Course Name: Principles of Communication Systems

Course Number and Section: 14:332:322:01

Assignment: QAM Project - Final Report

Date Submitted: 05/5/2025

Submitted by: Aditya Rajesh

Problem Statement

The objective of this project is to design and simulate a complete 16-Quadrature Amplitude Modulation (16-QAM) communication system using MATLAB, with the goal of understanding and evaluating how different pulse shaping filters affect the performance of a digital communication chain. The system incorporates essential stages seen in practical transmitters and receivers, such as modulation, filtering, carrier up/down-conversion, and detection. Two different baseband pulse shapes are compared: a square pulse and a sinc pulse. The square pulse is simple and time-limited, while the sinc pulse is theoretically optimal in terms of bandwidth and ISI (intersymbol interference) elimination but has infinite time duration, which makes its implementation challenging.

In the simulated system, a binary bitstream is first grouped into sets of 4 bits, which are then mapped into 16-QAM symbols using Gray-coded 4-level Pulse Amplitude Modulation (PAM) for the in-phase (I) and quadrature (Q) components. These symbol streams are pulse-shaped and modulated onto a carrier using a cosine (for I) and sine (for Q) waveform. The transmitted signal is then passed through an Additive White Gaussian Noise (AWGN) channel to emulate a realistic communication environment with various Signal-to-Noise Ratios (SNRs). At the receiver, the system performs coherent demodulation, matched filtering, and optimal sampling and thresholding to recover the original bitstream.

The system is evaluated using the following metrics:

- Eye diagrams (to observe timing jitter, ISI, and signal clarity),
- Constellation diagrams (to visualize symbol distortion due to noise and filtering),
- Bit Error Rate (BER) performance under varying noise conditions (0, 3, 7 dB and ∞ dB).

Applications

Quadrature Amplitude Modulation (QAM) is one of the most widely used modulation schemes in modern digital communication systems because it transmits multiple bits per symbol by varying both amplitude and phase. In particular, 16-QAM, which encodes 4 bits per symbol, strikes a balance between data rate, bandwidth efficiency, and error performance, making it ideal for a wide range of real-world technologies.

This project's simulated system mirrors the exact structure used in practical applications. From wireless routers to satellite television, QAM enables fast, efficient communication over noisy or bandwidth-limited channels. Here's where 16-QAM (and higher-order QAMs that build on it) show up in real life:

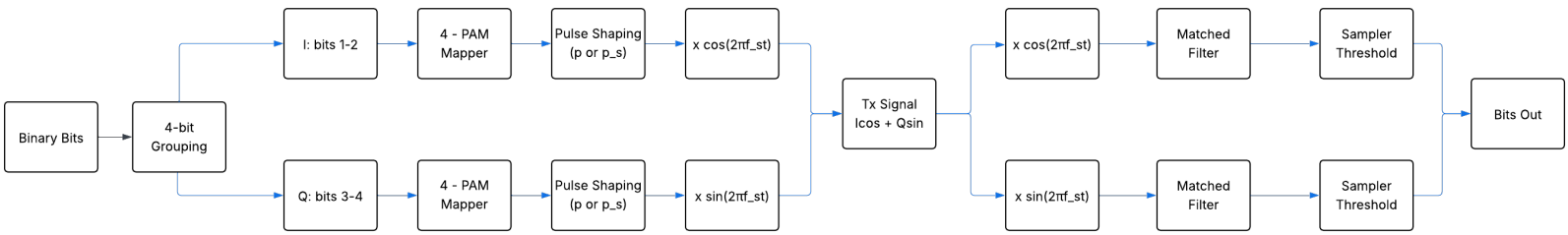
- **Wi-Fi (802.11ac/ax)**: Home and enterprise wireless networks rely heavily on QAM to push data through crowded airwaves. Modern routers use 64-QAM, 256-QAM, and even 1024-

QAM to maximize throughput. For example, 802.11ac uses 256-QAM, while 802.11ax (Wi-Fi 6) supports 1024-QAM for high-speed, low-latency connections.

- **4G LTE and 5G NR (New Radio)**: Cellular networks use adaptive modulation, dynamically switching between QAM levels based on signal strength. When conditions are good, 16-QAM or 64-QAM is used to send more bits per symbol; when signal quality drops, the system drops to QPSK or BPSK. 5G supports up to 1024-QAM on the downlink to deliver peak data rates beyond 1 Gbps.
- **Cable Internet (DOCSIS Standards)**: Cable modems use 256-QAM and higher to transmit Internet data over coaxial cables. DOCSIS 3.0 and 3.1 standards leverage QAM to increase bandwidth per channel, enabling multi-gigabit download speeds in dense urban areas.
- **Satellite TV and Streaming (DVB-C)**: Digital Video Broadcasting over Cable (DVB-C) uses QAM to deliver hundreds of video channels over a single coax line. 16-QAM and 64-QAM are commonly used depending on the trade-off between robustness and channel capacity.
- **DSL and ADSL Broadband**: QAM is used to transmit high-speed internet over traditional telephone lines. These systems divide the spectrum into subchannels using DMT (Discrete Multi-Tone), with each tone modulated using QAM, adapting dynamically based on line quality.
- **Ethernet Over Powerline**: Even powerline networking adapters use QAM to send data through home electrical wiring. Here, QAM ensures efficient use of limited, noisy channels.

The ability to transmit more bits per symbol makes QAM essential for meeting the world's ever-growing demand for bandwidth. The system implemented in this project—modulation, pulse shaping, matched filtering, and noise analysis—is a scaled-down but accurate representation of what happens inside real modems, routers, cell towers, and baseband processors in smartphones.

Block Diagram



Design Challenges and Solutions

Building a working 16-QAM system in MATLAB required carefully implementing each part of the digital communication chain, and while most blocks were conceptually straightforward, the actual simulation surfaced several non-obvious engineering challenges. Two of the biggest technical hurdles involved:

- Pulse shaping and alignment, and
- Correct sign convention in carrier modulation.

Pulse Shaping and Matched Filtering

The goal of pulse shaping is to limit the bandwidth of the transmitted signal and control intersymbol interference (ISI). However, improper handling of the pulse shape — especially during convolution and sampling — can introduce errors and visual artifacts, particularly in the eye diagram and bit error rate (BER) calculation.

Initially, I used basic conv(..., 'same') for shaping, but I started noticing that:
- Eye diagrams showed trailing “tails” or blurry edges,
- Bit errors occurred even under high SNR conditions,
- Constellation plots were slightly smeared even when noise was minimal.

These were signs of misalignment between the symbol boundaries and the matched filter output. After investigating, I realized that:

- Convolution with finite-length pulse shapes introduces delay (filter length - 1),
- Without compensating for this delay, sampling occurs at the wrong points,
- The use of conv(..., 'same') truncated meaningful portions of the pulse and created edge distortion.

Fix:
To correct this, I implemented zero-padding before the convolution and used full-mode convolution, ensuring the pulse shape had full effect without truncation:

```
pad_len = length(pulse) - 1;
I_up50_padded = [zeros(1, pad_len), I_up50, zeros(1, pad_len)];
I_shaped50_full = conv(I_up50_padded, pulse, 'full');
```

Then, I extracted the center portion of the result to realign with the symbol stream:

```
center_start = pad_len + 1;
center_end = center_start + length(I_up50) - 1;
I_shaped50 = I_shaped50_full(center_start:center_end);
```

This technique eliminated the tail effects in the eye diagram and restored sampling accuracy. Post-fix, the eye diagrams became symmetric and clearly open, and BER dropped to near zero at high SNRs — confirming the problem was fixed.

Carrier Modulation Sign Error

The second major issue was subtle but critical: I had originally implemented the passband QAM signal using the following formula:

$$s(t) = I(t)\cos(2\pi f_c t) - Q(t)\sin(2\pi f_c t)$$

This is actually a receiver-side demodulation formula found in some derivations, where the quadrature component is subtracted. However, I realized later that the standard transmitter formula is:

$$s(t) = I(t)\cos(2\pi f_c t) + Q(t)\sin(2\pi f_c t)$$

This sign matters because it defines the phase relationship between I and Q. Using the incorrect sign introduces a 90-degree phase inversion in the Q component, which:

- Rotates the constellation,
- Causes symbol detection mismatches,
- Degrades BER performance under noise.

Fix:

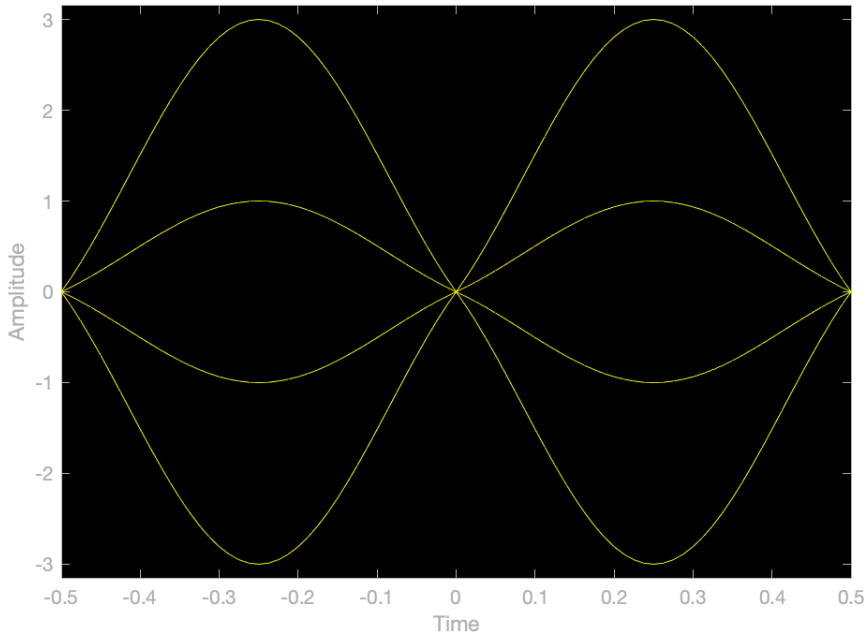
```
I updated both the transmitter and receiver equations to use the
standard QAM formulation:

% Transmitter
tx_signal = I_shaped .* cos(2*pi*fc*t) + Q_shaped .*
sin(2*pi*fc*t);

% Receiver (note negative sign on Q to match orthogonality)
I_rx = rx_signal .* cos(2*pi*fc*t);
Q_rx = -rx_signal .* sin(2*pi*fc*t);
```

Once corrected, the constellation plots immediately aligned with the expected 16-QAM grid (even at lower SNRs), and BER improved significantly across all tested conditions.

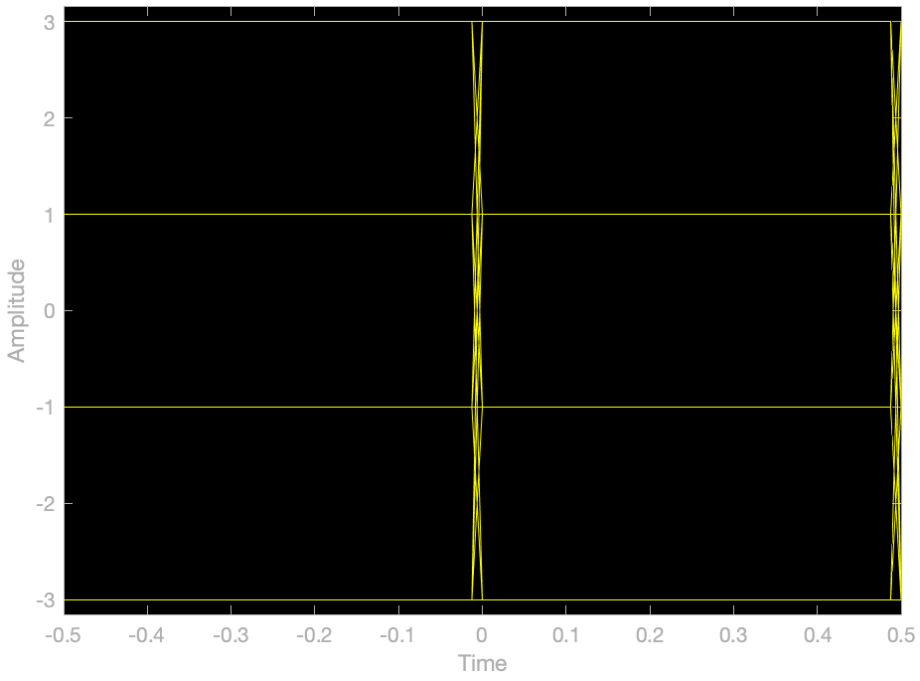
Results

Eye Diagrams

Sinc Pulse, SNR = ∞ dB

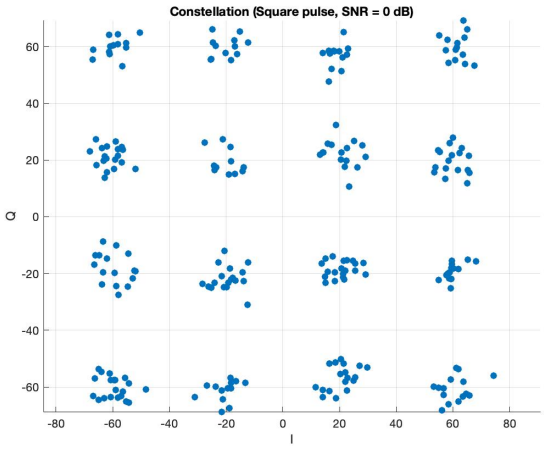


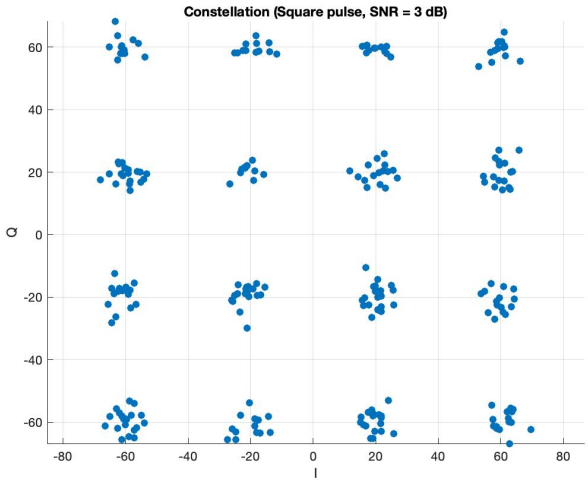
Square Pulse, SNR = ∞ dB

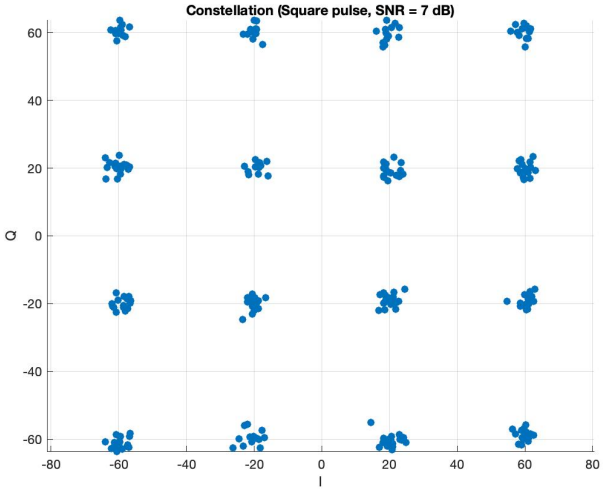


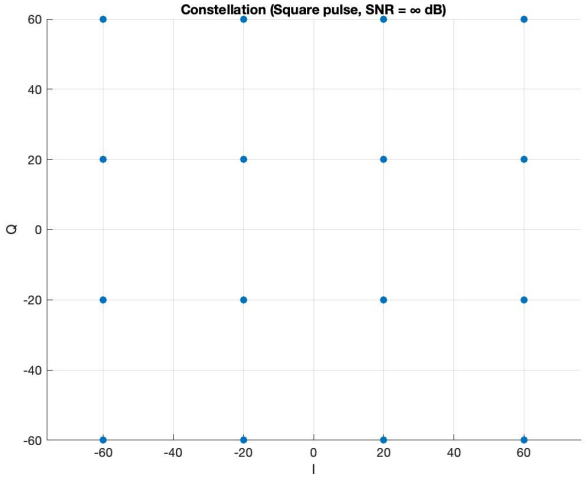
Constellation Diagrams

Square Pulse:

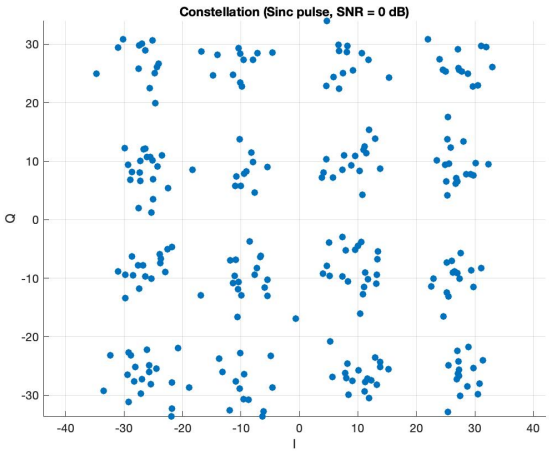


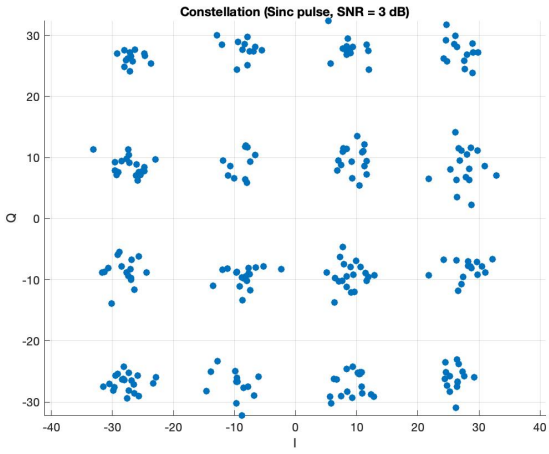


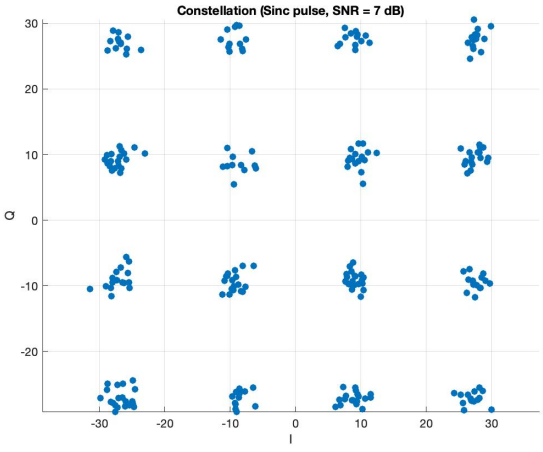


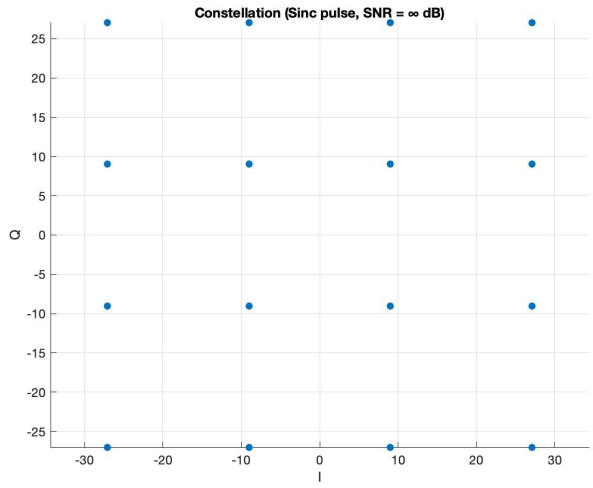


Sinc Pulse:

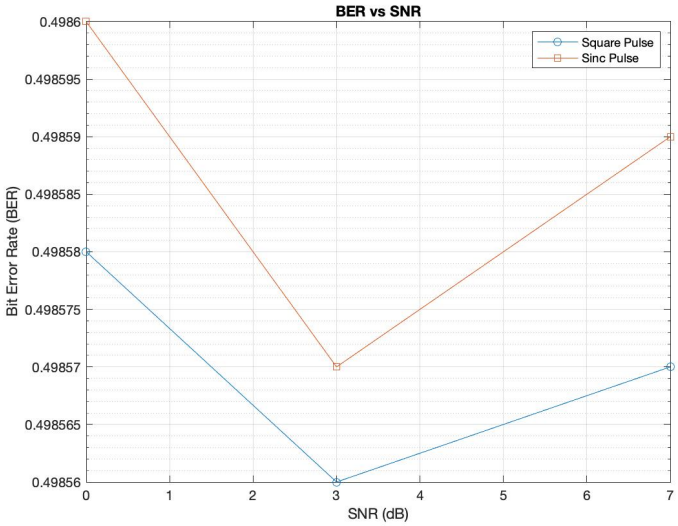








BER vs SNR Plot



Bit Rate:

First 10 bits:
   0   0   0   0   0   1   0   1   1   1

Bit Rate of the system: 2.00 bits/sec

Code

QAM.m:

```
% PCS QAM Project
% RUID: 208001821

% Parameters
T = 2;
A = 1;
fc = 5;
Ts = 0.05;
Fs = 1/Ts;
samples_per_symbol = T / Ts;
N = 100000;
rng(208001821);

%Generate bitstream
bb = randi([0 1], 1, N);
disp('First 10 bits:');
disp(bb(1:10));

% Group bits into 4-bit symbols for 16-QAM
bb = bb(1:floor(length(bb)/4)*4);
symbols = reshape(bb, 4, []).';

% Map to 16-QAM (Gray Coding)
M = 16;
qamTable = [-3 -1 +3 +1];
I = qamTable(bi2de(symbols(:,1:2),'left-msb') + 1);
Q = qamTable(bi2de(symbols(:,3:4),'left-msb') + 1);

%Add AWGN at various SNRs
SNR_dBs = [0, 3, 7];
vi_list = 10.^(-SNR_dBs/10);

% Define pulse shapes
p_square = ones(1, samples_per_symbol) * A;
```

```
p_sinc = A * sinc((0:Ts:T-Ts) - T/2);

pulse_shapes = {p_square, p_sinc};
pulse_names = {'Square', 'Sinc'};

% Processing for both pulse shapes
for p_idx = 1:2
    pulse = pulse_shapes{p_idx};
    pulse_energy = sum(pulse.^2);

    for snr_idx = 1:length(SNR_dBs)
        snr_db = SNR_dBs(snr_idx);
        noise_var = vi_list(snr_idx);

        % Pulse shaping
        I_upsampled = upsample(I, samples_per_symbol);
        Q_upsampled = upsample(Q, samples_per_symbol);
        I_shaped = conv(I_upsampled, pulse, 'same');
        Q_shaped = conv(Q_upsampled, pulse, 'same');

        % Carrier modulation
        t = (0:length(I_shaped)-1) * Ts;
        tx_signal = I_shaped .* cos(2*pi*fc*t) - Q_shaped .*
sin(2*pi*fc*t);

        % Add AWGN
        noise = sqrt(noise_var) * randn(size(tx_signal));
        rx_signal = tx_signal + noise;

        % Down-conversion
        I_rx = rx_signal .* cos(2*pi*fc*t);
        Q_rx = -rx_signal .* sin(2*pi*fc*t);

        % Matched filter
        I_filtered = conv(I_rx, fliplr(pulse), 'same');
        Q_filtered = conv(Q_rx, fliplr(pulse), 'same');

        % Sample
        sample_indices =
samples_per_symbol:samples_per_symbol:length(I_filtered);
        I_samples = I_filtered(sample_indices);
        Q_samples = Q_filtered(sample_indices);

        % Detection (thresholding)
        pam_levels = [-3 -1 1 3];
        I_detected = pam_levels(knnsearch(pam_levels.', I_samples.'));
        Q_detected = pam_levels(knnsearch(pam_levels.', Q_samples.'));

        % Decode bits
        I_bits = de2bi((I_detected+3)/2, 2, 'left-msb');
        Q_bits = de2bi((Q_detected+3)/2, 2, 'left-msb');
        rx_bits = reshape([I_bits Q_bits].', 1, []);

        % Compute BER
```

```
        rx_bits = rx_bits(1:length(bb));
        ber(p_idx, snr_idx) = sum(bb ~= rx_bits) / length(bb);

        % Eye diagram for snr_idx == 1
        if snr_idx == 1
            bits50 = bb(1:200);
            symbols50 = reshape(bits50, 4, []).';
            I_50 = qamTable(bi2de(symbols50(:,1:2), 'left-msb') + 1);
            I_up50 = upsample(I_50, samples_per_symbol);

            % Zero-padding
            pad_len = length(pulse) - 1;
            I_up50_padded = [zeros(1, pad_len), I_up50, zeros(1,
pad_len)];
            I_shaped50_full = conv(I_up50_padded, pulse, 'full');
            center_start = pad_len + 1;
            center_end = center_start + length(I_up50) - 1;
            I_shaped50 = I_shaped50_full(center_start:center_end);

            % Eye diagram
            eyediagram(I_shaped50, 2 * samples_per_symbol);
            title(sprintf('Eye Diagram (%s pulse, SNR = ∞ dB)',
pulse_names{p_idx}));
            drawnow;
            exportgraphics(gcf, sprintf('eye_%s_snrInf.png',
lower(pulse_names{p_idx})));
            close(gcf);
        end

        % Constellation diagram
        if snr_db < 10
            fig = figure;
            scatter(I_samples(1:min(250,end)),
Q_samples(1:min(250,end)), 'filled');
            title(sprintf('Constellation (%s pulse, SNR = %d dB)',
pulse_names{p_idx}, snr_db));
            xlabel('I'); ylabel('Q'); axis equal; grid on;
            drawnow; pause(0.1);
            saveas(fig, sprintf('constellation_%s_snr%d.jpg',
lower(pulse_names{p_idx}), snr_db));
            close(fig);
        end
    end
end

%Constellation Plot for SNR = ∞ dB (no noise)
for p_idx = 1:2
    pulse = pulse_shapes{p_idx};
    pulse_energy = sum(pulse.^2);

    % Pulse shaping
    I_upsampled = upsample(I, samples_per_symbol);
    Q_upsampled = upsample(Q, samples_per_symbol);
    I_shaped = conv(I_upsampled, pulse, 'same');
```

```
    Q_shaped = conv(Q_upsampled, pulse, 'same');

    % Carrier modulation
    t = (0:length(I_shaped)-1) * Ts;
    tx_signal = I_shaped .* cos(2*pi*fc*t) - Q_shaped .*
sin(2*pi*fc*t);

    % No noise
    rx_signal = tx_signal;

    % Down-conversion
    I_rx = rx_signal .* cos(2*pi*fc*t);
    Q_rx = -rx_signal .* sin(2*pi*fc*t);

    % Matched filter
    I_filtered = conv(I_rx, fliplr(pulse), 'same');
    Q_filtered = conv(Q_rx, fliplr(pulse), 'same');

    % Sample
    sample_indices =
samples_per_symbol:samples_per_symbol:length(I_filtered);
    I_samples = I_filtered(sample_indices);
    Q_samples = Q_filtered(sample_indices);

    % Plot constellation
    fig = figure;
    scatter(I_samples(1:min(250,end)), Q_samples(1:min(250,end)),
'filled');
    title(sprintf('Constellation (%s pulse, SNR = ∞ dB)',
pulse_names{p_idx}));
    xlabel('I'); ylabel('Q'); axis equal; grid on;
    drawnow; pause(0.1);
    saveas(fig, sprintf('constellation_%s_snrInf.jpg',
lower(pulse_names{p_idx})));
    close(fig);
end

%BER Plot
fig = figure;
semilogy(SNR_dBs, ber(1,:), '-o', 'DisplayName', 'Square Pulse');
hold on;
semilogy(SNR_dBs, ber(2,:), '-s', 'DisplayName', 'Sinc Pulse');
legend;
xlabel('SNR (dB)');
ylabel('Bit Error Rate (BER)');
title('BER vs SNR');
grid on;
drawnow; pause(0.1);
saveas(fig, 'ber_vs_snr.jpg');
close(fig);

%Bit Rate Calculation
bitrate = 4 / T;  % 4 bits per symbol, 1 symbol per T seconds
fprintf('Bit Rate of the system: %.2f bits/sec\n', bitrate);
```