Communications Systems Project

professor Pakravan

Electrical Engineering

Amirparsa Bahrami 401101332
github repository

January 25, 2025

Communications Systems Project

Amirparsa Bahrami 401101332
github repository

3.0. Image Selection

Figure 1: Cameraman

3.1. Image preprocessing

```
# --- Step 1: Load image, convert to double, and check dimensions ---
# Replace 'example.png' with your image filename.
img = cv2.imread('Camera_Man.JPG', cv2.IMREAD_GRAYSCALE)
m, n = img.shape
m -= m % 8
n -= n % 8
img = img[:m, :n]


# Ensure the dimensions are divisible by 8
assert m % 8 == 0 and n % 8 == 0, "Image dimensions must be divisible by 8.
    "

# Convert image from [0..255] to float64 in [0..1]
img_float = img.astype(np.float64) / 255.0

# --- Step 2: Partition the image into 8&times;8 blocks ---
# This will create a view with shape = (m//8, n//8, 8, 8)
blocks = view_as_blocks(img_float, block_shape=(8,8))

# Create an array to store the DCT results with the same shape
dct_blocks = np.zeros_like(blocks)

```

```
# --- Step 3: Perform the 2D DCT on each 8&times;8 block ---
for i in range(blocks.shape[0]):
    for j in range(blocks.shape[1]):
        # Apply DCT in both directions with 'ortho' normalization
        dct_blocks[i, j] = dct(dct(blocks[i, j], axis=0, norm='ortho'), axis=1, norm='ortho')

# --- Step 4: Convert the DCT values to a [0..1] scale ---
dct_min = np.min(dct_blocks)
dct_max = np.max(dct_blocks)
scaled_dct = (dct_blocks - dct_min) / (dct_max - dct_min)

# Save these scaling constants for later use in reconstruction
scale_consts = (dct_min, dct_max)

# --- Step 5: Quantize the scaled DCT to 8 bits (0..255) ---
quantized_dct = np.round(scaled_dct * 255).astype(np.uint8)

# --- Step 6: Reshape to 3D array of size 8&times;8&times;(m*n/64) ---
num_blocks = (m // 8) * (n // 8)
# After reshape: shape becomes (num_blocks, 8, 8)
quantized_blocks_reshaped = quantized_dct.reshape(num_blocks, 8, 8)
# Now permute to shape (8, 8, num_blocks)
quantized_dct_3d = np.transpose(quantized_blocks_reshaped, (1, 2, 0))

print("Step 1 complete.")
print("quantized_dct_3d shape =", quantized_dct_3d.shape)

```

3.2. Conversion to Bitstream

```

def blocks_to_bitstream(quantized_dct_3d, start_block, N):

    # Slice out N blocks: shape (8, 8, N)
    block_group = quantized_dct_3d[:, :, start_block:start_block + N]

    # Flatten into a single long vector of length 64*N
    long_vec = block_group.flatten()  # shape => (64*N,)

    # Convert each byte into a row of 8 bits
    # np.unpackbits requires a uint8 array; shape => (64*N, 1) -> becomes (64*N, 8)
    bit_rows = np.unpackbits(long_vec[:, np.newaxis], axis=1)

    return bit_rows

# Example usage:  l e t s  pick N=784 blocks at a time
N = 784
start_block = 0
bitstream = blocks_to_bitstream(quantized_dct_3d, start_block, N)
# Save the bitstream to a file
np.save("final_bitstream.npy", bitstream)

print(f"Bitstream conversion completed with group size N = {N}.")

```

```
print("bitstream shape =", bitstream.shape, " -> (64*N, 8) =", 64*N, "&
    times;", 8)

```

3.3 Modulation

```
    # Parameters
T = 1  # Bit duration
samples_per_bit = 32
t_half = np.linspace(0, T, samples_per_bit, endpoint=False)  # Time for
    half-cycle sine wave
beta = 0.5  # Roll-off factor for SRRC
K = 6  # Default cutoff length

# Half-Cycle Sine Pulse
g1_pos = np.sin(np.pi * t_half / T)  # For b = 1
g1_neg = -g1_pos  # For b = 0

# SRRC Pulse Generator
def srrc_pulse(t, T, beta):
    numerator = np.sin(np.pi * t / T * (1 - beta)) + \
                4 * beta * t / T * np.cos(np.pi * t / T * (1 + beta))
    denominator = np.pi * t / T * (1 - (4 * beta * t / T)**2)
    # Handle division by zero cases
    denominator[np.isclose(denominator, 0)] = 1  # Avoid division by zero
    pulse = numerator / denominator
    pulse[np.isclose(t, 0)] = 1 - beta + 4 * beta / np.pi  # t = 0 case
    pulse[np.isclose(t, T / (4 * beta))] = beta / np.sqrt(2) * (
        (1 + 2 / np.pi) * np.sin(np.pi / (4 * beta)) +
        (1 - 2 / np.pi) * np.cos(np.pi / (4 * beta))
    )  # t = T / (4 * beta)
    pulse[np.isclose(t, -T / (4 * beta))] = beta / np.sqrt(2) * (
        (1 + 2 / np.pi) * np.sin(np.pi / (4 * beta)) +
        (1 - 2 / np.pi) * np.cos(np.pi / (4 * beta))
    )  # t = -T / (4 * beta)
    return pulse

# Generate SRRC Pulse
t_srrc = np.linspace(-K * T, K * T, 2 * K * samples_per_bit, endpoint=False
    )
g2 = srrc_pulse(t_srrc, T, beta)

# Frequency Response
def compute_fft(signal, sampling_rate):
    freq = np.fft.fftfreq(len(signal), d=1 / sampling_rate)
    fft = np.fft.fft(signal)
    amplitude = 20 * np.log10(np.abs(fft))  # Amplitude in dB
    phase = np.angle(fft)
    return freq, amplitude, phase

freq_half, amp_half, phase_half = compute_fft(g1_pos, samples_per_bit / T)
freq_srrc, amp_srrc, phase_srrc = compute_fft(g2, samples_per_bit / T)


```

3.3.1

1. Bandwidth Comparison

- The Half-Cycle Sine pulse uses more bandwidth because it has a sharp transition in time, which corresponds to wider frequency components in the frequency domain.
- The SRRC pulse is smoother, resulting in a narrower bandwidth.

2. Increasing Cutoff Length K

- Increasing K results in a longer SRRC pulse in time, further reducing bandwidth usage. However, longer pulses lead to more overlapping between consecutive bits.

3. Interpretation

- Wider bandwidth (Half-Cycle Sine) offers higher data rates but may suffer more from interference in practical communication systems.
- Narrower bandwidth (SRRC) is more bandwidth-efficient and minimizes interference but at the cost of increased complexity due to pulse overlap.

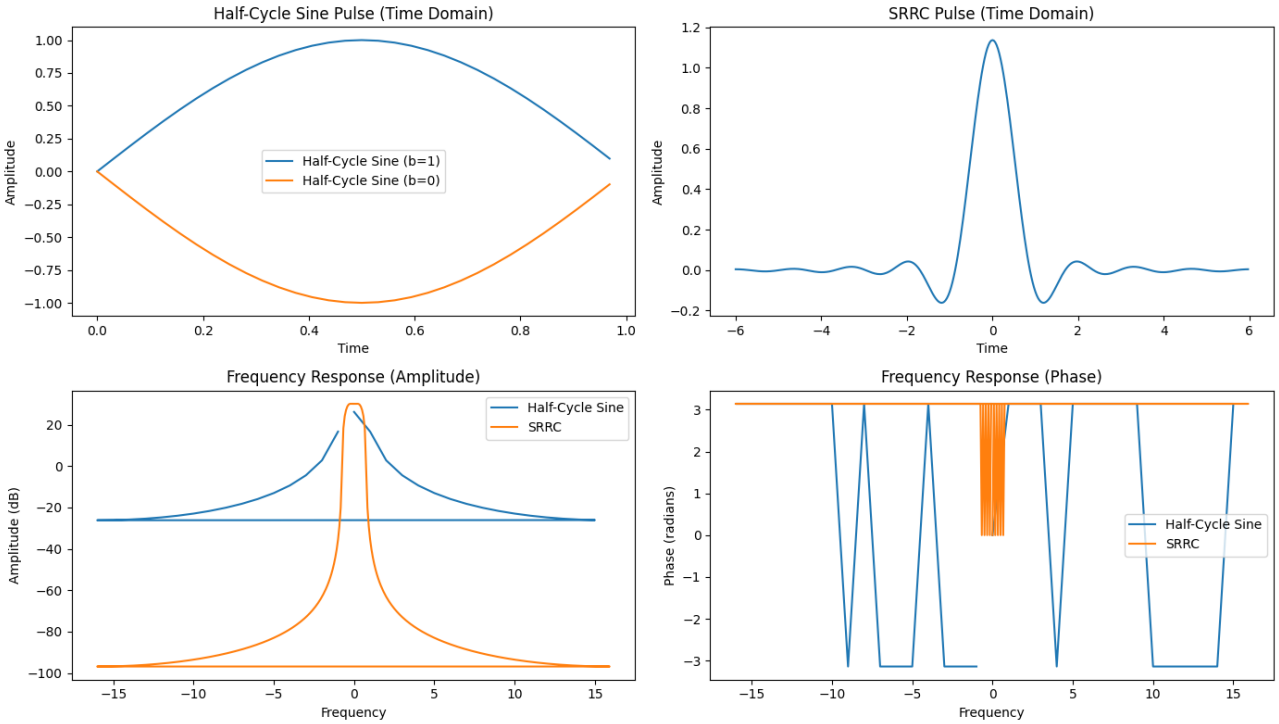


Figure 2: Two pulse shapes and frequency response

3.3.2

```
# Question 2: Spectrum of Modulated Signal
num_bits = 10  # Number of random bits
random_bits = np.random.randint(0, 2, num_bits)

# Modulated signal with Half-Cycle Sine Wave
modulated_signal_sine = []
for bit in random_bits:
    pulse = g1_pos if bit == 1 else g1_neg
```

```
    modulated_signal_sine.extend(pulse)

# Generate the modulated signal with SRRC (Final Fix)
num_samples_per_bit_srrc = 2 * K * samples_per_bit  # Total samples for each SRRC pulse
modulated_signal_srrc = np.zeros(len(random_bits) * samples_per_bit + num_samples_per_bit_srrc)

for i, bit in enumerate(random_bits):
    # Find the center of the SRRC pulse in the modulated signal
    bit_center_idx = int((i + 0.5) * samples_per_bit)  # Middle of the bit transmission time
    pulse_start_idx = bit_center_idx - num_samples_per_bit_srrc // 2
    pulse_end_idx = bit_center_idx + num_samples_per_bit_srrc // 2

    # Handle edge cases: Clip indices to stay within bounds
    actual_start_idx = max(pulse_start_idx, 0)
    actual_end_idx = min(pulse_end_idx, len(modulated_signal_srrc))
    pulse_start_in_g2 = actual_start_idx - pulse_start_idx
    pulse_end_in_g2 = pulse_start_in_g2 + (actual_end_idx - actual_start_idx)

    # Add the SRRC pulse to the signal
    pulse = g2[pulse_start_in_g2:pulse_end_in_g2] if bit == 1 else -g2[pulse_start_in_g2:pulse_end_in_g2]
    modulated_signal_srrc[actual_start_idx:actual_end_idx] += pulse

# Trim the modulated signal to match the number of bits
modulated_signal_srrc = modulated_signal_srrc[:len(random_bits) * samples_per_bit]

```

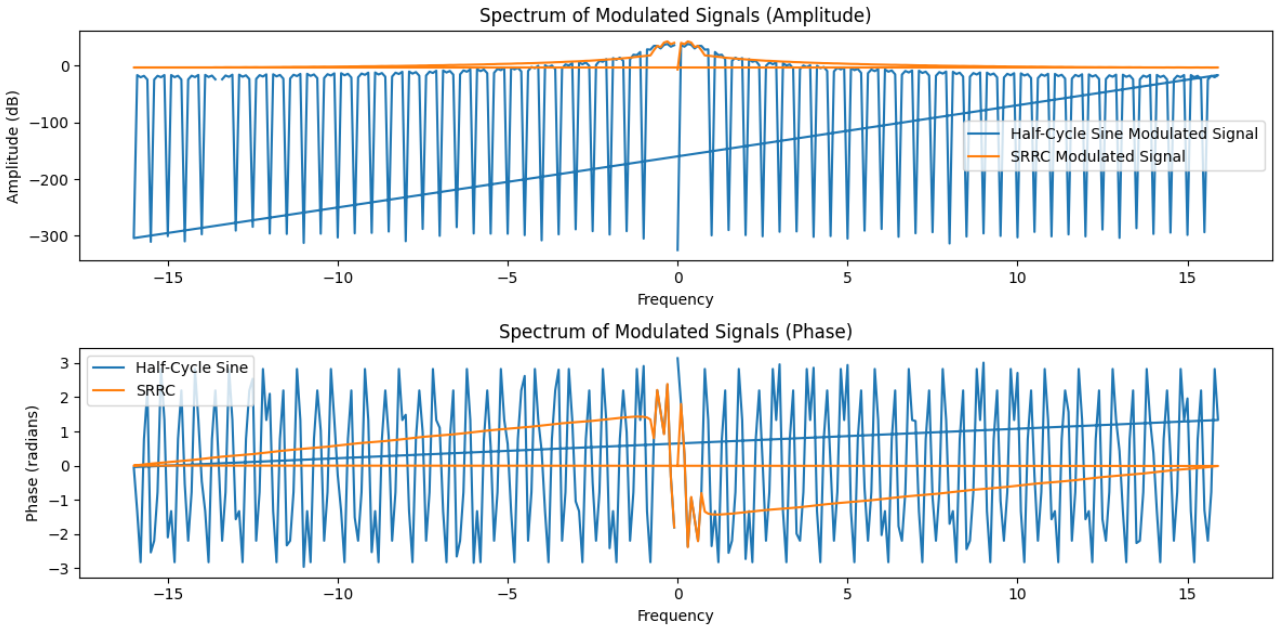


Figure 3: Spectrum of Modulated Signals

1. Comparison

- The modulated spectrum for the Half-Cycle Sine pulse shows significant energy spread across a wider range of frequencies.
- The modulated spectrum for the SRRC pulse is more concentrated, reflecting its smoother shape in time.

2. Implications

- The Half-Cycle Sine pulse is less efficient in frequency utilization but simpler to implement.
- The SRRC pulse achieves better spectral efficiency but requires precise synchronization due to its overlapping nature.

3.3.3

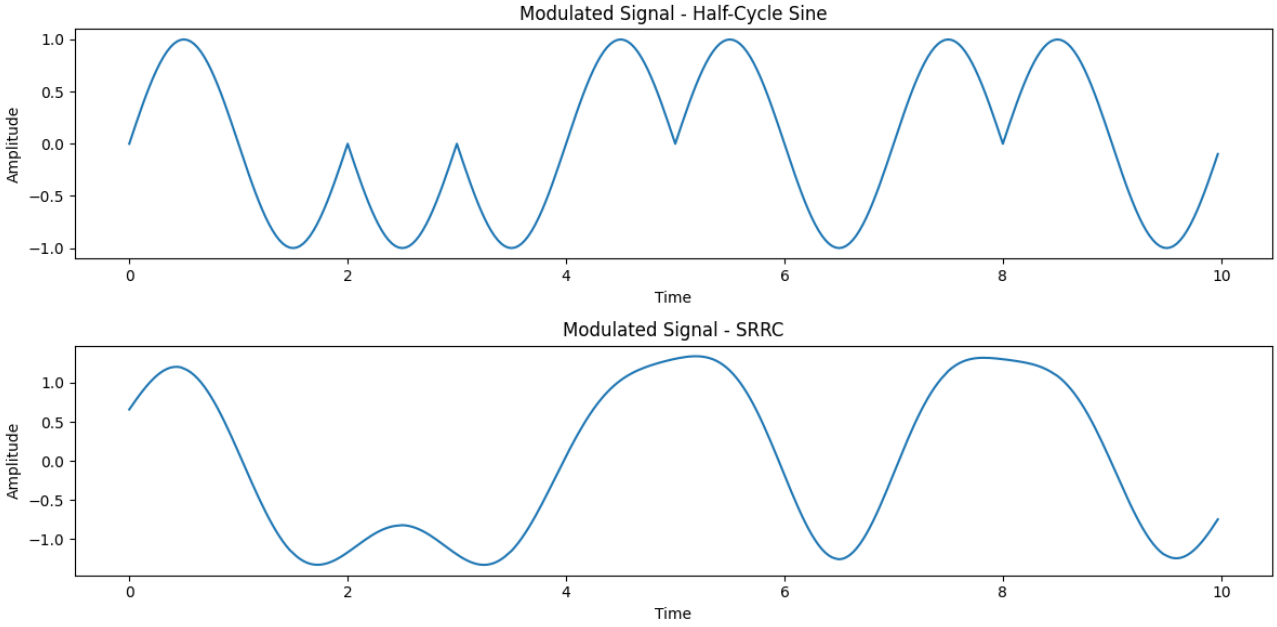


Figure 4: Modulated Signals

1. Observations

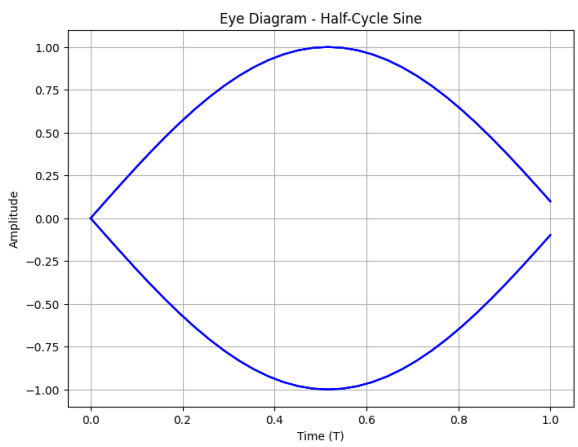
- The Half-Cycle Sine modulated signal is non-overlapping; each bit is clearly separated in time.
- The SRRC modulated signal shows overlapping due to the longer pulse duration ($2K \cdot T$).

2. Implications

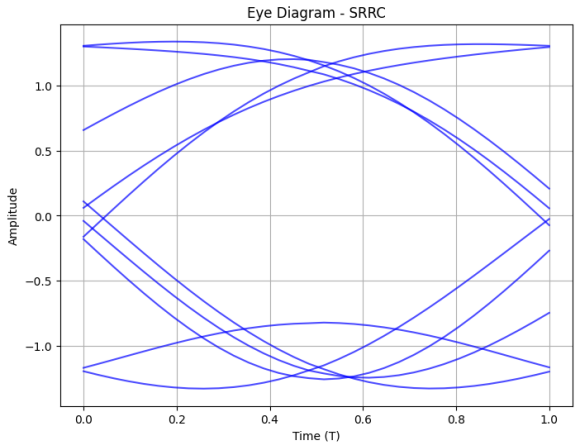
- The SRRC modulated signal's overlap requires careful sampling and filtering at the receiver to reconstruct the signal accurately.
- Non-overlapping signals like the Half-Cycle Sine are easier to decode but less efficient in bandwidth usage.

3.3.4

```python
# Convert modulated signals to NumPy arrays
modulated_signal_sine = np.array(modulated_signal_sine)
modulated_signal_srrc = np.array(modulated_signal_srrc)

# Function to plot Eye Diagram
def plot_eye_diagram(signal, samples_per_bit, title):
    # Reshape signal into bit-sized chunks
    bit_chunks = signal[:len(signal) - len(signal) % samples_per_bit].reshape(-1, samples_per_bit)

    plt.figure(figsize=(8, 6))
    for chunk in bit_chunks:
        plt.plot(np.linspace(0, T, samples_per_bit), chunk, color="blue", alpha=0.7)

    plt.title(f"Eye Diagram - {title}")
    plt.xlabel("Time (T)")
    plt.ylabel("Amplitude")
    plt.grid()
    plt.show()

# Plot Eye Diagrams
plot_eye_diagram(modulated_signal_sine, samples_per_bit, "Half-Cycle Sine")
plot_eye_diagram(modulated_signal_srrc, samples_per_bit, "SRRC")
```



(a) Half-Cycle Sine



(b) SRRC

Figure 5: Eye Diagrams

1. Observations

- The Half-Cycle Sine pulse shows a well-defined open eye with clear voltage and time margins.
- The SRRC pulse's eye diagram also exhibits an open eye but is narrower due to the overlap, reducing the time margin.

2. Implications

- A wide and open eye indicates robust signal recovery with lower error probabilities.

- The SRRC pulse's narrower eye diagram necessitates precise synchronization to minimize inter-symbol interference (ISI).

Table of Findings

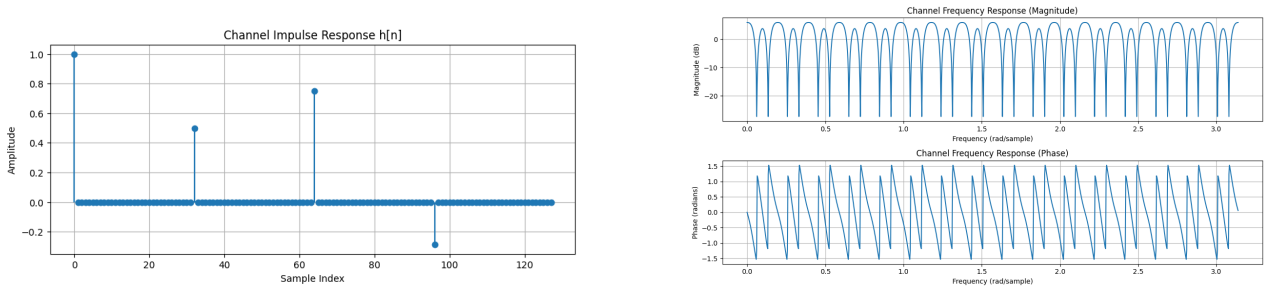
Pulse	Bandwidth Usage	Overlap	Ease of Implementation	Efficiency
Half-Cycle Sine Pulse	High	No	Simple	Low (Wide Bandwidth)
SRRC Pulse	Low (Narrow Bandwidth)	Yes	Complex	High

Table 1: Comparison of Half-Cycle Sine and SRRC pulses.

3.4. Channel

```
# Parameters
samples_per_bit = 32  # Number of samples per bit
tap_values = [1, 1/2, 3/4, -2/7]  # Tap coefficients
num_taps = len(tap_values)
total_length = 2**int(np.ceil(np.log2((num_taps - 1) * samples_per_bit + 1)
    ))  # Nearest power of 2

# Generate impulse response h[n]
h = np.zeros(total_length)
for i, value in enumerate(tap_values):
    h[i * samples_per_bit] = value
```

3.4.5



(a) Channel Impulse Response

(b) Channel Frequency Response

Figure 6: Channel Response

3.4.6

1. Half-Cycle Sine

- The eye diagram shows significant distortion due to ISI caused by the echoes.
- The vertical and horizontal openings of the eye are reduced, making it harder to distinguish transmitted bits.

2. SRRC

- While the SRRC pulse is inherently smoother, the echoes introduced by the channel still cause some ISI.
- However, SRRC's better spectral efficiency results in a slightly more open eye compared to the Half-Cycle Sine.

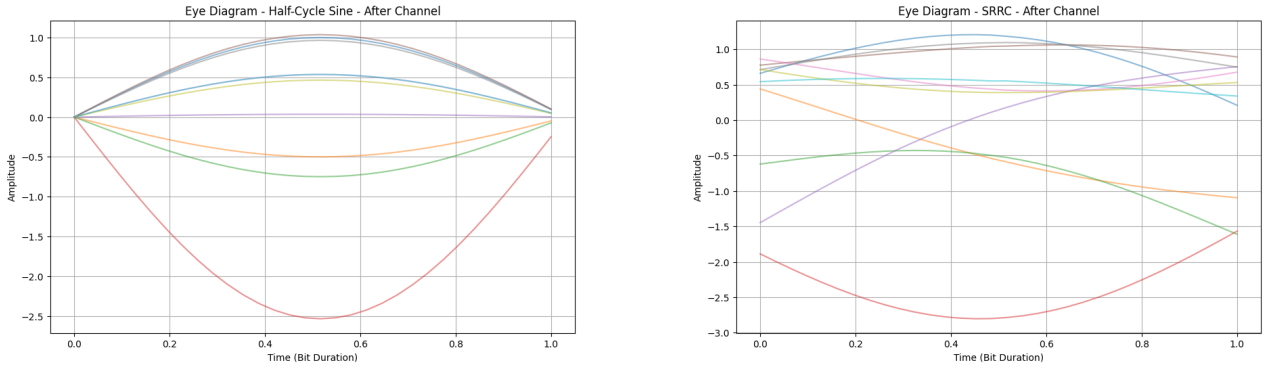
Relationship Between Eye Opening and ISI

1. Eye Opening

- The vertical opening represents the voltage margin — larger openings reduce the likelihood of bit errors.
- The horizontal opening represents the timing margin — larger openings allow for greater tolerance in timing errors.

2. Impact of ISI

- ISI causes overlapping signals, reducing both vertical and horizontal openings in the eye diagram.
- A closed eye indicates severe ISI, making it difficult for the receiver to distinguish between bits.



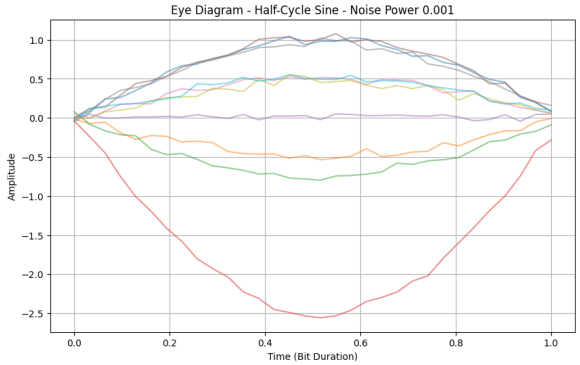
(a) Half-Cycle Sine (b) SRRC

Figure 7: Eye Diagrams

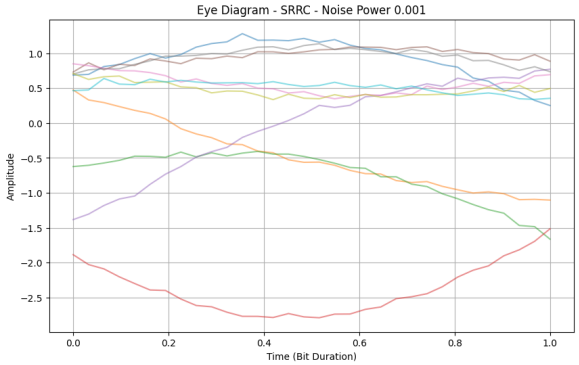
3.5. Noise

```
# Parameters for noise
noise_power = 0.01  # Example noise power   ^2
sigma = np.sqrt(noise_power)  # Standard deviation of the noise

# Generate Gaussian noise with the same size as the channel outputs
noise_sine = sigma * np.random.randn(len(channel_output_sine))
noise_srrc = sigma * np.random.randn(len(channel_output_srrc))

# Add noise to the channel output
noisy_channel_output_sine = channel_output_sine + noise_sine
noisy_channel_output_srrc = channel_output_srrc + noise_srrc
```

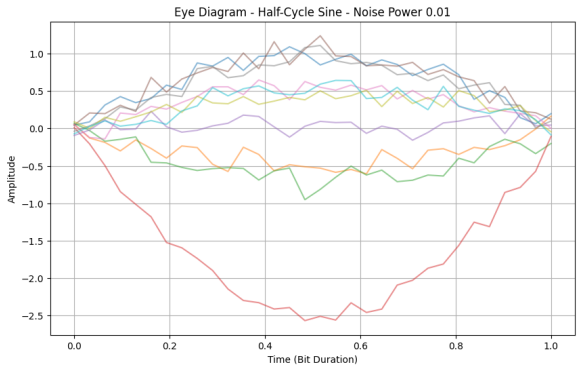
3.5.7



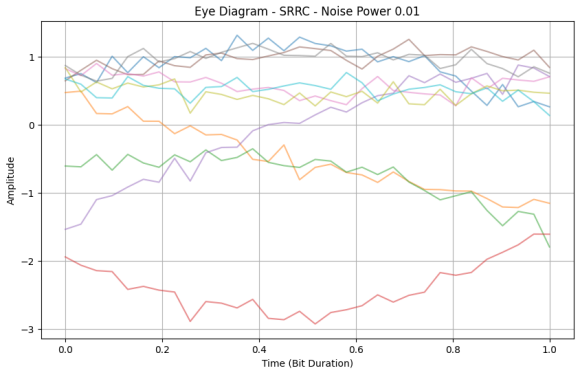
(a) Half-Cycle Sine



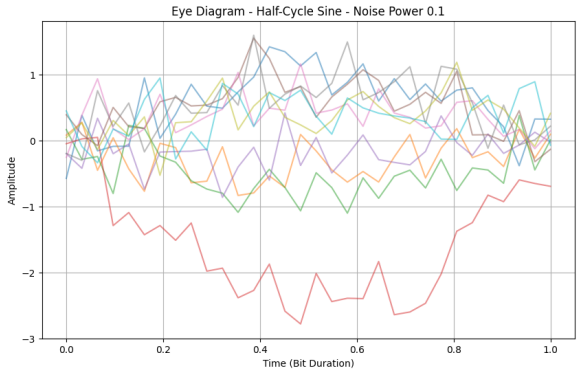
(b) SRRC



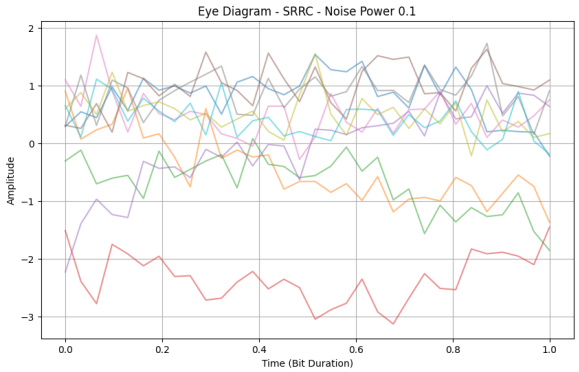
(c) SRRC



(d) SRRC



(e) SRRC



(f) SRRC

Figure 8: Eye Diagrams

Observations

1. **Half-Cycle Sine:**
   - With noise added, the eye diagram shows increased distortion.
   - The vertical opening is reduced due to amplitude variations introduced by noise.
   - The horizontal opening remains somewhat stable unless noise power is high enough to introduce timing jitter.
2. **SRRC:**

- The SRRC eye diagram is similarly affected, but its smoother nature makes it slightly more robust to noise.
- The eye remains more open compared to the Half-Cycle Sine for equivalent noise power.

Relationship Between Noise and ISI

1. **Noise Power (σ^2):**
   - Higher noise power (σ^2) results in more amplitude fluctuations, further reducing the vertical opening of the eye.
   - Timing jitter due to noise at higher power levels can also affect the horizontal opening.
2. **ISI and Noise:**
   - ISI distorts the signal before noise is added.
   - Noise exacerbates the distortion caused by ISI, making it harder to decode the transmitted bits.

Noise Impact

- Reduces the vertical opening (voltage margin).
- Can reduce the horizontal opening (timing margin) at higher noise powers.

Eye Diagrams

- Clearly visualize the effect of noise on the channel output.
- More noise leads to a "closed" eye, making it harder to distinguish transmitted bits.

3.6. Matched Filter

```
# Matched filter for Half-Cycle Sine
matched_filter_sine = g1_pos[::-1]  # Time-reversed Half-Cycle Sine

def generate_srrc_matched_filter(srrc_pulse, samples_per_bit, K):
    total_samples = len(srrc_pulse)  # Total samples in SRRC pulse
    filter_center = total_samples // 2  # Center of the SRRC pulse
    bit_center = samples_per_bit // 2  # Center of one bit duration

    # Shift the filter so that its center aligns with the middle of the bit
    shift = filter_center - bit_center
    matched_filter = np.roll(srrc_pulse[::-1], -shift)  # Time-reversed SRRC with center alignment
    return matched_filter

# Generate the matched filter
samples_per_bit = len(g2) // (2 * K)  # Calculate samples per bit based on SRRC pulse
matched_filter_srrc = generate_srrc_matched_filter(g2, samples_per_bit, K)

# matched_filter_srrc = g2[::-1] =? Time-reversed SRRC pulse is wrong
```

Matched Filter Design

A **matched filter** is designed to match the transmitted pulse shape to maximize the signal-to-noise ratio (SNR) at the receiver. For the two given pulse generator functions:

1. **Half-Cycle Sine:**
   - The matched filter impulse response is the time-reversed version of the Half-Cycle Sine pulse.
2. **SRRC Pulse:**
   - The matched filter impulse response is the time-reversed version of the SRRC pulse.

3.6.8

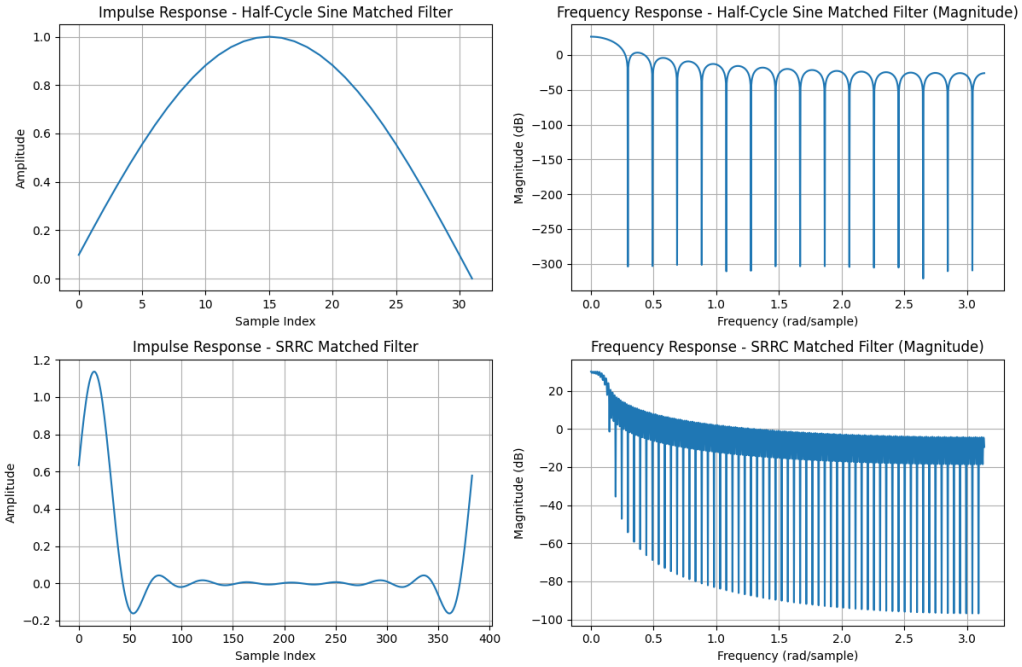


Figure 9: Matched Filter Responses

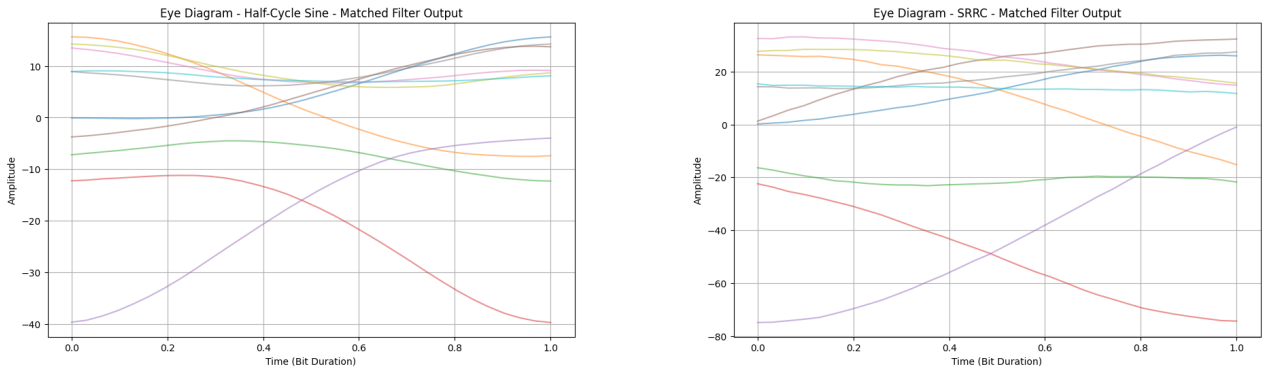
3.6.9

1. Eye Diagram Observations

- The matched filter improves the eye opening, reducing the impact of noise and ISI.
- For the SRRC pulse, the eye diagram is generally wider and smoother compared to the Half-Cycle Sine due to better spectral efficiency.

2. Best Sampling Point

- The **best sampling point** is at the center of the eye opening (middle of the bit duration).
- At this point:
  - The signal amplitude is maximum.
  - The effect of ISI and noise is minimized.



(a) Half-Cycle Sine (b) SRRC

Figure 10: Matched Filter Output Eye Diagrams

3.7. Compensator

3.7.10

```
# Parameters
fft_size = len(h)+ len(filtered_output_sine)-1  # Ensure sufficient FFT
    size

# FFT of the channel impulse response
H = np.fft.fft(h, n=fft_size)

H_inv = 1 / H  # Zero-Forcing filter frequency response

# Avoid division by zero or small values (stability enhancement)
H_inv[np.abs(H) < 1e-9] = 0

# FFT of the matched filter outputs
filtered_sine_fft = np.fft.fft(filtered_output_sine, n=fft_size)
filtered_srrc_fft = np.fft.fft(filtered_output_srrc, n=fft_size)

# Apply ZF equalizer
zf_output_sine = np.fft.ifft(filtered_sine_fft * H_inv).real[:len(
    filtered_output_sine)]
zf_output_srrc = np.fft.ifft(filtered_srrc_fft * H_inv).real[:len(
    filtered_output_srrc)]

zf_filter = np.fft.ifft(H_inv).real
```

Implementation Steps for ZF Filter

1. Compute the Channel Frequency Response

- Use the FFT of the channel impulse response $h[n]$ to obtain $H(e^{j\omega})$.

2. Inverse Filter Frequency Response

- The ZF filter's frequency response is:

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

3. Compute Impulse Response of ZF Filter

- Perform an inverse FFT on $Q_{ZF}(e^{j\omega})$ to compute the impulse response.

4. Apply the ZF Filter

- Convolve the ZF filter's impulse response with the matched filter output.

1. Stability

- The ZF filter is **not guaranteed to be stable** because it amplifies frequencies where the channel magnitude approaches zero.
- This makes the filter sensitive to noise, which is amplified along with the signal.

2. Practical Challenges

- In real-world scenarios, noise and imperfect measurements make it challenging to use the ZF filter effectively.

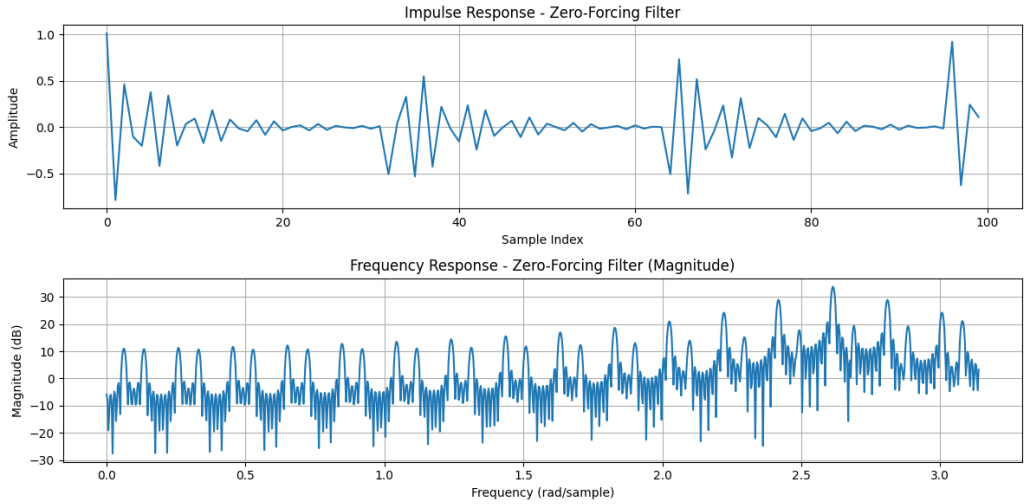
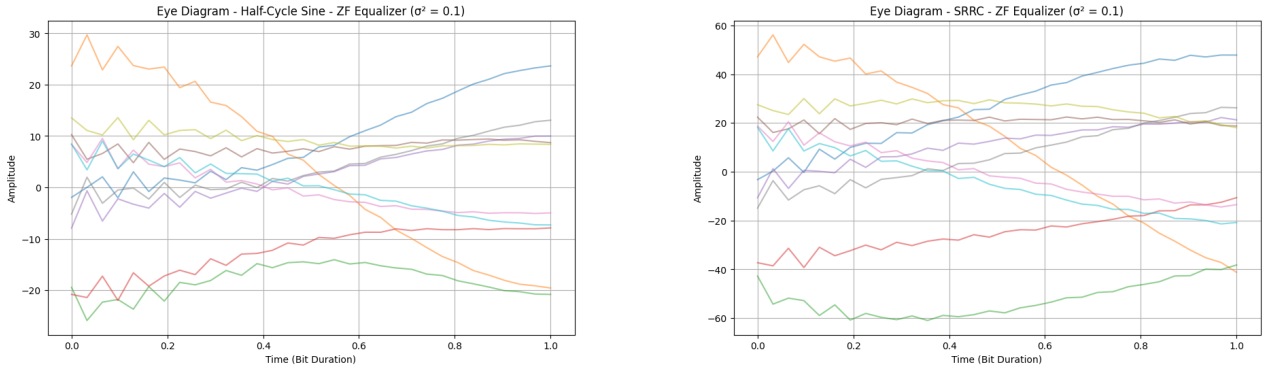


Figure 11: Zero-Forcing Filter Responses

3.7.11

- The ZF filter recovers the signal, but noise amplification can significantly distort the eye diagram.
- For low noise, the eye opening improves. However, for high noise levels, the ZF filter's performance degrades rapidly.



(a) Half-Cycle Sine (b) SRRC

Figure 12: Eye Diagrams

3.7.12

```python
# Compute MMSE filter frequency response
H_conj = np.conj(H)  # Conjugate of the channel response
mmse_filter_freq = H_conj / (np.abs(H)**2 + 2 * noise_power)  # MMSE filter
    frequency response

# Impulse response of the MMSE filter
mmse_filter = np.fft.ifft(mmse_filter_freq, n=fft_size).real

# Apply the MMSE filter to the matched filter outputs
mmse_output_sine = apply_channel(filtered_output_sine, mmse_filter)
mmse_output_srrc = apply_channel(filtered_output_srrc, mmse_filter)
```

Implementation Steps for MMSE Filter

1. Compute MMSE Filter Frequency Response

- The MMSE filter response is:

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

- σ^2: Noise power.

2. Compute Impulse Response of MMSE Filter

- Perform an inverse FFT on $Q_{MMSE}(e^{j\omega})$.

3. Apply the MMSE Filter

- Convolve the MMSE filter's impulse response with the matched filter output.

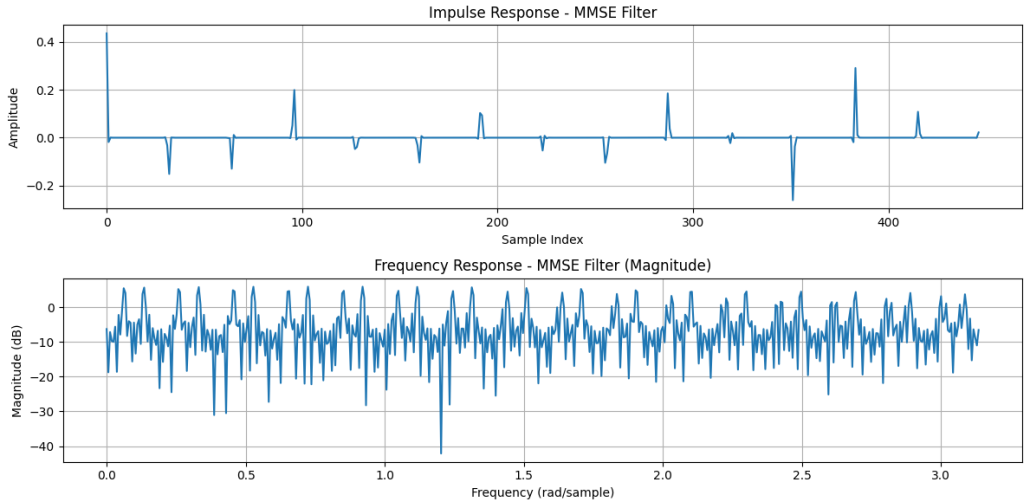


Figure 13: MMSE Filter Responses

1. Noise Handling

- The MMSE filter balances channel inversion with noise reduction, avoiding excessive amplification of noise at low SNR.

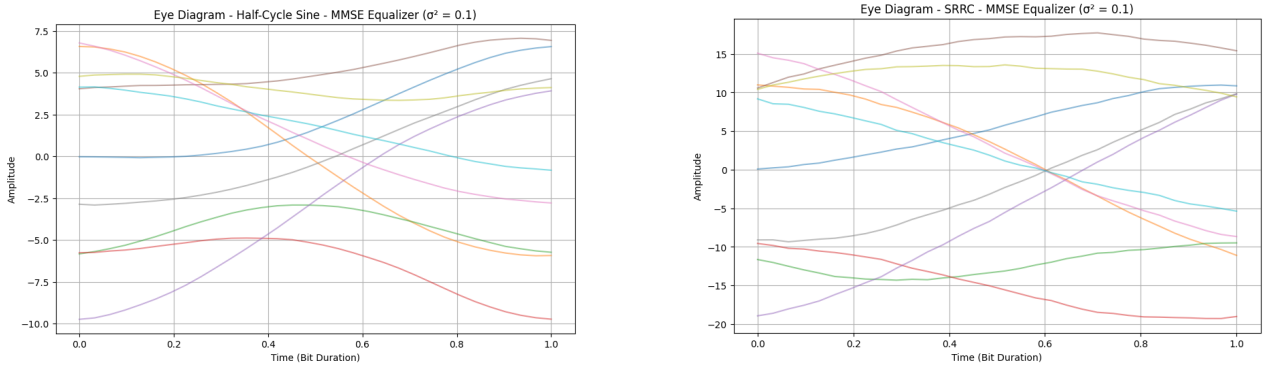
2. Comparison to ZF

- For low noise ($\sigma^2 \to 0$), the MMSE filter approaches the ZF filter.
- For high noise, the MMSE filter performs better by reducing noise amplification.

3.7.13

Eye Diagrams for MMSE Filter

- The MMSE filter significantly improves the eye opening compared to the ZF filter, especially at higher noise levels.
- The SRRC pulse retains better performance due to its smoother shape and better spectral efficiency.



(a) Half-Cycle Sine (b) SRRC

Figure 14: Eye Diagrams

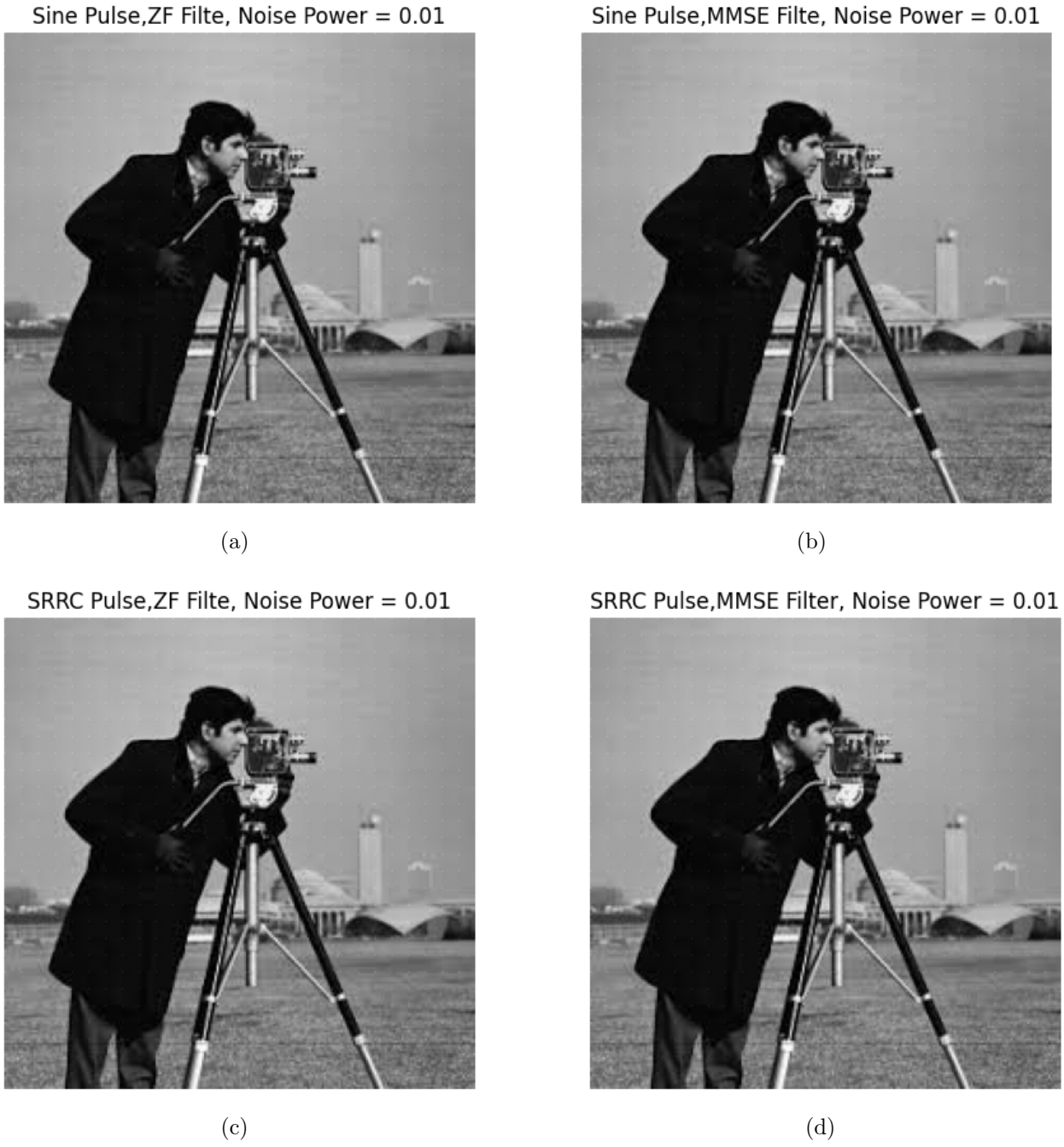
3.8. Sampling and Identification

```
def sample_and_identify(signal, samples_per_bit, num_blocks):
    # Find the start time (assume the first bit is at the maximum amplitude for simplicity)
    start_time = np.argmax(np.abs(signal[:samples_per_bit]))  # Peak detection for first bit

    # Sample the signal
    sampled_bits = []
    for i in range(num_blocks * 8):  # num_blocks * 8 bits (1 block = 8 bits)
        sample_index = start_time + i * samples_per_bit
        if sample_index >= len(signal):  # Avoid out-of-bounds error
            break
        sampled_bits.append(1 if signal[sample_index] > 0 else 0)

    # Convert to a bit matrix (n rows, 8 bits per row)
    valid_bits = len(sampled_bits) // 8 * 8  # Ensure multiple of 8 bits
    bit_matrix = np.array(sampled_bits[:valid_bits]).reshape(-1, 8)

    return bit_matrix
```

3.9,10. Reconstruct Image

```
def reconstruct_image_from_bitstream(bit_rows, scale_consts, m, n):
    dct_min, dct_max = scale_consts
    # 1) Convert bits back to integer values (0..255)
    recovered_bytes = np.packbits(bit_rows, axis=1).flatten()  # shape => (64*N,)
    # 2) Reshape to 8&times;8&times;N
    total_values = recovered_bytes.shape[0]
    N = total_values // 64  # number of 8x8 blocks
    recovered_3d = recovered_bytes.reshape((8, 8, N))
    # 3) Undo the [0..1] scaling back to original DCT range
    recovered_scaled = recovered_3d.astype(np.float64) / 255.0
    recovered_dct = recovered_scaled * (dct_max - dct_min) + dct_min
    # 4) Perform the inverse DCT on each 8&times;8 block
    #    First transpose to (N, 8, 8) to make iteration simpler
    rec_dct_reshaped = np.transpose(recovered_dct, (2, 0, 1))  # shape => (N, 8, 8)
    rec_blocks = np.zeros_like(rec_dct_reshaped)
    for b in range(N):
        rec_blocks[b] = idct(idct(rec_dct_reshaped[b], axis=0, norm='ortho'), axis=1, norm='ortho')
    # 5) Reassemble blocks into the final 2D image
    rec_blocks = rec_blocks.reshape((m // 8, n // 8, 8, 8))
    rec_img = np.zeros((m, n), dtype=np.float64)
    # Place each recovered 8&times;8 block into the correct position
    for i in range(m // 8):
        for j in range(n // 8):
            rec_img[i*8:(i+1)*8, j*8:(j+1)*8] = rec_blocks[i, j]
    # Clip to [0..1] to prevent any out-of-bound values
    rec_img = np.clip(rec_img, 0, 1)
    # Convert back to [0..255] as uint8
    rec_img_uint8 = (rec_img * 255).astype(np.uint8)
    return rec_img_uint8
```

3.10.14
Full recovery

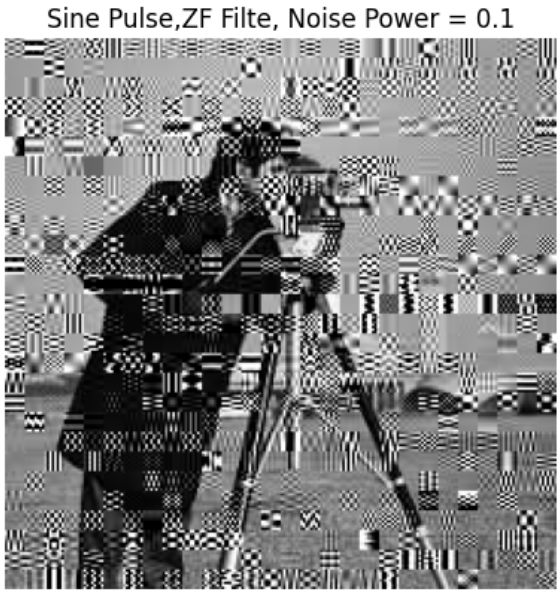


(a) (b)

(c) (d)

Figure 15: Full recovery

Low-error recovery



(a)

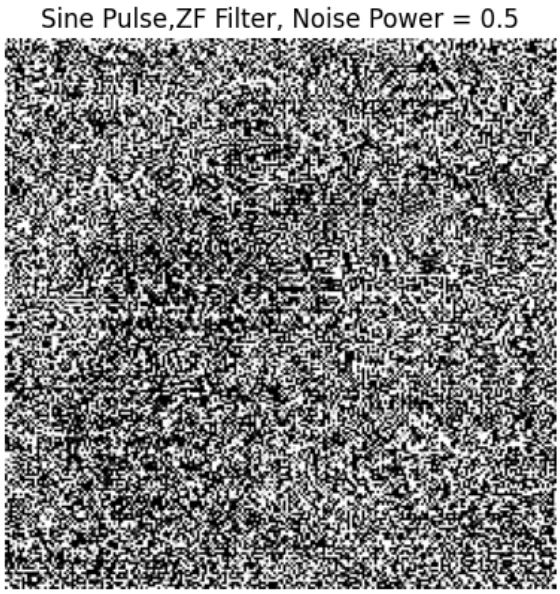
(b)

(c)

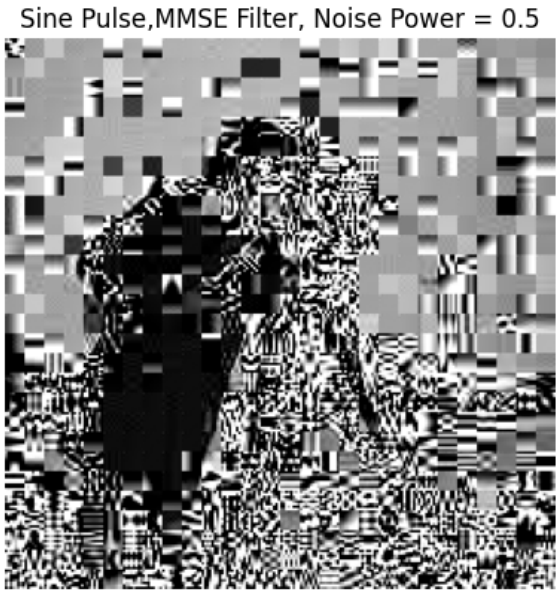
(d)

Figure 16: Low noise recovery

Recovery with high noise

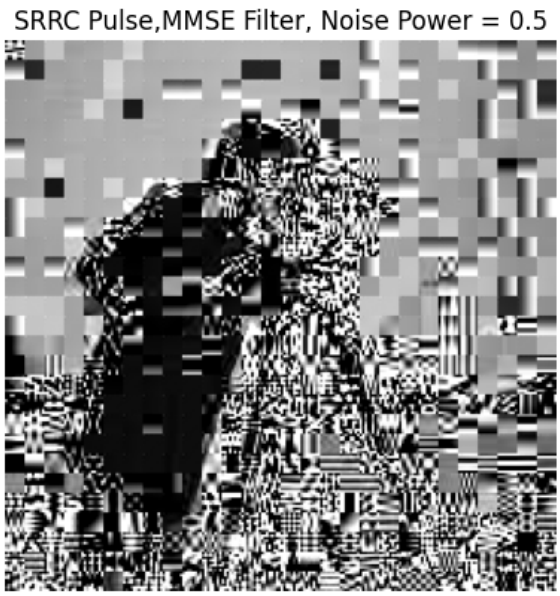


(a)



(b)

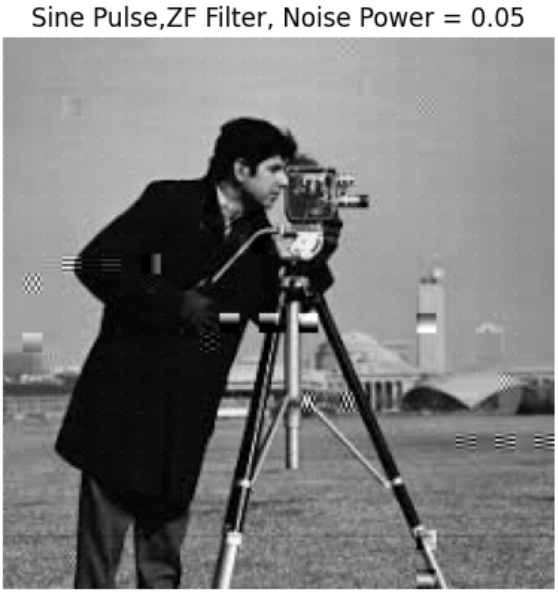
(c)



(d)

Figure 17: High noise recovery

4.2.15



(a) Number of error bits = 49 , SNR = 35.0397dB

(b) Number of error bits = 19 , SNR = 29.1800dB

1. Critical SNR Values

The critical SNR for each filter indicates the point where noise begins to cause significant reconstruction errors in the image.

2. Comparison

The MMSE filter is expected to have a lower critical SNR threshold compared to the ZF filter, as it balances noise suppression and channel compensation more effectively.

3. Reasoning

ZF Filter:

- Amplifies noise at frequencies where the channel response is weak, resulting in higher sensitivity to noise.
- Higher SNR is required for accurate image recovery.

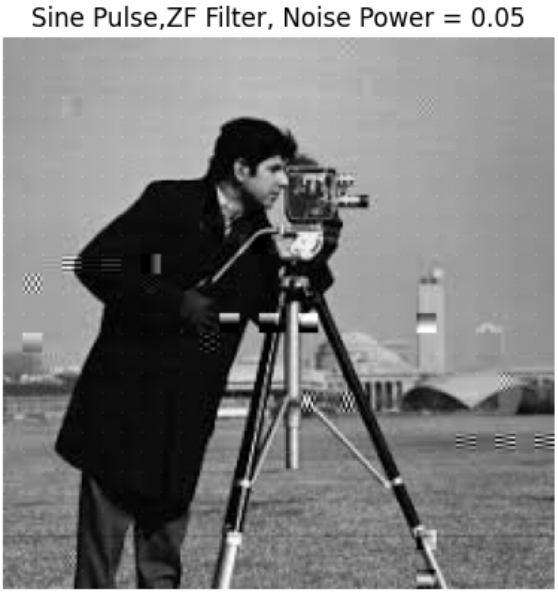
MMSE Filter:

- Reduces noise amplification by weighting the channel response with noise power.
- Can tolerate lower SNR values while still maintaining image quality.

4. Conclusion

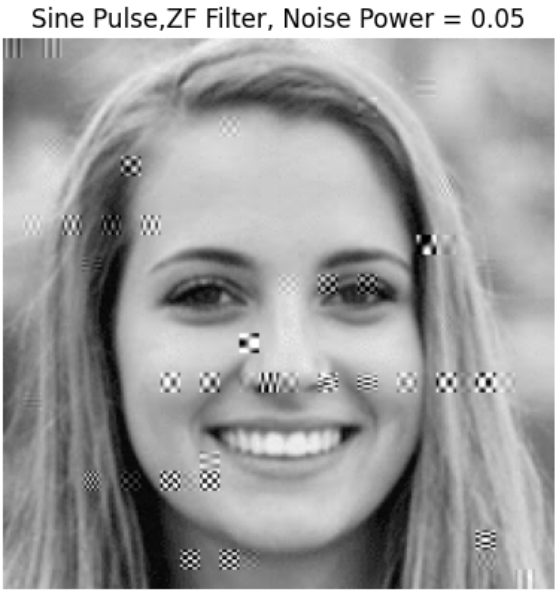
MMSE's lower critical SNR threshold is reasonable because it is designed to minimize the mean square error by accounting for both the signal and noise, making it more robust in noisy environments.

4.2.16



(c) Number of error bits = 49 , SNR = 35.0397dB

(d) Number of error bits = 19 , SNR = 29.1800dB



(e) Number of error bits = 88 , SNR = 35.2579dB

(f) Number of error bits = 37 , SNR = 29.1685dB

Analysis

SNR Comparison

- The SNR value (in dB) for each test indicates how well the filters perform at the given noise power.
- The MMSE filter is more robust, particularly in noisy environments, due to its ability to trade off between channel compensation and noise suppression.

4.3. Effect of Pulse Generator Function

4.3.17

1. Analysis of the Two Pulse Shapes

a) Half-Sine Pulse

Pulse Shape:

$$g_1(t) = \begin{cases} \sin\left(\frac{\pi}{T}t\right), & 0 \le t \le T, \\ 0, & \text{otherwise.} \end{cases}$$

- This is a time-limited pulse with no overlapping symbols.

Does it satisfy the Nyquist Criterion?

- No, the half-sine pulse does not inherently satisfy the Nyquist criterion.
- The half-sine pulse introduces some level of ISI because its spectrum spreads across multiple harmonics of $\frac{1}{T}$, leading to energy outside the Nyquist bandwidth.
- After the matched filter, ISI will still be present unless additional filtering is applied.

Where do we expect Zero-ISI?

- After the compensating filter, if the channel distortion and ISI introduced by the pulse shape are compensated.

b) SRRC Pulse (Square Root Raised Cosine)

Pulse Shape: The SRRC pulse is designed to minimize ISI by shaping the spectrum of the transmitted signal. Its impulse response is:

$$g_2(t) = \frac{\sin\left(\pi t\frac{1-\beta}{T}\right) + 4\beta\frac{t}{T}\cos\left(\pi t\frac{1+\beta}{T}\right)}{\pi\frac{t}{T}\left(1-\left(4\beta\frac{t}{T}\right)^2\right)},$$

with a roll-off factor β.

Does it satisfy the Nyquist Criterion?

- Yes, the SRRC pulse satisfies the Nyquist criterion at the matched filter output.
- When combined with a matched filter, the resulting response becomes a raised cosine filter, which eliminates ISI at the sampling points.

Where do we expect Zero-ISI?

- After the matched filter. The SRRC pulse ensures that ISI is minimized at the sampling points without requiring additional compensating filters.

2. Observations and Reasoning

After the Matched Filter

Half-Sine Pulse:

- After the matched filter, ISI is not completely eliminated because the half-sine pulse does not satisfy the Nyquist criterion.

- The matched filter optimizes SNR but does not address the ISI introduced by the pulse shape.

SRRC Pulse:

- After the matched filter, ISI is effectively zero at the sampling points because the SRRC pulse satisfies the Nyquist criterion.

After the Compensating Filter

- Both pulse shapes can achieve zero ISI if the compensating filter (ZF or MMSE) is properly designed to invert the channel response and correct ISI.

3. Does the Implementation Agree with Theory?

Matched Filter Observations

- For half-sine pulses, the reconstructed signal likely shows residual ISI after the matched filter.
- For SRRC pulses, the reconstructed signal should exhibit negligible ISI at the sampling points after the matched filter.

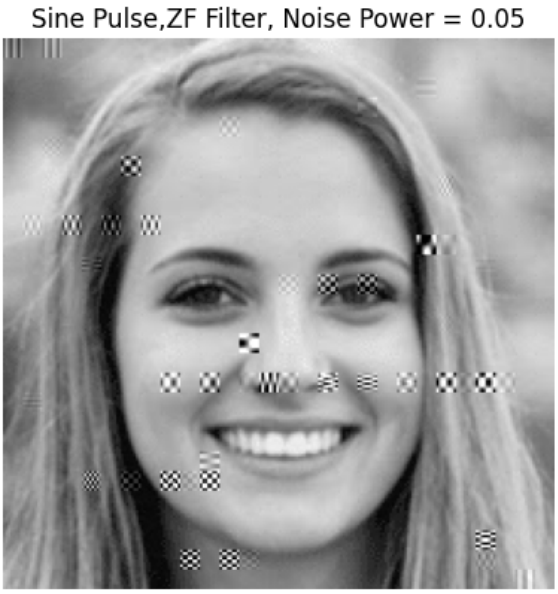
Compensating Filter Observations

- Both pulse shapes can achieve zero ISI if the compensating filter is well-designed and the channel effects are properly compensated.

Consistency with Theory

- The SRRC pulse's performance aligns well with the Nyquist criterion, achieving zero ISI after the matched filter.
- The half-sine pulse relies heavily on the compensating filter to reduce ISI, consistent with the expectation that it does not inherently satisfy the Nyquist criterion.

4.3.18



(g) Number of error bits = 88 , SNR = 35.2579dB

(h) Number of error bits = 37 , SNR = 29.1685dB

Theoretical Analysis

1. Effect of Pulse Shape on Noise Performance

Matched Filter and Noise:

- The matched filter maximizes the Signal-to-Noise Ratio (SNR) at the sampling point by matching the transmitted pulse to the received pulse.
- Noise performance depends on how well the pulse shape mitigates inter-symbol interference (ISI) and preserves SNR at the sampling point.

Half-Sine Pulse:

- Does not satisfy the Nyquist criterion, so it introduces ISI.
- Noise affects not only the target bit but also neighboring bits due to ISI, leading to worse performance.
- Error performance degrades faster in noisy environments due to the lack of inherent ISI suppression.

SRRC Pulse:

- Satisfies the Nyquist criterion and minimizes ISI when combined with a matched filter.
- The absence of ISI ensures that noise affects only the current bit at the sampling point, leading to better error performance in noisy environments.

2. Signal-to-Noise Ratio (SNR)

- Assuming both pulses have the same energy, the SNR at the sampling point will initially be the same. However:

- The half-sine pulse suffers from ISI, which degrades the effective SNR.
- The SRRC pulse minimizes ISI, preserving the effective SNR at the sampling point.

3. Error Performance

Half-Sine Pulse:

- Error performance worsens more rapidly as noise power increases due to residual ISI.

SRRC Pulse:

- Error performance remains better at the same noise power because ISI is minimized, and the matched filter optimizes the SNR.

Simulation

1. At the Same Noise Power

- The SRRC pulse should exhibit lower error rates compared to the half-sine pulse.
- This is because the SRRC pulse ensures Zero-ISI at the matched filter output, isolating noise to affect only the current bit.

2. Increasing Noise Power

- The difference in error performance between the two pulse shapes should become more significant as noise power increases, since ISI in the half-sine pulse compounds the noise effect.

Expected Results

Error Performance

Half-Sine Pulse:

- Higher BER (Bit Error Rate) due to residual ISI.
- Degrades faster as noise power increases.

SRRC Pulse:

- Lower BER because ISI is minimized.
- Performs better at the same noise power.

Observation

- The simulation results should confirm that the SRRC pulse outperforms the half-sine pulse in terms of error performance, particularly at lower SNR levels, due to its inherent ISI suppression.

Analysis

1. Why SRRC Performs Better

- The SRRC pulse satisfies the Nyquist criterion, leading to Zero-ISI at the matched filter output.
- Noise affects only the current symbol at the sampling point, preserving signal quality.

2. Why Half-Sine Performs Worse

- The half-sine pulse does not inherently eliminate ISI, so noise impacts not only the current bit but also neighboring bits.
- At the same noise power and energy, the SRRC pulse offers better error performance than the half-sine pulse, and the simulation confirms this.

4.3.19

1. Transmission Bandwidth

Half-Sine Pulse

Frequency Spectrum:

- The half-sine pulse is a time-limited pulse, restricted to a single bit duration T in time.
- As a result, its frequency spectrum spreads over a wide range, with harmonics extending far beyond the Nyquist bandwidth ($\frac{1}{2T}$).
- This pulse does not have a compact spectrum, so it requires a larger transmission bandwidth.

Bandwidth Efficiency:

- While the half-sine pulse can transmit bits quickly (short pulse duration), its high spectral spread leads to reduced spectral efficiency.

SRRC Pulse

Frequency Spectrum:

- The SRRC pulse is designed to confine its spectrum within the Nyquist bandwidth ($\frac{1}{2T}$).
- The roll-off factor β controls how much of the spectrum extends beyond the Nyquist bandwidth:
  - Lower β: Narrower bandwidth but longer pulse duration in time.
  - Higher β: Increased bandwidth but shorter pulse duration in time.

Bandwidth Efficiency:

- The SRRC pulse is more bandwidth-efficient because it is spectrally compact and limits interference with adjacent channels.

2. Effect of Pulse Length

Pulse Length in Bit Periods

- Pulse length refers to how many bit periods T the pulse spans.
- Longer pulse lengths result in more overlap between consecutive pulses, leading to ISI unless the Nyquist criterion is satisfied.

Half-Sine Pulse

Short Pulse Length:

- The half-sine pulse spans exactly 1 bit period, so it introduces no inherent overlap in time.

- However, it does not satisfy the Nyquist criterion, so its spectrum overlaps with adjacent symbols, causing ISI in the frequency domain.

Effect on Performance:

- The short time duration reduces ISI in the time domain, but the wide spectral spread increases ISI in the frequency domain, degrading performance.

SRRC Pulse

Long Pulse Length:

- The SRRC pulse spans multiple bit periods (e.g., $2K \cdot T$), introducing intentional overlap in time.
- However, it satisfies the Nyquist criterion, ensuring zero ISI at the sampling points.

Roll-Off Factor (β):

- Lower β: Narrow bandwidth, longer pulse length, more overlap.
- Higher β: Wider bandwidth, shorter pulse length, less overlap.

Effect on Performance:

- Longer pulse length minimizes ISI while maintaining bandwidth efficiency, though it may require more complex compensating filters.

3. Effect on System Performance

Half-Sine Pulse

Advantages:

- Short pulse length simplifies system implementation.
- Can transmit symbols quickly due to minimal time overlap.

Disadvantages:

- Requires a larger bandwidth, making it less suitable for bandwidth-limited systems.
- Does not suppress ISI, resulting in worse performance in noisy environments or over long-distance channels.

SRRC Pulse

Advantages:

- Highly spectrally efficient, consuming minimal bandwidth.
- Satisfies the Nyquist criterion, ensuring zero ISI at sampling points.
- Performs better in noisy environments due to reduced ISI and better matched filter performance.

Disadvantages:

- Longer pulse duration increases complexity in symbol alignment and filtering.

- Trade-off between bandwidth efficiency (β) and pulse length must be carefully managed.

4. Observations in the Implementation

Bandwidth Comparison:

- The SRRC pulse is more bandwidth-efficient, as observed in the frequency response plots.
- The half-sine pulse shows significant spectral leakage beyond the Nyquist bandwidth.

Pulse Length Trade-Off:

- The SRRC pulse exhibits overlapping symbols in time, but this does not cause ISI at the sampling points.
- The half-sine pulse shows residual ISI, especially under noisy conditions.

Conclusion

- The SRRC pulse is more suitable for practical communication systems due to its bandwidth efficiency and minimal ISI.
- The half-sine pulse, while simpler to implement, suffers from significant ISI and requires a larger transmission bandwidth.
- Pulse length trade-offs in SRRC provide a balance between bandwidth efficiency and system complexity, making it the preferred choice in modern communication systems.

4.3.20

Advantages and Disadvantages of Each Pulse

Half-Sine Pulse

Advantages:

- Simple to implement.
- Short pulse duration minimizes time-domain overlap.

Disadvantages:

- Requires more bandwidth.
- Does not eliminate ISI, leading to degraded performance.
- Poor performance in noisy environments.

SRRC Pulse

Advantages:

- Bandwidth-efficient and spectrally compact.
- Satisfies the Nyquist criterion, ensuring zero ISI.
- Better noise performance due to reduced ISI.

Disadvantages:

- More complex to implement due to longer pulse duration and overlap.

- Requires precise synchronization and filtering.

3. Final Recommendation

Better Pulse Shape:

The SRRC pulse is the better choice because it:

- Is more bandwidth-efficient.
- Satisfies the Nyquist criterion, eliminating ISI.
- Performs better in noisy environments.

Considerations:

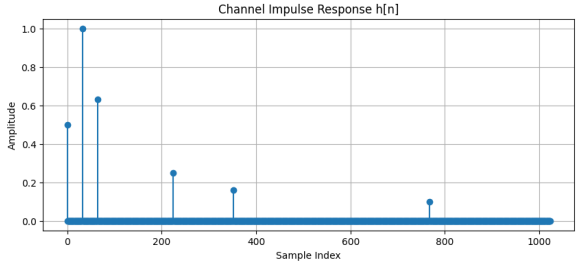
- While the SRRC pulse offers superior performance, it comes with increased complexity and longer processing times.
- The half-sine pulse may be preferred in scenarios where simplicity and implementation speed are more important than spectral efficiency or noise performance.

Conclusion

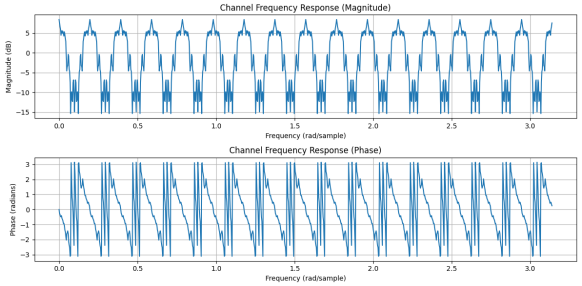
The SRRC pulse is the better choice for most practical communication systems due to its superior bandwidth efficiency, ability to eliminate ISI, and robustness in noisy environments. However, its increased complexity may make the half-sine pulse suitable for simpler systems where bandwidth and noise are not critical constraints.

4.4. Channel Effects

Open Space Channel

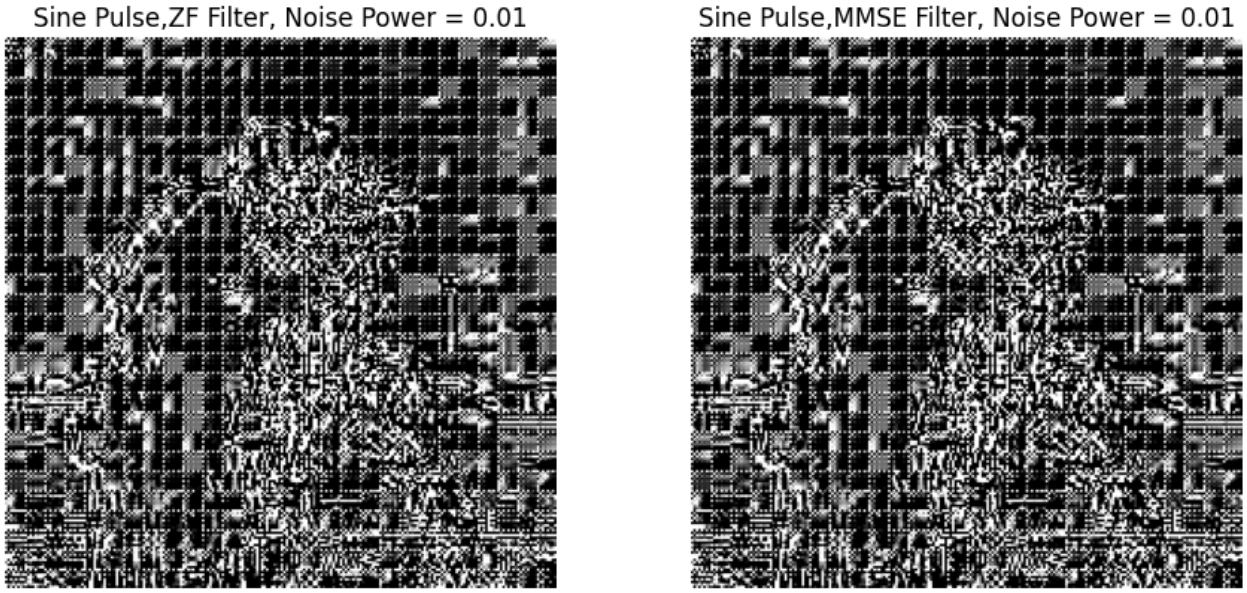


(i) Channe Impulse Response



(j) Channel Frequency Response

Figure 18: Open Space Channel Response



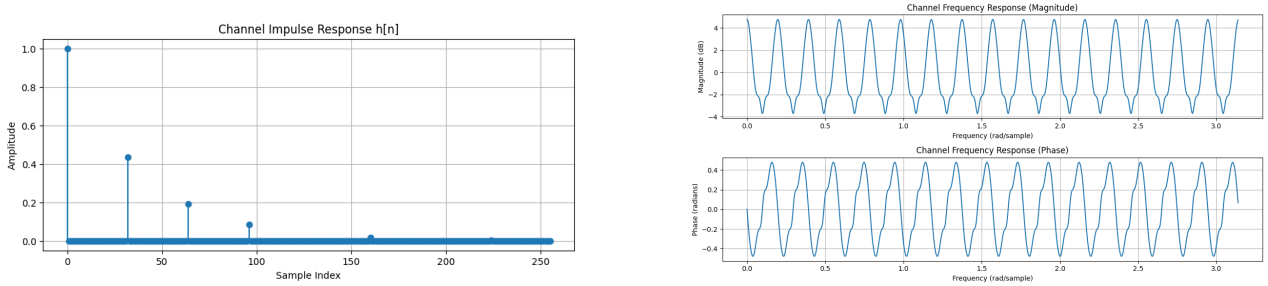
(a) (b)

Figure 19: Open Space Channel Reconstructed Images

Open Space Channel:

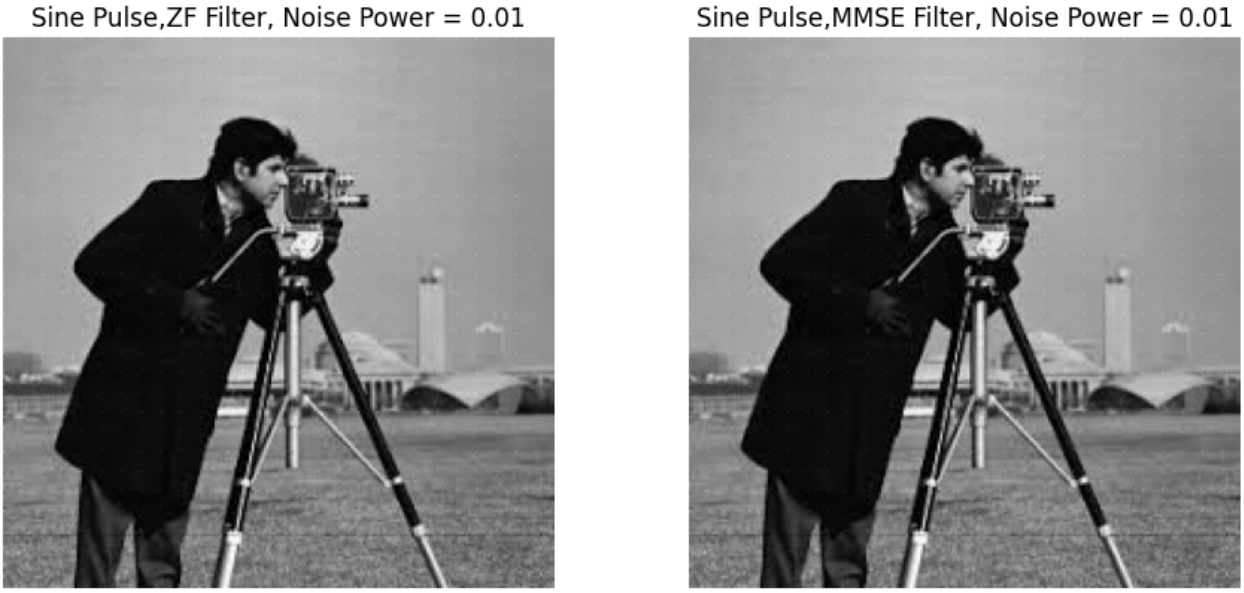
- Requires longer equalizers due to the long channel delays.
- May benefit from increasing the transport block size N to mitigate ISI more effectively.

Indoor Channel



(a) Channe Impulse Response (b) Channel Frequency Response

Figure 20: Indoor Channel Response



(a) (b)

Figure 21: Indoor Channel Reconstructed Images

Indoor Channel:

- Easier to compensate due to shorter delays.
- Performs better with default system parameters.

Results

Frequency Response Analysis

- The open space channel ($h_1[n]$) will show significant "deep drops" in the frequency response due to long delays and multipath effects.
- The indoor channel ($h_2[n]$) will have a smoother frequency response due to shorter delays.

Reconstructed Images

- The reconstructed image for the indoor channel should exhibit better quality because the channel introduces less distortion and ISI.
- The image for the open space channel may show more artifacts unless the compensator handles the long delays effectively.

Key Takeaways

- The indoor channel provides better system performance due to its smoother frequency response and shorter delays.
- The open space channel requires more sophisticated equalizers and longer filter responses to handle its deep frequency drops and long delays.
- Adjusting the transport block size N can help improve performance, especially for the open space channel.