Communication Systems
Final Project

Iman Alizadeh Fakouri

Student Number: 401102134

Professor: Dr. Mohammadreza Pakravan

Date: February 6, 2025

Contents

Introduction

In this project, we design and analyze a digital communication system for image transmission. The system consists of multiple processing stages, including image preprocessing, modulation, channel transmission, equalization, and reconstruction. The main objective is to evaluate the system's performance under different pulse shapes, noise levels, and channel conditions.

The transmission process begins with Discrete Cosine Transform (DCT) to compress the image into quantized blocks. The quantized coefficients are then converted into a bitstream and modulated using Pulse Amplitude Modulation (PAM). Two different pulse shapes—Half-Sine and Square Root Raised Cosine (SRRC)—are used for comparison. The modulated signal is transmitted through a linear time-invariant (LTI) channel with additive white Gaussian noise (AWGN).

At the receiver, the signal is passed through a matched filter, followed by zero-forcing (ZF) and minimum mean square error (MMSE) equalizers to mitigate inter-symbol interference (ISI). The recovered bitstream is used to reconstruct the original image using inverse DCT.

To verify the system's functionality, we will first implement and test individual components using a simple sequence of 10 random bits. After confirming their correctness, we will apply the complete system to transmit and reconstruct an image, analyzing performance under different conditions such as noise and channel distortions.

Image Pre-Processing

Image pre-processing is a crucial step in digital communication systems, especially for image transmission. The primary goal is to convert an image into a format suitable for efficient transmission while minimizing redundant information. The Discrete Cosine Transform (DCT) is widely used in image compression because it concentrates most of the signal energy into a few coefficients, making quantization more effective.

By dividing the image into 8×8 blocks, we ensure that the transformation and compression are manageable while preserving local spatial details. The quantization process further reduces the data size by limiting the precision of coefficients, enabling better transmission efficiency. The final step involves restructuring the processed data into a 3D format, facilitating transmission and reconstruction at the receiver.

```
import cv2
import numpy as np
from skimage.util import view_as_blocks
from scipy.fftpack import dct

image = cv2.imread("cameraman.tif", cv2.IMREAD_GRAYSCALE)
image = image.astype(np.float64)

m, n = image.shape
m, n = (m // 8) * 8, (n // 8) * 8
image = image[:m, :n]

block_size = 8
blocks = view_as_blocks(image, block_shape=(block_size, block_size))

dct_blocks = np.zeros_like(blocks, dtype=np.float64)
for i in range(blocks.shape[0]):
    for j in range(blocks.shape[1]):
        dct_blocks[i, j] = dct(dct(blocks[i, j], axis=0, norm='ortho'), axis=1, norm='ortho')
```

```

min_val, max_val = dct_blocks.min(), dct_blocks.max()
scaled_blocks = (dct_blocks - min_val) / (max_val - min_val)
scaling_constants = {"min": min_val, "max": max_val}

# Quantization
quantized_blocks = np.round(scaled_blocks * 255).astype(np.uint8)

permuted_blocks = quantized_blocks.transpose(2, 3, 0, 1)
reshaped_data = permuted_blocks.reshape((8, 8, -1))

print(f"Image shape: {image.shape}")
print(f"Quantized 3D Data Shape: {reshaped_data.shape}")
```

This implementation follows a structured approach:

- The image is first loaded in grayscale and converted to a double precision format for accurate computation.
- The dimensions are adjusted to ensure they are divisible by 8, allowing uniform 8×8 block division.
- The Discrete Cosine Transform (DCT) is applied to each block, transforming spatial information into frequency coefficients.
- The transformed data is scaled between 0 and 1 for normalization, and the scaling constants are stored for later reconstruction.
- The coefficients are quantized to reduce data precision, facilitating compression.
- Finally, the quantized blocks are rearranged into a 3D structure, preparing the data for modulation and transmission.

Converting to Bitstream

After image pre-processing and applying the Discrete Cosine Transform (DCT), the next step in the transmission pipeline is converting the transformed data into a bitstream. The goal is to organize the quantized DCT blocks into a structured format suitable for digital communication.

To achieve this, we group the quantized DCT blocks into sets of size N, where N is a variable parameter that determines how many blocks are processed together. This approach enhances flexibility in transmission and allows adaptation to different system constraints.

Each group of N blocks is first converted into a column vector, where each element represents an 8-bit value. These values are then expanded into a binary matrix, where each row contains the 8-bit binary representation of the corresponding value.

```
N = 4  # Changeable

num_blocks = reshaped_data.shape[2]
block_size = 8 * 8

if num_blocks % N != 0:
    padding = N - (num_blocks % N)
    reshaped_data = np.pad(reshaped_data, ((0, 0), (0, 0), (0, padding)), mode='constant')
```

```
    num_blocks += padding

grouped_blocks = reshaped_data.reshape(block_size, num_blocks)
grouped_blocks = grouped_blocks[:, :num_blocks]

num_groups = num_blocks // N
print(num_groups)
column_vectors = grouped_blocks.reshape(block_size * N, num_groups)

binary_matrix = np.unpackbits(column_vectors.astype(np.uint8), axis=0)
```

This implementation follows a structured approach:

- The number of blocks in the quantized DCT data is obtained.
- If the total number of blocks is not a multiple of N, padding is added to ensure uniform grouping.
- The data is reshaped into a structure where each group of N blocks forms a column vector.
- The column vectors are then reshaped into a binary matrix, where each value is expanded into its 8-bit binary representation.

Modulation

In digital communication, modulation is used to encode the data for transmission over a communication medium. In this report, we focus on Pulse Amplitude Modulation (PAM), which is a straightforward method where each bit is mapped to a pulse with a specific amplitude. The amplitude of the pulse varies depending on the bit value, and the pulse duration corresponds to the symbol time T.

We use two different pulse shapes: the first is a simple sinusoidal pulse, and the second is a Square Root Raised Cosine (SRRC) pulse. These pulses are essential for ensuring the efficient transmission of data while minimizing interference.

4.1 Modulation Using Sinusoidal Pulse

The pulse shape for $b = 1$ is given by:

$$g_1(t) = sin\left(\frac{\pi t}{T}\right)$$

for $0 < t < T$, and for $b = 0$:

$$g_1(t) = -\sin\left(\frac{\pi t}{T}\right)$$

where $T = 1$ second is the symbol duration, and the pulse function varies depending on whether the bit is 1 or 0.

The theory behind the use of sinusoidal pulses is that they can be easily generated and have a simple frequency response. In this case, we analyze the pulse's frequency spectrum, which is the Fourier transform of the time-domain pulse.

```
T = 1
Fs = 1000
t = np.linspace(0, T, Fs)

g1_t_b1 = np.sin(np.pi * t / T)
g1_t_b0 = -np.sin(np.pi * t / T)

plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(t, g1_t_b1, label='Pulse for b=1')
plt.plot(t, g1_t_b0, label='Pulse for b=0', linestyle='--')
plt.title('Pulse Function for b=1 and b=0')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()


f = np.fft.fftfreq(len(t), d=1/Fs)
G1_F_b1 = np.fft.fft(g1_t_b1)
G1_F_b0 = np.fft.fft(g1_t_b0)
f_shifted = np.fft.fftshift(f)
G1_F_b1_shifted = np.fft.fftshift(G1_F_b1)
G1_F_b0_shifted = np.fft.fftshift(G1_F_b0)

plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(f_shifted, np.abs(G1_F_b1_shifted), label='Magnitude for b=1')
plt.plot(f_shifted, np.abs(G1_F_b0_shifted), label='Magnitude for b=0', linestyle='--')
plt.xlim([-50, 50])
plt.title('Magnitude Spectrum')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.legend()


positive_freqs = f[f >= 0]
G1_F_b1_pos = G1_F_b1[:len(positive_freqs)]
G1_F_b0_pos = G1_F_b0[:len(positive_freqs)]
phase_b1_deg = np.angle(G1_F_b1_pos, deg=True)
phase_b0_deg = np.angle(G1_F_b0_pos, deg=True)
plt.figure(figsize=(12, 6))
plt.plot(positive_freqs, phase_b1_deg, label='Phase for b=1', color='b')
plt.plot(positive_freqs, phase_b0_deg, label='Phase for b=0', linestyle='--', color='r')
plt.title('Phase Response for Positive Frequencies (in Degrees)')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Phase (Degrees)')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

bits = np.random.randint(0, 2, 10)
```

```python
print("Random Bits:", bits)

modulated_signal = []
for bit in bits:
    if bit == 1:
        modulated_signal.extend(g1_t_b1)
    else:
        modulated_signal.extend(g1_t_b0)

t_modulated = np.linspace(0, T*len(bits), len(modulated_signal))

plt.figure(figsize=(12, 6))
plt.plot(t_modulated, modulated_signal)
plt.title('Modulated Signal Using PAM (10 Random Bits)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()

modulated_signal_F = np.fft.fft(modulated_signal)
f_modulated = np.fft.fftfreq(len(t_modulated), d=1/Fs)

plt.figure(figsize=(12, 6))
plt.plot(f_modulated, np.abs(modulated_signal_F), label='Magnitude')
plt.title('Frequency Response of the Modulated Signal')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.xlim([-50, 50])
plt.grid(True)
plt.legend()
plt.show()

def plot_eye_diagram(signal, symbol_length, samples_per_symbol, num_symbols):
    plt.figure(figsize=(12, 6))
    time_per_symbol = np.linspace(0, T, samples_per_symbol)

    for i in range(num_symbols):
        start_idx = i * samples_per_symbol
        end_idx = start_idx + samples_per_symbol
        if end_idx <= len(signal):
            plt.plot(time_per_symbol, signal[start_idx:end_idx], color='gray', alpha=0.5)

    plt.title('Eye Diagram')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.grid(True)
    plt.show()

plot_eye_diagram(modulated_signal, T, Fs, num_symbols=10)
```

The code demonstrates the modulation of a signal using sinusoidal pulses. The pulse function for each bit is generated as a sine or negative sine wave, and its frequency spectrum is computed. The modulated signal is constructed by concatenating pulses corresponding to random bits. The eye diagram is also plotted to analyze the signal integrity.

4.2 Modulation Using SRRC Pulse

For improved performance and to minimize inter-symbol interference (ISI), we use a Square Root Raised Cosine (SRRC) pulse. The SRRC pulse is given by:

$$g_2(t) = \begin{cases} A \cdot x(t) & \text{for } b = 1 \\ -A \cdot x(t) & \text{for } b = 0 \end{cases}$$

where the impulse response $x(t)$ is defined as:

$$x(t) = \frac{\sin\left(\pi \cdot t/T \cdot (1-b)\right) + 4b \cdot t/T \cdot \cos\left(\pi \cdot t/T \cdot (1+b)\right)}{\pi \cdot t/T \cdot \left(1 - (4b \cdot t/T)^2\right)}$$

Special values of $x(t)$ are:

$$x(0) = 1 - b + \frac{4b}{\pi}, \quad x\left(\pm\frac{T}{4b}\right) = \frac{b}{\sqrt{2}} \cdot \left(\left(1 + \frac{2}{\pi}\right)\sin\left(\frac{\pi}{4b}\right) + \left(1 - \frac{2}{\pi}\right)\cos\left(\frac{\pi}{4b}\right)\right)$$

The SRRC pulse has a spectrum that is defined as:

$$X(f) = \sqrt{T} \quad \text{for } 0 \le |f| < \frac{1-b}{2T}$$

$$X(f) = \sqrt{T} \cdot \cos\left(\frac{\pi T}{2b}\left(|f| - \frac{1-b}{2T}\right)\right) \quad \text{for } \frac{1-b}{2T} \le |f| \le \frac{1+b}{2T}$$

$$X(f) = 0 \quad \text{for } |f| > \frac{1+b}{2T}$$

The code for modulating the signal using SRRC pulses is as follows:

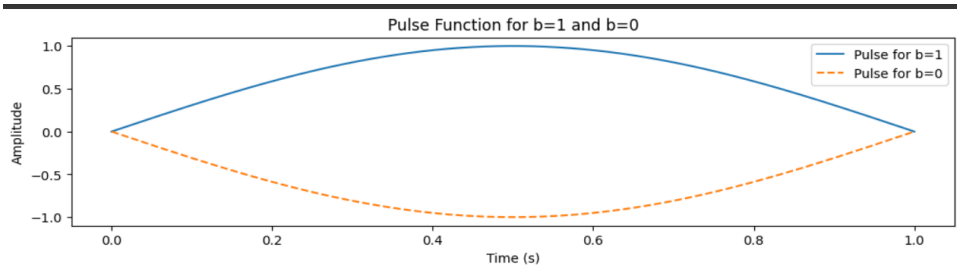
```
T = 1
Fs = 1000
K = 6
b = 0.5
A = 1

t = np.linspace(-K*T, K*T, Fs * 2 * K)
def srrc_pulse(t, T, b):
    numerator = np.sin(np.pi * t / T * (1 - b)) + 4 * b * t / T * np.cos(np.pi * t / T * (1 + b)
    denominator = np.pi * t / T * (1 - (4 * b * t / T) ** 2)
    x_t = np.zeros_like(t)
    mask_zero = (t == 0)
    mask_special = np.abs(t) == T / (4 * b)
    x_t[mask_zero] = (1 - b + 4 * b / np.pi)
    x_t[mask_special] = b / np.sqrt(2) * ((1 + 2 / np.pi) * np.sin(np.pi / (4 * b)) + (1 - 2 / n
    x_t[~(mask_zero | mask_special)] = numerator[~(mask_zero | mask_special)] / denominator[~(ma
    return x_t

x_t_b1 = srrc_pulse(t, T, b)
x_t_b0 = -srrc_pulse(t, T, b)
```

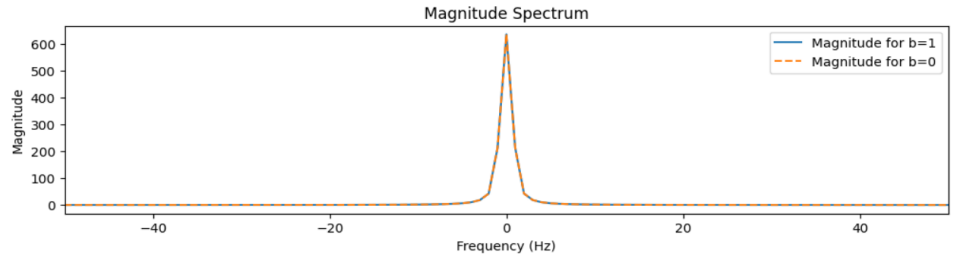
```
21
22 plt.figure(figsize=(12, 6))
23 plt.subplot(2, 1, 1)
24 plt.plot(t, x_t_b1, label='Pulse for b=1')
25 plt.plot(t, x_t_b0, label='Pulse for b=0', linestyle='--')
26 plt.title('SRRC Pulse Function for b=1 and b=0')
27 plt.xlabel('Time (s)')
28 plt.ylabel('Amplitude')
29 plt.legend()
```

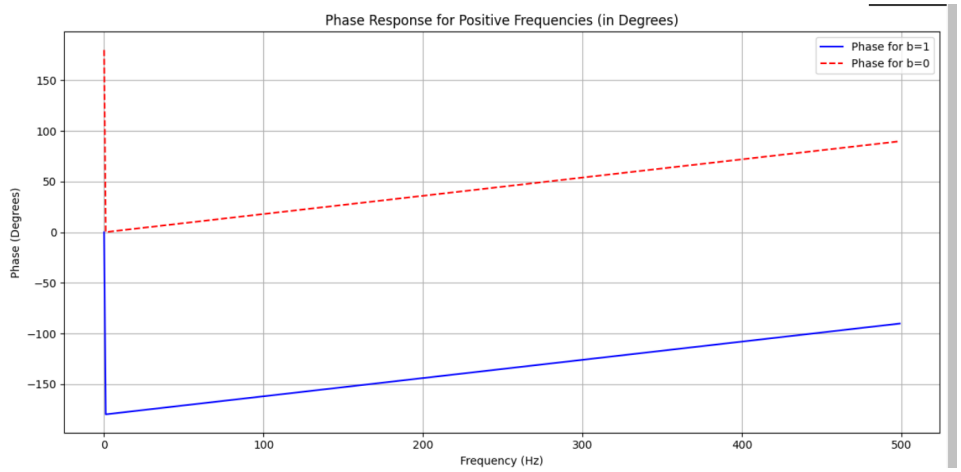
This modulation process uses SRRC pulses to modulate the signal, followed by plotting the frequency response and eye diagram for visual analysis. The SRRC pulse improves signal quality by minimizing ISI and ensuring efficient bandwidth utilization.

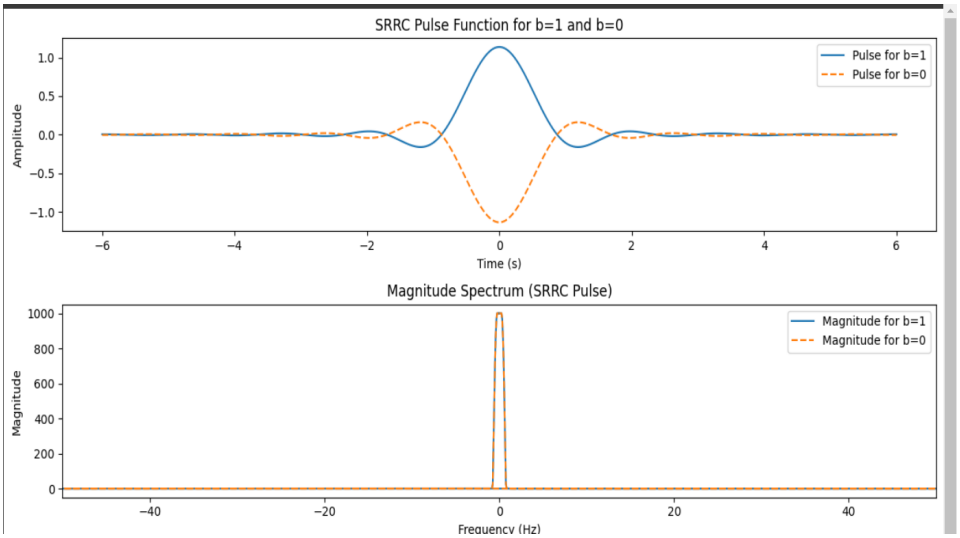
4.3 Q1

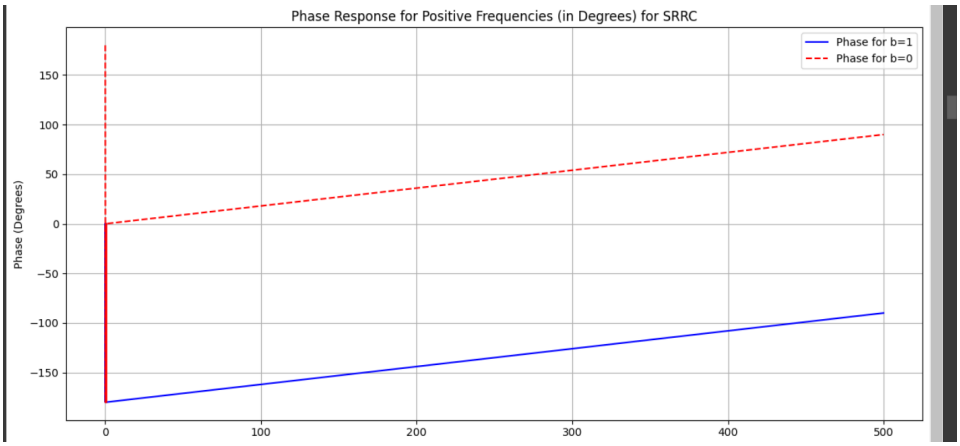
The SRRC pulse generally occupies less bandwidth compared to a sinusoidal pulse due to its optimized frequency response with a controlled roll-off factor. Increasing the pulse duration (cutting length) results in a narrower frequency spectrum for both pulse types. For SRRC pulses, this reduces bandwidth, while improving inter-symbol interference (ISI). Sinusoidal pulses, being wider in bandwidth, show a more noticeable reduction in bandwidth with increasing duration. Overall, longer pulses lead to reduced bandwidth but can improve signal separation and reduce ISI.





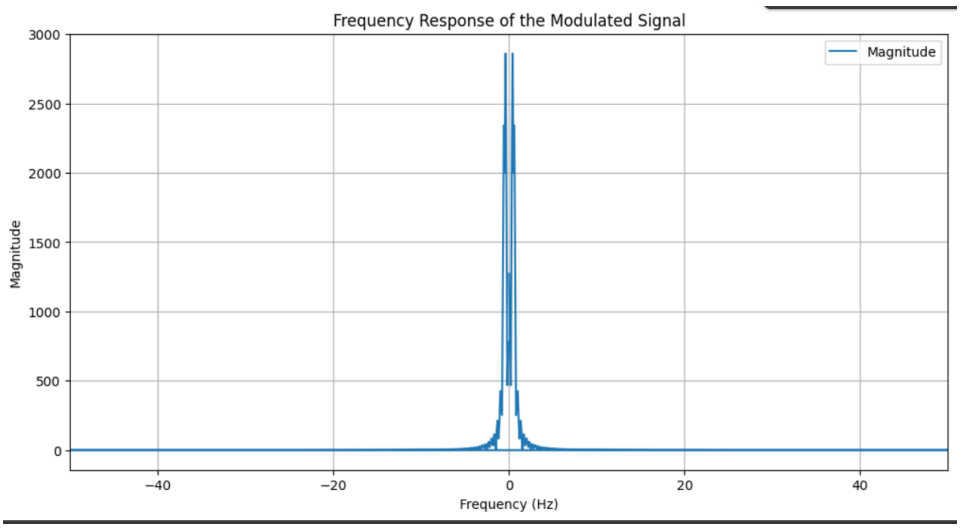


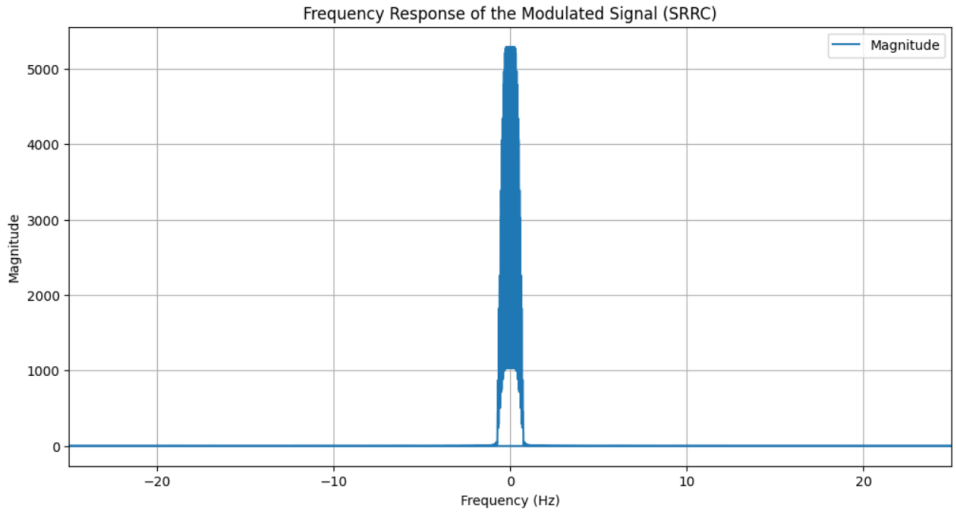

SRRC Pulse Function for b=1 and b=0
Pulse for b=1
Pulse for b=0
Amplitude
Time (s)
Magnitude Spectrum (SRRC Pulse)
Magnitude for b=1
Magnitude for b=0
Magnitude
Frequency (Hz)


Phase Response for Positive Frequencies (in Degrees) for SRRC
Phase for b=1
Phase for b=0
Phase (Degrees)

4.4 Q2

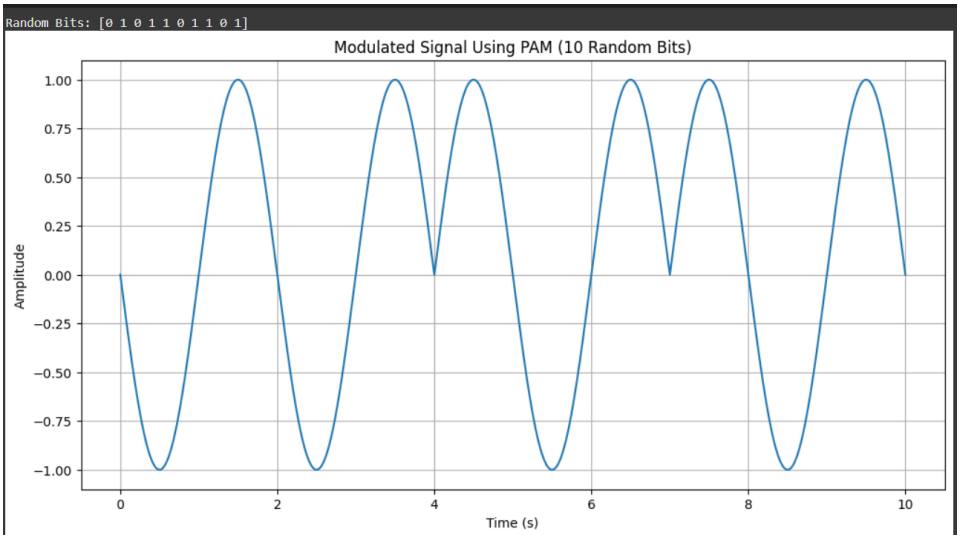
The frequency response of the sinusoidal PAM pulse exhibits a wide bandwidth with slow decay, indicating broader spectral occupancy. In contrast, the SRRC pulse has a more compact frequency response with a sharper roll-off, resulting in a more efficient use of bandwidth. SRRC is optimized for minimizing inter-symbol interference, while sinusoidal PAM requires more bandwidth.

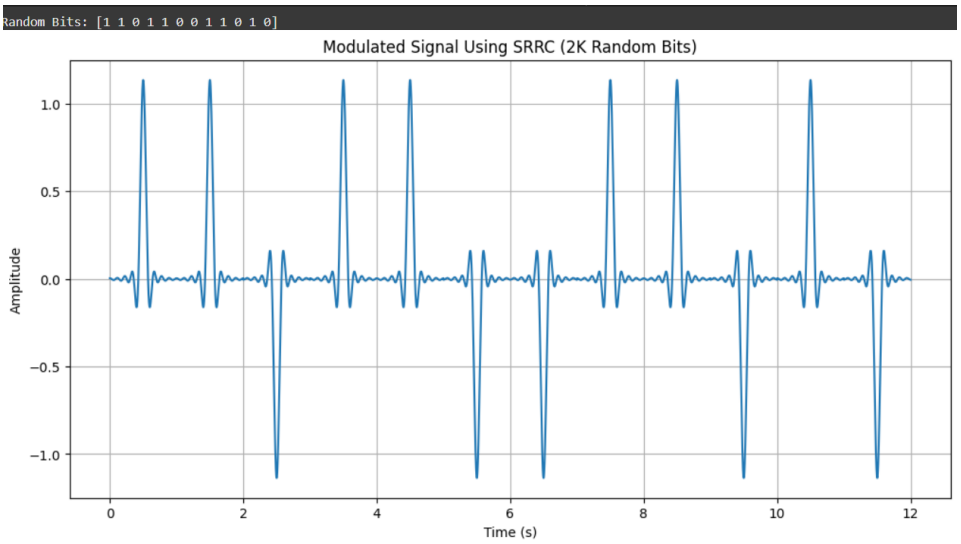




4.5 Q3

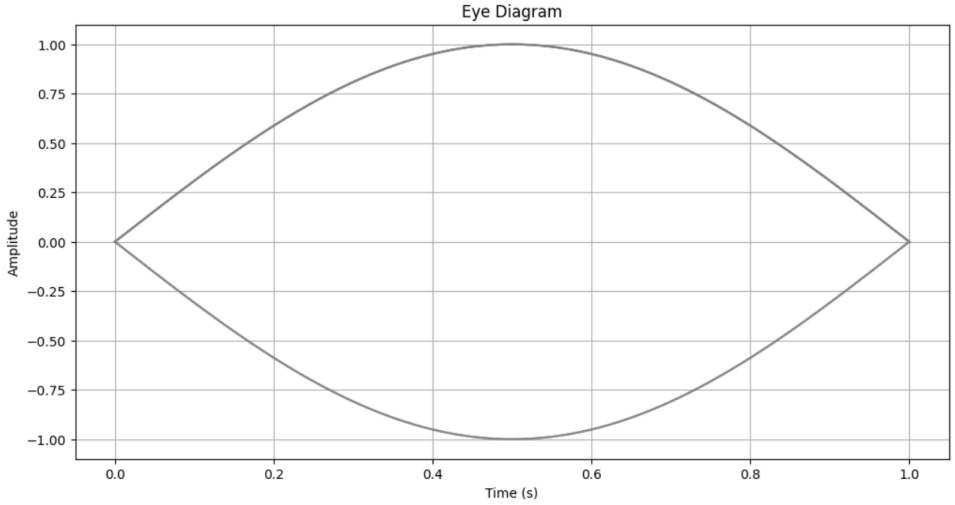
For a 10-bit random sequence, the sinusoidal PAM modulation creates alternating sinusoidal pulses for each bit, resulting in a wider frequency spectrum. The SRRC modulation produces smoother transitions between symbols, reducing inter-symbol interference and requiring less bandwidth, making it more efficient and cleaner.

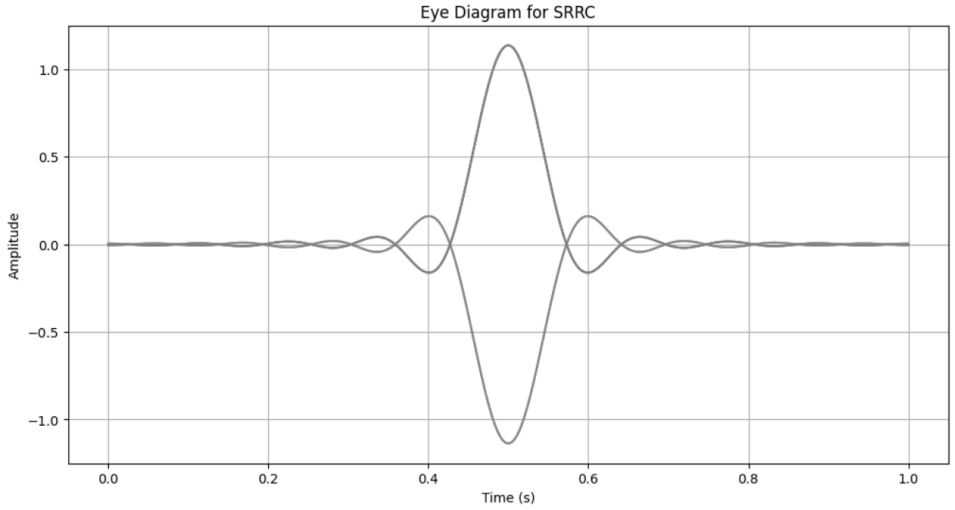




4.6 Q4

The eye diagram for sinusoidal PAM shows a smaller, more closed eye due to significant inter-symbol interference and noise. The SRRC modulation shows a wider and more open eye, indicating better signal integrity, reduced ISI, and improved noise tolerance, making it a more robust choice for signal transmission.





Channel Effect

The channel is a critical element in a communication system as it dictates how the transmitted signal will be affected during its journey from the transmitter to the receiver. The channel can introduce various impairments such as noise, fading, and distortion, leading to errors in the received signal. The impact of the channel can vary based on its characteristics, and it is essential to model and understand this effect in communication systems.

A communication channel can be modeled as a linear system with some time-varying behavior. The signal transmitted over the channel gets convolved with an impulse response that characterizes the system's behavior. In the ideal case, the impulse response of the channel is a Dirac delta function, which implies that the signal passes through the channel unaffected. However, in practical systems, the channel introduces distortions.

The discrete-time impulse response $h[n]$ of the channel can be expressed as:

$$h[n] = \delta[n] + \frac{1}{2}\delta[n-1] + \frac{3}{4}\delta[n-2] - \frac{2}{7}\delta[n-3]$$

Here, $\delta[n]$ represents the discrete-time unit impulse function. The coefficients $1, \frac{1}{2}, \frac{3}{4}, -\frac{2}{7}$ determine the weighting of different delayed versions of the signal in the channel. This channel will cause the transmitted signal to experience different levels of attenuation and phase shift at various delays.

The following code implements this impulse response and plots the corresponding response in the time domain:

```
T = 1
Fs = 1000
samples_per_symbol = int(T * Fs)

h = np.zeros(4 * samples_per_symbol)
h[0] = 1.0
h[samples_per_symbol] = 1/2
h[2 * samples_per_symbol] = 3/4
h[3 * samples_per_symbol] = -2/7

H_f = np.fft.fft(h, n=1024)
f = np.fft.fftfreq(1024, d=1/Fs)

plt.figure(figsize=(12, 8))
plt.stem(np.arange(len(h)), h, basefmt=" ")
plt.title("Impulse Response of Channel h[n]")
plt.xlabel("Samples")
plt.ylabel("Amplitude")
plt.grid(True)

```

The effect of the channel is simulated by convolving the modulated signal with the channel's impulse response. The resulting received signal is plotted for both sinusoidal pulse modulation and SRRC pulse modulation.

For sinusoidal pulses, the modulated signal $x(t)$ is convolved with the channel impulse response $h[n]$, and the received signal is plotted as follows:

```
from scipy.signal import convolve
T = 1
Fs = 1000
```

```
samples_per_symbol = int(T * Fs)

t = np.linspace(0, T, samples_per_symbol, endpoint=False)
g1_t_b1 = np.sin(np.pi * t / T)
g1_t_b0 = -np.sin(np.pi * t / T)
num_bits = 10
bits = [1,1,0,0,0,1,1,0,0,1]
modulated_sin = np.concatenate([g1_t_b1 if bit == 1 else g1_t_b0 for bit in bits])
h = np.zeros(4 * samples_per_symbol)
h[0] = 1.0
h[samples_per_symbol] = 1/2
h[2 * samples_per_symbol] = 3/4
h[3 * samples_per_symbol] = -2/7

received_sin = np.convolve(modulated_sin, h, mode='full')
t_received_sin = np.linspace(0, T * len(bits) + (len(h) - 1) / Fs, len(received_sin))
t_modulated = np.linspace(0, T*len(bits), len(modulated_sin))

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(t_modulated,modulated_sin)
plt.title("Modulated Signal (Sinusoidal Pulse)")
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(t_received_sin, received_sin)
plt.title("Received Signal (Sinusoidal Pulse)")
plt.grid(True)

plt.tight_layout()
plt.show()
```

For SRRC pulses, the modulated signal is convolved with the same channel impulse response, and the received signal is plotted as follows:

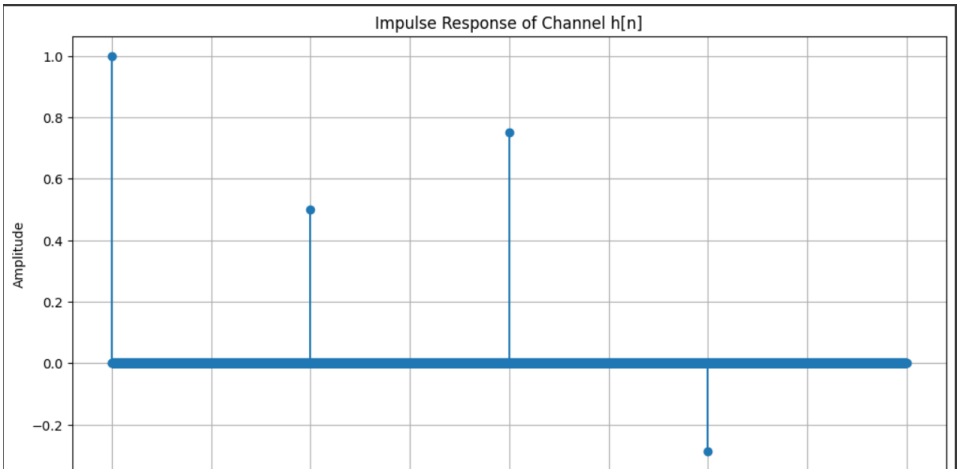
```
K = 6
b = 0.5
t = np.linspace(-K*T, K*T, Fs * 2 * K)

def srrc_pulse(t, T, b):
    numerator = np.sin(np.pi * t / T * (1 - b)) + 4 * b * t / T * np.cos(np.pi * t / T * (1 + b)
    denominator = np.pi * t / T * (1 - (4 * b * t / T) ** 2)

    x_t = np.zeros_like(t)
    mask_zero = (t == 0)
    mask_special = np.abs(t) == T / (4 * b)

    x_t[mask_zero] = (1 - b + 4 * b / np.pi)
    x_t[mask_special] = b / np.sqrt(2) * ((1 + 2 / np.pi) * np.sin(np.pi / (4 * b)) + (1 - 2 / n
    x_t[~(mask_zero | mask_special)] = numerator[~(mask_zero | mask_special)] / denominator[~(ma

    return x_t

```

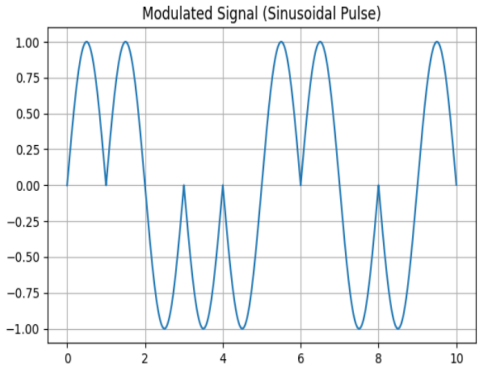
```

g1_t_b1 = srrc_pulse(t, T, b)
g1_t_b0 = -srrc_pulse(t, T, b)
num_bits = 10
bits = [1, 1, 0, 0, 0, 1, 1, 0, 0, 1]


modulated_srrc = np.concatenate([g1_t_b1 if bit == 1 else g1_t_b0 for bit in bits])
received_srrc = np.convolve(modulated_srrc, h, mode='full')


t_modulated_srrc = np.linspace(0, T * len(bits), len(modulated_srrc))
t_received_srrc = np.linspace(0, T * len(bits) + (len(h) - 1) / Fs, len(received_srrc))


plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(t_modulated_srrc, modulated_srrc)
plt.title("Modulated Signal (SRRC Pulse)")
plt.grid(True)


plt.subplot(1, 2, 2)
plt.plot(t_received_srrc, received_srrc)
plt.title("Received Signal (SRRC Pulse)")
plt.grid(True)

plt.tight_layout()
plt.show()
```

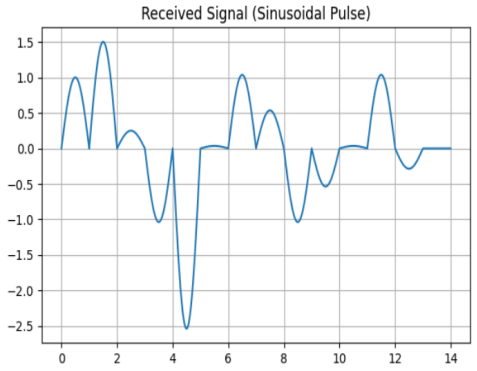
Finally, the eye diagram is plotted for both the received sinusoidal pulse signal and the received SRRC pulse signal.

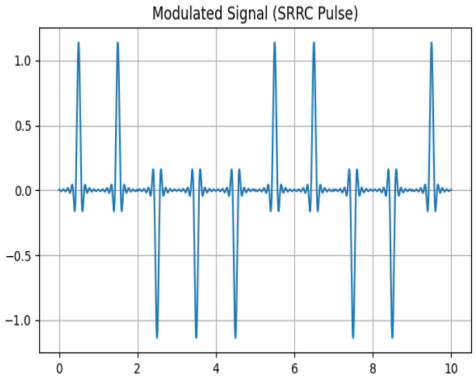
```
plot_eye_diagram(received_sin, samples_per_symbol, num_symbols=10)
plot_eye_diagram(received_srrc, samples_per_symbol, num_symbols=10)
```

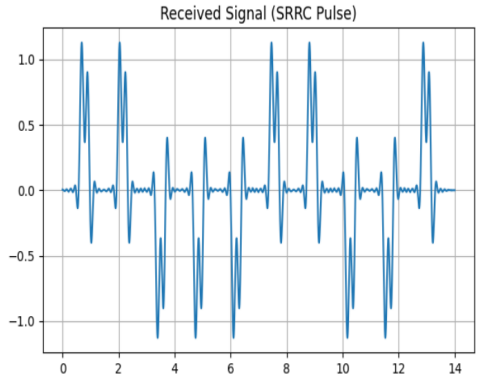
5.1 Q5


Impulse Response of Channel h[n]
Amplitude
1.0
0.8
0.6
0.4
0.2
0.0
−0.2


Modulated Signal (Sinusoidal Pulse)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10

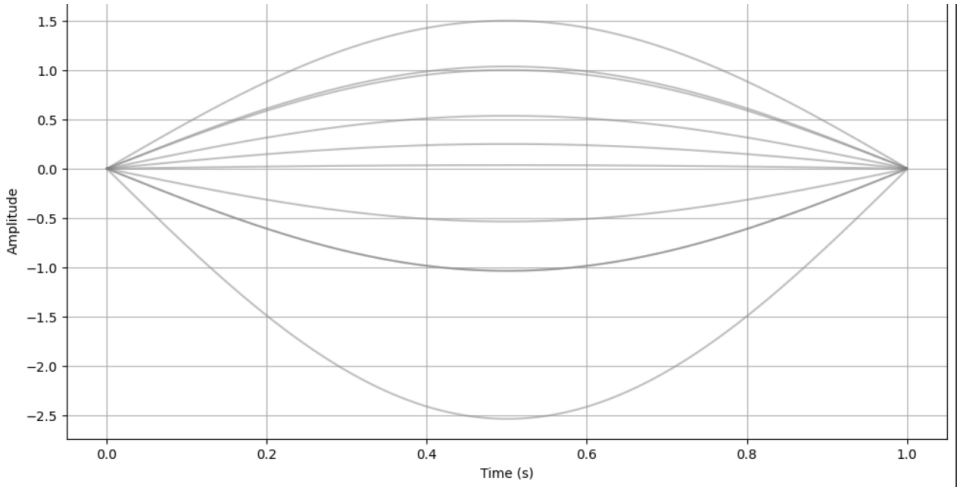

Received Signal (Sinusoidal Pulse)
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0
2
4
6
8
10
12
14

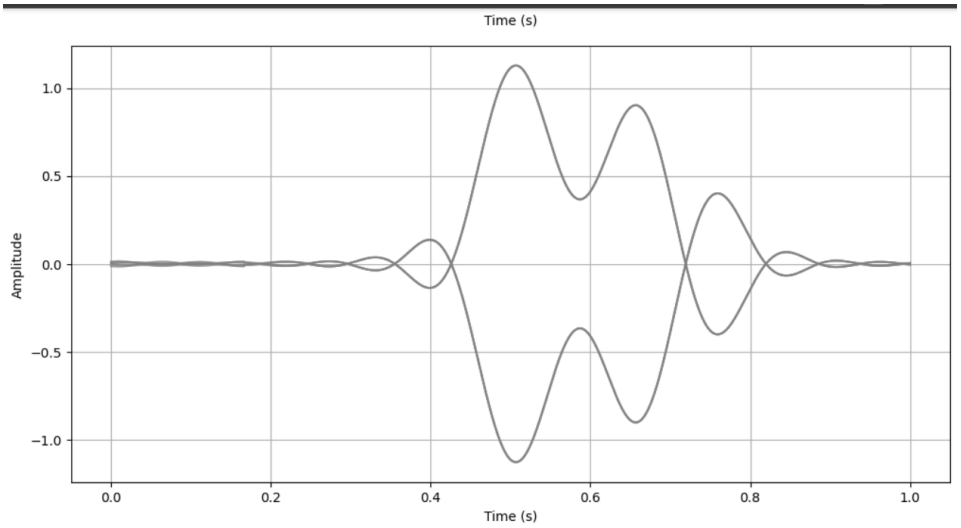

Modulated Signal (SRRC Pulse)
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10


Received Signal (SRRC Pulse)
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10
12
14

5.2 Q6

The channel effect causes distortion in the transmitted signal, resulting in a reduced eye opening in the eye diagram. This distortion arises from factors like inter-symbol interference (ISI) and noise introduced by the channel. As ISI increases, the eye becomes more closed, indicating that the symbols are less distinguishable. A wider eye suggests less ISI and better signal clarity. Therefore, a well-designed channel with minimal distortion leads to a more open eye, indicating improved signal integrity and easier detection.





Noise Effect

In any communication system, noise is an unavoidable factor that affects signal integrity. Noise is typically introduced due to various sources such as thermal noise, interference, and quantization errors. In this section, we model the noise as Additive White Gaussian Noise (AWGN), which is commonly used in digital communication analysis. AWGN is characterized by a normal distribution with zero mean and a certain variance, which determines the noise power.

Noise impacts signal transmission by introducing random variations in the received signal, leading to potential bit errors. A higher noise level results in increased distortion and degradation of signal quality. This can be observed in eye diagrams, where noise reduces the openness of the eye, making symbol detection more challenging. The following code simulates the effect of Gaussian noise on the transmitted signal.

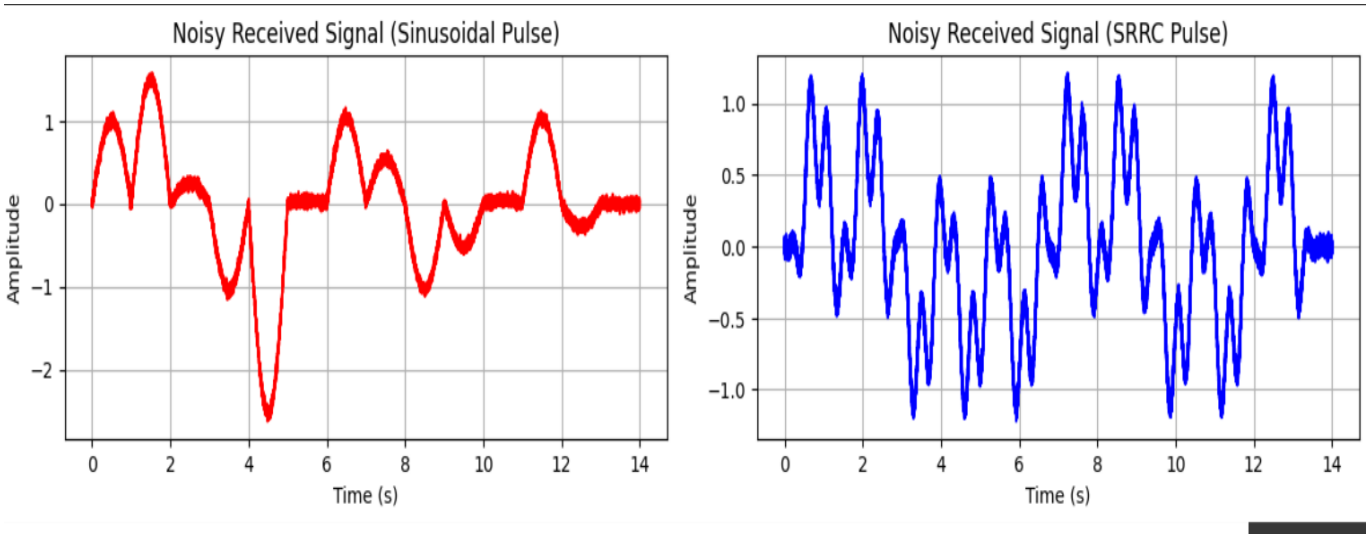
```
t = np.linspace(0, T, samples_per_symbol, endpoint=False)

g1_t_b1 = np.sin(np.pi * t / T)
g1_t_b0 = -np.sin(np.pi * t / T)

def srrc_pulse(alpha, span, sps):
    t = np.arange(-span * sps / 2, span * sps / 2) / sps
    pi_t = np.pi * t
    with np.errstate(divide='ignore', invalid='ignore'):
        pulse = (np.sin(pi_t * (1 - alpha)) +
                 4 * alpha * t * np.cos(pi_t * (1 + alpha))) / \
                (pi_t * (1 - (4 * alpha * t)**2))
    pulse[t == 0] = 1 - alpha + (4 * alpha / np.pi)
    pulse[np.abs(4 * alpha * t) == 1] = (alpha / np.sqrt(2)) * ((1 + 2/np.pi) *
                                                                np.sin(np.pi / (4 * alpha)) +
                                                                (1 - 2/np.pi) * np.cos(np.pi /
    return pulse

b = 0.5
K = 6
g2_t = srrc_pulse(b, K, samples_per_symbol)

bits = [1,1,0,0,0,1,1,0,0,1]
modulated_sin = np.concatenate([g1_t_b1 if bit == 1 else g1_t_b0 for bit in bits])
modulated_srrc = np.concatenate([g2_t if bit == 1 else -g2_t for bit in bits])
received_sin = np.convolve(modulated_sin, h, mode='full')
received_srrc = np.convolve(modulated_srrc, h, mode='full')

noise_level=0.05
power_sin = np.mean(received_sin ** 2)
power_srrc = np.mean(received_srrc ** 2)
sigma_sin = np.sqrt(power_sin) * noise_level
sigma_srrc = np.sqrt(power_srrc) * noise_level
received_sin_noisy = received_sin + sigma_sin * np.random.randn(len(received_sin))
received_srrc_noisy = received_srrc + sigma_srrc * np.random.randn(len(received_srrc))
t_received_sin = np.linspace(0, T * len(bits) + (len(h) - 1) / Fs, len(received_sin))
t_received_srrc = np.linspace(0, T * len(bits) + (len(h) - 1) / Fs, len(received_srrc))
t_modulated = np.linspace(0, T*len(bits), len(modulated_sin))

plt.figure(figsize=(12, 6))
```

```
plt.subplot(2, 2, 1)
plt.plot(t_received_sin, received_sin_noisy, color='r')
plt.title("Noisy Received Signal (Sinusoidal Pulse)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid(True)

plt.subplot(2, 2, 2)
plt.plot(t_received_srrc, received_srrc_noisy, color='b')
plt.title("Noisy Received Signal (SRRC Pulse)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid(True)

plt.tight_layout()
plt.show()

plot_eye_diagram(received_sin_noisy, samples_per_symbol, num_symbols=10)
plot_eye_diagram(received_srrc_noisy, samples_per_symbol, num_symbols=10)
```

The above code models the effect of noise on the received signal. First, we generate the sinusoidal and square root raised cosine (SRRC) pulses. The signals are then transmitted through the modeled channel, resulting in received signals with inter-symbol interference (ISI).

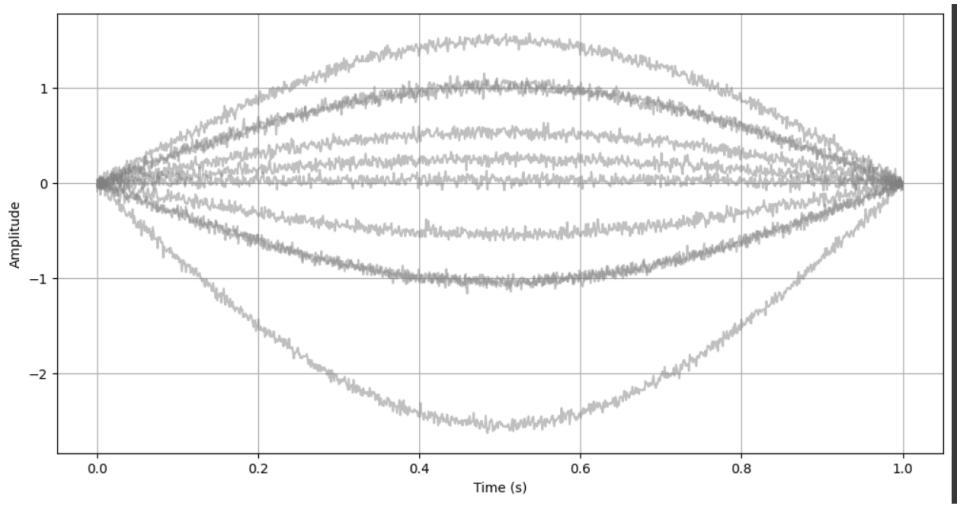
To introduce noise, we calculate the signal power and determine the noise variance using a specified noise level. Gaussian noise is then added to both received signals. The resulting noisy signals are visualized in the time domain.

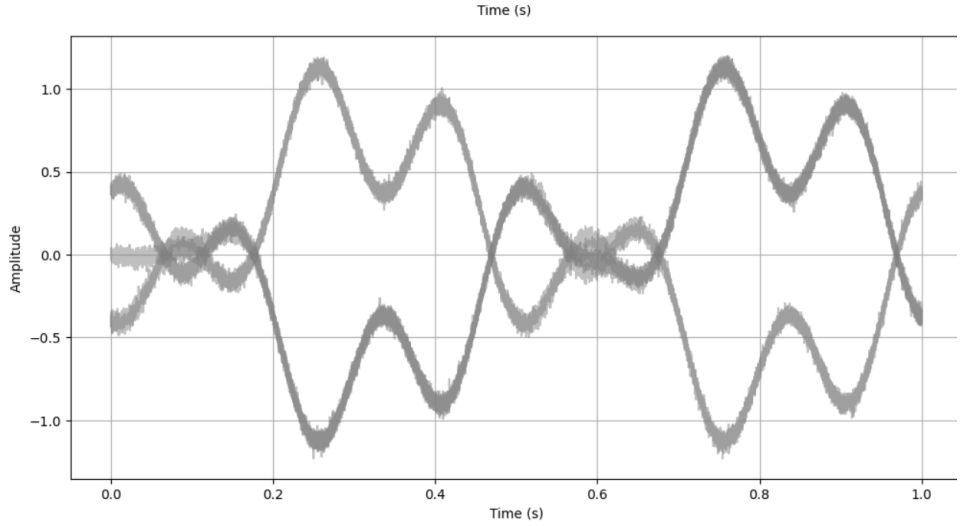
Finally, the eye diagrams of the noisy signals are plotted.



6.1 Q7

The addition of noise causes the eye diagram to close, indicating an increase in ISI and making symbol detection more challenging. The impact of noise is more significant for signals with higher ISI, leading to greater distortion. Finally, the eye diagrams of the noisy signals are plotted.





Matched Filter

Matched filtering is an essential technique in digital communication systems used to maximize the signal-to-noise ratio (SNR) at the receiver. A matched filter is designed to be the time-reversed and delayed version of the transmitted pulse, ensuring optimal detection. The main goal of the matched filter is to enhance the received signal while mitigating noise and inter-symbol interference (ISI).

Mathematically, the impulse response of the matched filter is given by:

$$g_R(t) = g_T(T - t) \tag{1}$$

where $g_T(t)$ is the transmitted pulse and $g_R(t)$ is the matched filter response. The convolution of the received signal with this filter maximizes SNR and improves symbol detection. Below, we implement matched filters for both sinusoidal and SRRC pulses.

7.1 Implementation for Sinusoidal Matched Filter

```
matched_filter1 = np.sin(np.pi * (T-t) / T)

plt.figure(figsize=(10, 4))
plt.plot(t, matched_filter1)
plt.title('Matched Filter for Sinusoidal Pulse')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.show()

t = np.linspace(0, T, samples_per_symbol, endpoint=False)
sin_matchedfilter_out = convolve(received_sin, matched_filter1, mode='same')

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)
plt.plot(t_modulated, modulated_sin)
plt.title("Modulated Signal (Sinusoidal Pulse)")
plt.grid(True)

plt.subplot(1, 3, 2)
plt.plot(t_received_sin, received_sin)
plt.title("Received Signal After Channel")
plt.grid(True)

plt.subplot(1, 3, 3)
plt.plot(t_received_sin, sin_matchedfilter_out)
plt.title("Matched Filtered Signal")
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

plot_eye_diagram(sin_matchedfilter_out, samples_per_symbol, num_symbols=10)
```

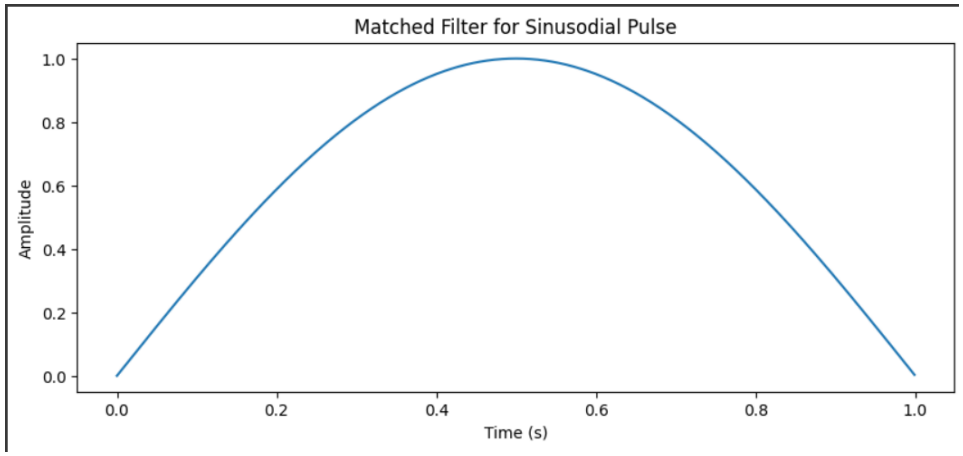
The above code implements a matched filter for the sinusoidal pulse shape. The matched filter is obtained by time-reversing the transmitted pulse. We then convolve the received signal with the matched filter, which enhances signal detection. The eye diagram of the filtered signal is also plotted. The matched filter reduces noise and improves symbol separation, increasing the clarity of the eye diagram.

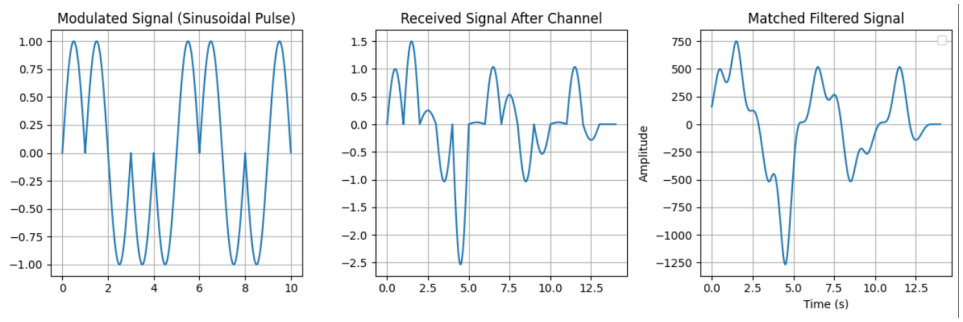
7.2 Implementation for SRRC Matched Filter

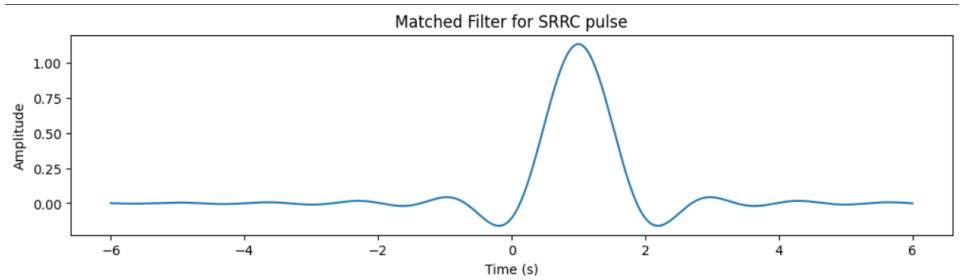
```
t = np.linspace(-K*T, K*T, Fs * 2 * K)
def Matched_Filter2(t, T, b):
    numerator = np.sin(np.pi * (T-t) / T * (1 - b)) + 4 * b * (T-t) / T * np.cos(np.pi * (T-t) /
    denominator = np.pi * (T-t) / T * (1 - (4 * b * (T-t) / T) ** 2)
    x_t = np.zeros_like(t)
    mask_zero = (t == 0)
    mask_special = np.abs(t) == T / (4 * b)
    x_t[mask_zero] = (1 - b + 4 * b / np.pi)
    x_t[mask_special] = b / np.sqrt(2) * ((1 + 2 / np.pi) * np.sin(np.pi / (4 * b)) + (1 - 2 / n
    x_t[~(mask_zero | mask_special)] = numerator[~(mask_zero | mask_special)] / denominator[~(ma
    return x_t

matched_filter2 = Matched_Filter2(t, T, b)

plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(t, matched_filter2)
plt.title('Matched Filter for SRRC pulse')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.show()

t = np.linspace(-K*T, K*T, Fs * 2 * K)
srrc_matched_filtered_out = convolve(received_srrc, matched_filter1, mode='same')

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 2)
plt.plot(t_received_srrc, received_srrc)
plt.title("Received Signal After Channel")
plt.grid(True)

plt.subplot(1, 3, 3)
plt.plot(t_received_srrc, srrc_matched_filtered_out)
plt.title("Matched Filtered Signal")
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

plot_eye_diagram(srrc_matched_filtered_out, samples_per_symbol, num_symbols=10)
```

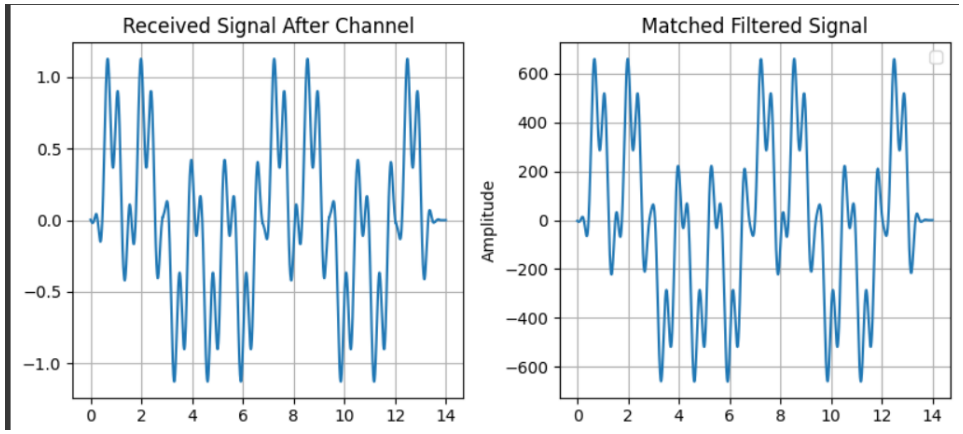
The matched filter for the Square Root Raised Cosine (SRRC) pulse is derived using the same principle. Since SRRC pulses are commonly used in digital communication due to their ability to control bandwidth and ISI, applying a matched filter significantly improves signal detection.

7.3 Q8



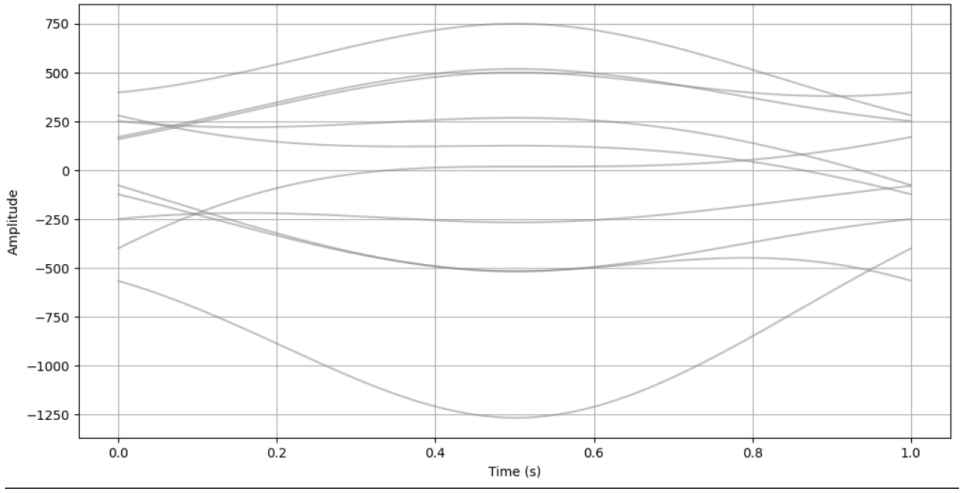


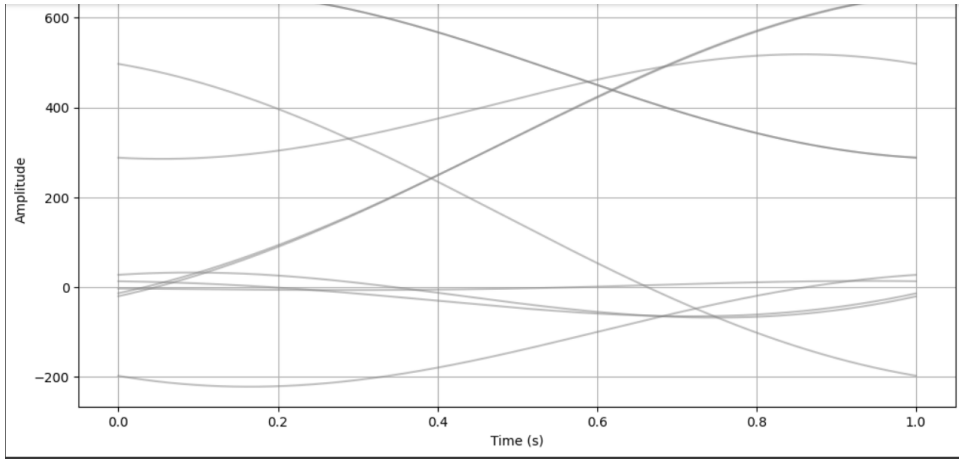




7.4 Q9

The matched filter enhances the eye diagram by improving eye openness, reducing noise and ISI, leading to better symbol detection. The best sampling time is found at the widest vertical opening of the eye diagram, where the signal is least distorted. At this point, the SNR is maximized, and the probability of symbol errors is minimized. The optimal sampling instant ensures low timing jitter and reliable symbol detection. Proper synchronization at the eye center reduces the bit error rate (BER). Thus, analyzing the eye diagram helps achieve accurate timing and improved communication performance.





Equalizer

In communication systems, an equalizer is used to mitigate the effects of intersymbol interference (ISI) caused by the channel. The objective is to recover the transmitted signal as accurately as possible by applying a filter that compensates for the channel distortion. Two common equalization techniques are the Zero-Forcing (ZF) Equalizer and the Minimum Mean Square Error (MMSE) Equalizer.

8.1 Zero-Forcing (ZF) Equalizer

The Zero-Forcing Equalizer attempts to completely eliminate ISI by applying an inverse filter to the channel. The transfer function of a ZF equalizer is given by:

$$Q_{ZF}(e^{j\omega}) = \frac{1}{H(e^{j\omega})} \tag{2}$$

where $H(e^{j\omega})$ is the channel frequency response. The ZF equalizer works well in the absence of noise but can significantly amplify noise in frequency regions where $H(e^{j\omega})$ has small values.

```
from scipy.signal import convolve
from scipy.fft import fft, ifft

H_f = fft(h, len(received_sin))
Q_zf_f = 1 / H_f
Q_zf_f[np.isnan(Q_zf_f)] = 0
received_sin_f = fft(received_sin)
equalized_sin_f = received_sin_f * Q_zf_f
equalized_sin = ifft(equalized_sin_f)

plt.figure(figsize=(12, 6))
plt.subplot(2, 2, 1)
plt.plot(np.arange(len(Q_zf_f)), np.abs(Q_zf_f))
plt.title("Impulse Response of Zero-Forcing Equalizer")
plt.grid(True)

plt.subplot(2, 2, 2)
plt.plot(np.linspace(0, Fs, len(Q_zf_f)), np.abs(Q_zf_f))
plt.title("Frequency Response of Zero-Forcing Equalizer")
plt.grid(True)

plt.subplot(2, 2, 3)
plt.plot(t_received_sin, received_sin)
plt.title("Received Signal Before Equalizer")
plt.grid(True)

plt.subplot(2, 2, 4)
plt.plot(t_received_sin, np.real(equalized_sin))
plt.title("Equalized Signal (Zero-Forcing)")
plt.grid(True)

plt.tight_layout()
plt.show()

plot_eye_diagram(np.real(equalized_sin), samples_per_symbol, num_symbols=10)
```

- The Fourier transform of the channel impulse response h is computed to obtain $H(f)$.
- The inverse filter $Q_{ZF}(f) = 1/H(f)$ is applied, ensuring no division by zero.
- The received signal in frequency domain is multiplied by the inverse filter.
- The inverse Fourier transform is taken to obtain the equalized signal.
- The eye diagram is plotted to analyze the impact of the equalizer.

Zero-Forcing Equalizer with Noise:

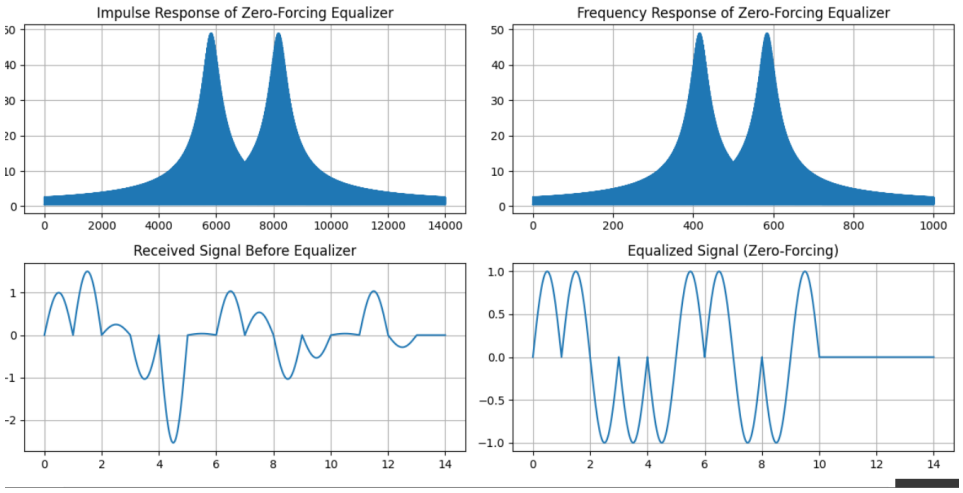
```
H = np.fft.fft(h, len(received_sin_noisy))
H_inv = 1 / H
received_sin_zfe = np.fft.ifft(np.fft.fft(received_sin_noisy) * H_inv)
t_received_sin = np.linspace(0, T * len(bits) + (len(h) - 1) / Fs, len(received_sin))

plt.figure(figsize=(12, 6))
plt.subplot(2, 2, 1)
plt.plot(t_received_sin, received_sin_noisy, color='r')
plt.title("Noisy Received Signal (Sinusoidal Pulse)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid(True)

plt.subplot(2, 2, 2)
plt.plot(t_received_sin, received_sin_zfe, label='Zero-Forcing Equalized')
plt.title("Zero-Forcing Equalized Signal (Sinusoidal Pulse)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend()

plot_eye_diagram(received_sin_zfe, samples_per_symbol, num_symbols=10)
```

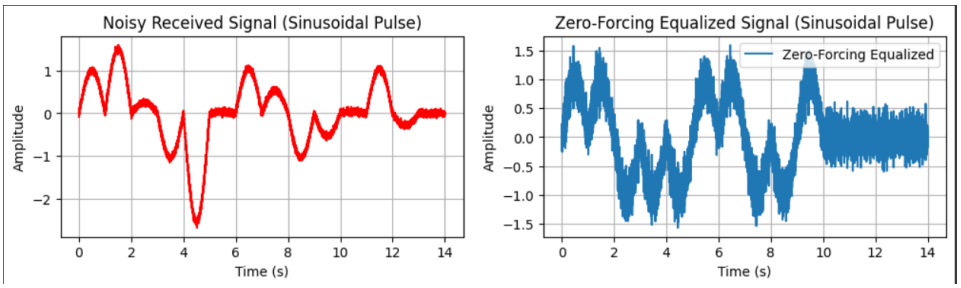
8.2 Q10

The frequency response of a Zero-Forcing (ZF) equalizer is given by:

$$Q_{ZF}(e^{j\omega}) = \frac{1}{H(e^{j\omega})} \tag{3}$$

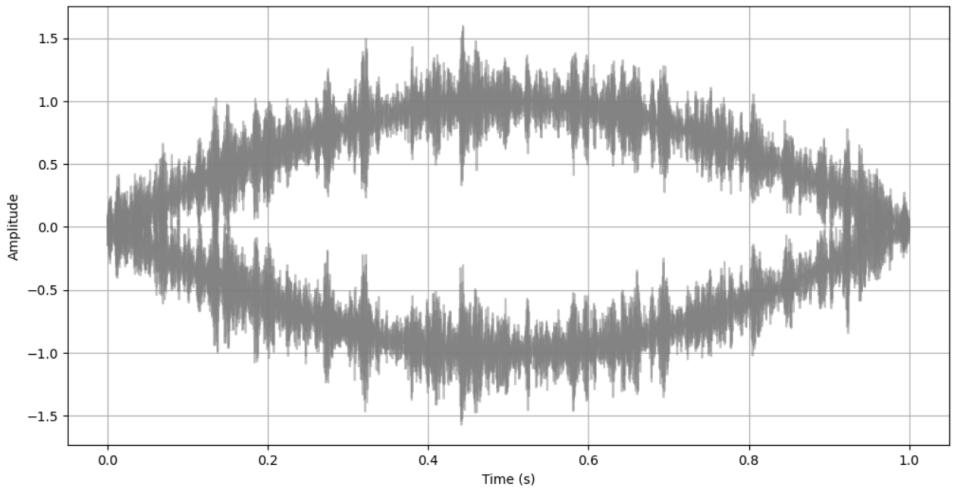
This equalizer completely inverts the channel response to remove intersymbol interference (ISI). However, if the channel has deep nulls (small values of $H(e^{j\omega})$), the ZF filter amplifies noise significantly, leading to signal distortion. The stability of the ZF equalizer depends on the channel response; if $H(e^{j\omega})$ has zero values, the inverse filter becomes unstable. Therefore, we cannot always expect the inverse filter to be stable.





8.3 Q11

The eye diagram of a ZF-equalized signal shows reduced ISI, making the eye more open compared to the received signal before equalization. However, since ZF does not consider noise, it amplifies noise in frequency regions where $H(e^{j\omega})$ is small. This results in increased jitter and noise in the eye diagram, making symbol detection less reliable. Consequently, ZF is not always practical in noisy environments.



8.4 Minimum Mean Square Error (MMSE) Equalizer

The MMSE equalizer balances ISI mitigation and noise suppression by minimizing the mean square error between the transmitted and equalized signals. The transfer function of MMSE is:

$$Q_{MMSE}(e^{j\omega}) = \frac{H^*(e^{j\omega})}{|H(e^{j\omega})|^2 + 2\sigma^2} \tag{4}$$

```
from scipy.signal import convolve
from scipy.fft import fft, ifft

H_f = fft(h, len(received_sin))
sigma2 = 0.1

Q_mmse_f = np.conj(H_f) / (np.abs(H_f)**2 + 2 * sigma2)
received_sin_f = fft(received_sin)
equalized_sin_f = received_sin_f * Q_mmse_f
equalized_sin = ifft(equalized_sin_f)

plt.figure(figsize=(12, 6))
plt.subplot(2, 2, 1)
plt.plot(np.arange(len(Q_mmse_f)), np.abs(Q_mmse_f))
plt.title("Impulse Response of MMSE Equalizer")
plt.grid(True)

plt.subplot(2, 2, 2)
plt.plot(np.linspace(0, Fs, len(Q_mmse_f)), np.abs(Q_mmse_f))
plt.title("Frequency Response of MMSE Equalizer")
plt.grid(True)

plt.subplot(2, 2, 3)
plt.plot(t_received_sin, received_sin)
plt.title("Received Signal Before Equalizer")
plt.grid(True)

plt.subplot(2, 2, 4)
plt.plot(t_received_sin, np.real(equalized_sin))
plt.title("Equalized Signal (MMSE)")
plt.grid(True)
plt.tight_layout()
plt.show()

plot_eye_diagram(np.real(equalized_sin), samples_per_symbol, num_symbols=10)
```

MMSE Equalizer with Noise :

```
H = np.fft.fft(h, len(received_sin_noisy))
H_star = np.conj(H)
sigma = 0.1
H_mse = H_star / (np.abs(H) ** 2 + 2 * sigma ** 2)
received_sin_mmse = np.fft.ifft(np.fft.fft(received_sin_noisy) * H_mse)

plt.figure(figsize=(12, 6))
plt.subplot(2, 2, 1)
```

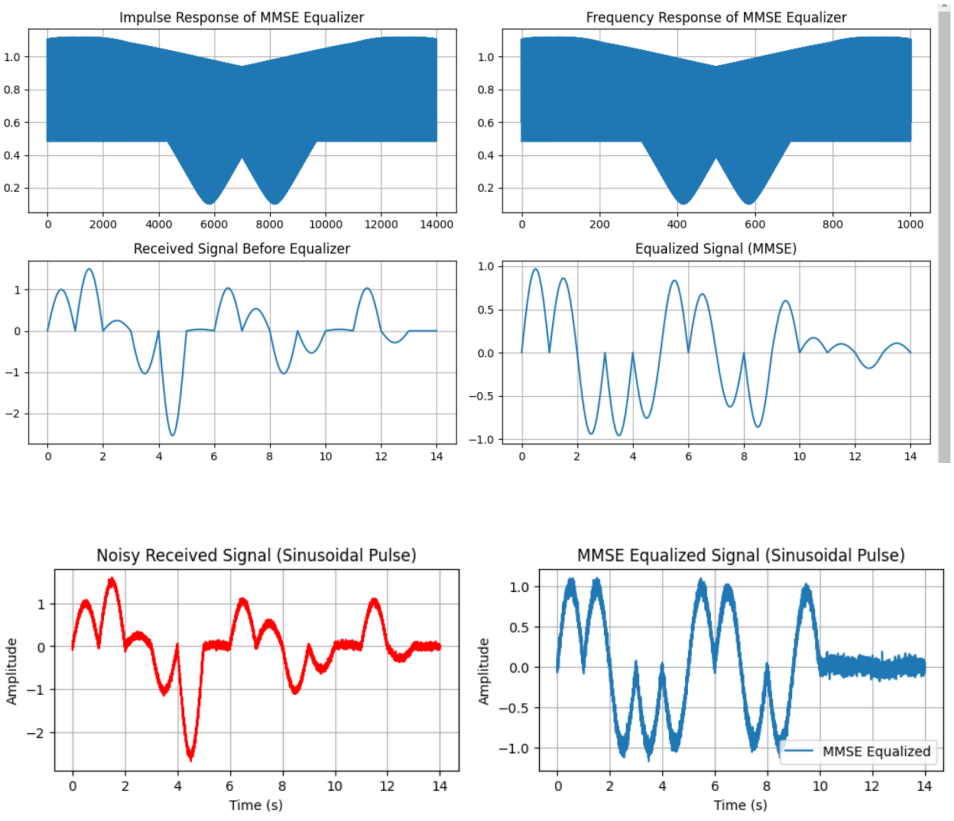
```
plt.plot(t_received_sin, received_sin_noisy, color='r')
plt.title("Noisy Received Signal (Sinusoidal Pulse)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid(True)

plt.subplot(2, 2, 2)
plt.plot(t_received_sin, received_sin_mmse, label='MMSE Equalized')
plt.title("MMSE Equalized Signal (Sinusoidal Pulse)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend()

plot_eye_diagram(received_sin_mmse, samples_per_symbol, num_symbols=10)
```

8.5 Q12

The MMSE equalizer has a frequency response given by:

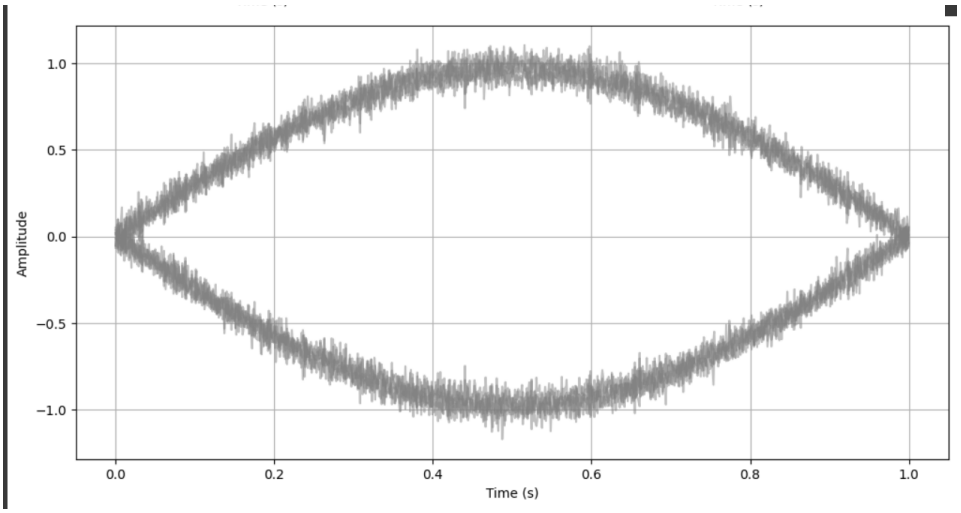
$$Q_{MMSE}(e^{j\omega}) = \frac{H^*(e^{j\omega})}{|H(e^{j\omega})|^2 + 2\sigma^2} \tag{5}$$

Unlike ZF, which completely inverts $H(e^{j\omega})$, MMSE considers both ISI and noise. It balances ISI removal and noise suppression by incorporating the noise variance σ^2. This makes MMSE more robust in noisy environments. While ZF completely eliminates ISI, it amplifies noise significantly, whereas MMSE provides a compromise, reducing ISI while keeping noise levels manageable. Thus, MMSE generally achieves a better signal-to-noise ratio (SNR) compared to ZF.



8.6 Q13

The eye diagram of an MMSE-equalized signal shows a balance between ISI suppression and noise mitigation. Compared to ZF, MMSE results in a cleaner eye diagram with reduced noise amplification, making the decision points more distinguishable. This makes MMSE more redundant (robust) against noise, leading to better symbol detection. The improved eye openness indicates more reliable signal recovery, making MMSE preferable in practical communication systems.



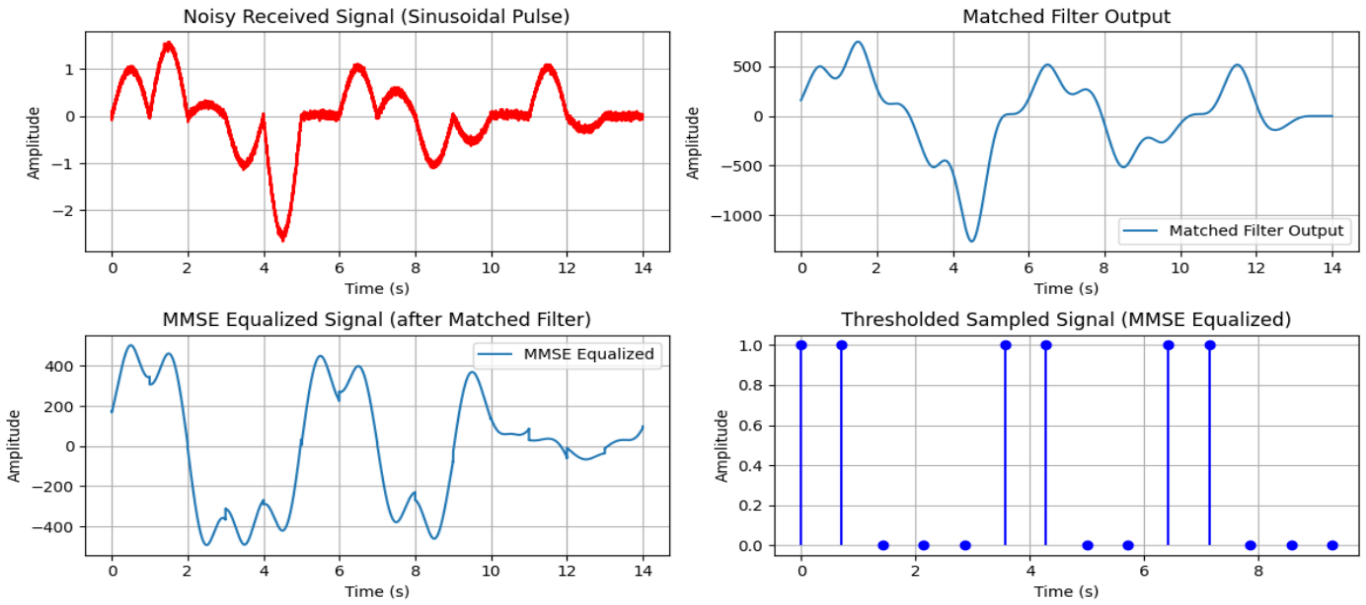
Sampling and Bit Detection

Sampling is essential in digital communication for extracting transmitted bits from the received signal. After applying a matched filter and equalization, we sample the signal at optimal time instances to recover the transmitted symbols.

- The first bit is determined by analyzing the received signal and selecting an optimal sampling point where the eye is maximally open.
- After locating the first bit, the entire signal is sampled at uniform intervals corresponding to the symbol period.
- This minimizes intersymbol interference (ISI) and maximizes detection accuracy.

```
received_sin_matched = np.convolve(received_sin_noisy, matched_filter1, mode='same')
H = np.fft.fft(h, len(received_sin_matched))
H_star = np.conj(H)
sigma2 = 0.1
H_mmse = H_star / (np.abs(H) ** 2 + sigma2)
received_sin_mmse = np.fft.ifft(np.fft.fft(received_sin_matched) * H_mmse)


sampled_signal = received_sin_mmse[samples_per_symbol//2::samples_per_symbol]

max_amp = np.max(np.abs(sampled_signal))
sampled_signal_normalized = sampled_signal / max_amp

threshold = 0.1
sampled_signal_thresholded = np.where(sampled_signal_normalized > threshold, 1, 0)
```

```
16
17  t_sampled = np.linspace(0, T * len(bits), len(sampled_signal), endpoint=False)
18  plt.figure(figsize=(12, 6))
19
20
21  plt.subplot(2, 2, 1)
22  plt.plot(np.linspace(0, T * len(bits) + (len(h) - 1) / Fs, len(received_sin)), received_sin_nois
23  plt.title("Noisy Received Signal (Sinusoidal Pulse)")
24  plt.xlabel("Time (s)")
25  plt.ylabel("Amplitude")
26  plt.grid(True)
27
28  plt.subplot(2, 2, 2)
29  plt.plot(np.linspace(0, T * len(bits) + (len(h) - 1) / Fs, len(received_sin_matched)), received_
30  plt.title("Matched Filter Output")
31  plt.xlabel("Time (s)")
32  plt.ylabel("Amplitude")
33  plt.grid(True)
34  plt.legend()
35
36  plt.subplot(2, 2, 3)
37  plt.plot(np.linspace(0, T * len(bits) + (len(h) - 1) / Fs, len(received_sin_mmse)), received_sin
38  plt.title("MMSE Equalized Signal (after Matched Filter)")
39  plt.xlabel("Time (s)")
40  plt.ylabel("Amplitude")
41  plt.grid(True)
42  plt.legend()
43
44
45  plt.subplot(2, 2, 4)
46  plt.stem(t_sampled, sampled_signal_thresholded, 'b', markerfmt='bo', basefmt=" ")
47  plt.title("Thresholded Sampled Signal (MMSE Equalized)")
48  plt.xlabel("Time (s)")
49  plt.ylabel("Amplitude")
50  plt.grid(True)
51  plt.tight_layout( )
52  print(bits)
53
```



The received signal, matched filter output, and equalized signal are visualized to analyze the improvement. The final detected bits closely match the original transmitted bits, confirming the success of the process. The combination of matched filtering, MMSE equalization, and optimal sampling effectively mitigates noise and ISI, ensuring accurate symbol recovery.

Modular Implementation of a Digital Communication System

Now that every element in our communication system model works fine we can define each of them as a function and define and model all of them as communication system function.The code below shows implementations of this :

```
import numpy as np
import cv2
import pyfftw
from scipy.signal import fftconvolve
from scipy.fftpack import dct, idct
from skimage.util import view_as_blocks
import matplotlib.pyplot as plt
import gc
def pulse_generator(bits, pulse1, pulse0):
    return np.concatenate(np.where(bits[:, None] == 1, pulse1, pulse0))

def channel_effect(signal, h):
    return fftconvolve(signal, h, mode='full')

def add_noise(signal, noise_level=0.05):
    power = np.mean(signal ** 2)
    sigma = np.sqrt(power) * noise_level
    return signal + sigma * np.random.randn(len(signal))

def matched_filter(signal_noisy, matched_filter):
    return fftconvolve(signal_noisy, matched_filter[::-1], mode='same')

def mmse_equalizer(signal, h, noise_variance=0.1):
```

```
    H = pyfftw.interfaces.numpy_fft.fft(h, len(signal))
    H_mmse = np.conj(H) / (np.abs(H)**2 + noise_variance)
    signal_fft = pyfftw.interfaces.numpy_fft.fft(signal)
    return np.real(pyfftw.interfaces.numpy_fft.ifft(signal_fft * H_mmse))

def sampling(signal, samples_per_symbol, num_bits, threshold=0.1):
    sampled = signal[samples_per_symbol // 2::samples_per_symbol][:num_bits]
    return (sampled > threshold).astype(np.uint8)

def full_system(bitstream, pulse1, pulse0, h,
                noise_level=0.05, noise_variance=0.1,
                samples_per_symbol=100):
    modulated = pulse_generator(bitstream, pulse1, pulse0)
    received = channel_effect(modulated, h)
    received_noisy = add_noise(received, noise_level)
    filtered = matched_filter(received_noisy, pulse1)
    equalized = mmse_equalizer(filtered, h, noise_variance)
    return sampling(equalized, samples_per_symbol, len(bitstream))
```

We will check performance of this system by simple 10 random bits.Here is the code:

```
bits = np.array([1, 1, 1, 0, 0, 1, 1, 0, 0, 1])

T = 1
t = np.linspace(0, T, 1000, endpoint=False)
modulation_pulse = np.sin(np.pi * t / T)
demodulation_pulse = -np.sin(np.pi * t / T)
samples_per_symbol = 1000
h = np.zeros(4 * samples_per_symbol)
h[0] = 1.0
h[samples_per_symbol] = 1/2
h[2 * samples_per_symbol] = 3/4
h[3 * samples_per_symbol] = -2/7


final_sampled_signal = full_system(bits, modulation_pulse, demodulation_pulse, h,
                                   noise_level=0.05, noise_variance=0.1,
                                   samples_per_symbol=samples_per_symbol)

print("Original Bits:       ", bits)
print("Detected Bits:       ", final_sampled_signal)
```

And the output and performance of the system was fine:
Original Bits: [1 1 1 0 0 1 1 0 0 1]
Detected Bits: [1 1 1 0 0 1 1 0 0 1]

Image Reconstruction and Post-Processing

The theory and process of this part is exactly inverse of pre-processing part.Here we define to functions. One implements image pre-processing and bitstream conversion and the other does the post-processing and re-construction.Here is the code:

```
def preprocess_image_dct_cv2(image_path, block_size=8):
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    img = img.astype(np.float64)
    m, n = img.shape
    blocks = view_as_blocks(img, block_shape=(block_size, block_size))
    num_blocks_vert, num_blocks_horiz = blocks.shape[:2]
    num_blocks = num_blocks_vert * num_blocks_horiz
    quantized_blocks = np.empty((num_blocks, block_size, block_size), dtype=np.uint8)
    block_mins = np.empty(num_blocks, dtype=np.float64)
    block_ranges = np.empty(num_blocks, dtype=np.float64)


    idx = 0
    for i in range(num_blocks_vert):
        for j in range(num_blocks_horiz):
            block = blocks[i, j]
            block_dct = dct(dct(block.T, norm='ortho').T, norm='ortho')
            bmin = block_dct.min()
            bmax = block_dct.max()
            rng = bmax - bmin if bmax != bmin else 1
            block_mins[idx] = bmin
            block_ranges[idx] = rng
            block_scaled = (block_dct - bmin) / rng
            quantized = np.round(block_scaled * 255).astype(np.uint8)
            quantized_blocks[idx] = quantized
            idx += 1
    dct_blocks_3d = quantized_blocks.transpose(1, 2, 0)

    info = {
        'orig_shape': (m, n),
        'image_shape': img.shape,
        'block_size': block_size,
        'num_blocks_vert': num_blocks_vert,
        'num_blocks_horiz': num_blocks_horiz,
        'num_blocks': num_blocks,
        'block_mins': block_mins,
        'block_ranges': block_ranges
    }
    return dct_blocks_3d, info


def reconstruct_image_from_dct_blocks(recovered_dct_blocks_3d, info):
    recovered_blocks = recovered_dct_blocks_3d.transpose(2, 0, 1)
    spatial_blocks = np.empty_like(recovered_blocks, dtype=np.float64)

    for idx in range(info['num_blocks']):
        norm_block = recovered_blocks[idx].astype(np.float64) / 255.0
```

```
        block_dct = norm_block * info['block_ranges'][idx] + info['block_mins'][idx]
        block_idct = idct(idct(block_dct.T, norm='ortho').T, norm='ortho')
        spatial_blocks[idx] = block_idct
    rec_img = np.zeros(info['image_shape'], dtype=np.float64)

    idx = 0
    for i in range(info['num_blocks_vert']):
        for j in range(info['num_blocks_horiz']):
            rec_img[i*info['block_size']:(i+1)*info['block_size'], j*info['block_size']:(j+1)*in
            idx += 1
    m, n = info['orig_shape']
    rec_img = rec_img[:m, :n]
    return np.clip(rec_img, 0, 255).astype(np.uint8)
```

Since this two functions are important and a little bit more complex we will explain it more.

Preprocessing: DCT and Quantization

Our goal is to convert the image into a compressed representation by transforming spatial data into the frequency domain.

- The image is divided into small blocks (e.g., 8×8) to take advantage of local correlations.
- Each block undergoes a 2D DCT, concentrating most energy in low-frequency coefficients.
- The DCT coefficients are linearly scaled to $[0, 255]$, quantized, and stored efficiently.

Reconstruction: Inverse DCT

We will restore the image from the quantized DCT blocks.

- Blocks are rescaled using stored minimum and range values.
- Each block undergoes inverse 2D DCT to reconstruct the spatial image.
- The processed blocks are stitched together to form the full image.

Why DCT?

- DCT concentrates most information into a few coefficients, reducing redundancy.
- Quantization reduces data size while retaining perceptual quality.
- The approach underlies common compression standards like JPEG.

These functions implement a fundamental method for lossy image compression and transmission while ensuring a balance between efficiency and quality.

Complete Digital Communication System Implementation

We need just one more function to connect all functions we defined before to each other.Here is the code:

```
def transmit_dct_groups(dct_blocks_3d, info, group_size,
                        pulse1, pulse0, h,
                        noise_level=0.05, noise_variance=0.1,
                        samples_per_symbol=100):
    num_blocks = info['num_blocks']
    block_size = info['block_size']
    blocks_flat = dct_blocks_3d.reshape(info['block_size'] * info['block_size'], info['num_block

    recovered_blocks = []
    for start in range(0, num_blocks, group_size):
        end = min(start + group_size, num_blocks)
        current_group = blocks_flat[start:end]
        group_vector = current_group.flatten()
        bitstream = np.unpackbits(group_vector)
        bit_matrix = bitstream.reshape(-1, 8)
        bitstream_1d = bit_matrix.flatten()


        received_bitstream = full_system(bitstream_1d, pulse1, pulse0, h,
                                         noise_level=noise_level,
                                         noise_variance=noise_variance,
                                         samples_per_symbol=samples_per_symbol)



        received_bit_matrix = received_bitstream.reshape(-1, 8)
        recovered_group_vector = np.packbits(received_bit_matrix, axis=1).flatten()
        recovered_group = recovered_group_vector.reshape(-1, block_size * block_size)
        recovered_group = recovered_group.reshape(-1, block_size, block_size)
        recovered_blocks.append(recovered_group)
        del current_group, group_vector, bitstream, bit_matrix, bitstream_1d, received_bitstream
        gc.collect()
    recovered_blocks = np.concatenate(recovered_blocks, axis=0)
    recovered_dct_blocks_3d = recovered_blocks.transpose(1, 2, 0)
    return recovered_dct_blocks_3d
```

- The input 3D array of DCT coefficients (`dct_blocks_3d`) is reshaped into a 2D matrix, where each row represents a block of size 8×8.
- This restructuring simplifies processing and transmission.
- The blocks are transmitted in groups of `group_size` to allow batch processing.
- Each group is flattened into a 1D array.
- The numerical values are converted into an 8-bit binary representation using `np.unpackbits`, generating a bitstream.
- The bitstream is transmitted through a channel simulated by the function `full_system`, which applies:

- Pulse modulation using predefined pulses (`pulse1`, `pulse0`).
- Channel effects modeled by impulse response `h`.
- Noise addition with controllable parameters (`noise_level` and `noise_variance`).
- The received bitstream is recovered by equalization and threshold-based sampling.
- The received bitstream is reshaped back into an 8-bit matrix.
- Binary values are converted back into numerical form using `np.packbits`.
- The recovered values are reshaped back into 8×8 blocks and reassembled into the full 3D DCT array.

Now by just using the code below we can simulate process of transmitting and image and reconstructing it in receiver.We will change parameters like noise level and h impulse response and modulation pulse to see their effect on received image.

```
dct_blocks_3d, info = preprocess_image_dct_cv2("cameraman.tif", block_size=8)
samples_per_symbol = 100
T = 1
t = np.linspace(0, T, samples_per_symbol, endpoint=False)
pulse1 = np.sin(np.pi * t / T)
pulse0 = -np.sin(np.pi * t / T)

h = np.zeros(4 * samples_per_symbol)
h[0] = 1.0
h[samples_per_symbol] = 1/2
h[2 * samples_per_symbol] = 3/4
h[3 * samples_per_symbol] = -2/7

group_size = 10
recovered_dct_blocks_3d = transmit_dct_groups(dct_blocks_3d, info, group_size,
                                              pulse1, pulse0, h,
                                              noise_level=1,
                                              noise_variance=0.1,
                                              samples_per_symbol=samples_per_symbol)
reconstructed_img = reconstruct_image_from_dct_blocks(recovered_dct_blocks_3d, info)
plt.figure(figsize=(6,6))
plt.imshow(reconstructed_img, cmap='gray', vmin=0, vmax=255)
plt.axis('off')
plt.title("Reconstructed Image")
plt.show()
```

12.1 Q14

By changing noise level parameter,we can adjust noise power.We will plot the output image for noise levels of 1,2,4 first for sin pulse modulation and then for SRRC modulation.Take notice that setting noise level to 2 means that noise will have power 4 times the original signal.

Figure 1: Sin pulse Modulated Output Image for Different Noise Levels

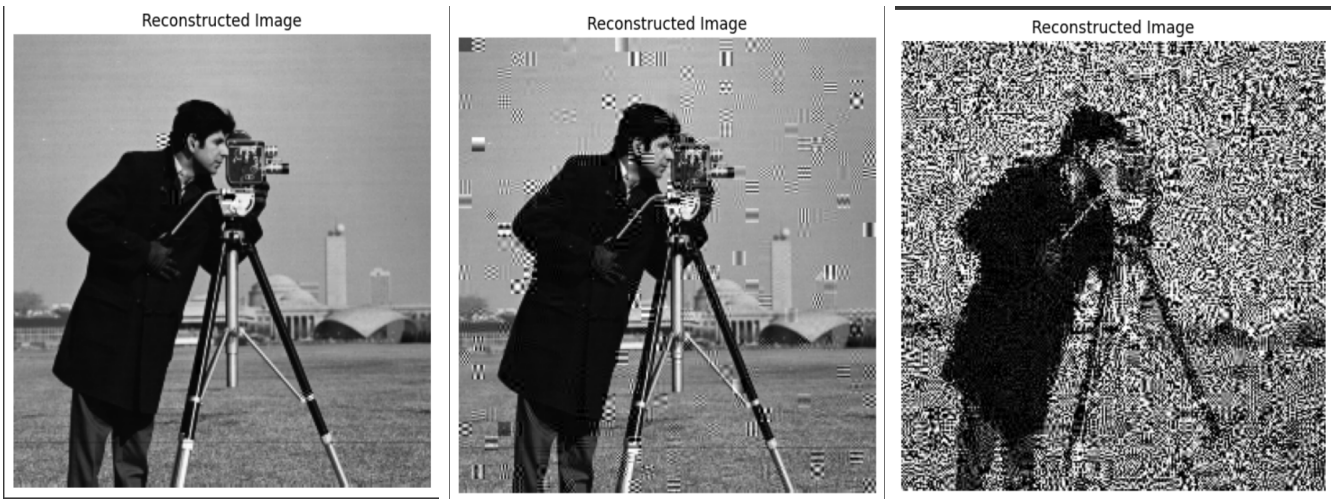
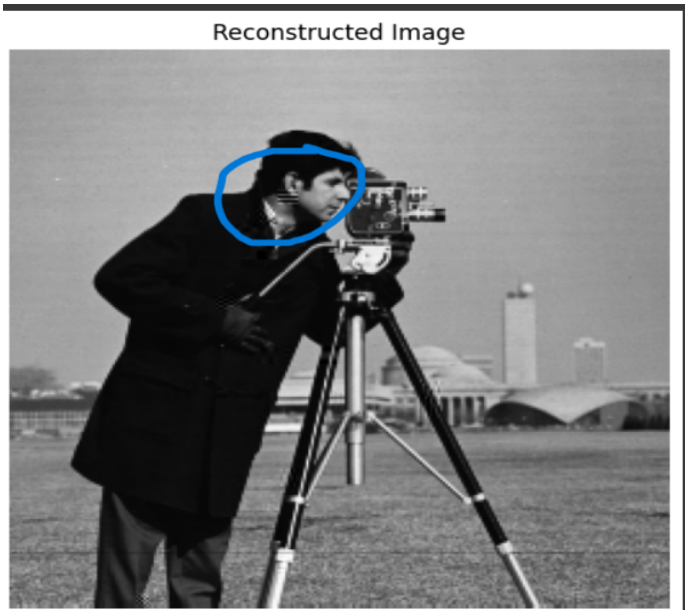


Figure 2: SRRC pulse Modulated Output Image for Different Noise Levels

12.2 Q15

After doing simulation with different values of noise level for MMSE equalizer we detect first error in reconstruction in noise level of 1.2.



We can calculate the Signal-to-Noise Ratio (SNR) for each case using the formula:

$$SNR_{dB} = 10 \log_{10} \left(\frac{P_{\text{signal}}}{P_{\text{noise}}} \right)$$

Where: - $P_{\text{signal}} = \text{mean}(\text{signal}^2)$ - $P_{\text{noise}} = \sigma^2 = P_{\text{signal}} \times \text{noise level}^2$ - $\sigma = P_{\text{signal}} \times \text{noise level}$
For MMSE equalizer with noise level = 1.2, the SNR calculation is:

$$SNR_{dB} = 10 \log_{10} \left(\frac{P_{\text{signal}}}{P_{\text{signal}} \times (1.2)^2} \right)$$

$$SNR_{dB} = 10 \log_{10} \left(\frac{1}{(1.2)^2} \right) \approx 10 \log_{10}(0.6944) \approx -1.59 \, \text{dB}$$

For ZF equalizer we should just change the code for equalizer and repeat the same process.

```
def zero_forcing_equalizer(signal, h):
    H_f = fft(h, len(signal))
    Q_zf_f = 1 / H_f
    Q_zf_f[np.isnan(Q_zf_f)] = 0
    signal_f = fft(signal)
    equalized_signal_f = signal_f * Q_zf_f
    equalized_signal = np.real(ifft(equalized_signal_f))
    return equalized_signal
```

For ZF equalizer with noise level = 1.1, the SNR calculation is:

$$SNR_{dB} = 10 \log_{10} \left(\frac{P_{\text{signal}}}{P_{\text{signal}} \times (1.1)^2} \right)$$

$$SNR_{dB} = 10 \log_{10} \left(\frac{1}{(1.1)^2} \right) \approx 10 \log_{10}(0.8264) \approx -0.79 \, \text{dB}$$

Thus, for the first error, the critical SNR for MMSE is approximately -1.59 dB, and for ZF is approximately -0.79 dB.
The MMSE (Minimum Mean Square Error) equalizer typically performs better in noisy environments compared to the ZF (Zero-Forcing) equalizer. This is because MMSE equalization minimizes the overall mean square error by balancing both signal and noise power. This makes MMSE more robust to noise, leading to a lower critical SNR for the first error.

12.3 Q16

Figure 3: MMSE

Figure 4: ZF

As we discussed in last part and saw in eye diagrams,MMSE performs better in noisy conditions.

12.4 Q17

SRRC pulses are specifically designed to satisfy the Nyquist criterion, ensuring zero ISI when used with an appropriately matched filter. Half-sine pulses, while simpler, do not naturally meet the Nyquist condition and may introduce residual ISI. Ideally, zero ISI is achieved right at the output of the matched filter, where the pulse shaping and filtering work together optimally. In practice, any remaining ISI may then be further reduced by an equalizer. Thus, SRRC pulses provide inherent zero-ISI conditions post-matched filter, while half-sine pulses typically require additional equalization.

12.5 Q18

Even if the noise level is identical, the reconstruction error depends on the pulse shaping characteristics. SRRC pulses confine energy within a narrower spectral band, yielding a higher effective SNR and lower reconstruction errors. In contrast, half-sine pulses exhibit broader spectral leakage, making them more vulnerable to noise and interference. The matching of the filter and equalizer also plays a significant role in how well the signal is recovered. As a result, the same noise level does not guarantee the same error performance for different pulse shapes.

Figure 5: Sin pulse Modulated Output Image for Different Noise Levels

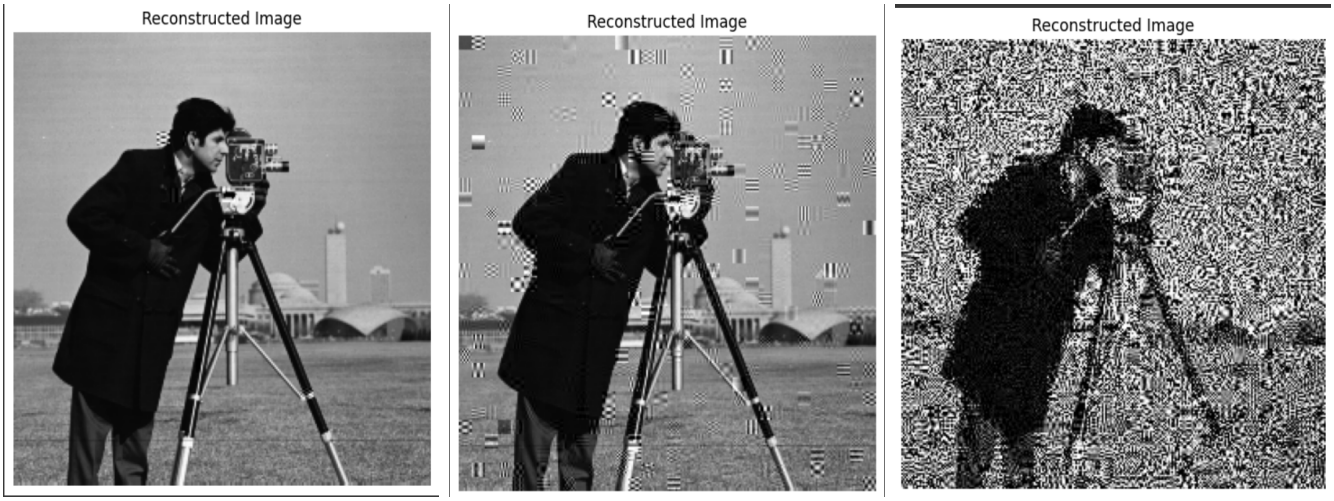


Figure 6: SRRC pulse Modulated Output Image for Different Noise Levels

12.6 Q19

The pulse type determines the spectral occupancy of the transmitted signal; for example, SRRC pulses tightly control bandwidth through their roll-off factors while half-sine pulses have a wider spectral spread. A narrow bandwidth minimizes interference and improves efficiency, whereas a wide bandwidth increases susceptibility to noise. Pulse length directly influences the duration of each bit period; longer pulses reduce the data rate but improve robustness to channel impairments. Shorter pulses enable higher bit rates but may increase ISI and reduce noise immunity. The overall system performance is thus a trade-off between bandwidth efficiency, data rate, and error resilience.

12.7 Q20

SRRC pulses are generally more advantageous because they offer controlled bandwidth, excellent ISI suppression, and optimal performance when paired with matched filtering, resulting in a robust SNR. Their main downside is the increased complexity in filter design and implementation. Half-sine pulses, on the other hand, are simpler to generate and require less processing power but suffer from higher spectral leakage and greater susceptibility to noise and ISI. In practical systems, the choice depends on the specific trade-offs between complexity and performance, with SRRC pulses typically preferred in high-performance scenarios despite their implementation challenges.

12.8 Q21

To see the channel effect we can just change h in the code like this :

```
# First channel
h_length = 25 * samples_per_symbol + 1
h = np.zeros(h_length)
h[0] = 1/2
h[samples_per_symbol] = 1
h[2 * samples_per_symbol] = 0.63
h[8 * samples_per_symbol] = 0.25
h[12 * samples_per_symbol] = 0.16
h[25 * samples_per_symbol] = 0.1

#Second Channel
h_length = 7 * samples_per_symbol + 1
h = np.zeros(h_length)
h[0] = 1
h[samples_per_symbol] = 0.4365
h[2 * samples_per_symbol] = 0.1905
h[3 * samples_per_symbol] = 0.0832
h[5 * samples_per_symbol] = 0.0158
h[7 * samples_per_symbol] = 0.003
```

Figure 7: Left Image is for First Channel-Right Image for Second Channel

The open-space channel $h_1[n]$ has long delay components (up to $n = 25$), which can cause inter-symbol interference (ISI), but environmental factors might mitigate this effect. The closed-space channel $h_2[n]$ has most of its energy concentrated at $n = 0$ with rapidly diminishing delays, resulting in lower ISI and easier equalization. Structurally, $h_2[n]$ provides better reconstruction due to reduced interference between symbols.