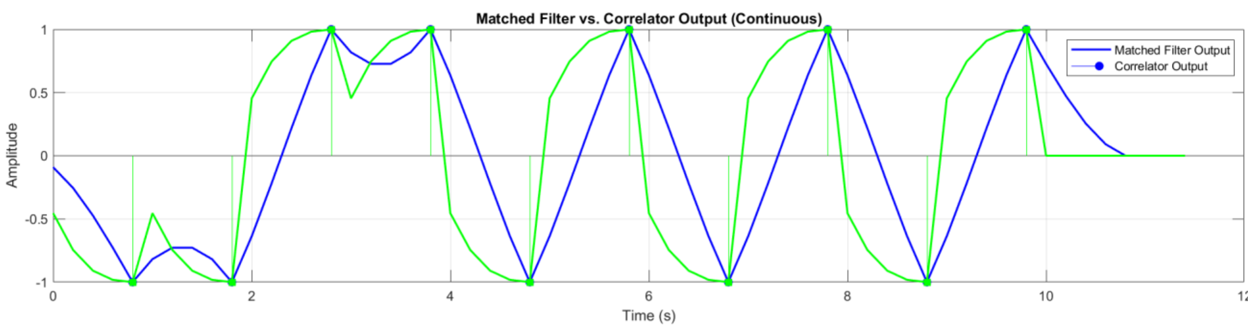

Figure 5: Output of the matched filter and correlator

Comment

The output of the correlator is the result of convolving the received signal with the pulse shaping function, which is equivalent to the output of the matched filter at the sampling instances. By sampling at the symbol rate Ts, this operation maximizes the signal-to-noise ratio (SNR) at the sampling points, ensuring optimal detection. The correlator output can be expressed as:

$$Correlator\ out = \int_0^{Ts} x(t) * p(t)\, dt$$

Noise Analysis

Impulse Train, Pulse Shaping, and Noise Addition for 10K Bits

The process involves expanding the initial bit sequence of 10,000 bits, generating Gaussian noise with zero mean and unity variance matching the size of the sequence. This noise is then scaled to achieve a variance of N0/2.

Code Snippet

```
bits_for_noise = randi([0, 1], 1, 10000); % random 10000 bits
data_for_noise = 2 * bits_for_noise - 1;

%% Impulse Train for noise
impulse_train_for_noise = upsample(data_for_noise, samples_per_symbol);
t_impulse_for_noise = (0:length(impulse_train_for_noise)-1) *
(Ts/samples_per_symbol);

% Generate and scale noise
N0 = 1/(10 ^ (-2/10));  % since Eb=1 & N0=1/(10^(SNR/10)),starting from Eb/N0 = -2
dB
Noise_scaled = sqrt(N0/2) * randn(size(y_tx_for_noise));

% Add noise to signal
V = y_tx_for_noise + Noise_scaled; % The Transmitted pulse shaped signal added to
noise
t_v = 0:Ts/samples_per_symbol:(length(V)-1)*Ts/samples_per_symbol;
```

Comment

In this code, we first generate a random sequence of 10,000 binary bits using randi([0, 1], 1, 10000). These bits are then mapped to bipolar values (1 → +1, 0 → -1) using data_for_noise = 2 * bits_for_noise - 1. Next, we create an impulse train by upsampling the bipolar bitstream to match the symbol rate, inserting zeros between the bits using upsample(data_for_noise, samples_per_symbol). The time vector t_impulse_for_noise is created to represent the time instances corresponding to the impulse train. We then generate additive white Gaussian noise (AWGN) with zero mean and unit variance using randn, which generates random values from normal distribution. The noise sequence is scaled to match the desired noise power spectral density (N0) using Noise_scaled = sqrt(N0/2) * randn(size(y_tx_for_noise)), ensuring the correct variance based on the target SNR. Finally, the noise is added to the pulse-shaped transmitted signal y_tx_for_noise to simulate the noisy received signal, resulting in V = y_tx_for_noise + Noise_scaled, and the time vector t_v is created to represent the time instances corresponding to the noisy signal.

Graphs

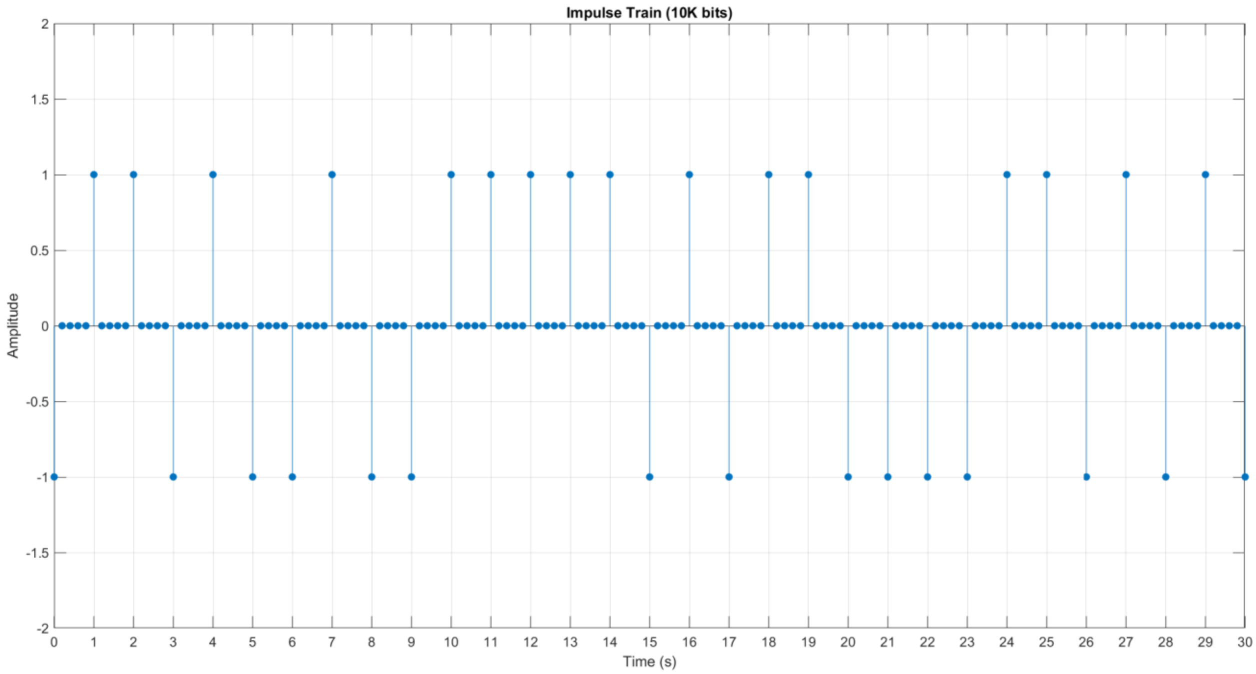

Figure 6: Impulse Train of 10,000 Bits

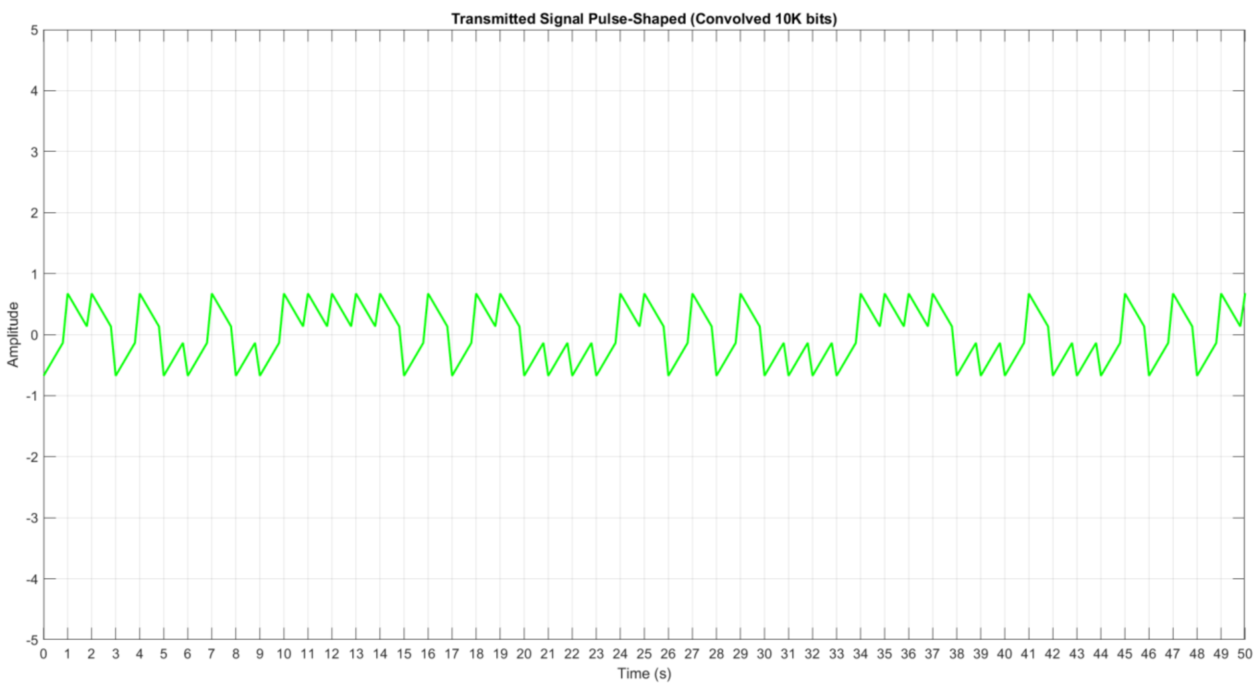

Figure 7: Pulse-Shaped Transmitted Signal

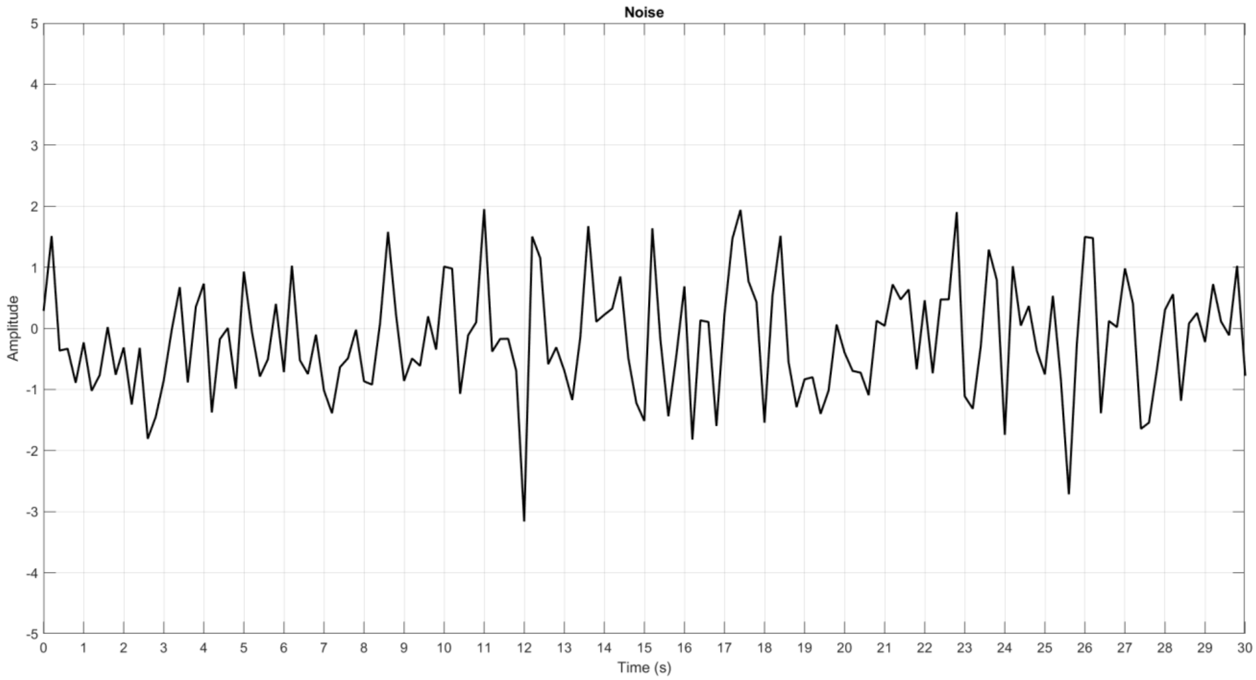

Figure 8: Scaled Additive White Gaussian Noise (AWGN)

Matched Filter Noisy Output

The noisy signal is then passed through a matched filter using convolution to obtain y_matched_noisy, which enhances the detection of symbols by maximizing the signal-to-noise ratio (SNR) at the sampling instants. A time vector t_v corresponding to the duration of the noisy received signal V is created. The time steps are spaced according to the sampling interval (Ts divided by the number of samples per symbol), ensuring correct alignment for plotting or further analysis. Subsequently, the filtered signal is sampled at the symbol rate (Ts), which is achieved by selecting every samples_per_symbol-th value from the filtered output, resulting in the matched_noisy_sampled signal. The corresponding time vector t_matched_noisy_sampled is created to reflect the symbol rate spacing, providing the exact time instances at which the symbols are detected for further decision or error analysis.

Code Snippet

```
%% Matched Filter Noisy Output
y_matched_noisy= conv(V, matched_filter, 'full');
t_matched_noisy = 0:Ts/samples_per_symbol:(length(y_matched_noisy)-
1)*Ts/samples_per_symbol;


%% Sampled Matched Filter (Symbol Rate Ts)
matched_noisy_sampled = y_matched_noisy(samples_per_symbol:samples_per_symbol:end);
t_matched_noisy_sampled = 0:Ts:(length(matched_noisy_sampled)-1)*Ts;
```

Graphs

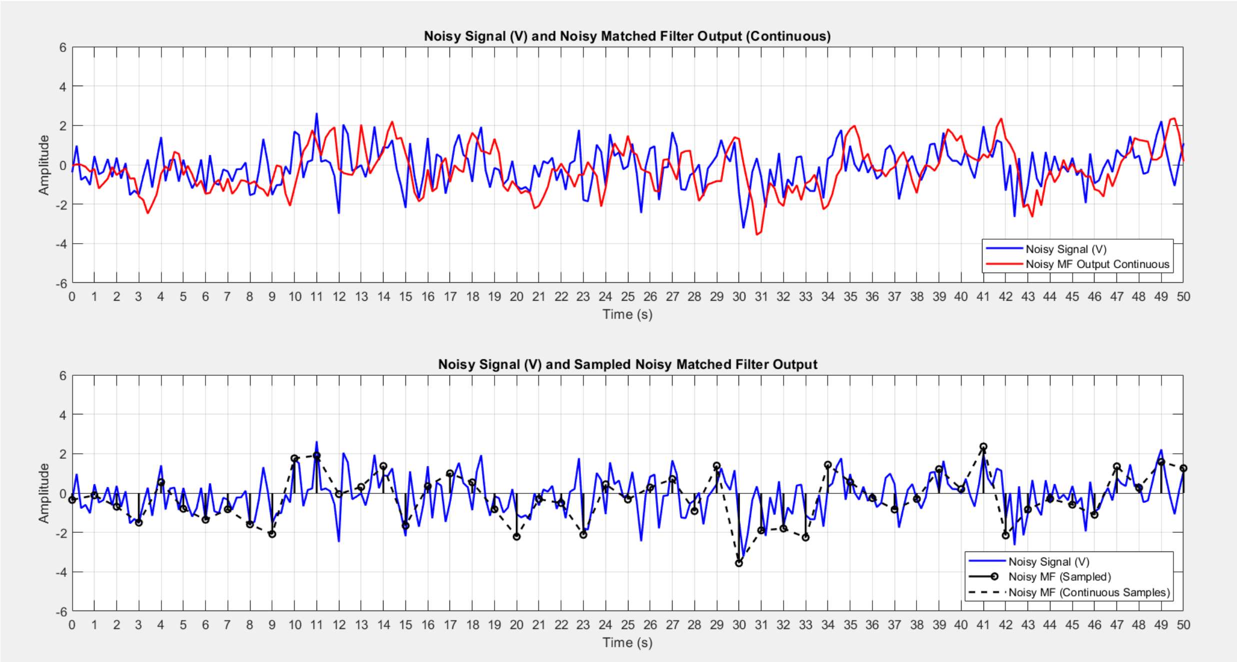

Figure 9: Noisy Signal and Noise/Sampled Matched Filter Output

Requirement 2: The BER

Code Snippet:

```
%% BER Calculation
N0_Values_linear = [1/(10 ^ (-2/10)),1/(10 ^ (-1/10)),1/(10 ^ (-0/10)),1/(10 ^
(1/10)),1/(10 ^ (2/10)),1/(10 ^ (3/10)),1/(10 ^ (4/10)),1/(10 ^ (5/10))];
EbN0_Values_linear = 1 ./ N0_Values_linear;
EbN0_Values_dB = [-2,-1,0,1,2,3,4,5];
BER_theoretical = 0.5 * erfc(sqrt(EbN0_Values_linear));


MF_BER_array = calculate_MF_BER(N0_Values_linear, y_tx_for_noise, bits_for_noise,
matched_filter, samples_per_symbol);
```

Graphs

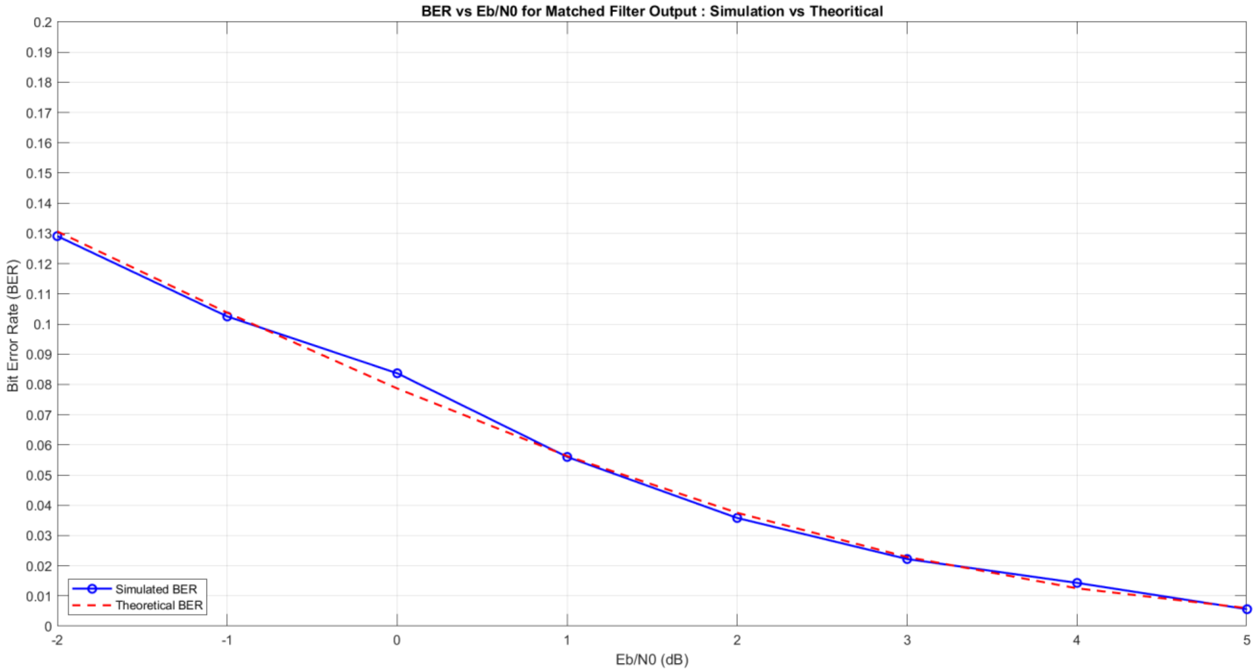

Figure 10: BER vs Eb/N0 for Matched Filter Output : Simulation vs Theoritical

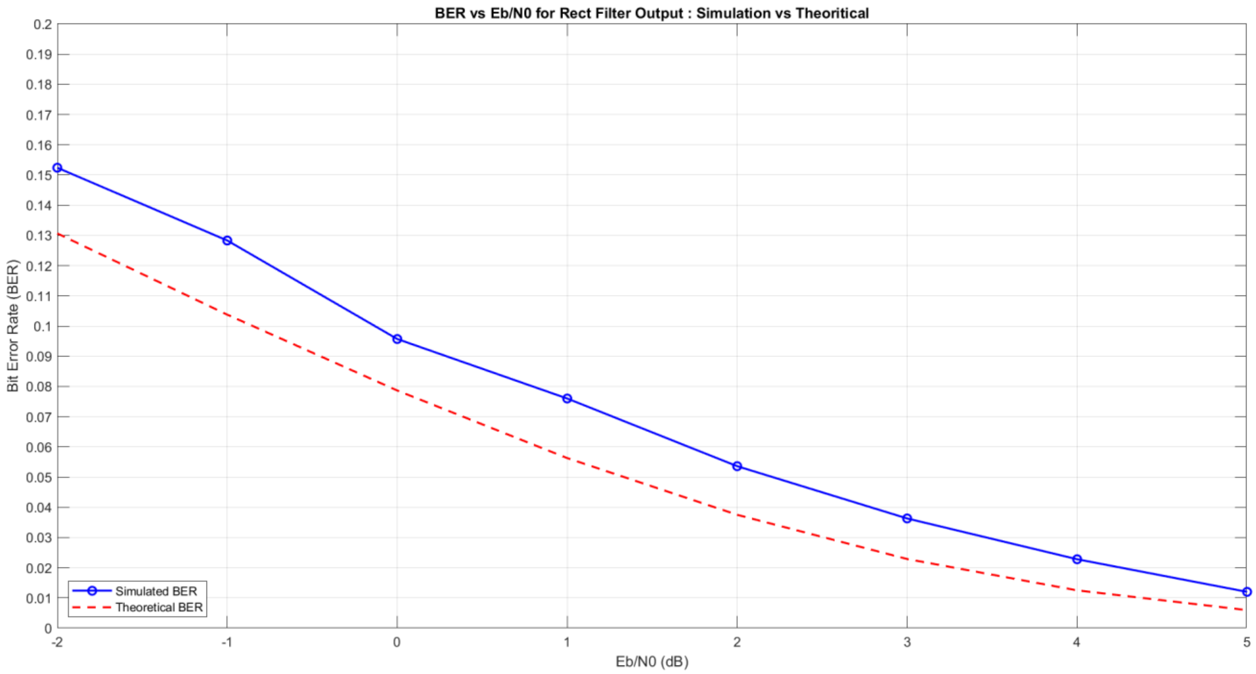

Figure 11: BER vs Eb/N0 for Rect Filter Output: Simulation vs Theoretical

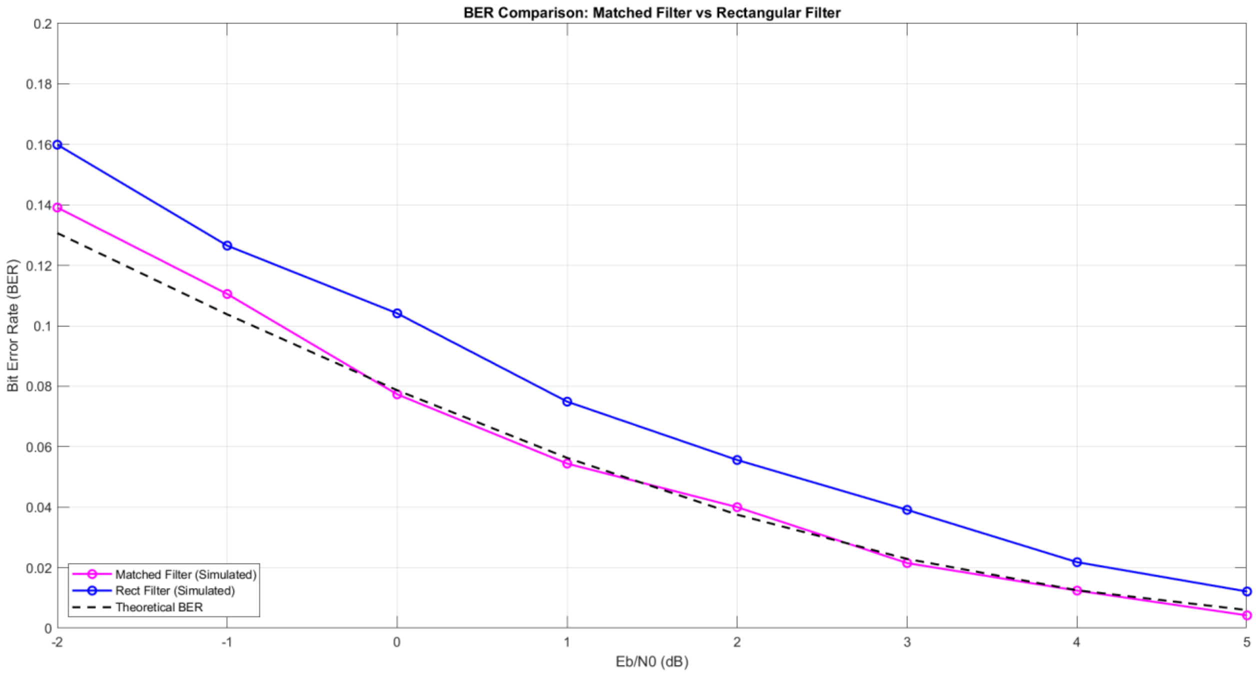

Figure 12: BER Comparison: Matched Filter vs Rectangular Filter for 10000 bits

Comment

Matched filters are designed to maximize the peak Signal-to-Noise Ratio (SNR), making them more efficient. So even with equivalent Energy to Noise Spectral Ratio (Eb/No), the Bit Error Rate (BER) tends to be lower with a matched filter than with an unmatched filter. Also, the matched filter BER is nearly equal to the theoretical BER: BER = $\frac{1}{2}$ erfc ($\sqrt{\frac{Eb}{No}}$)

So, to produce more accurate results, we used a larger number of bits (2,000,000)

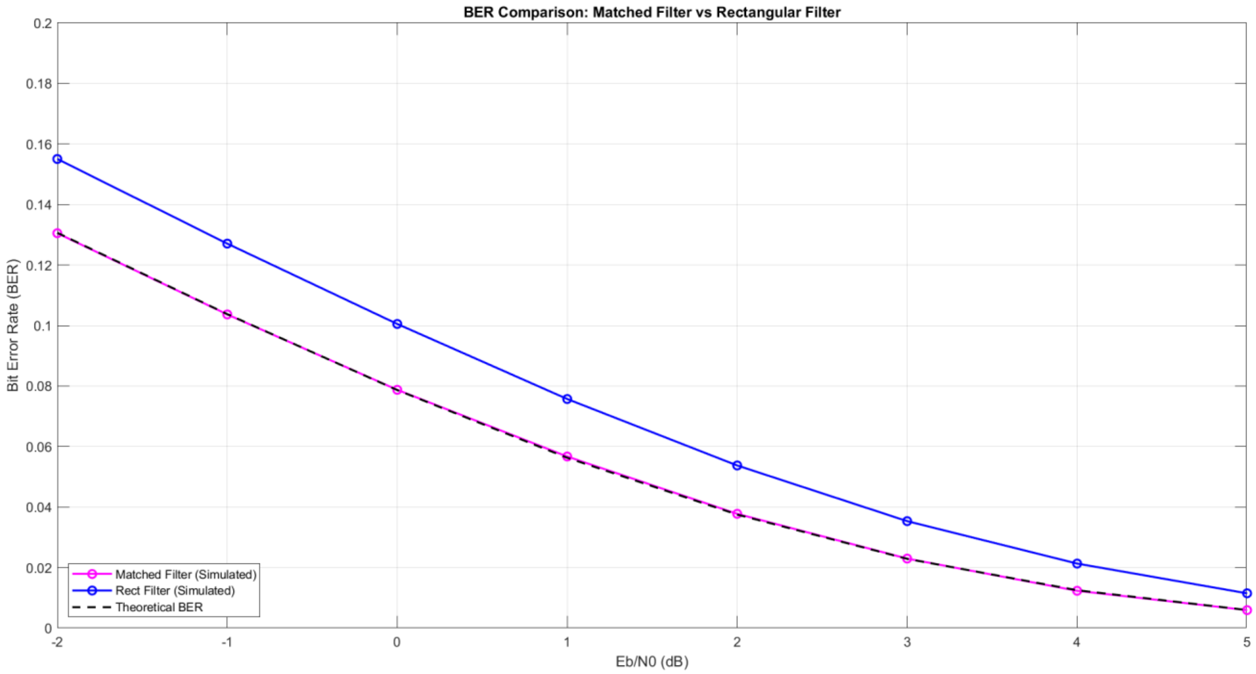

Figure 13: BER Comparison: Matched Filter vs Rectangular Filter for 2000000 bits

As the number of bits approaches infinity, the BER of the matched filter approaches the theoretical result.

ISI and raised cosine

Signal Processing – Generation and Filtering

Code Snippet

```
%% ISI and Raised cosine
Data = randi([0, 1], 1, 100);%generation of 100 random bits data
Data_forISI = 2 * Data - 1;%mapping bits to 1 & -1
impulse_train_Data_forISI = upsample(Data_forISI, samples_per_symbol);


delay_values = [2, 8];%delay
R_values = [0, 1];%roll-off factor



for R = R_values
   for delay = delay_values
       rcos_filter = rcosine(1/Ts, samples_per_symbol, 'sqrt', R,
delay);%generation of raisedcosine filter
       figure;%ploting the filter
       plot(rcos_filter);
       title("rcosine filter R: " + R + "Delay: " + delay);
       xlabel("time_samples");
       ylabel("Amplitude");

       %passing signals through tx_filter then channel has no effect-------
      % A = filter(rcos_filter, 1, impulse_train_Data_forISI);
       A = conv(impulse_train_Data_forISI , rcos_filter , 'same');
       %then passing through rx filter-------------------------------
      % B = filter(rcos_filter, 1, A);
       B = conv(A , rcos_filter , 'same');

       %ploting  A,B signals--------------------------------------------
       fig_tit = ['R = ', num2str(R), ', Delay = ', num2str(delay)];
       figure;
       t_A = (0:length(A)-1) * (Ts/samples_per_symbol);
       plot(t_A, A);
       title(['Signal after transmit: ', fig_tit]);
       xlabel('Time (s)');
       ylabel('Amplitude');
       grid on;

       figure;
       t_B = (0:length(B)-1) * (Ts/samples_per_symbol);
       plot(t_B, B);
       title(['Signal after Receive : ', fig_tit]);
       xlabel('Time (s)');
       ylabel('Amplitude');
       grid on;
```

Comment:

If an ADC is sampling with a certain rate (F_S), then each sample is quantized according to accuracy of quantizer. Such that each sample of ADC is mapped to n bits $=\log_2(M)$, where M is quantizer levels. Bitrate (R_b) = n*F_s. Then the symbol rate = $\frac{Rb}{\#\ of\ bits\ per\ symbol}$.

To ensure efficient transmission while minimizing Inter-Symbol Interference (ISI), we use the square root raised cosine (SRRC) filter. Raised cosine is a more practical approach to Nyquist's ideal channel bandwidth BW_{Ideal}= $\frac{Rs}{2}\ x\ BW\ extended$. We extend bandwidth by certain factor (roll-off factor) BW = BW_{Ideal}(1+α) = $\frac{Rb}{2}\ (1+\alpha)$ since in binary PAM the symbol rate is same as bitrate.

In this simulation, we implement a noise-free communication system using SRRC filters at both the transmitter and receiver. These filters are chosen because their combined response, when cascaded, approximates an ideal raised cosine filter, which satisfies the Nyquist criterion and minimizes ISI. However, an ideal SRRC filter has an infinite impulse response and cannot be used in practice. Therefore, we use a finite-length approximation controlled by the delay parameter. The roll-off factor (R) controls the excess bandwidth, with higher values of R providing a larger bandwidth but potentially impacting the signal's spectral efficiency.

To observe the effects of ISI, we simulate four different configurations by varying the roll-off factor RRR (values of 0 and 1) and delay (values of 2 and 8). We generate a 100-bit random BPSK signal, upsample it, and pass it through the transmit SRRC filter. The filtered signal is then passed through the receive SRRC filter. The resulting filtered signals are visualized to observe the system's time-domain response, eye patterns, and how different filter settings influence the system's potential for ISI. By varying the filter parameters, we can see how the roll-off factor and delay affect the eye diagrams and the overall system performance.

Graphs

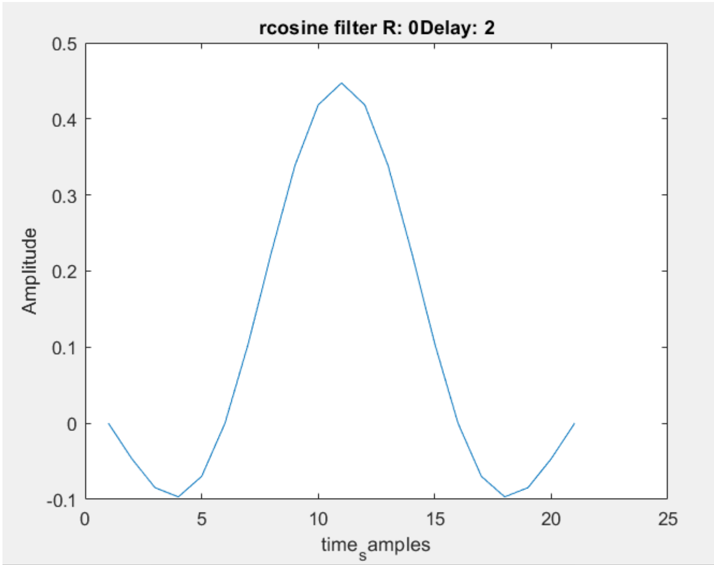

Figure 14: Square Root Raised Cosine with R=0, Delay=2

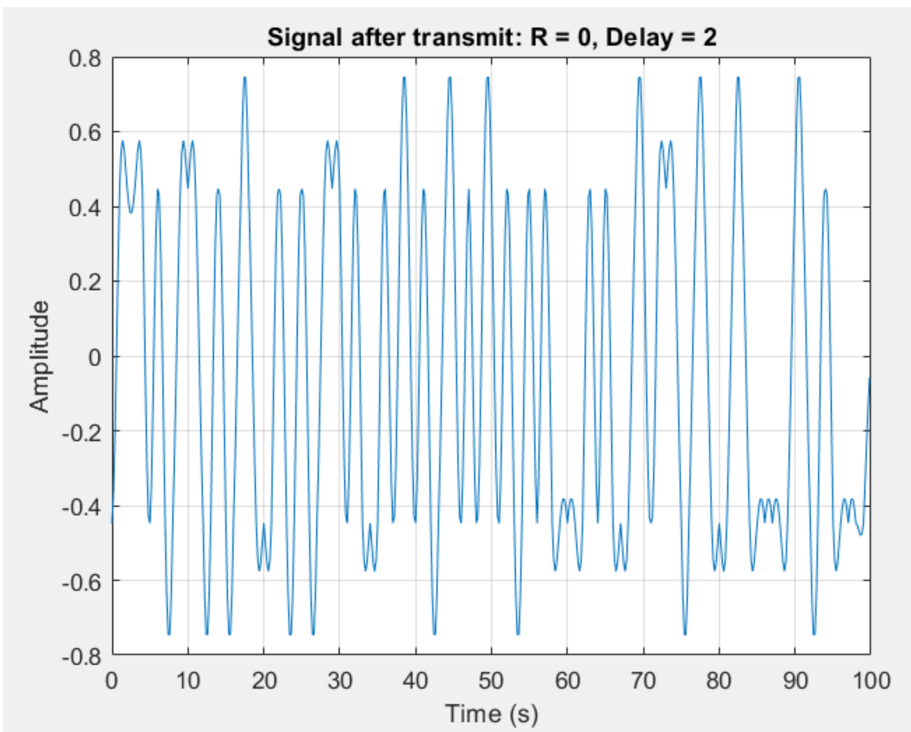

Figure 15: Transmitted Signal after Square Root Raised Cosine Filtering with R = 0, Delay=2

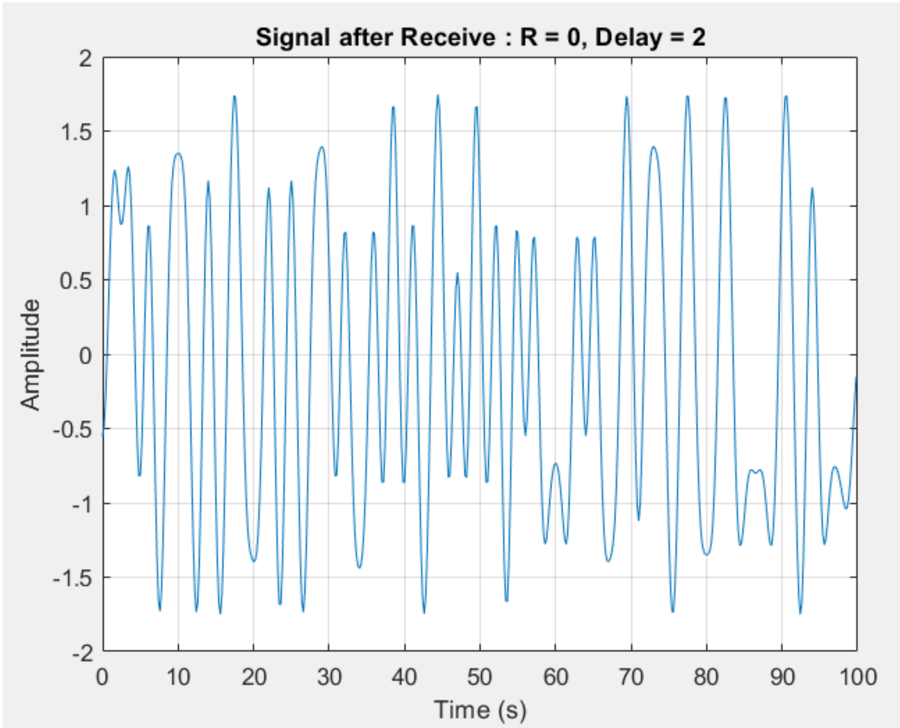

Figure 16: Received Signal after Square Root Raised Cosine Filtering with R = 0, Delay=2

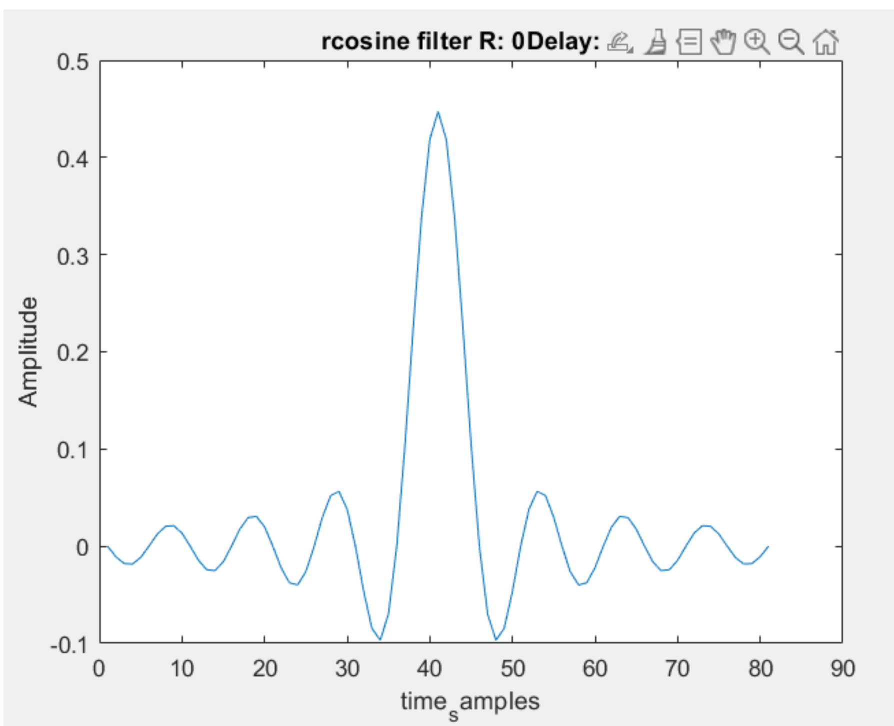

Figure 17: Square Root Raised Cosine with R=0, Delay=8

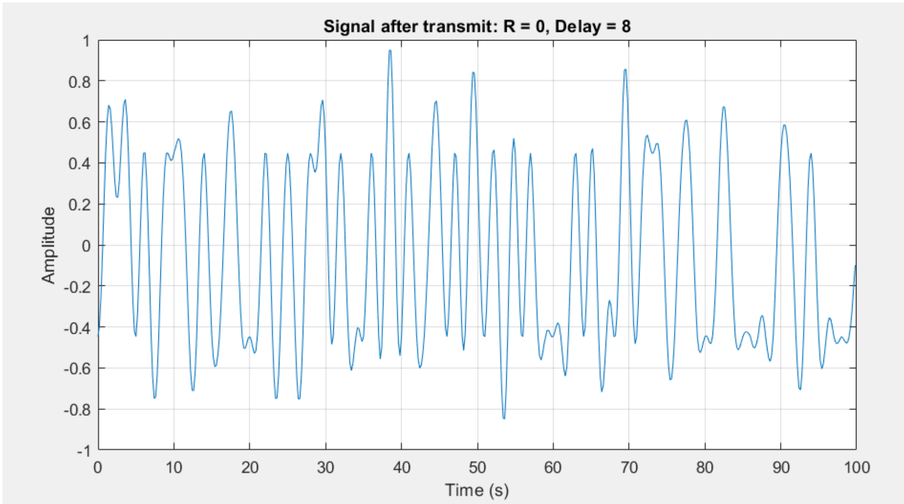

Figure 18: Transmitted Signal after Square Root Raised Cosine Filtering with R = 0, Delay=8

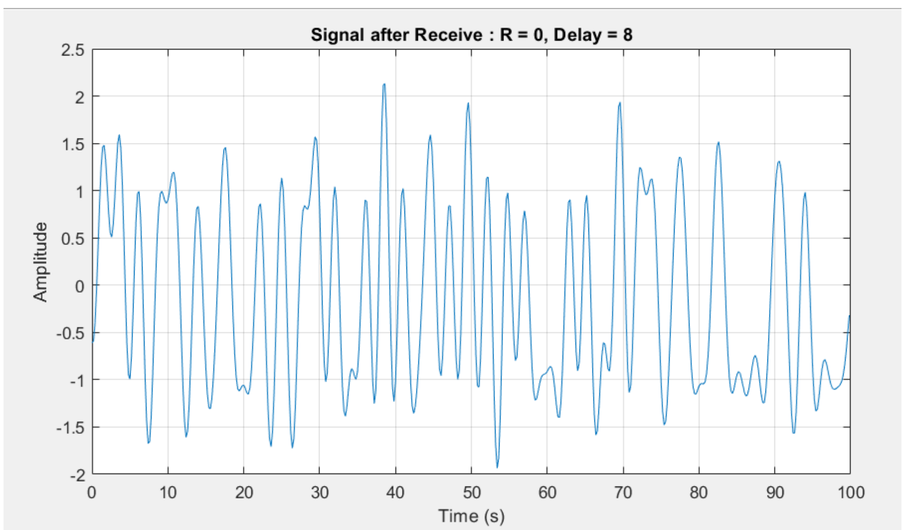

Figure 19: Received Signal after Square Root Raised Cosine Filtering with R=0, Delay=8

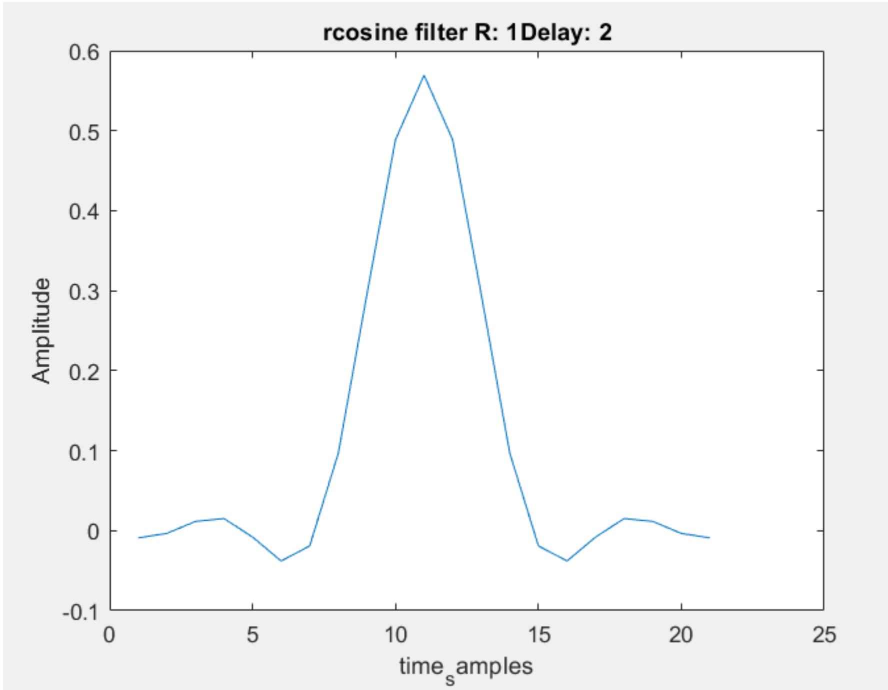

Figure 20: Square Root Raised Cosine with R=1, Delay=2

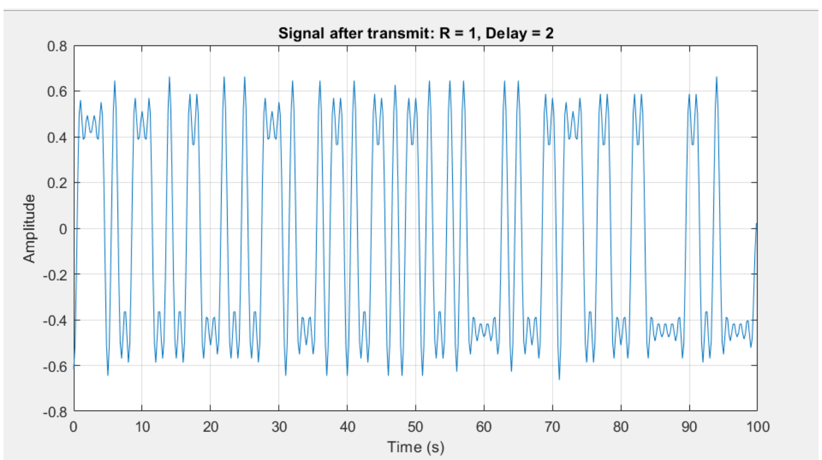

Figure 21: Transmitted Signal after Square Root Raised Cosine Filtering with R=1, Delay=2

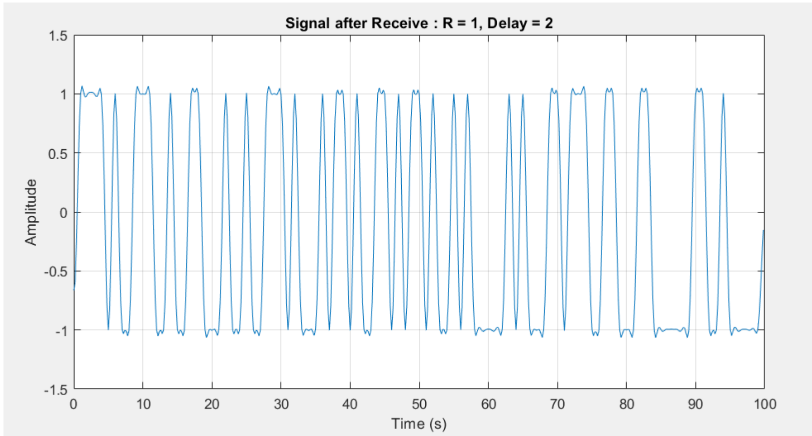

Figure 22: Received Signal after Square Root Raised Cosine Filtering with R=1, Delay=2

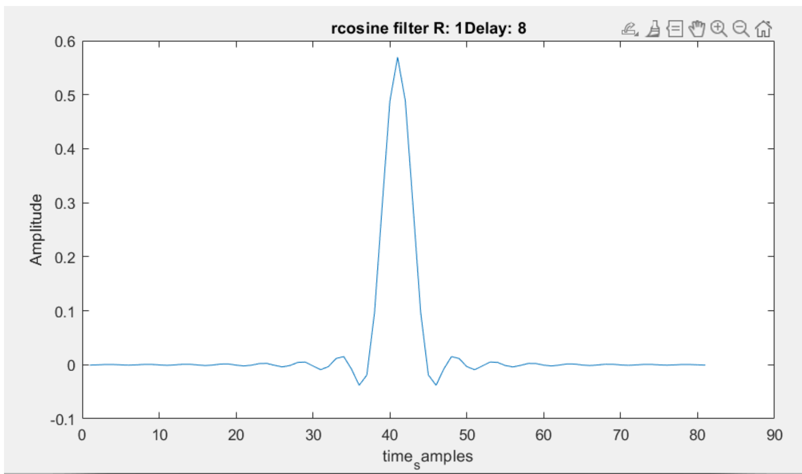

Figure 23: Square Root Raised Cosine with R=1, Delay=8

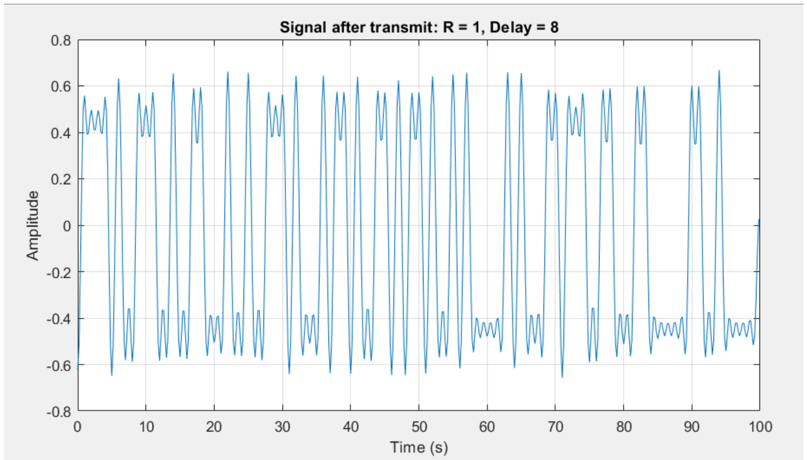

Figure 24: Transmitted Signal after Square Root Raised Cosine Filtering with R=1, Delay=8

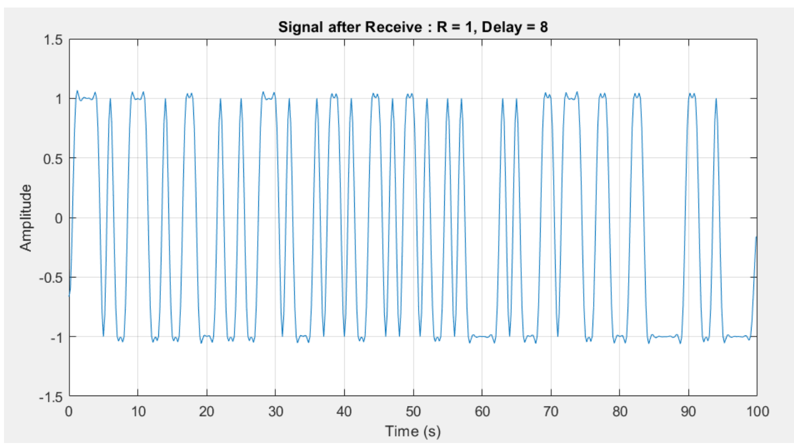

Figure 25: Received Signal after Square Root Raised Cosine Filtering with R=1, Delay=8

Requirement 3: Eye Diagram

At Point A: Transmitted signal

Graphs

A: R = 0, delay = 2.

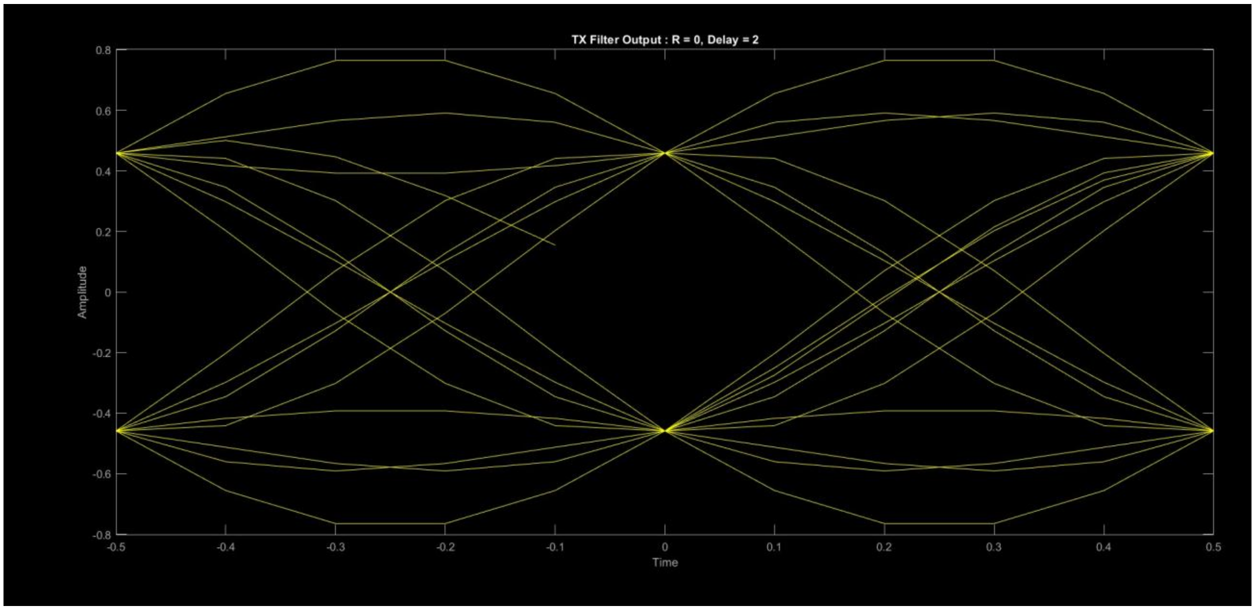

Figure 26: Transmitted signal Eye diagram at R=0 & Delay=2

B: R = 0, delay =8.

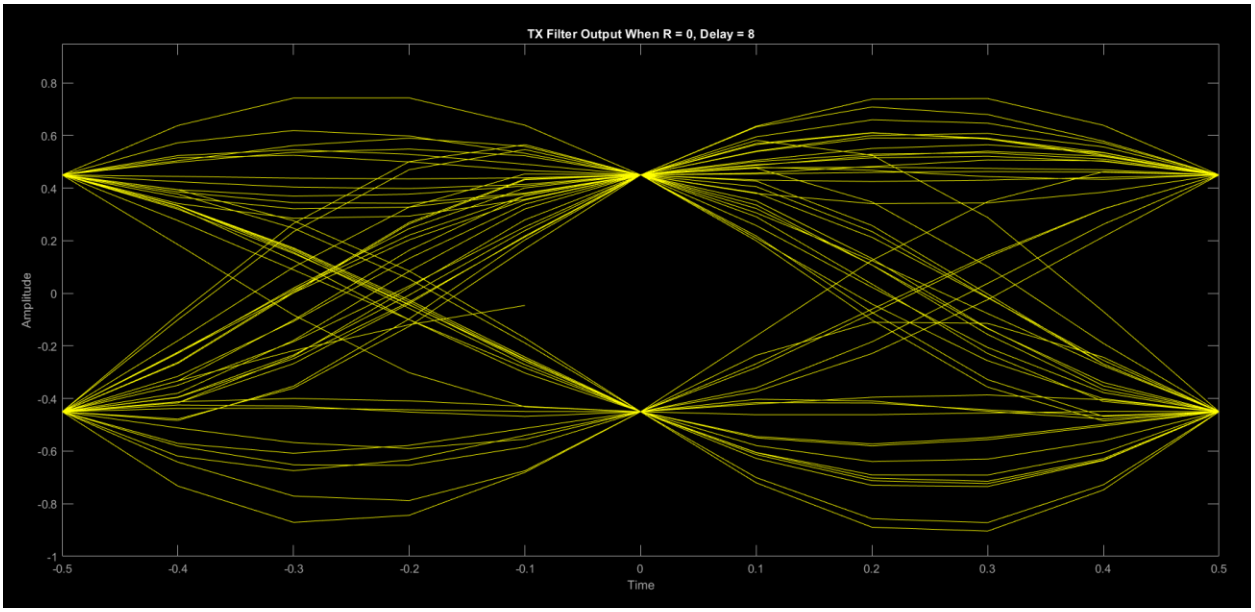

Figure 27: Transmitted signal Eye diagram at R=0 & Delay=8

C: R = 1, delay =2.

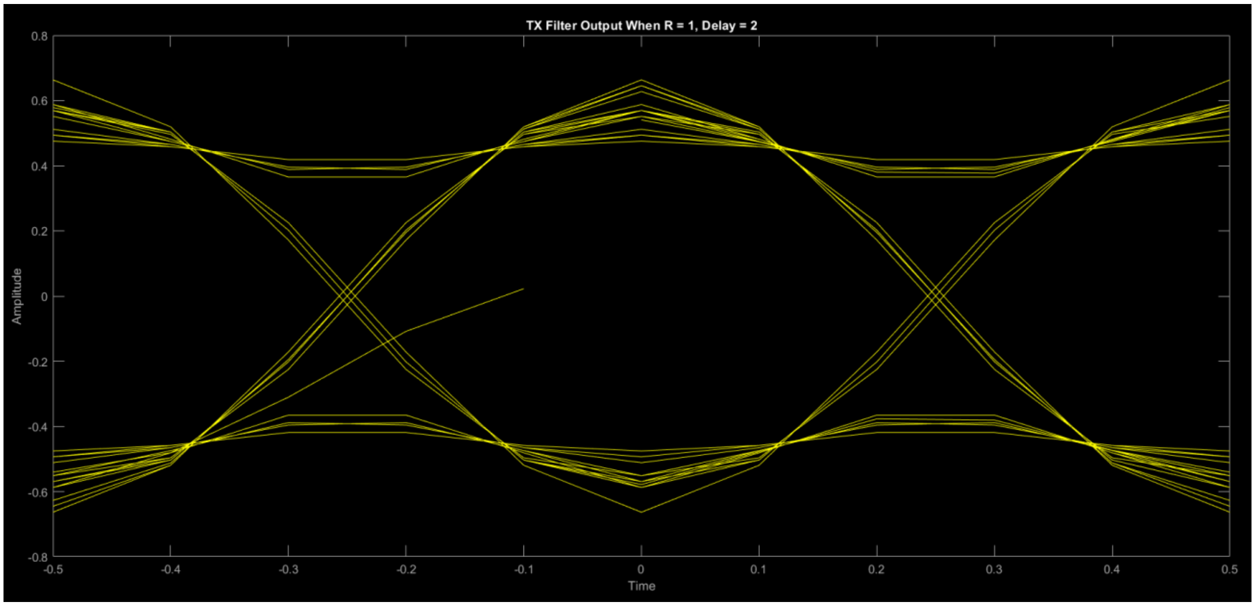

Figure 28: Transmitted signal Eye diagram at R=1 & Delay=2

D: R = 1, delay =8.

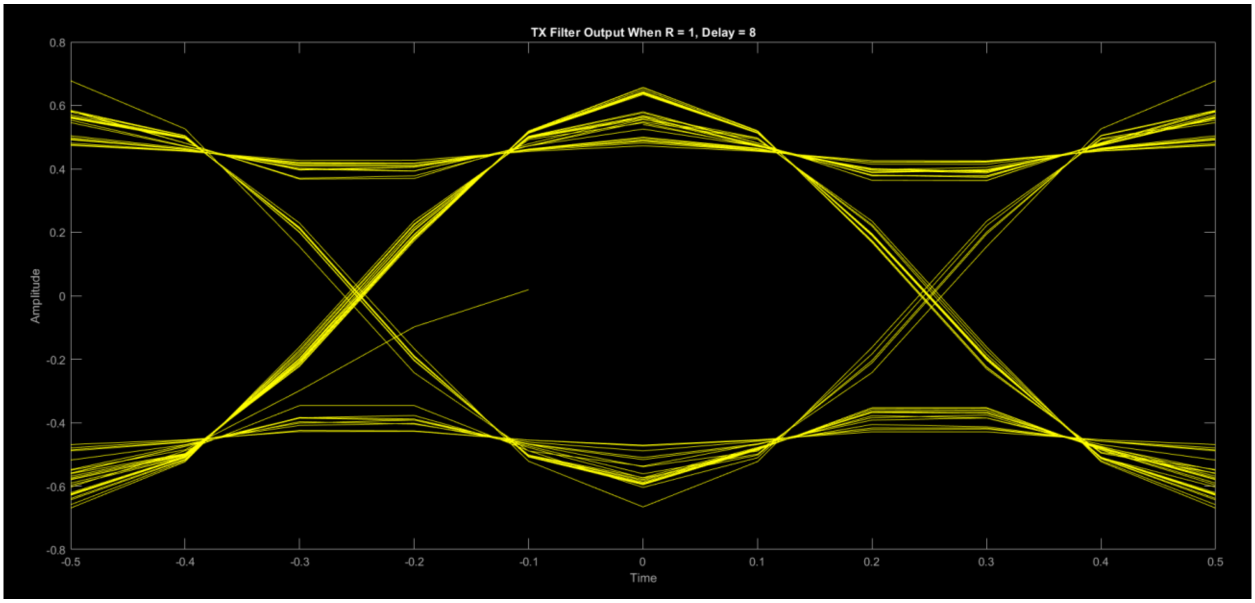

Figure 29: Transmitted signal Eye diagram at R=1 & Delay=8

Comment:

The eye diagrams at the transmitter show how the signal is shaped before it enters the channel. The best sampling point is at the center of the eye opening. A wider eye means better timing tolerance and less risk of symbol errors.

- **For R = 0**: The SRRC filter behaves like a sinc function, which takes longer to fade. At delay = 2, the eye is not very wide because the filter is too short. At delay = 8, the filter is longer, so the eye becomes clearer and more open.
- **For R = 1**: The pulse fades quickly, so the eye diagram is clean even with a short delay. Both delays = 2 and 8 show a wide eye opening, meaning less inter-symbol interference (ISI) and better signal quality.

At Point B: Received signal

Graphs

A: R = 0, delay =2.

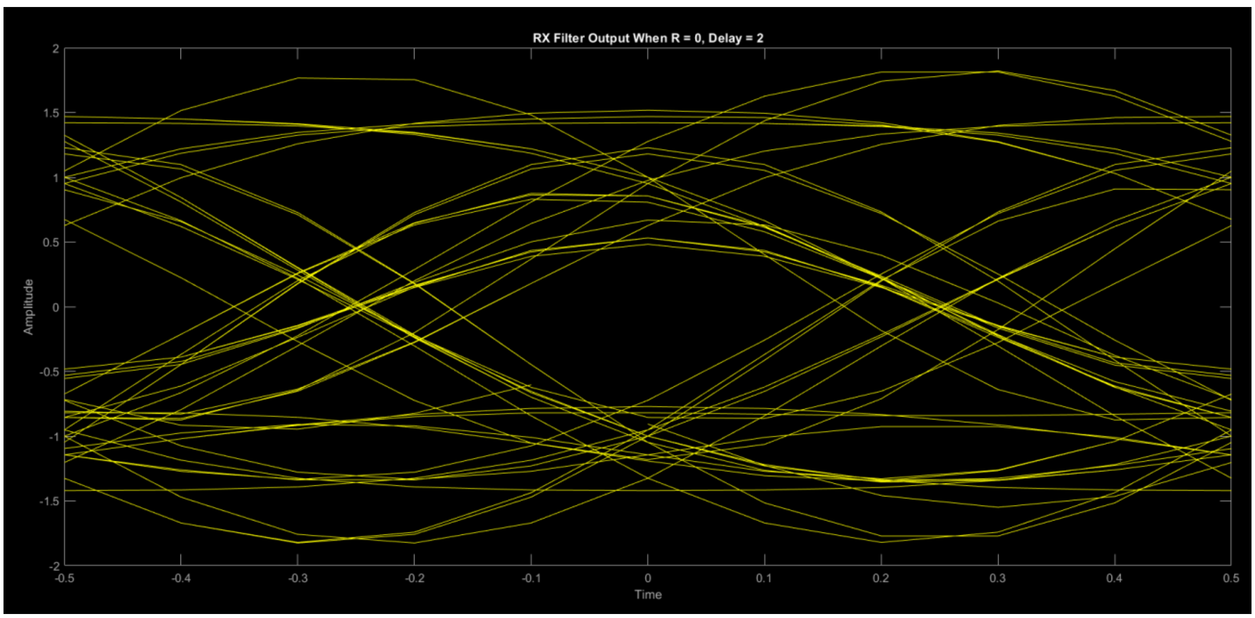

Figure 30: Received signal Eye diagram at R=0 & Delay=2

B: R = 0, delay =8.

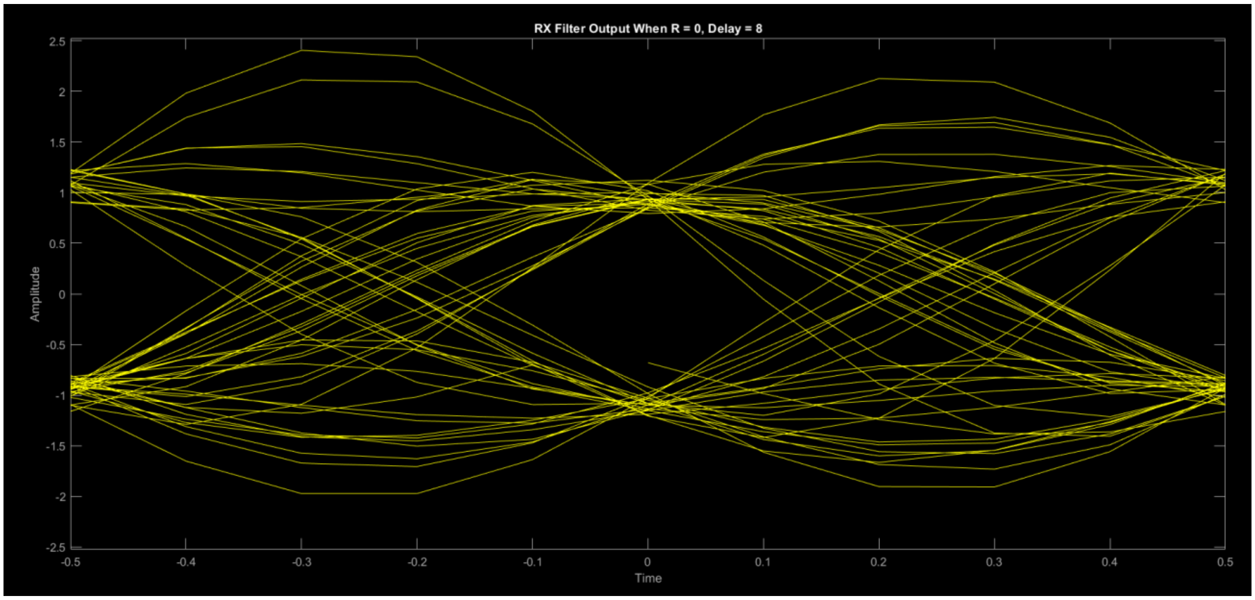

Figure 31: Received signal Eye diagram at R=0 & Delay=8

C: R = 1, delay =2.

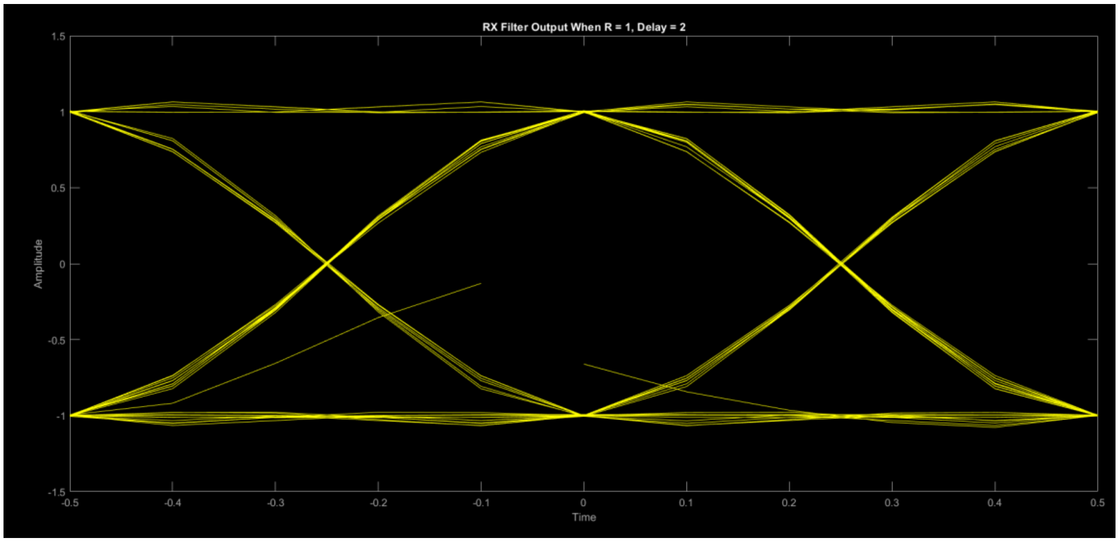

Figure 32: Received signal Eye diagram at R=1 & Delay=2

D: R = 1, delay =8.

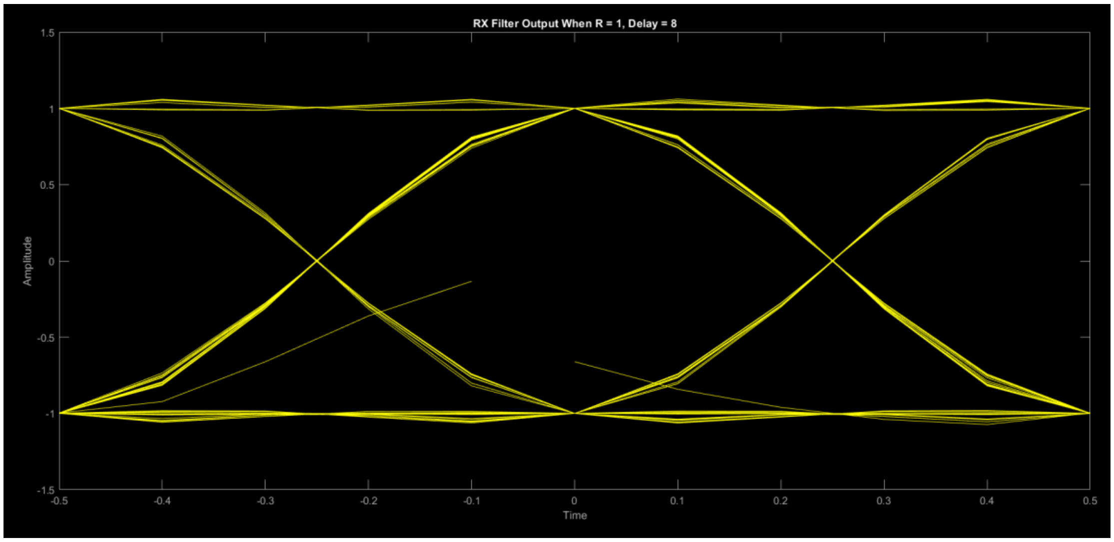

Figure 33: Received signal Eye diagram at R=1 & Delay=8

Comment:

At the receiver, the signal passes through another SRRC filter, which makes the overall response like a full raised cosine filter. This helps reduce ISI even more.

- **For R = 0**: At delay = 2, the eye is still not very open because the sinc-like shape overlaps symbols. With delay = 8, the eye becomes wider and clearer due to better pulse shaping.
- **For R = 1**: The received signal shows clear and wide eyes at both delays. The filter removes ISI effectively, even with short delay.

Also, the signal at the receiver looks a bit stronger in amplitude than at the transmitter, especially when R = 0. That's because it passed through two filters (one at the transmitter and one at the receiver), which makes the main pulse stand out more.

Full Code :

```
% Parameters
Ts = 1;                 % Symbol duration in seconds
samples_per_symbol = 5;                 % Samples per Ts (sampling frequency)
dt = 1/samples_per_symbol;              % Time between samples (200 ms)
N_bits = 10;            % Number of bits
% a) Generate Random Bits
bit_stream = randi([0 1], 1, N_bits);
% b) Map Bits to Symbols (+1 for 1, -1 for 0)
symbols = 2 * bit_stream - 1;
% c) Generate Impulse Signal (upsample by 5)
impulse_train = upsample(symbols, samples_per_symbol);
% Pulse Shaping Filter (normalized)
p = [5 4 3 2 1]/sqrt(55);
% Transmitted Signal (Pulse-Shaped)
tx_signal = conv(impulse_train, p);
t_tx = 0:dt:(length(tx_signal)-1)*dt;
% Plot Bitstream, Symbols, and Impulses
t_impulse = 0:dt:(length(impulse_train)-1)*dt;
figure;
subplot(3,1,1);
stem(t_impulse, impulse_train, 'filled');
title('Impulse Signal (Sampled Every 200 ms)');
xlabel('Time (s)'); ylabel('Amplitude'); ylim([-1.2 1.2]); grid on;

% Plot Transmitted Signal
figure;
plot(t_tx, tx_signal, 'LineWidth', 1.5);
title('Transmitted Signal y[n] (After Pulse Shaping)');
xlabel('Time (s)'); ylabel('Amplitude'); grid on;

%% Matched Filter
matched_filter = fliplr(p);
matched_output = conv(tx_signal, matched_filter);
t_matched = 0:dt:(length(matched_output)-1)*dt;
%.................................
% Correct total system delay after matched filtering
pulse_len = length(p);
total_delay = (pulse_len - 1);  % Transmit + matched filter

% Sampling offset: delay introduced by the filters
sample_offset = total_delay + 1;   % +1 because MATLAB indexing starts at 1
sample_indices = sample_offset : samples_per_symbol: sample_offset + (N_bits-
1)*samples_per_symbol;

sampled_matched = matched_output(sample_indices);
t_samples = t_matched(sample_indices); % common time vector
```

```
%..............................
figure;
subplot(2,1,1);
plot(t_matched, matched_output, 'b-', 'LineWidth', 1.5); hold on;
stem(t_samples, sampled_matched, 'r^', 'filled', 'LineWidth', 1.2);
title('Output After Matched Filter ');
xlabel('Time (s)');
ylabel('Amplitude');
grid on;
legend('Matched Filter Output', 'Sampled Points');
hold off;
%% Rectangular Filter (ideal) — Energy Normalized
rect_filter = ones(1, samples_per_symbol)/sqrt(samples_per_symbol);
rect_output = conv(tx_signal, rect_filter);
t_rect = 0:dt:(length(rect_output)-1)*dt;
sampled_rect = rect_output(sample_indices);
figure;
subplot(2,1,1);
plot(t_rect, rect_output, 'b-', 'LineWidth', 1.5); hold on;
stem(t_samples, sampled_rect, 'r^', 'filled', 'LineWidth', 1.2);
title('Output After Rectangular Filter (Energy Normalized)');
xlabel('Time (s)');
ylabel('Amplitude');
grid on;
legend('Rectangular Filter Output', 'Sampled Points');
hold off;


%% Compare Both Filters on Same Plot
figure;
subplot(2,1,1);
plot(t_matched, matched_output, 'b', 'LineWidth', 1.5); hold on;
plot(t_rect, rect_output, 'r--', 'LineWidth', 1.5);
legend('Matched Filter Output', 'Rectangular Filter Output');
title('Continuous-Time Output of Both Filters');
xlabel('Time (s)'); ylabel('Amplitude'); grid on;



% Plot Sampled Values Only
subplot(2,1,2);
stem(t_samples, sampled_matched, 'bo', 'filled'); hold on;
stem(t_samples, sampled_rect, 'r^', 'filled');
legend('Matched Filter Samples', 'Rectangular Filter Samples');
title('Sampled Outputs at Symbol Timing Instants');
xlabel('Time (s)'); ylabel('Amplitude'); grid on;
%% correlator

% Correlator (continuous) - slide the pulse and take dot product
```

```
correlator_output = zeros(1, length(tx_signal));
for i = 5:5:length(tx_signal)
    for k=0:1:4
    correlator_output(i-k) = sum(tx_signal(i-5+1:i-k) .* p(1:end-k));
end
end
% Pad correlator output to match length of matched filter output
corr_padded = [correlator_output, zeros(1, length(matched_output) -
length(correlator_output))];

% Time vector (same as matched filter for visual alignment)
t_corr_padded = t_matched;
%t_corr = 0:dt:(length(correlator_output)-1)*dt;
sampled_correlator = corr_padded(sample_indices);
 % trim to avoid index issues
t_corr_samples = t_matched(sample_indices(1:end));
% Plot both on same axis
figure;
subplot(2,1,1);
plot(t_matched, matched_output, 'b', 'LineWidth', 1.5); hold on;
stem(t_corr_samples, sampled_matched, 'bo', 'filled'); hold on;
plot(t_corr_padded, corr_padded, 'g', 'LineWidth', 1.5);
stem(t_corr_samples, sampled_correlator, 'gs', 'filled');

legend('Matched Filter Output', 'Correlator Output');
title('Matched Filter vs. Correlator Output (Continuous)');
xlabel('Time (s)'); ylabel('Amplitude'); grid on;

%%-------------------------------With Noise-----------------------

bits_for_noise = randi([0, 1], 1, 10000); % random 10000 bits
data_for_noise = 2 * bits_for_noise - 1;

%% Impulse Train for noise
impulse_train_for_noise = upsample(data_for_noise, samples_per_symbol);
t_impulse_for_noise = (0:length(impulse_train_for_noise)-1) *
(Ts/samples_per_symbol);

figure;
stem(t_impulse_for_noise, impulse_train_for_noise, 'filled', 'MarkerSize',
5);
xlabel('Time (s)');
ylabel('Amplitude');
title('Impulse Train (10K bits)');
ylim([-2 2]);
xlim([0 30]); % plotting it from 0 to 30 not to 10000 for easy plotting
grid on;
xticks(0:1:max(t_impulse_for_noise));
```

```
y_tx_for_noise = conv(impulse_train_for_noise, p, 'full');
t_tx_for_noise = 0:Ts/samples_per_symbol:(length(y_tx_for_noise)-
1)*Ts/samples_per_symbol;


figure;
plot(t_tx_for_noise, y_tx_for_noise,'g','LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Amplitude');
title('Transmitted Signal Pulse-Shaped (Convolved 10K bits)');
ylim([-5 5]);
xlim([0 50]);% plotting it from 0 to 50 not to 10000 for easy plotting
grid on;
xticks(0:1:max(t_tx_for_noise));



% Generate and scale noise
N0 = 1/(10 ^ (-2/10));  % since Eb=1 & N0=1/(10^(SNR/10)),starting from Eb/N0
= -2 dB
Noise_scaled = sqrt(N0/2) * randn(size(y_tx_for_noise));

figure;
plot(t_tx_for_noise, Noise_scaled,'k','LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Amplitude');
title('Noise');
ylim([-5 5]);
xlim([0 30]);% plotting it from 0 to 30 not to 10000 for easy plotting
grid on;
xticks(0:1:max(t_tx_for_noise));






% Add noise to signal
V = y_tx_for_noise + Noise_scaled; % The Transmitted pulse shaped signal
added to noise
t_v = 0:Ts/samples_per_symbol:(length(V)-1)*Ts/samples_per_symbol;

%% Matched Filter Noisy Output
y_matched_noisy= conv(V, matched_filter, 'full');
t_matched_noisy = 0:Ts/samples_per_symbol:(length(y_matched_noisy)-
1)*Ts/samples_per_symbol;
```

```
%% Sampled Matched Filter (Symbol Rate Ts)
matched_noisy_sampled =
y_matched_noisy(samples_per_symbol:samples_per_symbol:end);
t_matched_noisy_sampled = 0:Ts:(length(matched_noisy_sampled)-1)*Ts;

% Plot comparison using subplots
figure;

% Subplot 1: V signal and Continuous Matched Filter Output
subplot(2,1,1);
plot(t_v, V, 'b-', 'LineWidth', 1.5);
hold on;
plot(t_matched_noisy, y_matched_noisy, 'r-', 'LineWidth', 1.5);
title('Noisy Signal (V) and Noisy Matched Filter Output (Continuous)');
legend('Noisy Signal (V)', 'Noisy MF Output Continuous', 'Location',
'SouthEast');
xlabel('Time (s)'); ylabel('Amplitude'); grid on;
xlim([0 50]);
ylim([-6 6]);
xticks(0:1:max(t_tx_for_noise));

% Subplot 2: V signal and Sampled Matched Filter Output
subplot(2,1,2);
plot(t_v, V, 'b-', 'LineWidth', 1.5);
hold on;
stem(t_matched_noisy_sampled, matched_noisy_sampled, 'ko', 'LineWidth', 1.5,
'MarkerSize', 5);
plot(t_matched_noisy_sampled, matched_noisy_sampled, 'k--', 'LineWidth',
1.5);
title('Noisy Signal (V) and Sampled Noisy Matched Filter Output');
legend('Noisy Signal (V)', 'Noisy MF (Sampled)', 'Noisy MF (Continuous
Samples)', 'Location', 'SouthEast');
xlabel('Time (s)'); ylabel('Amplitude'); grid on;
xlim([0 50]);
ylim([-6 6]);
xticks(0:1:max(t_tx_for_noise));

%% BER Calculation
N0_Values_linear = [1/(10 ^ (-2/10)),1/(10 ^ (-1/10)),1/(10 ^ (-0/10)),1/(10
^ (1/10)),1/(10 ^ (2/10)),1/(10 ^ (3/10)),1/(10 ^ (4/10)),1/(10 ^ (5/10))];
EbN0_Values_linear = 1 ./ N0_Values_linear;
EbN0_Values_dB = [-2,-1,0,1,2,3,4,5];
BER_theoretical = 0.5 * erfc(sqrt(EbN0_Values_linear));


MF_BER_array = calculate_MF_BER(N0_Values_linear, y_tx_for_noise,
bits_for_noise, matched_filter, samples_per_symbol);
% Plotting BER of Matched Filter
figure;
```

```
plot(EbN0_Values_dB, MF_BER_array, 'b-o', 'LineWidth',
1.5,'DisplayName','Simulated BER');
hold on;
plot(EbN0_Values_dB, BER_theoretical, 'r--', 'LineWidth', 1.5, 'DisplayName',
'Theoretical BER');
hold off;
xlabel('Eb/N0 (dB)');
ylabel('Bit Error Rate (BER) ');
title('BER vs Eb/N0 for Matched Filter Output : Simulation vs Theoritical');
grid on;
legend('Location', 'southwest');
ylim([0 0.2]);
yticks(0:(1e-2):0.2);
xticks(EbN0_Values_dB);



Rect_BER_array = calculate_Rect_BER(N0_Values_linear, y_tx_for_noise,
bits_for_noise, rect_filter, samples_per_symbol);
% Plotting BER of Rect Filter
figure;
plot(EbN0_Values_dB, Rect_BER_array, 'b-o', 'LineWidth',
1.5,'DisplayName','Simulated BER');
hold on;
plot(EbN0_Values_dB, BER_theoretical, 'r--', 'LineWidth', 1.5, 'DisplayName',
'Theoretical BER');
hold off;
xlabel('Eb/N0 (dB)');
ylabel('Bit Error Rate (BER)');
title('BER vs Eb/N0 for Rect Filter Output : Simulation vs Theoritical');
grid on;
legend('Location', 'southwest');
ylim([0 0.2]);
yticks(0:(1e-2):0.2);
xticks(EbN0_Values_dB);


% Combining MF BER & Rect BER Plots
figure;
plot(EbN0_Values_dB, MF_BER_array, 'm-o', 'LineWidth', 1.5, 'DisplayName',
'Matched Filter (Simulated)');
hold on;
plot(EbN0_Values_dB, Rect_BER_array, 'b-o', 'LineWidth', 1.5, 'DisplayName',
'Rect Filter (Simulated)');
plot(EbN0_Values_dB, BER_theoretical, 'k--', 'LineWidth', 1.5, 'DisplayName',
'Theoretical BER');
hold off;
```

```
xlabel('Eb/N0 (dB)');
ylabel('Bit Error Rate (BER)');
title('BER Comparison: Matched Filter vs Rectangular Filter');
grid on;
legend('Location', 'southwest');
ylim([0, 0.2]);
xticks(EbN0_Values_dB);


%% ISI and Raised cosine
Data = randi([0, 1], 1, 100);%generation of 100 random bits data
Data_forISI = 2 * Data - 1;%mapping bits to 1 & -1
impulse_train_Data_forISI = upsample(Data_forISI, samples_per_symbol);


delay_values = [2, 8];%delay
R_values = [0, 1];%roll-off factor



for R = R_values
   for delay = delay_values
       %rcos_filter = rcosine(1/Ts, samples_per_symbol, 'sqrt', R,
delay);%generation of raisedcosine filter
       rcos_filter = rcosdesign(R, 2*delay, samples_per_symbol, 'sqrt');

       figure;%ploting the filter
       plot(rcos_filter);
       title("rcosine filter R: " + R + "Delay: " + delay);
       xlabel("time_samples");
       ylabel("Amplitude");

       %passing signals through tx_filter then channel has no effect-------
      % A = filter(rcos_filter, 1, impulse_train_Data_forISI);
       A = conv(impulse_train_Data_forISI , rcos_filter , 'same');
       %then passing through rx filter-------------------------------
      % B = filter(rcos_filter, 1, A);
       B = conv(A , rcos_filter , 'same');

       %ploting  A,B signals---------------------------------------------
       fig_tit = ['R = ', num2str(R), ', Delay = ', num2str(delay)];
       figure;
       t_A = (0:length(A)-1) * (Ts/samples_per_symbol);
       plot(t_A, A);
       title(['Signal after transmit: ', fig_tit]);
       xlabel('Time (s)');
       ylabel('Amplitude');
       grid on;
```

```
        figure;
        t_B = (0:length(B)-1) * (Ts/samples_per_symbol);
        plot(t_B, B);
        title(['Signal after Receive : ', fig_tit]);
        xlabel('Time (s)');
        ylabel('Amplitude');
        grid on;


        %eye diagram-----------------------------------------------------------
---------
        %ploting each diagram alone of A,B----------------
        figure;
        eyediagram(A, samples_per_symbol*2);
        title(['TX Filter Output : ', fig_tit]);

        figure;
        eyediagram(B, samples_per_symbol*2);
        title(['RX Filter Output : ', fig_tit]);
        %----------------------------------------------------
        %ploting eye diagram A,B together for the same R&delay-------
        %figure;
        %eye_fig = eyediagram( [A ; B]' , samples_per_symbol*2);
        %set(eye_fig,'Name',"eyediagram for R :" + R + " Delay: " + delay);
    end
end



%%  Function for calculating the BER for different EbN0 after MF filtering
function BER_array_MF = calculate_MF_BER(N0_Values_linear, y_tx_for_noise,
bits_for_noise, matched_filter, samples_per_symbol)
    % Initialize BER array
    BER_array_MF = zeros(1, length(N0_Values_linear));

    % Loop over each EbN0 value
    for i = 1:length(N0_Values_linear)

        Noise_scaled = sqrt(N0_Values_linear(i)/2) *
randn(size(y_tx_for_noise));

        V = y_tx_for_noise + Noise_scaled;

        % Matched filtering
        y_matched_noisy = conv(V, matched_filter, 'full');

        % Sample at symbol rate
        matched_noisy_sampled =
y_matched_noisy(samples_per_symbol:samples_per_symbol:end);
```

```
        matched_noisy_sampled =
matched_noisy_sampled(1:length(bits_for_noise));
        % Decision at Threshold = 0 if >=0 → 1, else → 0
        detected_bits = matched_noisy_sampled >= 0;
        % errors counting
        num_errors = sum(detected_bits ~= bits_for_noise);
        % Calculating BER
        BER_array_MF(i) = num_errors / length(bits_for_noise);

        EbN0_dB = 10 * log10(1 / N0_Values_linear(i));
        fprintf('MF Out Bit Error Rate (BER) (@Eb/N0=%.0f dB) = %.5f\n',
EbN0_dB, BER_array_MF(i));
    end
end


%%  Function for calculating the BER for different EbN0 after Rect filtering
function BER_array_Rect = calculate_Rect_BER(N0_Values_linear,
y_tx_for_noise, bits_for_noise, rect_filter, samples_per_symbol)
    % Initialize BER array
    BER_array_Rect = zeros(1, length(N0_Values_linear));
    % Loop over each EbN0 value
    for i = 1:length(N0_Values_linear)

        Noise_scaled = sqrt(N0_Values_linear(i)/2) *
randn(size(y_tx_for_noise));

        V = y_tx_for_noise + Noise_scaled;


        y_rect_noisy = conv(V, rect_filter, 'full');

        % Sample at symbol rate
        rect_noisy_sampled =
y_rect_noisy(samples_per_symbol:samples_per_symbol:end);

        rect_noisy_sampled = rect_noisy_sampled(1:length(bits_for_noise));
        % Decision at Threshold = 0 if >=0 → 1, else → 0
        detected_bits = rect_noisy_sampled >= 0;
        % errors counting
        num_errors = sum(detected_bits ~= bits_for_noise);
        % Calculating BER
        BER_array_Rect(i) = num_errors / length(bits_for_noise);

        EbN0_dB = 10 * log10(1 / N0_Values_linear(i));
        fprintf('Rect Filter Bit Error Rate (BER) (@Eb/N0=%.0f dB) = %.5f\n',
EbN0_dB, BER_array_Rect(i));
    end
end
```