Communication Theory

Communication Model

E.Druvitha
2022102029

P.Jyothi sri
2022102056

Introduction

Building the following communication model

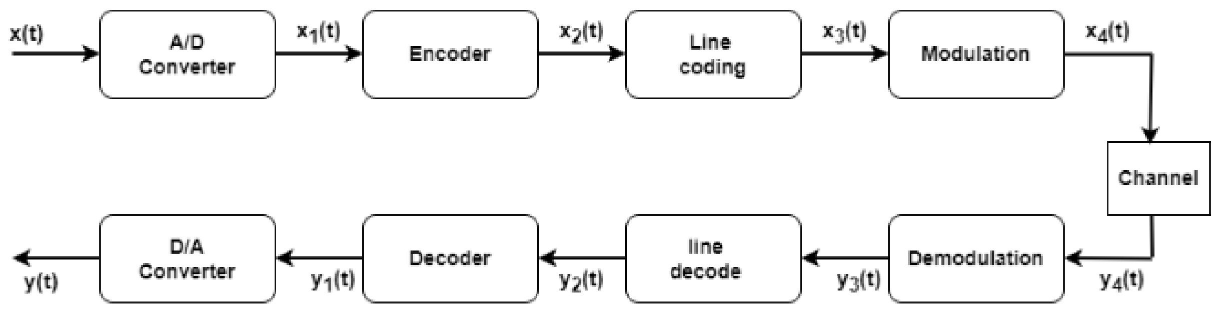
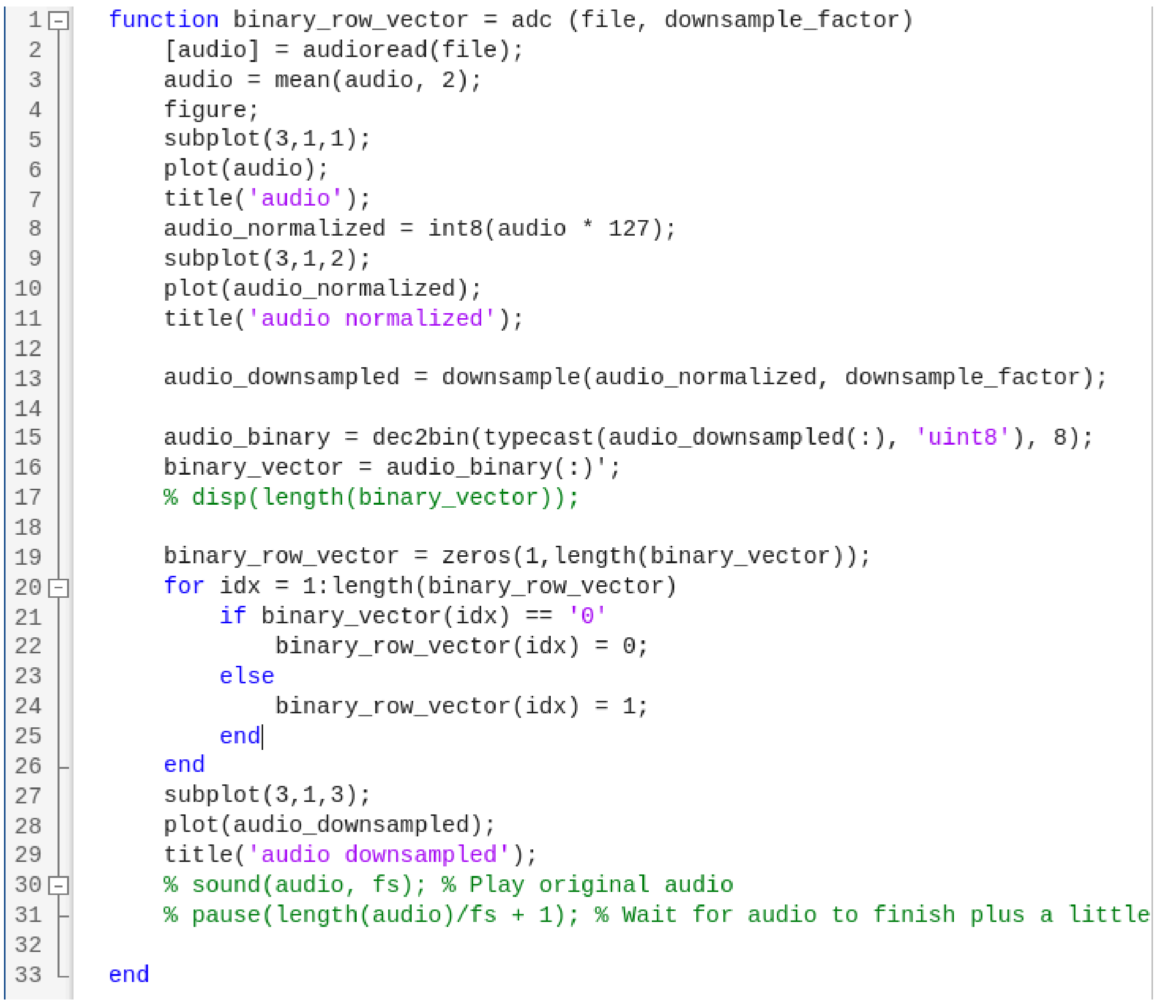


Figure 1: System model

1. ADC (Analog-to-Digital Converter): Converts analog signals into digital data for processing or transmission.
2. Encoder: Converts digital data into specific symbols for efficient transmission.
3. Line coding: Maps digital data to physical signals suitable for transmission over a communication channel. Essentially involves a pulse p(t).
4. Modulation: Embeds digital data into a carrier signal for efficient transmission through a communication medium.
5. Channel: The medium through which signals (modulated data) travel from sender to receiver. There is noise to the signal through this medium.
6. Demodulation: Extracts digital data from a modulated carrier signal at the receiver's end.
7. Line decoder: Converts received physical signals back into digital data after demodulation.
8. Decoder: Translates encoded digital data (symbols) back into bits.
9. DAC (Digital-to-Analog Converter): Converts digital signals back into analog form for output or display.

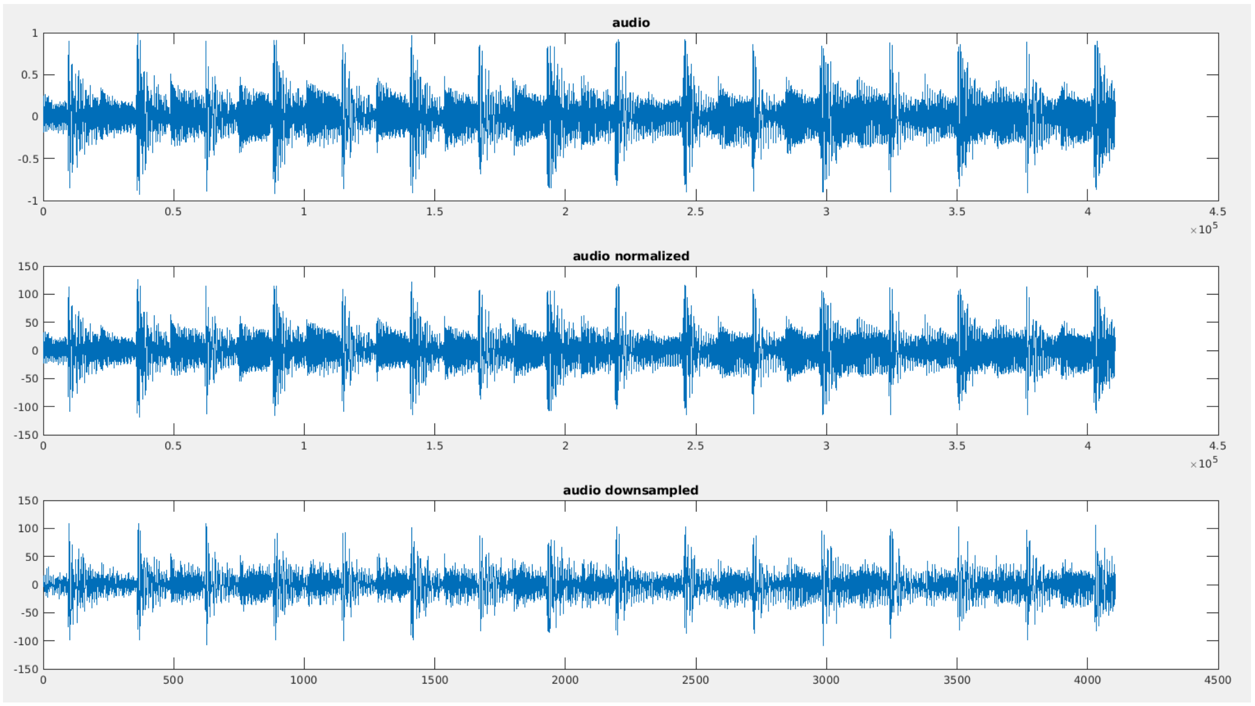
ADC

In this block, we convert the .wav file and convert it into bits as an input to the model.



```
function binary_row_vector = adc (file, downsample_factor)
    [audio] = audioread(file);
    audio = mean(audio, 2);
    figure;
    subplot(3,1,1);
    plot(audio);
    title('audio');
    audio_normalized = int8(audio * 127);
    subplot(3,1,2);
    plot(audio_normalized);
    title('audio normalized');

    audio_downsampled = downsample(audio_normalized, downsample_factor);

    audio_binary = dec2bin(typecast(audio_downsampled(:), 'uint8'), 8);
    binary_vector = audio_binary(:)';
    % disp(length(binary_vector));

    binary_row_vector = zeros(1,length(binary_vector));
    for idx = 1:length(binary_row_vector)
        if binary_vector(idx) == '0'
            binary_row_vector(idx) = 0;
        else
            binary_row_vector(idx) = 1;
        end
    end
    subplot(3,1,3);
    plot(audio_downsampled);
    title('audio downsampled');
    % sound(audio, fs); % Play original audio
    % pause(length(audio)/fs + 1); % Wait for audio to finish plus a little extra

end
```

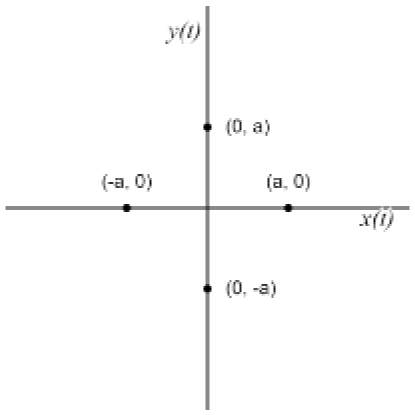
`audioread(file)` reads data from the file named `file`, and returns sampled data, `audio`, and a sample rate for that data, `Fs`. Since, the given audio is stereo (2 channels) we convert it into mono by taking the mean of the two. The audioread function returns normalized data (-1 to 1), thus we amplify it wrt int8. Also, since the given audio is long, we downsampled the audio by a factor of 100 (we get around 30000 bits, which takes some time to give outputs). We finally convert this into a binary vector for further processing.



Encoder

This block is responsible for conversion of the bit stream/sequence into a sequence of symbols a_k.

Given below is the constellation of QPSK:



(a) QPSK type 1

Mapping: (Gray coding for lesser error)

00 — [a,0]

01 — [0,a]

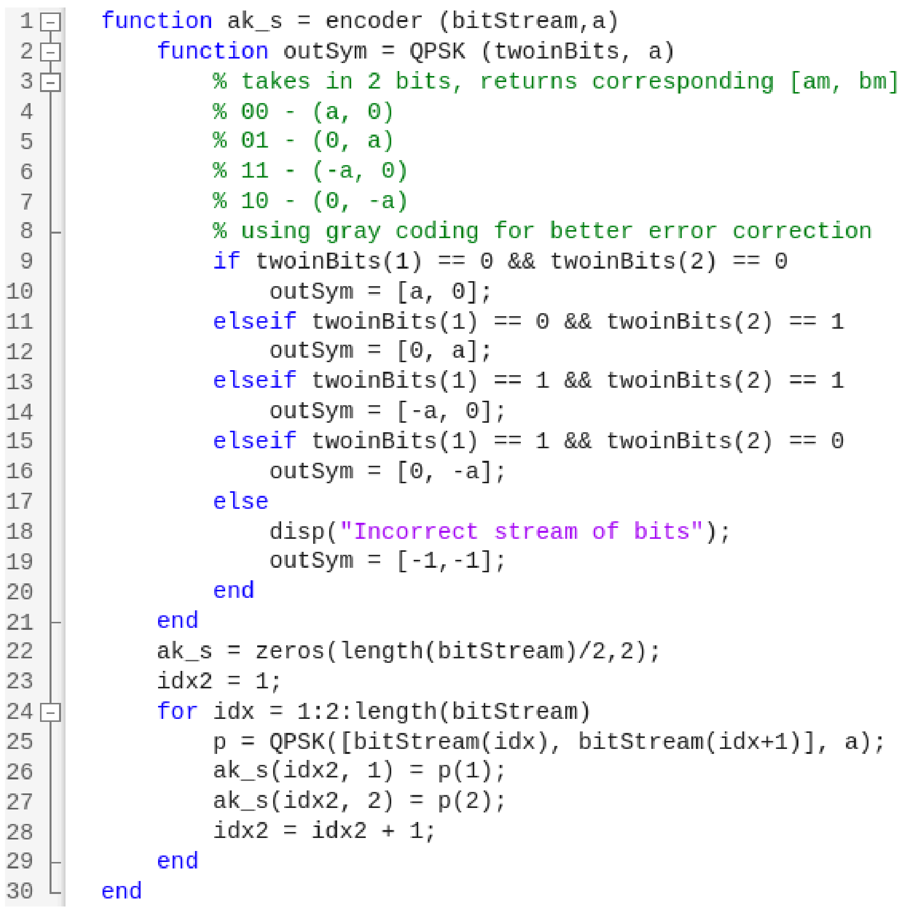
11 — [-a,0]

10 — [0,-a]

Input Arguments: bitStream, a

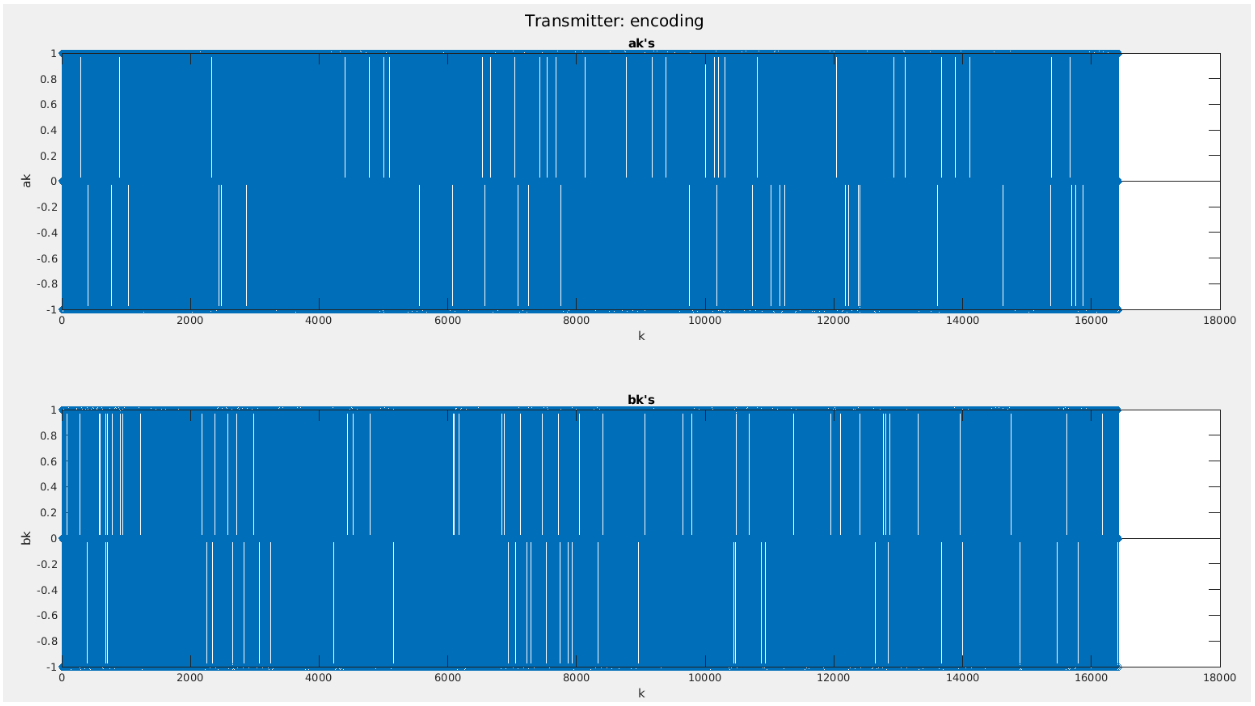
Output: ak_s : matrix of number_of_symbols x 2, where column 1 consists of a_m's and column 2 consists of b_m's.

a_0	b_0
a_1	b_1
a_2	b_2
.	.
.	.
a_k	b_k



```
function ak_s = encoder (bitStream,a)
    function outSym = QPSK (twoinBits, a)
        % takes in 2 bits, returns corresponding [am, bm]
        % 00 - (a, 0)
        % 01 - (0, a)
        % 11 - (-a, 0)
        % 10 - (0, -a)
        % using gray coding for better error correction
        if twoinBits(1) == 0 && twoinBits(2) == 0
            outSym = [a, 0];
        elseif twoinBits(1) == 0 && twoinBits(2) == 1
            outSym = [0, a];
        elseif twoinBits(1) == 1 && twoinBits(2) == 1
            outSym = [-a, 0];
        elseif twoinBits(1) == 1 && twoinBits(2) == 0
            outSym = [0, -a];
        else
            disp("Incorrect stream of bits");
            outSym = [-1,-1];
        end
    end
    ak_s = zeros(length(bitStream)/2,2);
    idx2 = 1;
    for idx = 1:2:length(bitStream)
        p = QPSK([bitStream(idx), bitStream(idx+1)], a);
        ak_s(idx2, 1) = p(1);
        ak_s(idx2, 2) = p(2);
        idx2 = idx2 + 1;
    end
end
```

Plot for the given audio file (output of encoder for the adc output):



More about QPSK Type - 1:

SNR per bit $\Rightarrow E_b/N_0$ Assuming AWGN Memoryless

Lets Find E_b

$$E_b = \frac{E_s}{\log_2 M} \quad \text{for QPSK } M = 4$$

$$\Rightarrow E_b = \frac{E_s}{\log_2 4} = \frac{E_s}{2} \text{ ——— } (1)$$

$$E_s = \frac{\sum E_{si}}{M}$$

$E_{si} \rightarrow$ Energy of signal transmitted for i^{th} Symbol

$00 \longrightarrow a\,p(t)\cos\omega_c t \rightarrow s_1$

$01 \longrightarrow a\,p(t)\sin\omega_c t \rightarrow s_2$

$10 \longrightarrow -a\,p(t)\sin\omega_c t \rightarrow s_3$

$11 \longrightarrow -a\,p(t)\cos\omega_c t \rightarrow s_4$

$$\epsilon_{s_1} = \int_{-\infty}^{\infty} (a p(t)\cos\omega_c t)^2\,dt = \int_0^{T_s} a^2 (p(t))^2 \cos^2\omega_c t\,dt$$

$$= \int_0^{T_s} a^2 (p(t))^2 \left(\frac{1+\cos 2\omega_c t}{2}\right) dt$$

$$\Rightarrow \frac{a^2}{2}\int_0^{T_s} (p(t))^2\,dt + \frac{a^2}{2}\int_0^{T_s} \cancelto{0}{(p(t))^2 \cos(2\omega_c t)\,dt}$$

$$\epsilon_{s_1} = \frac{a^2}{2}\epsilon_p$$

Similarly $\epsilon_{s_1} = \epsilon_{s_2} = \epsilon_{s_3} = \epsilon_{s_4} = \frac{a^2 \epsilon_p}{2}$

as $\omega_c T_s >> 1$ we can ignore $\int_0^{T_s} (p(t))^2 \cos(2\omega_c t)\,dt$

$$\therefore \quad \epsilon_s = \frac{\epsilon_{s_i} \times 4}{4} = \epsilon_{s_i} = \frac{a^2 \epsilon_p}{2}$$

$\Rightarrow$ from ① $\quad \epsilon_b = \frac{\epsilon_s}{2} = \frac{a^2\epsilon_p}{4}$

$$\Rightarrow \boxed{\epsilon_b = \frac{a^2\epsilon_p}{4}} \qquad \text{where } \epsilon_p = \int_0^{T_s} (p(t))^2\,dt$$

Taking $a = 1$, $T_s = 2$, $N_0 = 2$

(i) $p(t) \rightarrow$ Rect

$\Rightarrow \quad E_p = T_s$

So $\quad E_b = \frac{(1)\, T_s}{4} = \frac{T_s}{4} = 0.5$

(ii) $p(t) \longrightarrow$ Raised Cosine

$E_p = 1.75$

```
% Calculate the energy of p_t2 using numerical integration
integrand_squared = @(tt) (sin(pi*tt/Ts)./tt *Ts/pi.*cos(pi*rolloff*tt/Ts)./(1-4*(rolloff^2)*tt.^2/(Ts^2)
energy_p_t2 = integral(integrand_squared, -inf, inf);
```

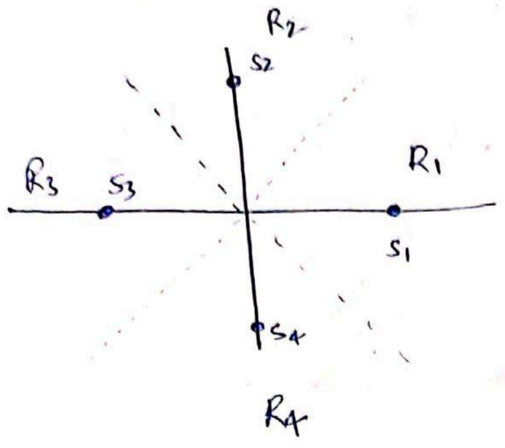
yields `1.7500`

So $\quad E_b = \frac{(1) \times 1.75}{4} \doteq 0.4375$

So for $p(t) =$ Rect $\quad$ SNR is $\Rightarrow \underline{\underline{\frac{0.5}{2} \Rightarrow 0.25}}$

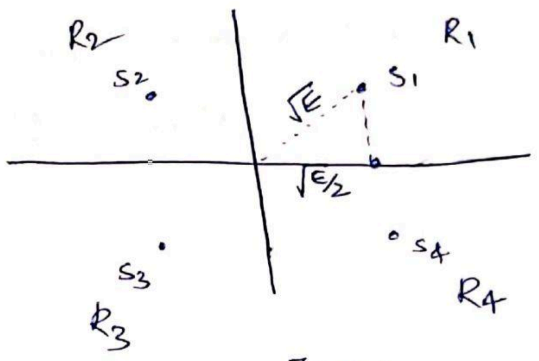
for $p(t) =$ Raised Cosine $\quad$ SNR is $\Rightarrow \underline{\underline{\frac{0.4375}{2} = 0.21875}}$

BER



Type-1

When you shift/rotate this by 45° anticlockwise the Regions shift but probability Remains same. Thus, finding ~~BE~~ BER of type1 is Equivalent to finding BER of type-2



Type-2

$$P(c/m_1) = P(s_1 + n_1 \in R_1)$$

$$\Rightarrow \quad = P\left(\sqrt{\frac{E}{2}} + n_1 > 0,\ \sqrt{\frac{E}{2}} + n_2 > 0\right)$$

$$\Rightarrow P(c/m_1) = P\left(-\sqrt{\frac{E}{2}} < n_1\right) P\left(n_2 > -\sqrt{\frac{E}{2}}\right)$$

$$= \left(1 - Q\left(\sqrt{\frac{E_s}{2\sigma_n^2}}\right)\right)^2$$

$$P(C/m_1) = \left(1 - Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\right)^2$$

$$\Rightarrow P(c) = \sum_{i=1}^{4} P(C/m_i)\, P(m_i) = \left(1 - Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\right)^2$$

$$\Rightarrow BER \Rightarrow P_b = 1 - \left(1 - Q\left(\sqrt{\frac{2E_b}{N_0}}\right)\right)^2$$

$$\underline{\text{Power}} \Rightarrow \frac{E_s}{T_s} = \frac{\text{Energy per symbol}}{\text{Symbol Period}}$$

$$\Rightarrow \boxed{\frac{a^2 E_p}{2\,T_s} = \text{Power}} \quad \text{where } E_p = \int_0^{T_s} (p(t))^2 dt$$

for $p(t) \rightarrow$ Rect and $a = 1,\ T_s = 2$

$\Rightarrow$ we have Power $= \underline{\underline{\frac{1 \times 2}{2 \times 2} \rightarrow 0.5}}$

$p(t) \rightarrow$ Raised Cosine $(E_p = 1.75)$

Power $= \underline{\underline{\frac{1 \times 1.75}{2 \times 2} \Rightarrow 0.4375}}$

The power efficiency of QPSK modulation stems from its ability to achieve a reliable BER in noisy environments, its spectral efficiency in utilizing available bandwidth, and its adaptability to varying channel conditions.QPSK modulation demonstrates strong power efficiency by achieving a BER in the 10^(-3) range over an AWGN channel. Its spectral efficiency, enabled by transmitting two bits per symbol, maximizes data throughput without increasing bandwidth. QPSK strikes a balance between spectral and power efficiency, outperforming higher-order modulations in noise immunity and power consumption per bit. Additionally, its adaptability to channel conditions through techniques like adaptive modulation further enhances power efficiency, making it a preferred choice for various communication applications.

LineCoding

This block is generates the output

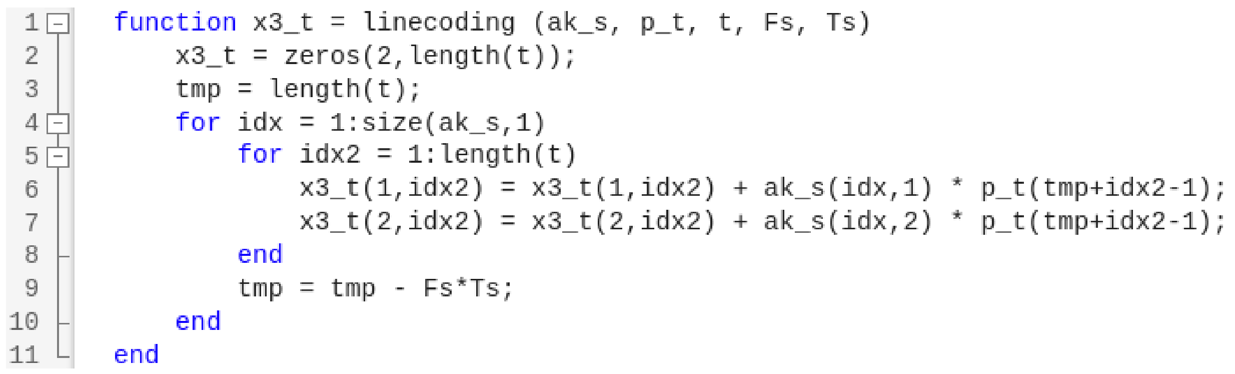
$$\mathbf{x}_3(t) = \sum_k a_k p(t - kT_b),$$

Since, we have two basis functions in QPSK,

Input Arguments: ak_s (output of the encoder), p_t (pulse defined over -n*Ts to n*Ts), t (0:1/Fs:n*Ts), Fs (sampling frequency), Ts (symbol time).

Here, n is the number of symbols = number of bits / 2.

Output: A matrix 2 x length(t) = 2 x n*Fs*Ts where row 1 corresponds to $\Sigma\, a_k p(t - kT_s)$ and row 2 corresponds to $\Sigma\, b_k p(t - kT_s)$ for all k.



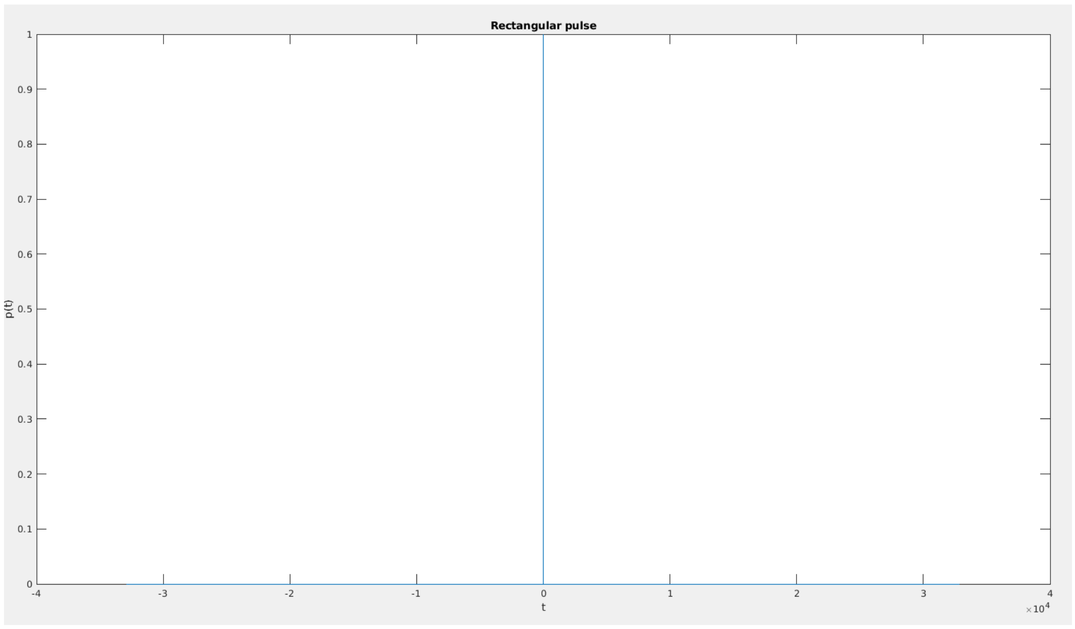
```
function x3_t = linecoding (ak_s, p_t, t, Fs, Ts)
    x3_t = zeros(2,length(t));
    tmp = length(t);
    for idx = 1:size(ak_s,1)
        for idx2 = 1:length(t)
            x3_t(1,idx2) = x3_t(1,idx2) + ak_s(idx,1) * p_t(tmp+idx2-1);
            x3_t(2,idx2) = x3_t(2,idx2) + ak_s(idx,2) * p_t(tmp+idx2-1);
        end
        tmp = tmp - Fs*Ts;
    end
end
```

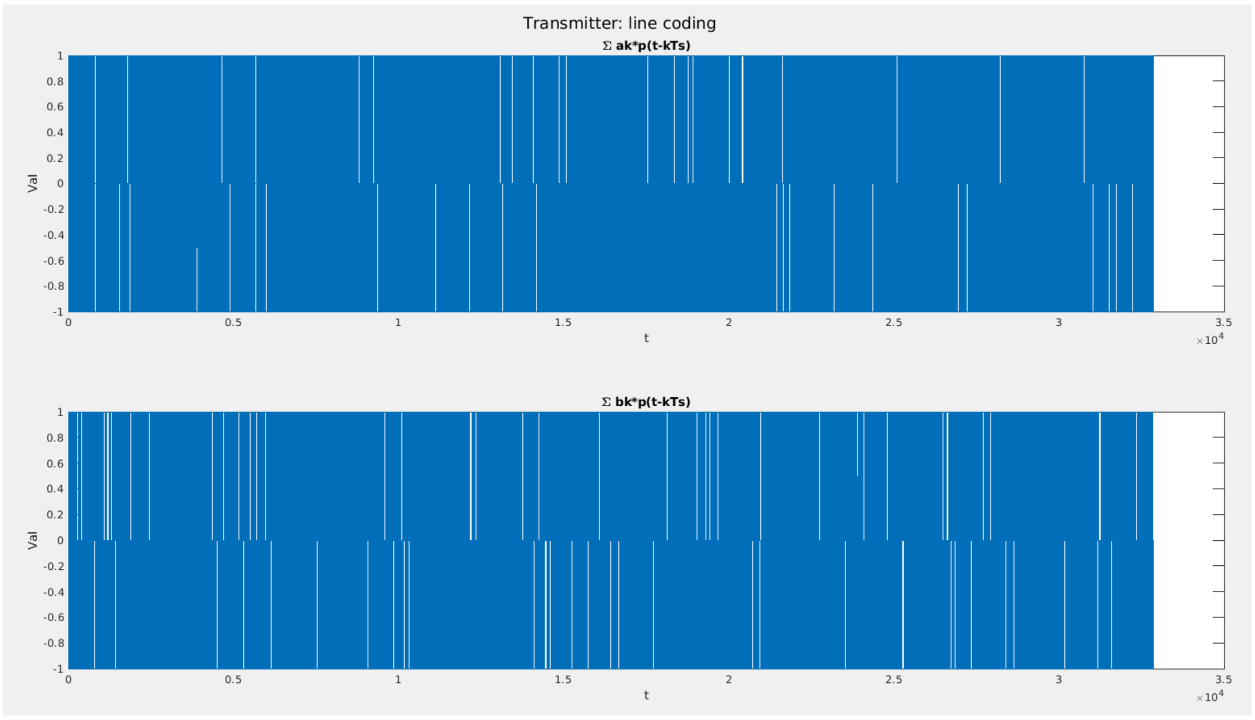
This code performs line coding by multiplying each symbol in `ak_s` with a pulse shape `p_t` shifted over time and sums these products to create a modulated waveform `x3_t` with two channels.

For Fs = 100, Ts = 2, a = 1,

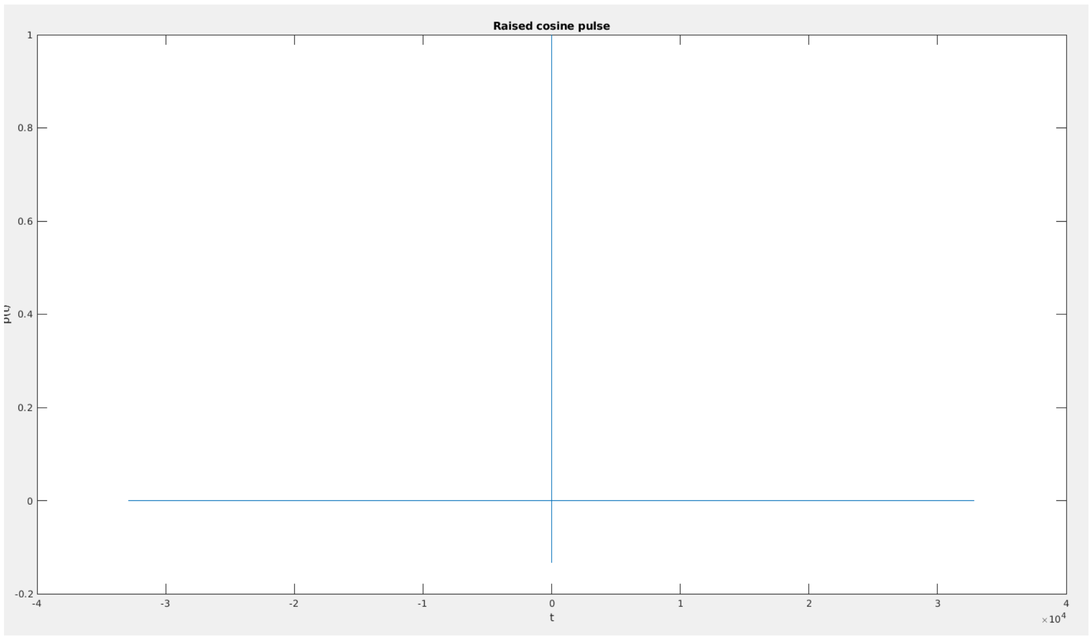
given below are the plots of pulses and their corresponding line coder output:

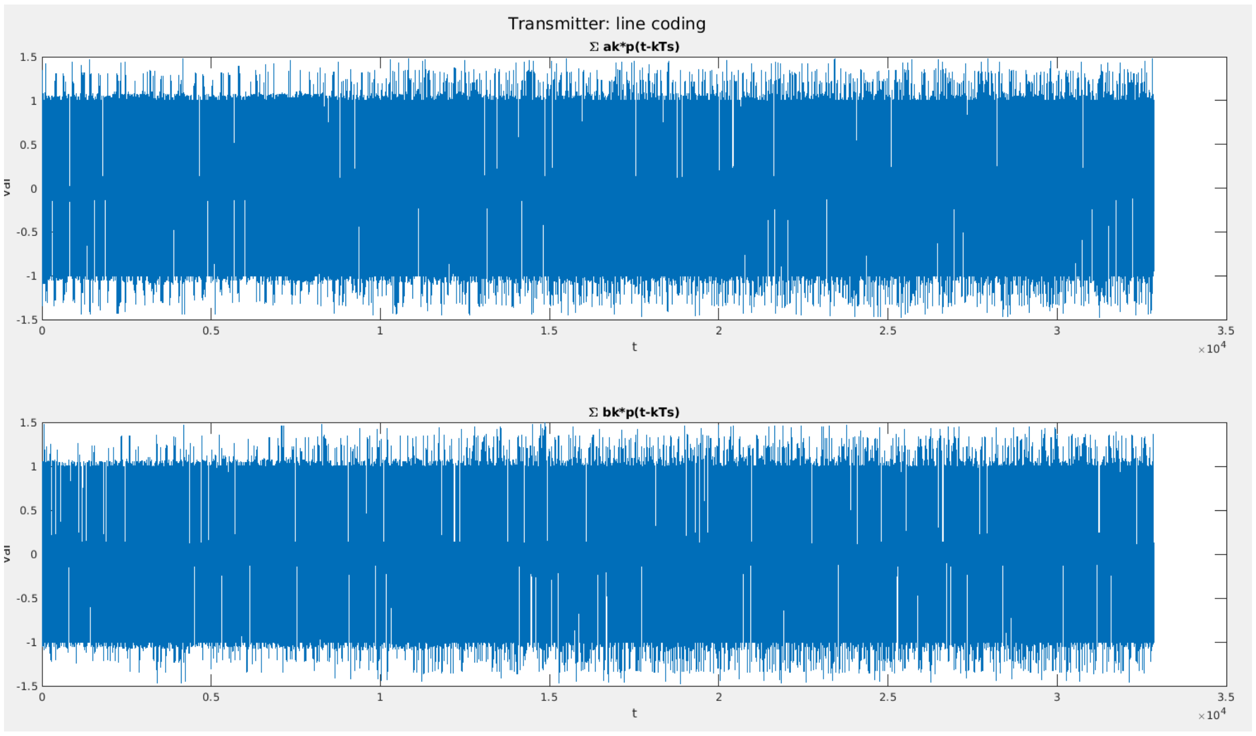
For rectangular pulse:





For raised cosine pulse:





Modulation

In this block we modulate the output of the line coding block to obtain the signal which can be passed to the channel.

For QPSK, (phase modulation), we get following output for modulation:

$$\Sigma\, a_k p(t - kT_s)\cos(w_c t) \;+\; \Sigma\, b_k p(t - kT_s)\sin(w_c t)$$

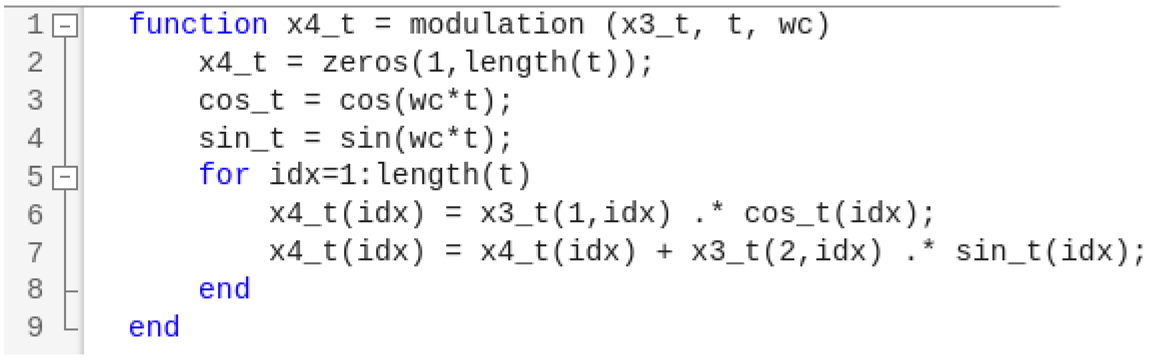
This is the equation since for QPSK (or M-PSK, in general) the basis functions are cos(w_ct) and sin(w_ct). This equation is similar to QAM, except the condition that a_k and b_k are such that they lie on a circle (thus, the energy of each symbol is the same).

Input Arguments: x3_t (output of the line coder), t (0:1/Fs:n*Ts), wc (carrier frequency)

Here, n is the number of symbols = number of bits / 2.

Output: A row vector of length(t) = n*Ts*Fs + 1, such that

$$x_4(t) \;=\; \Sigma\, a_k p(t - kT_s)\cos(w_c t) \;+\; \Sigma\, b_k p(t - kT_s)\sin(w_c t)$$

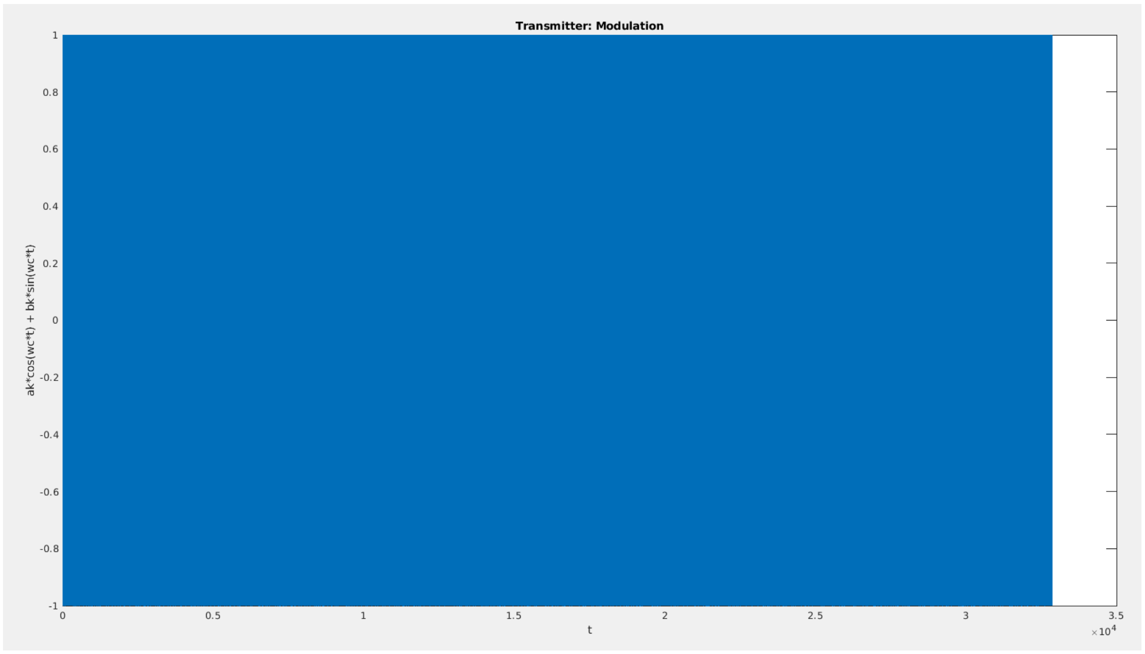


```
function x4_t = modulation (x3_t, t, wc)
    x4_t = zeros(1,length(t));
    cos_t = cos(wc*t);
    sin_t = sin(wc*t);
    for idx=1:length(t)
        x4_t(idx) = x3_t(1,idx) .* cos_t(idx);
        x4_t(idx) = x4_t(idx) + x3_t(2,idx) .* sin_t(idx);
    end
end
```

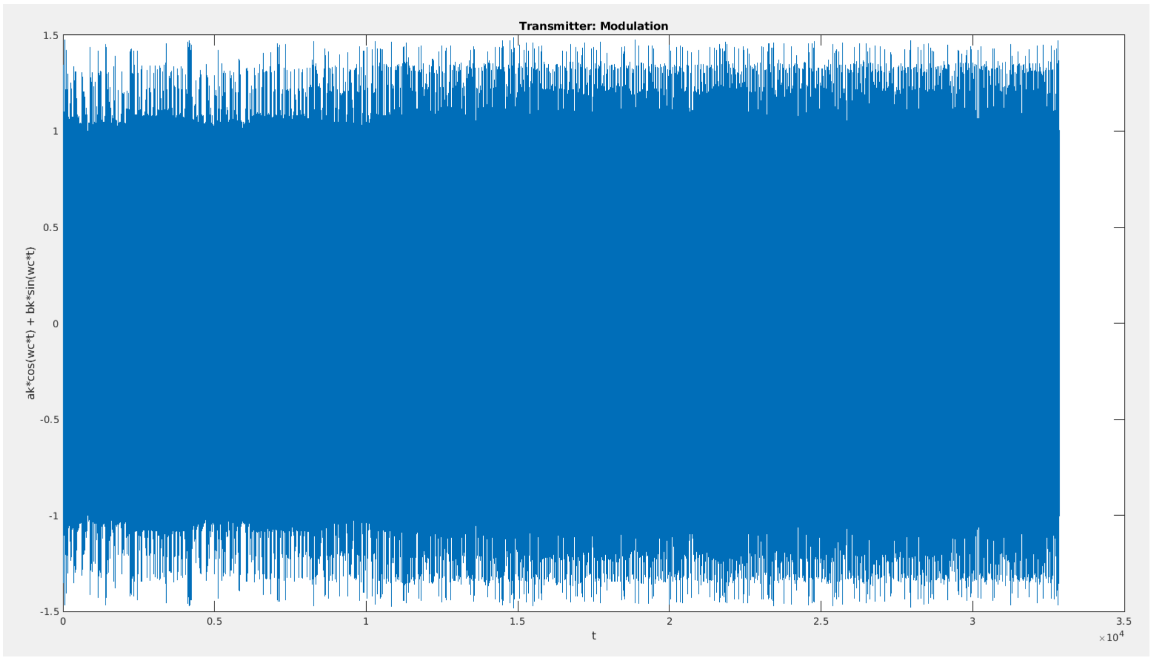
We simply take the first row of output of the line coder and multiply with cos(w_ct), take the second row and multiply with sin(w_ct) and add them to get x_4(t). This signal is then ready to be passed to the channel.

For w_c = 1MHz,

For rectangular pulse p(t):



For raised cosine p(t):



Channel

After modulating the signal, we pass it through the two following channels,

1. Memoryless AWGN channel

$$r(t) = s(t) + n(t)$$

2. AWGN channel with memory

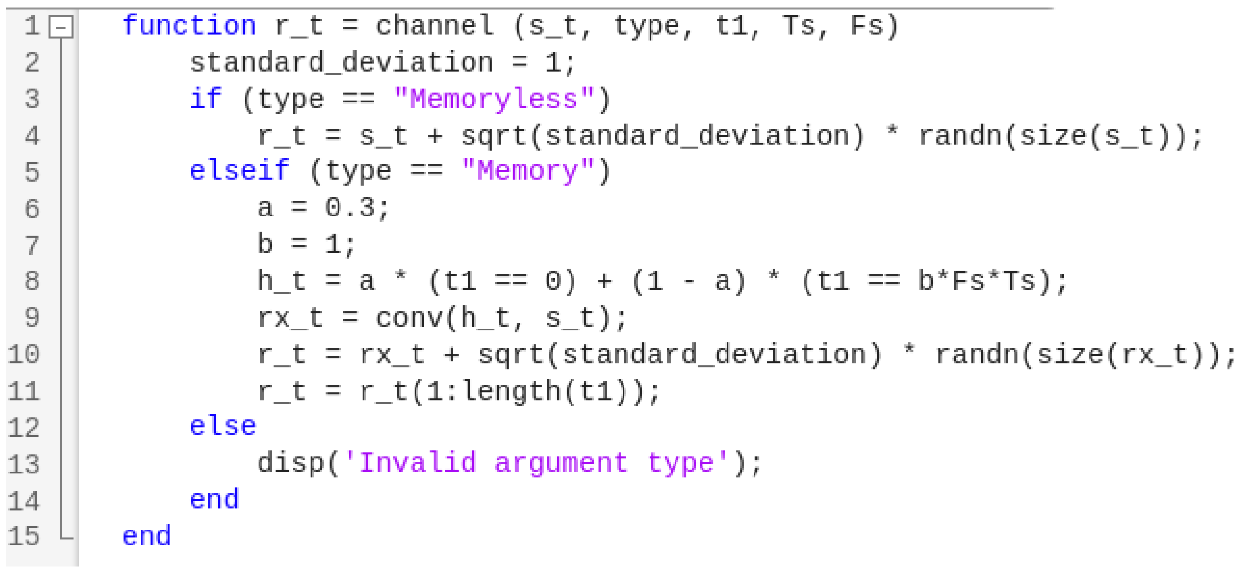
$$r(t) = h(t) * s(t) + n(t)$$

where, $h(t) = a\delta(t) + (1-a)\delta(t - bT_b)$

Where s(t) is the modulated signal, that is, x_4(t) and n(t) is the random noise, and r(t) is the output of the channel.

Input Arguments: x4_t (output of the modulator), t (0:1/Fs:n*Ts), type (‘Memoryless’, ‘Memory’), Ts (symbol duration), Fs (sampling frequency)

Output: A row vector r_t of length same as s(t)



```
function r_t = channel (s_t, type, t1, Ts, Fs)
    standard_deviation = 1;
    if (type == "Memoryless")
        r_t = s_t + sqrt(standard_deviation) * randn(size(s_t));
    elseif (type == "Memory")
        a = 0.3;
        b = 1;
        h_t = a * (t1 == 0) + (1 - a) * (t1 == b*Fs*Ts);
        rx_t = conv(h_t, s_t);
        r_t = rx_t + sqrt(standard_deviation) * randn(size(rx_t));
        r_t = r_t(1:length(t1));
    else
        disp('Invalid argument type');
    end
end
```

`randn(size(n))` returns a matrix of size of n of normally distributed random numbers.

Memoryless AWGN channel

In the Memoryless AWGN channel, we simply add this random generated normally distributed numbers. This channel has no memory, meaning each received symbol is only affected by the noise present at that instant and not by past symbols or noise.

Since the channel has no memory, equalization techniques (demodulation, and further processes) can be simpler and often involve compensating for amplitude and phase distortions caused by the channel but without having to deal with intersymbol interference (ISI) caused by memory effects.

Since, the size of s(t) is same as size of n(t) -> size of r(t) is same as size of s(t)

AWGN channel with Memory

Whereas, in AWGN channel with memory, we convolute with a impulse response is

h(t) = a* delta(t) + (1-a)*delta(t-bTs).
The channel impulse response h(t) which includes memory effects, such as delays and attenuations, which can affect the received signal over time. In this case, h(t) is defined as a weighted sum of two delta functions, indicating a channel with memory that includes both instantaneous and delayed responses.

Equalization becomes crucial and more challenging in such channels as it needs to mitigate ISI effects. Techniques like linear equalization, decision feedback equalization (DFE), or adaptive equalization are often used to combat these challenges.

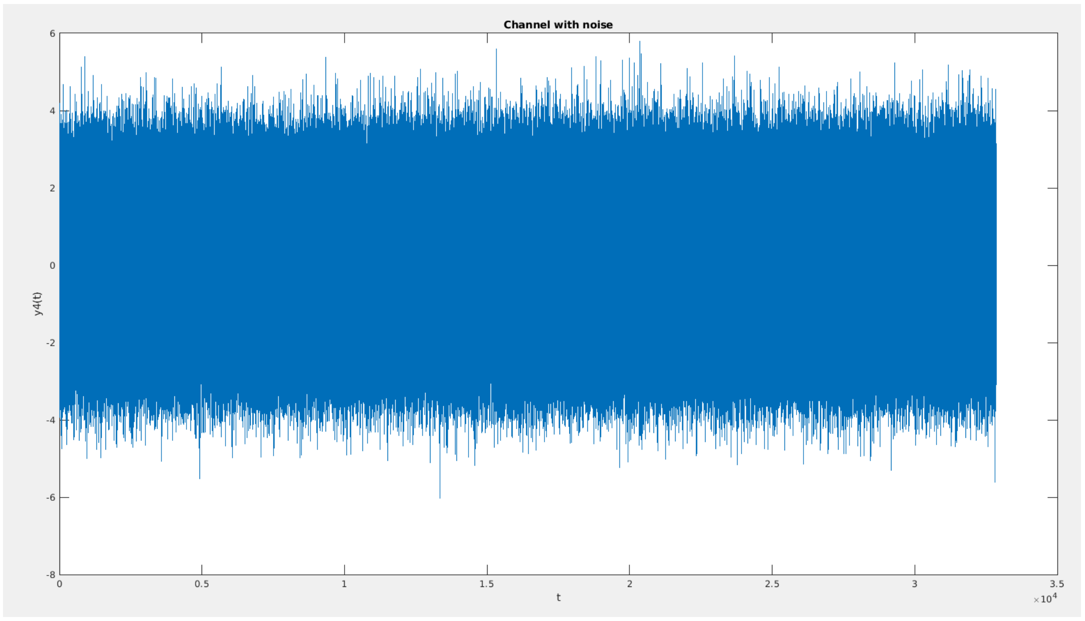
Here, we have used the same demodulation scheme for comparison purposes.

Since, the size of r(t) depends on b (specifically, exceeds size of s(t) by b), we truncate it to the same size as s(t).

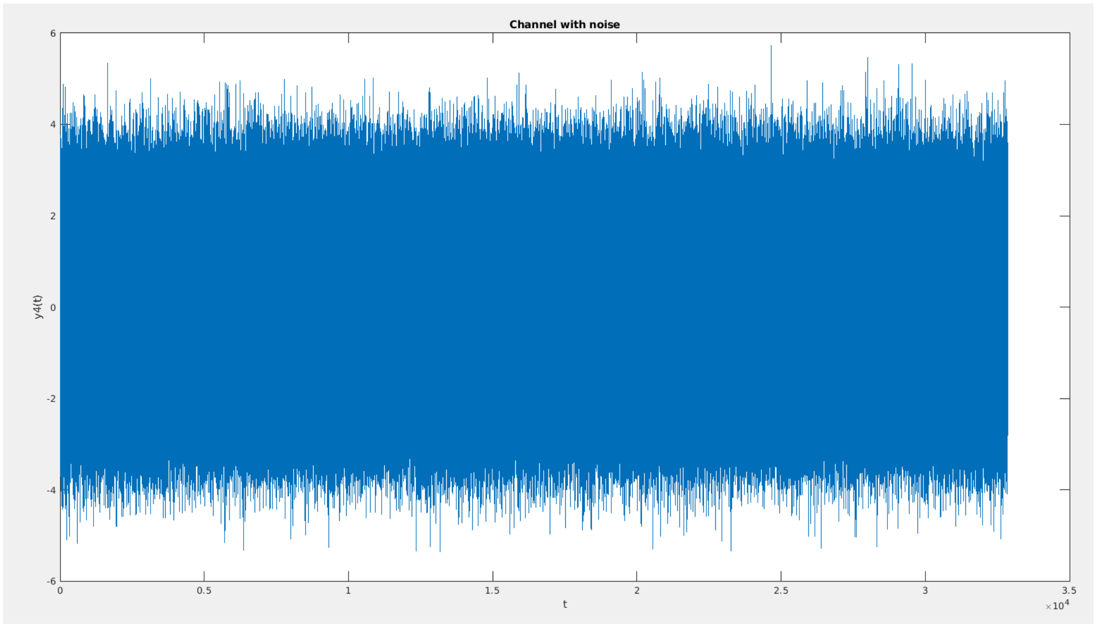
For variance = 1,

i) For Memoryless AWGN,

For raised cosine p(t),

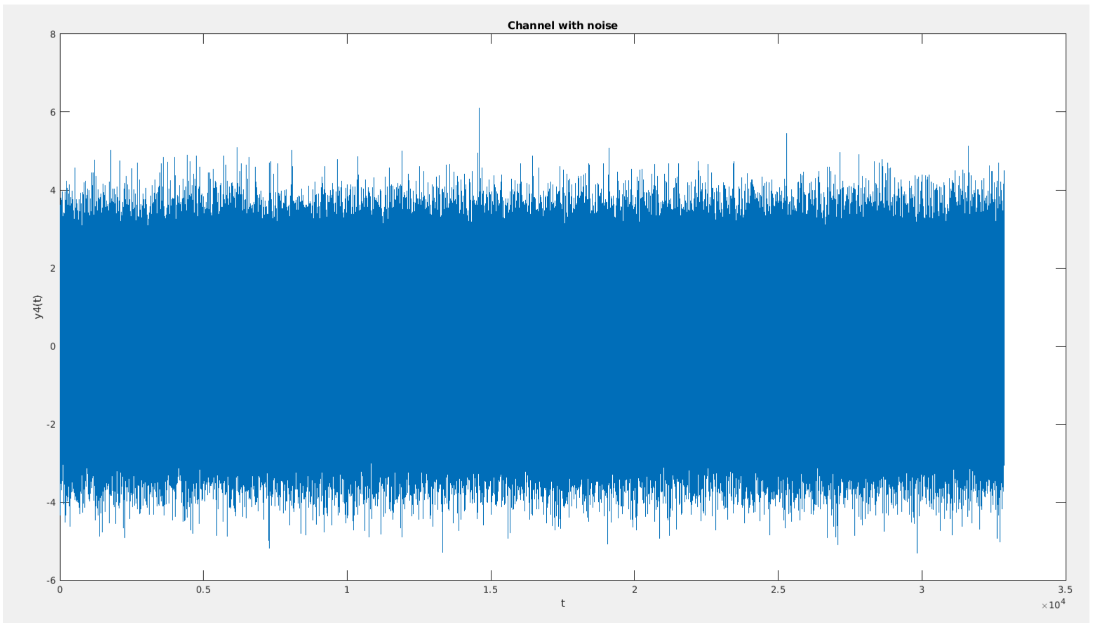


For rectangular p(t),

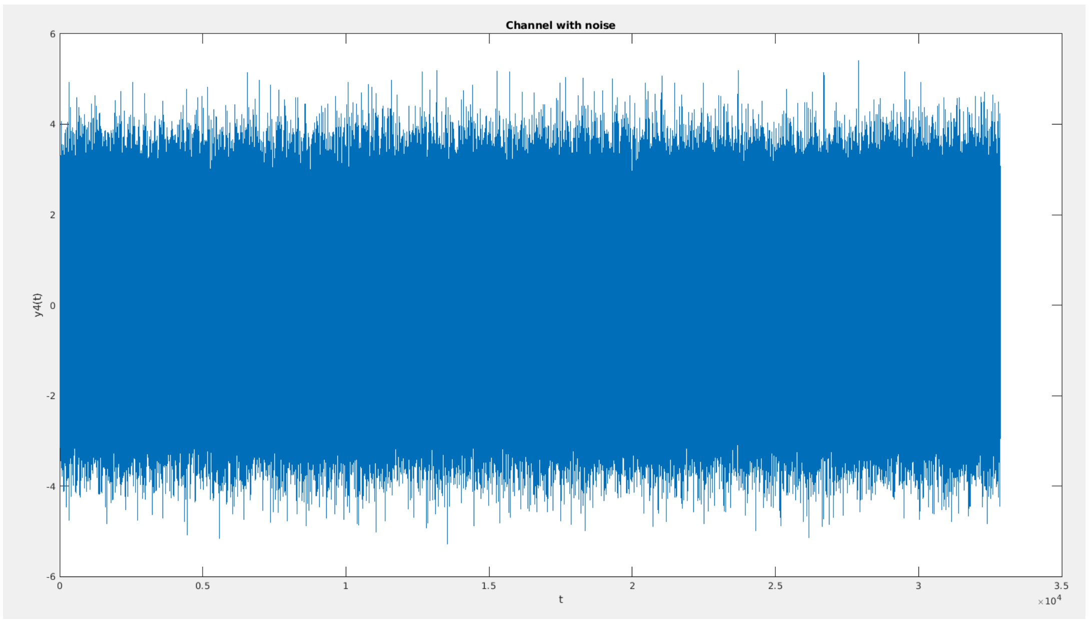


ii) For Memory AWGN,

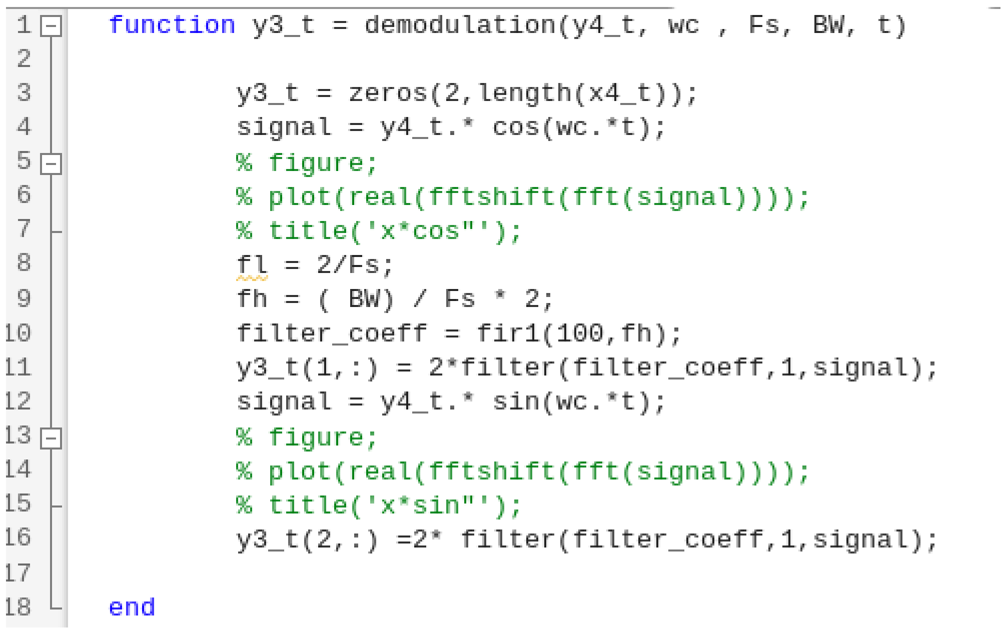
For raised cosine p(t),



For rectangular p(t),



Demodulation



```
function y3_t = demodulation(y4_t, wc , Fs, BW, t)

        y3_t = zeros(2,length(x4_t));
        signal = y4_t.* cos(wc.*t);
        % figure;
        % plot(real(fftshift(fft(signal))));
        % title('x*cos"');
        fl = 2/Fs;
        fh = ( BW) / Fs * 2;
        filter_coeff = fir1(100,fh);
        y3_t(1,:) = 2*filter(filter_coeff,1,signal);
        signal = y4_t.* sin(wc.*t);
        % figure;
        % plot(real(fftshift(fft(signal))));
        % title('x*sin"');
        y3_t(2,:) =2* filter(filter_coeff,1,signal);

end
```

In this block we demodulate the given signal by multiplying it with basis functions (cos(wct) and sin(wct)) and then passing them through low pass filters.

As input signal for this block is in this form

y4_t = ak * p(t) * cos(wct) + bk * p(t) * sin(wct)

We multiply it with the basis functions (cos and sin)
For example consider cos

y4_t * cos(wct) = ak * p(t) * cos^2 (wct) + bk * p(t) * sin(wct) * cos(wct)
= ak * p(t) * (1 + cos(2wct))/2 + bk * p(t) * sin(wct) * cos(wct)
= ak * p(t) / 2 + ak * p(t) * cos(2wct) /2 + bk * p(t) * sin(2wct) /2

In Frequency domain it is

ak * P(f) /2 + ak * P (f+2fc) /2 + bk * j P(f+2fc)

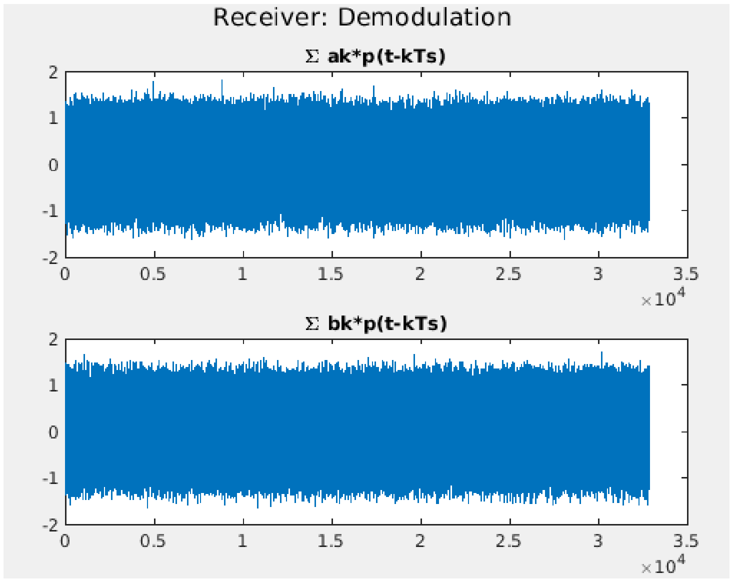
When passing through a low-pass filter, all frequency components at 2fc get attenuated, and only ak * P(f) /2 at low frequencies remains as output.

As in Encoder we are taking a = 1 So ak's and bk's are either 1 or 0 So output after lowpass filter is P(f)/2 .Thus we are setting the gain of filter as 2 to get P(f) as output of demodulation block.

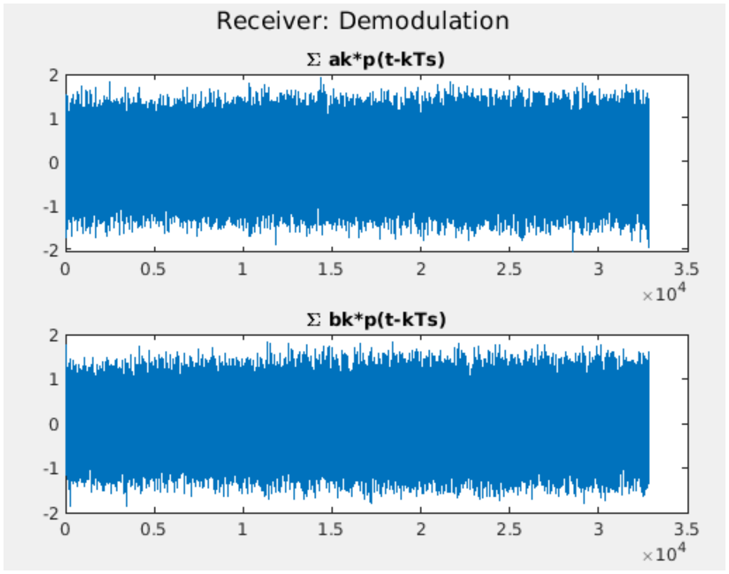
As we have two basis functions the output of demodulation block is 2d thus y3_t(:,1) is one which we get by multiplying y4_t with cos(wct) and y3_t(:,2) is one which we get by multiplying y4_t with sin(wct). Basically y3_t(:,1) = ak * p(t) and y3_t(:,2) = bk * p(t).

For AWGN noise (Memoryless) :

1. Output Plot for p(t)-> rect

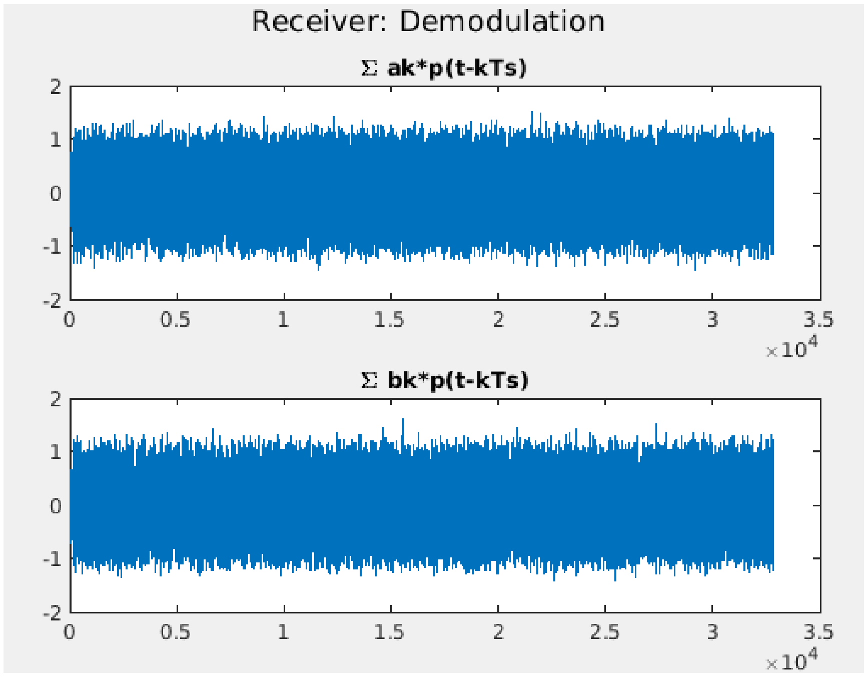


2. Output Plot for p(t)-> Raised Cosine

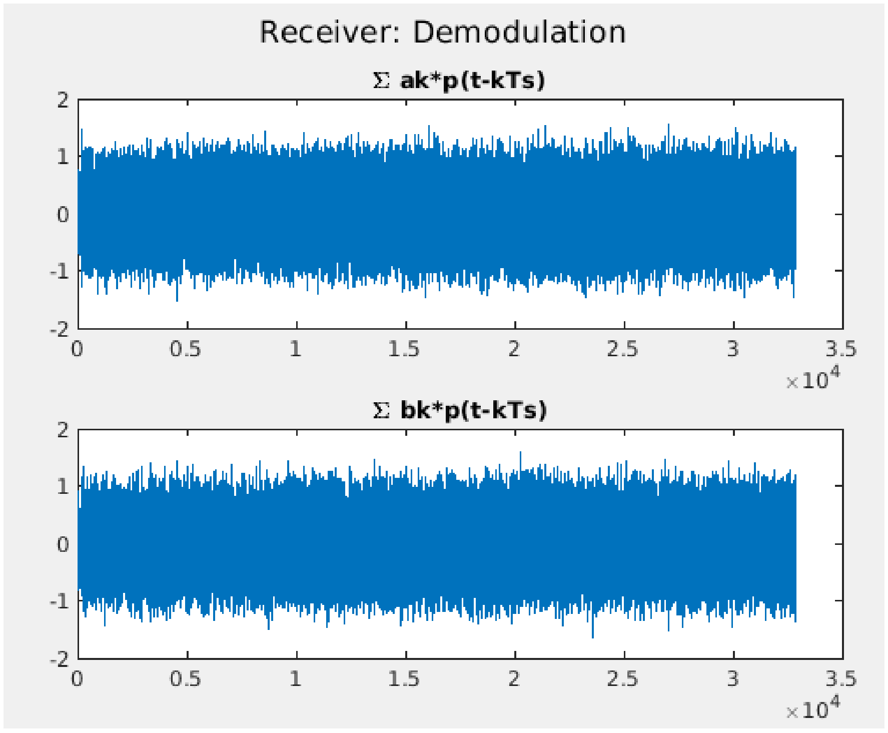


For AWGN noise with memory

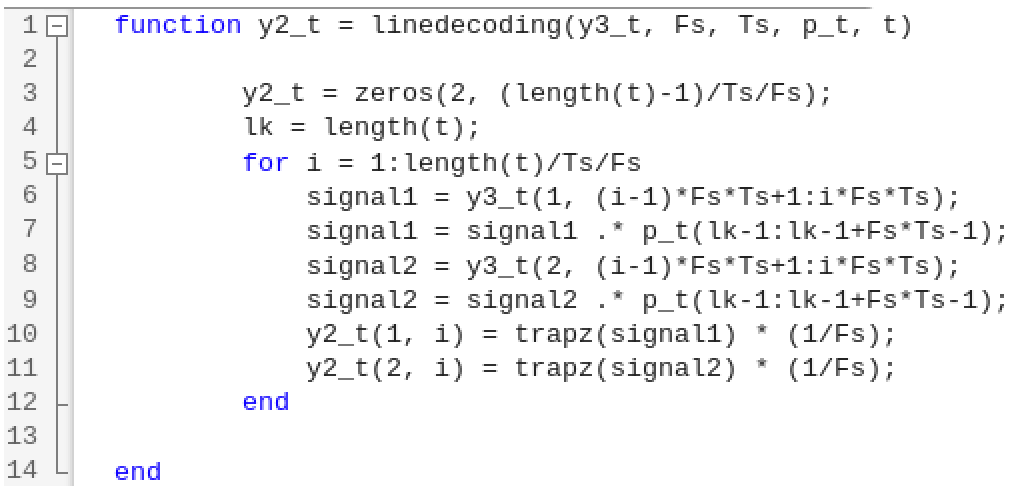
1. Output Plot for p(t)-> rect



2. Output Plot for p(t)-> Raised Cosine

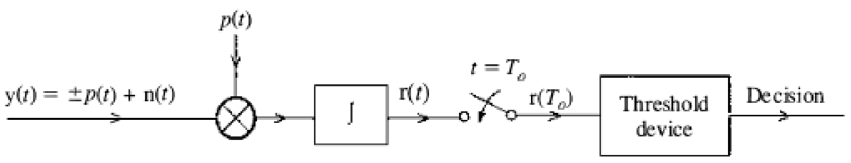


Linedecoding



```
function y2_t = linedecoding(y3_t, Fs, Ts, p_t, t)

        y2_t = zeros(2, (length(t)-1)/Ts/Fs);
        lk = length(t);
        for i = 1:length(t)/Ts/Fs
            signal1 = y3_t(1, (i-1)*Fs*Ts+1:i*Fs*Ts);
            signal1 = signal1 .* p_t(lk-1:lk-1+Fs*Ts-1);
            signal2 = y3_t(2, (i-1)*Fs*Ts+1:i*Fs*Ts);
            signal2 = signal2 .* p_t(lk-1:lk-1+Fs*Ts-1);
            y2_t(1, i) = trapz(signal1) * (1/Fs);
            y2_t(2, i) = trapz(signal2) * (1/Fs);
        end

end
```

For Linecoding part We used correlation detector rather than matched filter just because as integration it is easy to implement in matlab other than filter part.
Below is the derivation of how correlator detection is reduced to just integration.



Hence,

$$r(t) = \int_{-\infty}^{\infty} y(x)p\,(x + T_o - t)\,dx \qquad (10.16a)$$

At the decision-making instant $t = T_o$, we have

$$r(T_o) = \int_{-\infty}^{\infty} y(x)\,p\,(x)\,dx \qquad (10.16b)$$

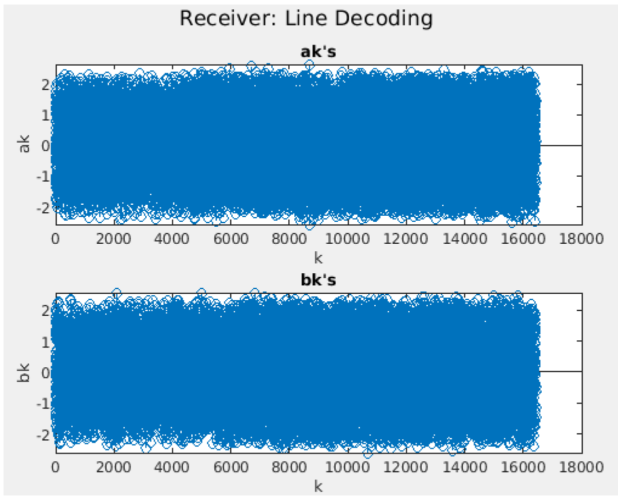
Because the input $y(x)$ is assumed to start at $x = 0$ and $p\,(x) = 0$ for $x > T_o$, we have the decision variable

$$r(T_o) = \int_{0}^{T_o} y(x)\,p\,(x)\,dx \qquad (10.16c)$$

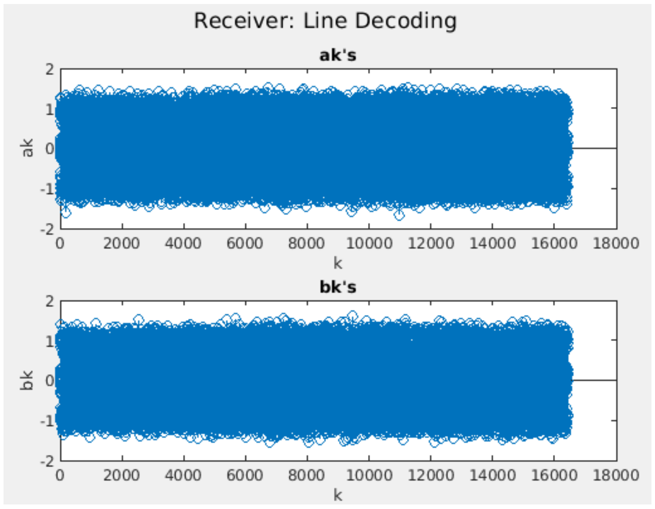
So for this line decoding block we are taking demodulation output and p_t as input and multiplying them and then integrating them in interval 0 to Ts i.e Symbol period to get the value ak and bk.
By integrating with y3_t(1,:) we get ak's and by integrating with y3_t(2,:) we get bk's in this way we get ak's and bk's as y2(1,:) and 2(2,:) . In this block the length of signal is reduced by Fs*Ts times from previous block signal length because we are integrating them over that period using matlab trapz inbuilt function.

For AWGN Noise (Memoryless) :

1. Output Plot for p(t)-> rect

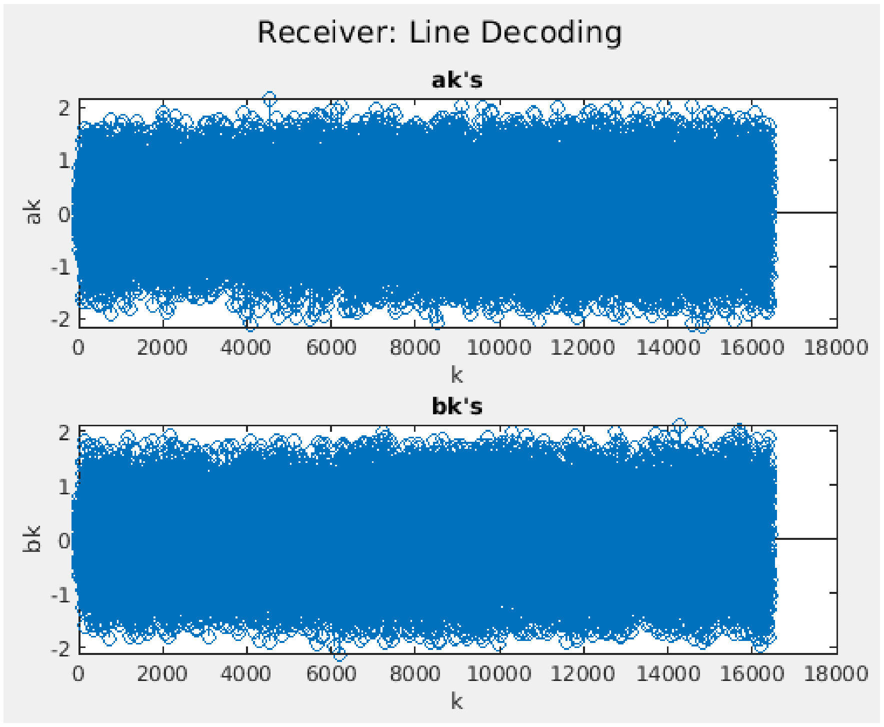


2. Output Plot for p(t)-> Raised Cosine

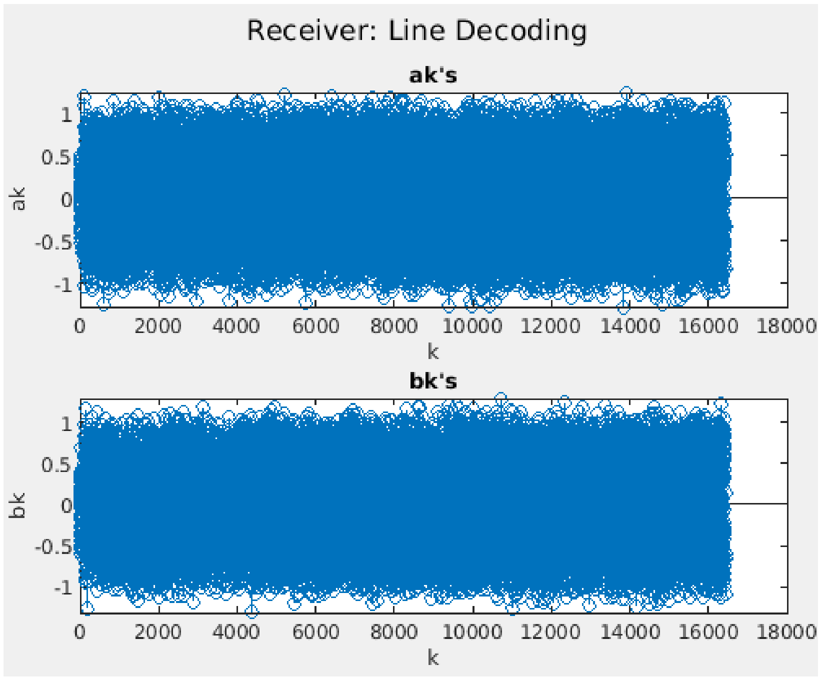


For AWGN noise with memory

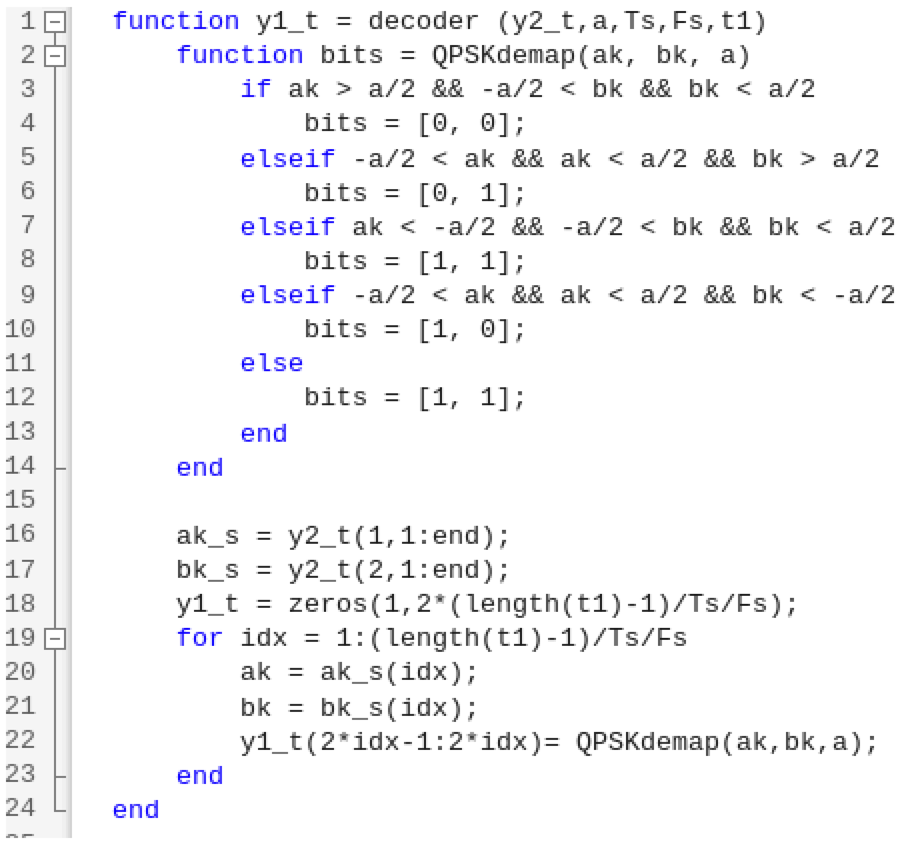
1. Output Plot for p(t)-> rect



2. Output Plot for p(t)-> Raised Cosine



Decoder



```
function y1_t = decoder (y2_t,a,Ts,Fs,t1)
    function bits = QPSKdemap(ak, bk, a)
        if ak > a/2 && -a/2 < bk && bk < a/2
            bits = [0, 0];
        elseif -a/2 < ak && ak < a/2 && bk > a/2
            bits = [0, 1];
        elseif ak < -a/2 && -a/2 < bk && bk < a/2
            bits = [1, 1];
        elseif -a/2 < ak && ak < a/2 && bk < -a/2
            bits = [1, 0];
        else
            bits = [1, 1];
        end
    end

    ak_s = y2_t(1,1:end);
    bk_s = y2_t(2,1:end);
    y1_t = zeros(1,2*(length(t1)-1)/Ts/Fs);
    for idx = 1:(length(t1)-1)/Ts/Fs
        ak = ak_s(idx);
        bk = bk_s(idx);
        y1_t(2*idx-1:2*idx)= QPSKdemap(ak,bk,a);
    end
end
```

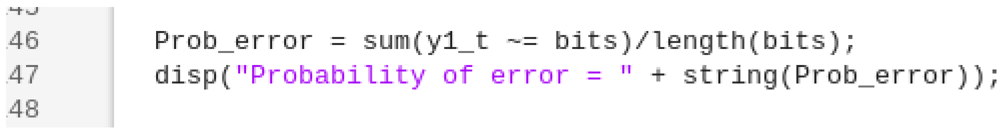
Now here we are demapping the ak’s and bk’s to their respective bits

Demapping

a/2 < ak	&&	-a/2 < bk < a/2	—->	00
-a/2 < ak < a/2	&&	a/2 < bk	—->	01
ak < -a/2	&&	-a/2 < bk < a/2	—->	10
-a/2 < ak < a/2	&&	bk < -a/2	—->	11

Now from the previous block output we are taking each ak and corresponding bk and then using the above demapping function we demap them to bits. These decoded bits are output of the decoder block represented as y1_t.

And then Probability of Error is calculated by comparing decoded bits and original bits by total number of bits using below code snippet

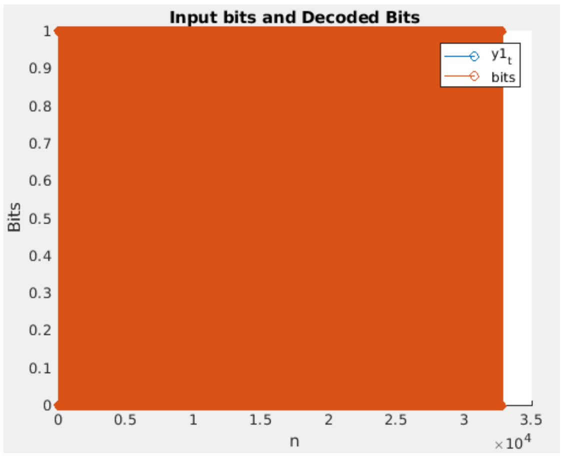


```
Prob_error = sum(y1_t ~= bits)/length(bits);
disp("Probability of error = " + string(Prob_error));
```

Plotting decoded bits and original bits on the same plot:

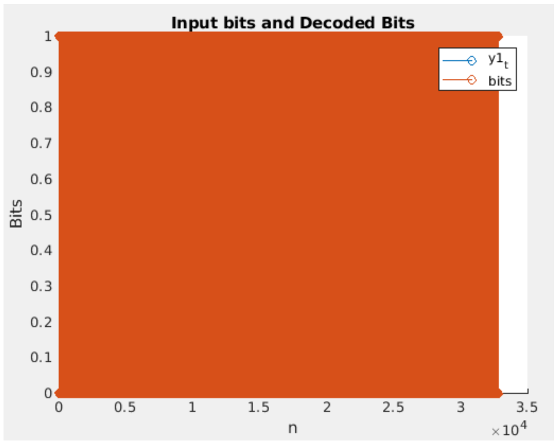
For AWGN noise (Memoryless) :

1. Output Plot for p(t)-> rect



Probability of error = 0.12171

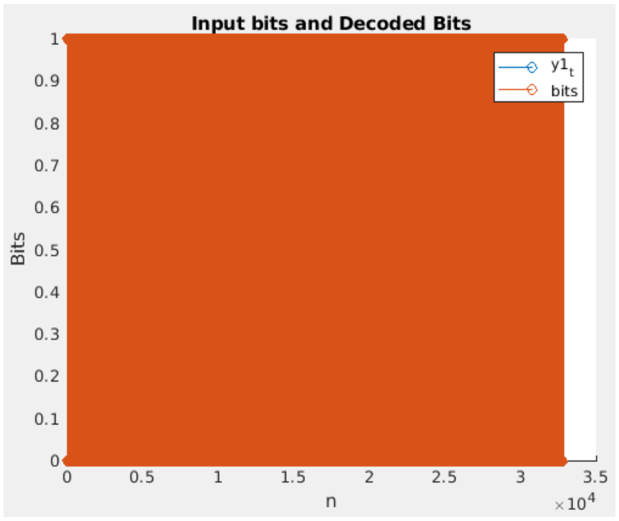
2. Output Plot for p(t)-> Raised Cosine



Probability of error = 0.0008522

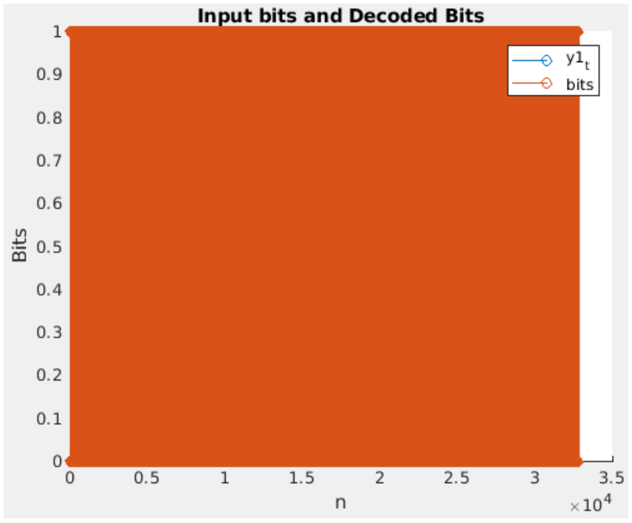
For AWGN noise with memory

1. Output Plot for p(t)-> rect



Probability of error = 0.45495

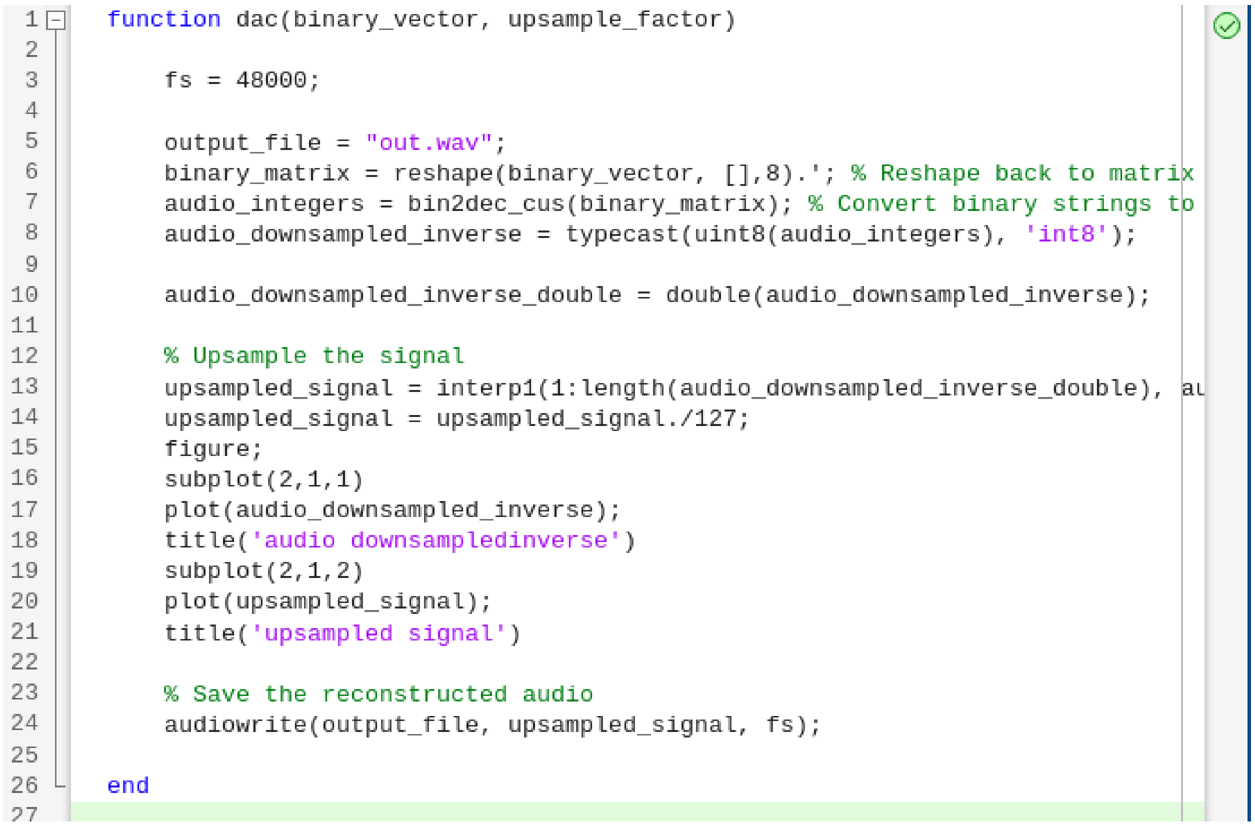
2. Output Plot for p(t)-> Raised Cosine



Probability of error = 0.37963

Here we can observe that AWGN noise with memory has more BER than memoryless AWGN because of overlaps in symbol period with other symbol signals (feedback). And even in AWGN memoryless due to zero ISI criteria and decay time < 1/t BER of raised cosine is less than rect.

DAC



```
function dac(binary_vector, upsample_factor)

    fs = 48000;

    output_file = "out.wav";
    binary_matrix = reshape(binary_vector, [],8).'; % Reshape back to matrix
    audio_integers = bin2dec_cus(binary_matrix); % Convert binary strings to
    audio_downsampled_inverse = typecast(uint8(audio_integers), 'int8');

    audio_downsampled_inverse_double = double(audio_downsampled_inverse);

    % Upsample the signal
    upsampled_signal = interp1(1:length(audio_downsampled_inverse_double), au
    upsampled_signal = upsampled_signal./127;
    figure;
    subplot(2,1,1)
    plot(audio_downsampled_inverse);
    title('audio downsampledinverse')
    subplot(2,1,2)
    plot(upsampled_signal);
    title('upsampled signal')

    % Save the reconstructed audio
    audiowrite(output_file, upsampled_signal, fs);

end
```

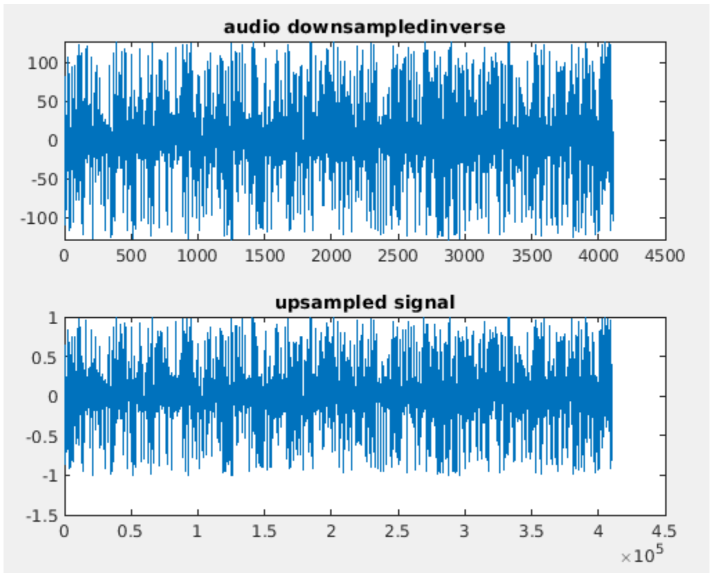
As in ADC we converted dec to 8 bits binary here we are creating a matrix of 8 x length(bits)/8 where it contains row of 8 binary bits and then convert each binary row to decimal using bin2dec_Cus function (just takes each 8binary bit number and then converts to decimal).
Then other line of code converts an array of unsigned 8-bit integers (uint8) into an array of signed 8-bit integers (int8) using the typecast function in MATLAB or Octave.

Now we are upsampling these because we downsampled them in adc so using interp1 inbuilt function we interpolate the samples and upsample the signal for fair reasoning and then we normalize it to audiowrite in out.wav file.

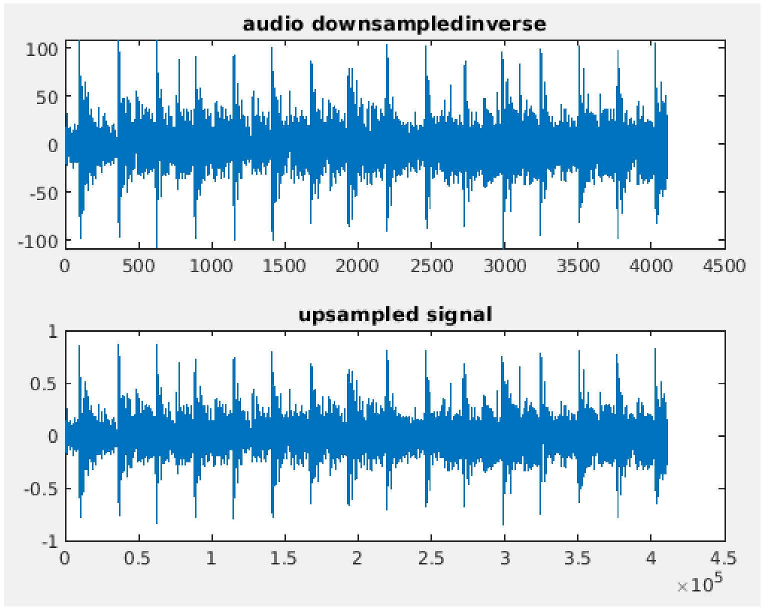
Plots observed are:

For AWGN noise (Memoryless) :

1. Output Plot for p(t)-> rect

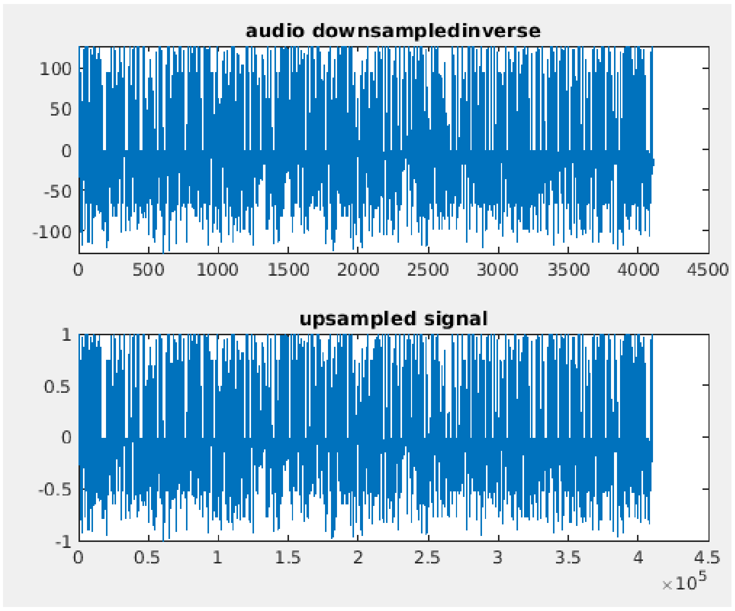


2. Output Plot for p(t)-> Raised Cosine

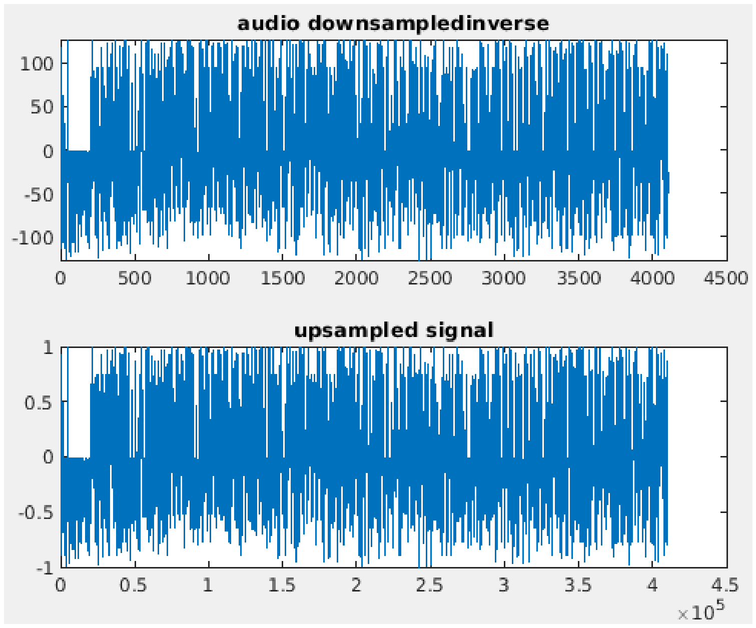


For AWGN noise with memory

1. Output Plot for p(t)-> rect

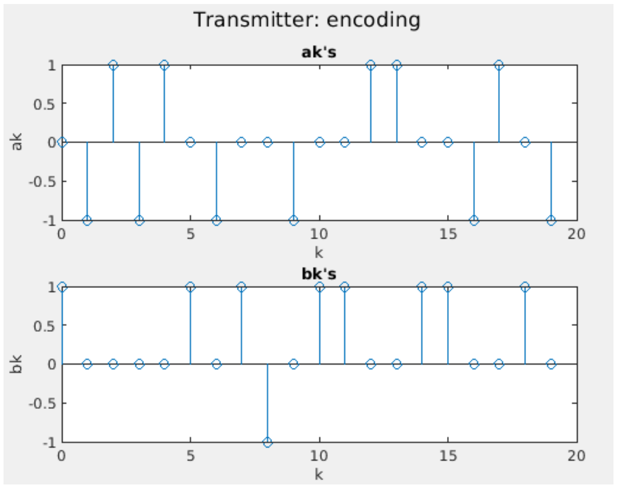


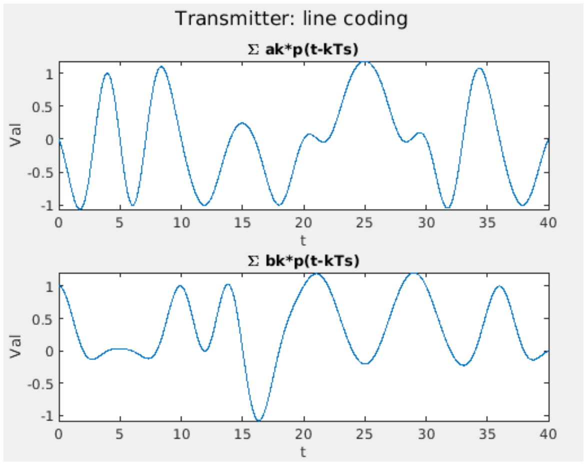
2. Output Plot for p(t)-> Raised Cosine

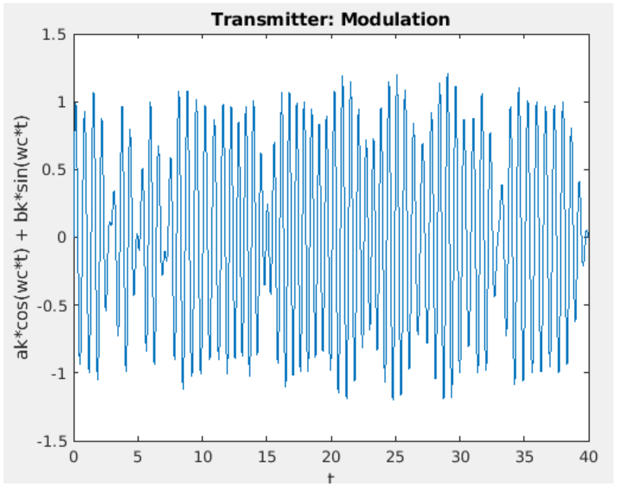


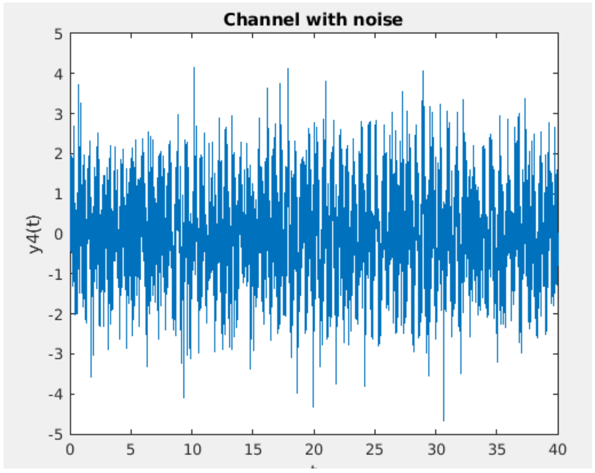
For Good visualization of plots let's take upsampling factor = 100000 (So that no of bits decreases) and wc = 10 (For good visualization during modulation)

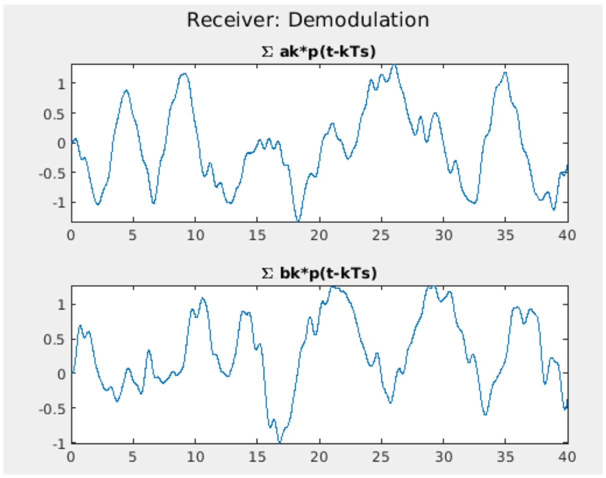
For sinc:

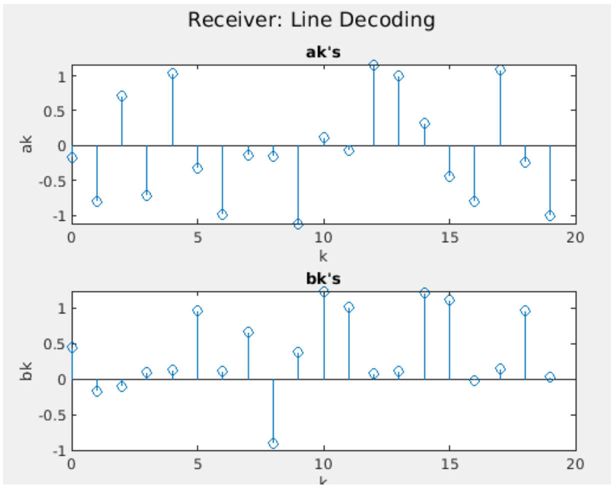




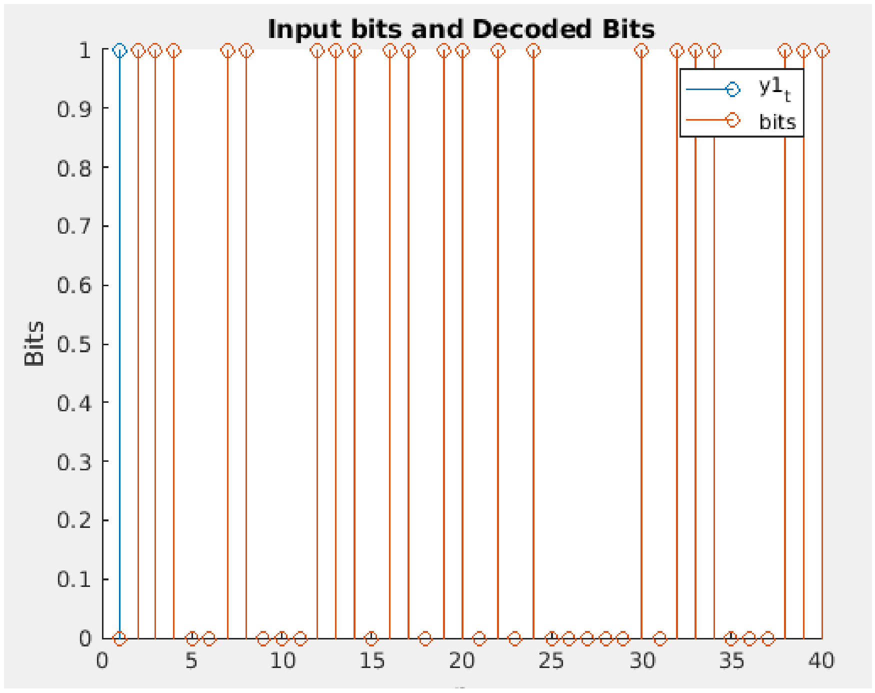








`



Using a raised cosine pulse shaping filter for demodulation and decoding offers several advantages over using a rectangular pulse shaping filter:

Reduced Interference: Raised cosine filters limit interference from adjacent channels, improving signal-to-noise ratio and reducing intersymbol interference (ISI).

Lower Out-of-Band Emissions: Raised cosine filters produce fewer emissions outside the desired frequency band, reducing interference with other systems.

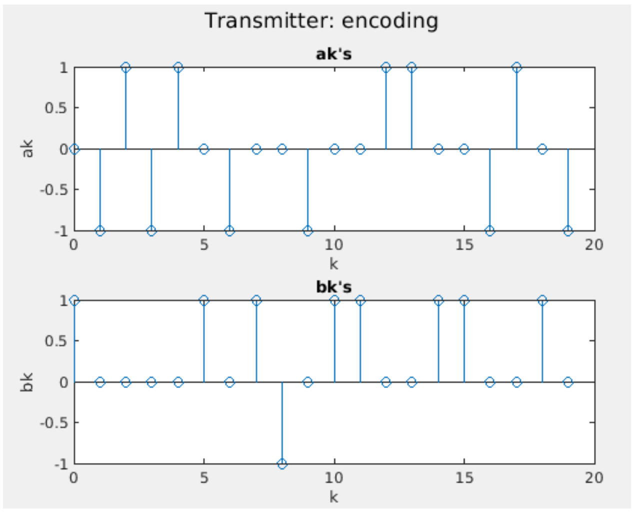
Improved Symbol Recovery: Smoother transitions between symbols reduce distortion and enhance symbol recovery at the receiver, leading to zero ISI.

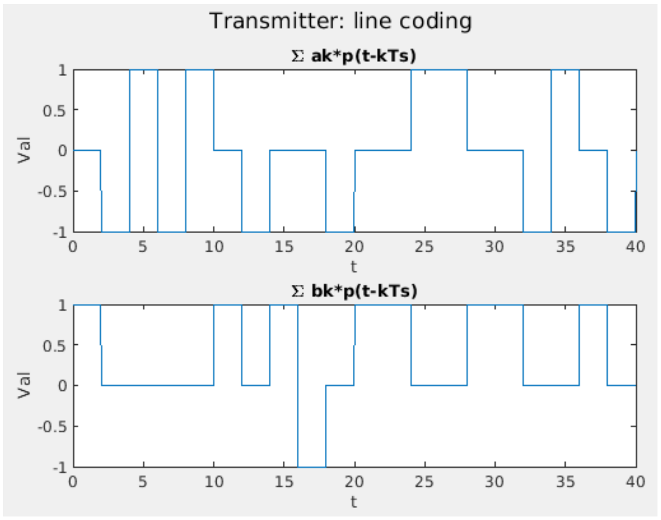
Enhanced Bandwidth Efficiency: Raised cosine filters allow for tighter packing of symbols in the frequency domain, enabling higher data rates within the same bandwidth.

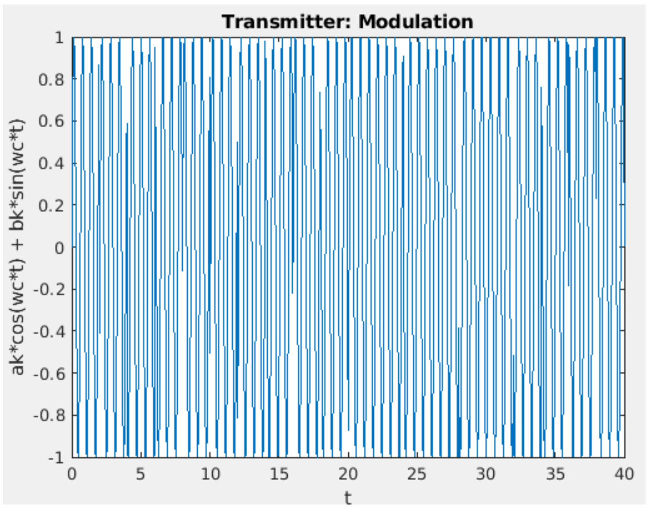
Shorter Decay Time: The decay time of raised cosine filters is typically shorter compared to rectangular filters, facilitating faster symbol recovery and reducing the effects of ISI.

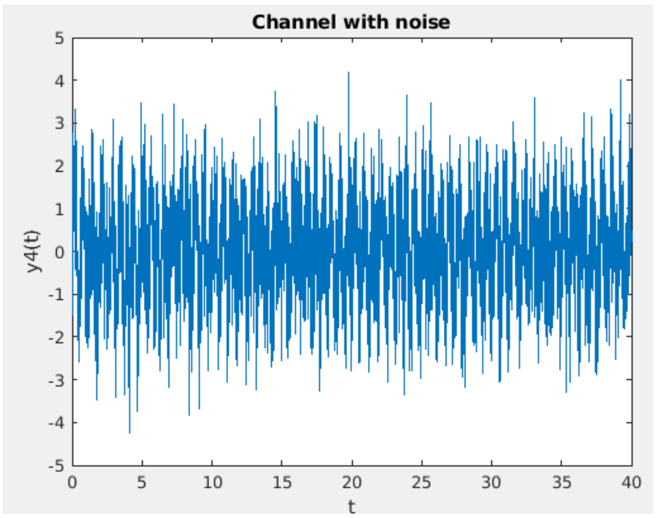
Thus we have good demodulation and decoding signals for raised cosine than for the rect pulse.

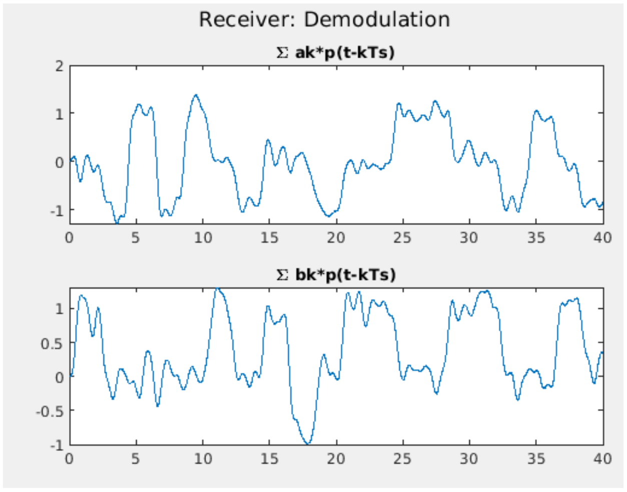
For rect:

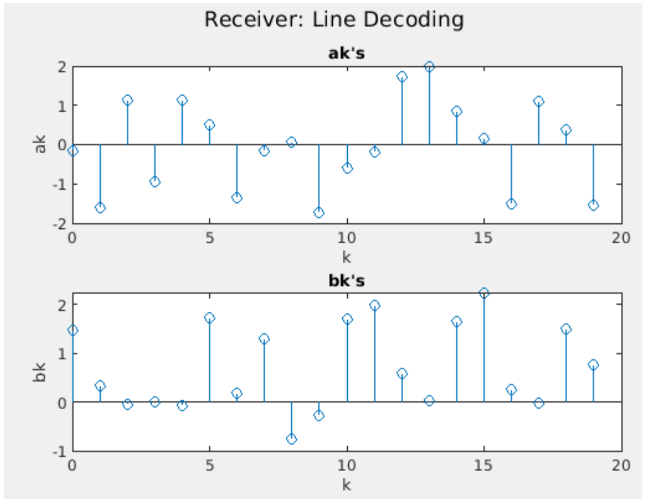

Transmitter: encoding
ak's
ak
k
bk's
bk
k

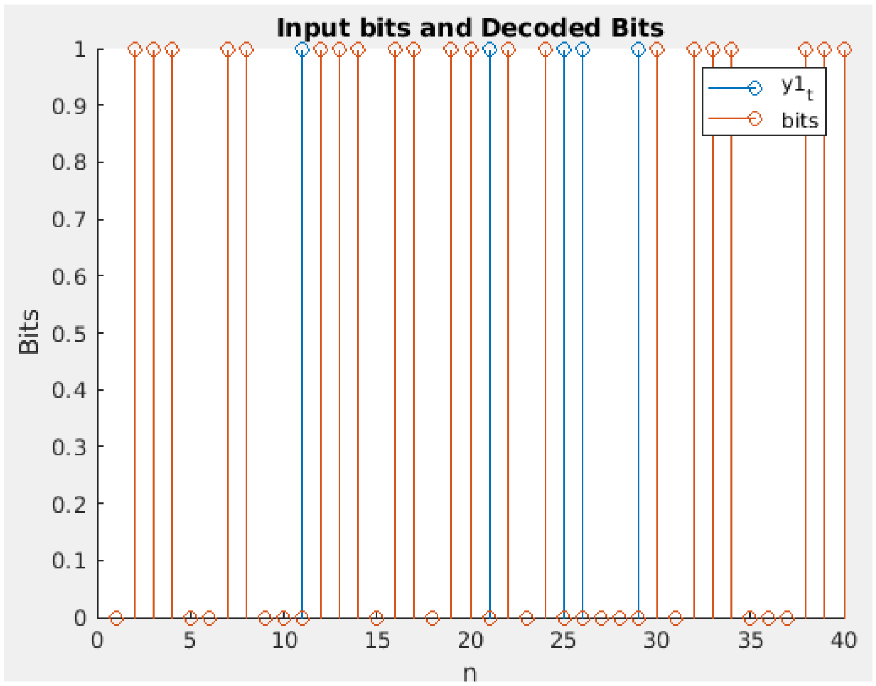

Transmitter: line coding
Σ ak*p(t-kTs)
Val
t
Σ bk*p(t-kTs)
Val
t


Transmitter: Modulation
ak*cos(wc*t) + bk*sin(wc*t)
t


Channel with noise
y4(t)
t


Receiver: Demodulation
Σ ak*p(t-kTs)
Σ bk*p(t-kTs)

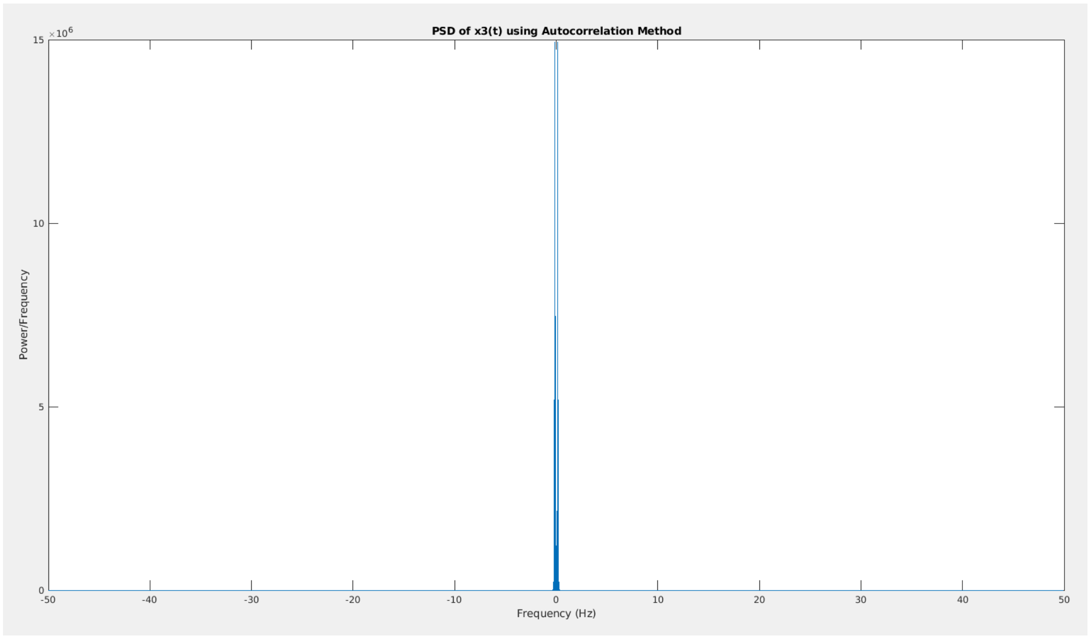

Receiver: Line Decoding
ak's
ak
k
bk's
bk
k

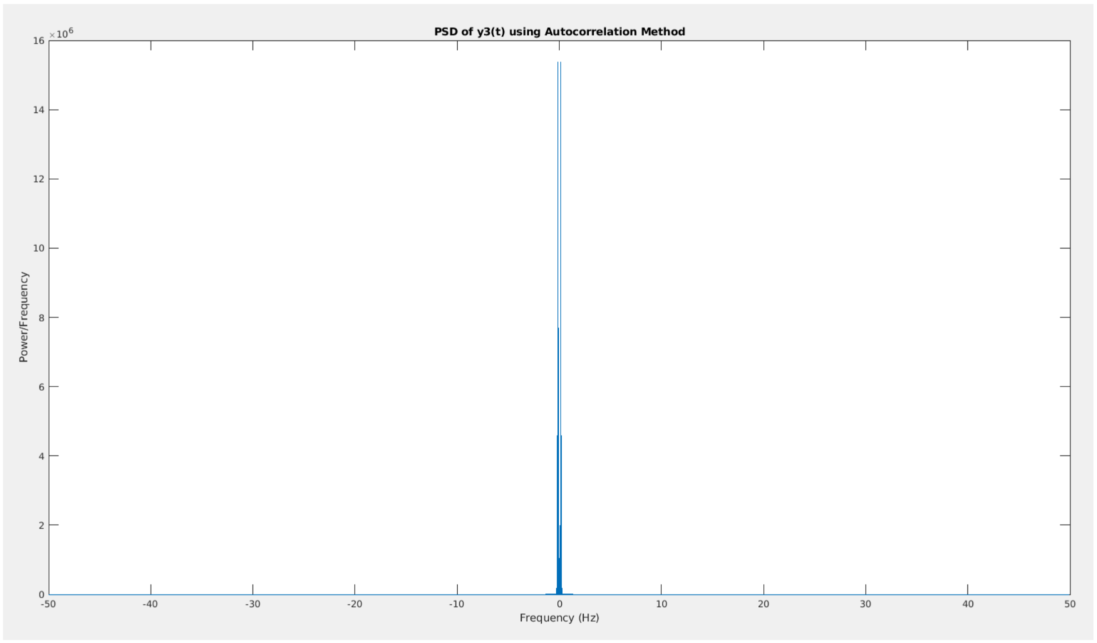


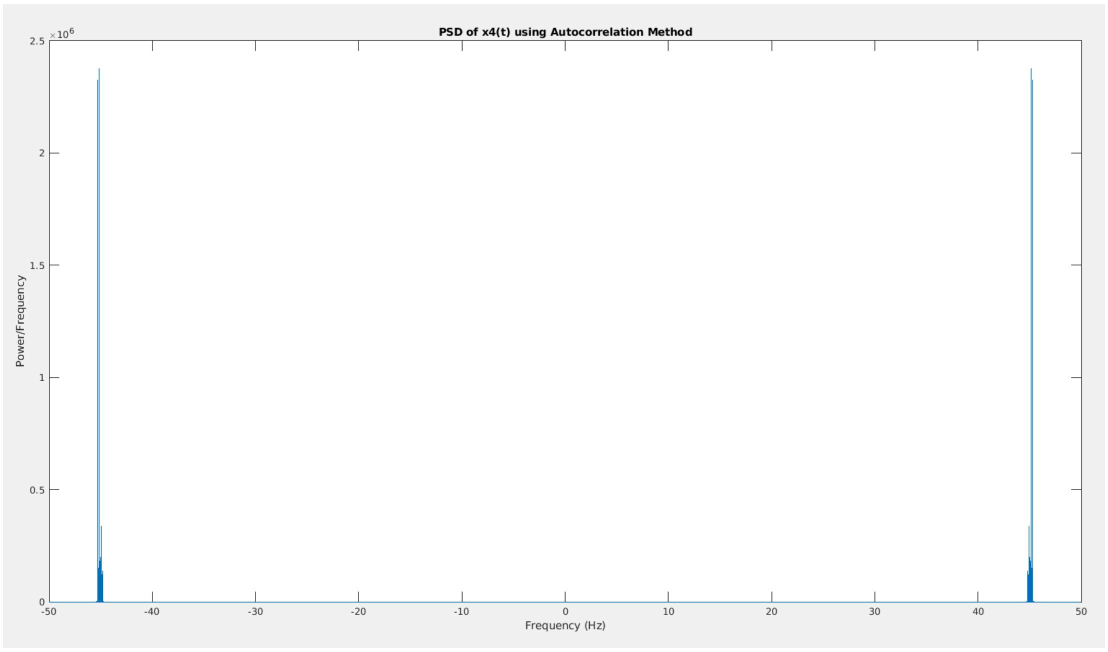
You can clearly see that the probability of error of rect is more. For linecoding and decoding plots we can observe they are similar in raised cosine case but they are varying a lot for rect case because in demodulation we pass it through a low pass filter due to infinite bandwidth of rect and finite bandwidth of raised cosine this difference is being appeared. Thus we are having greater probability of error for rect case.

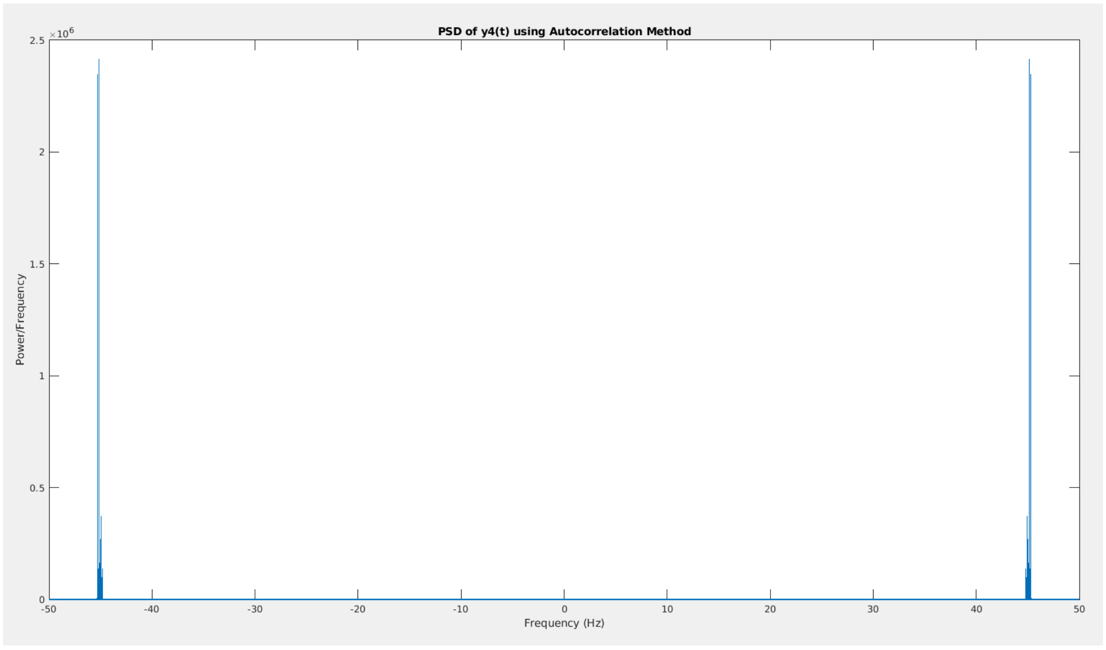
Plots of Power Spectral Density (PSD's):

1. Memoryless channel & Raised Cosine

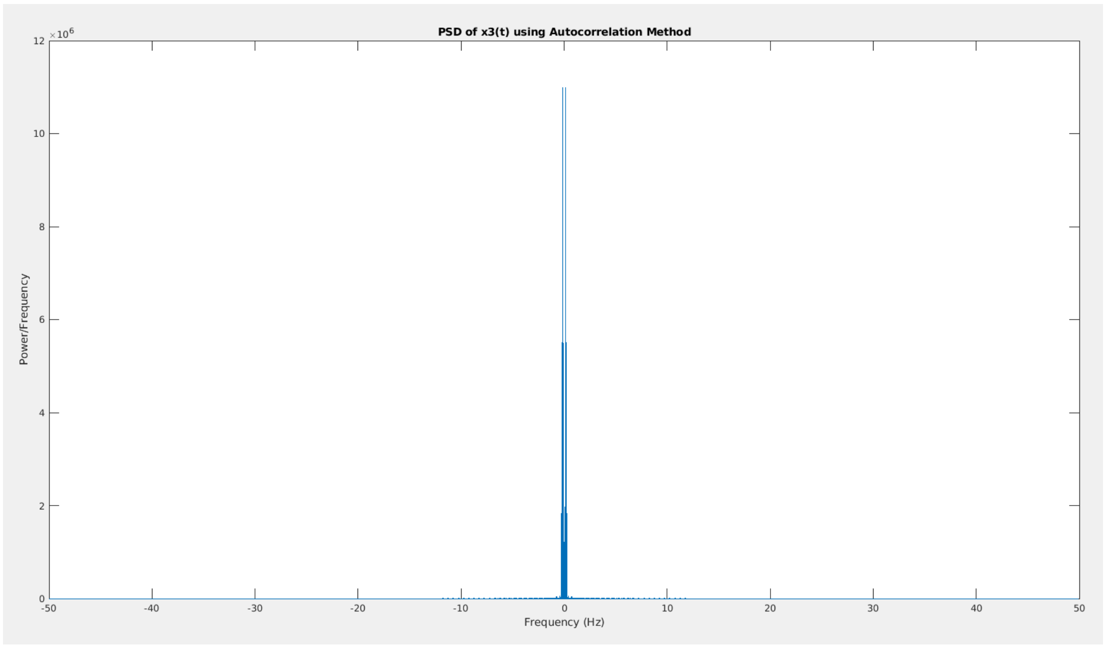


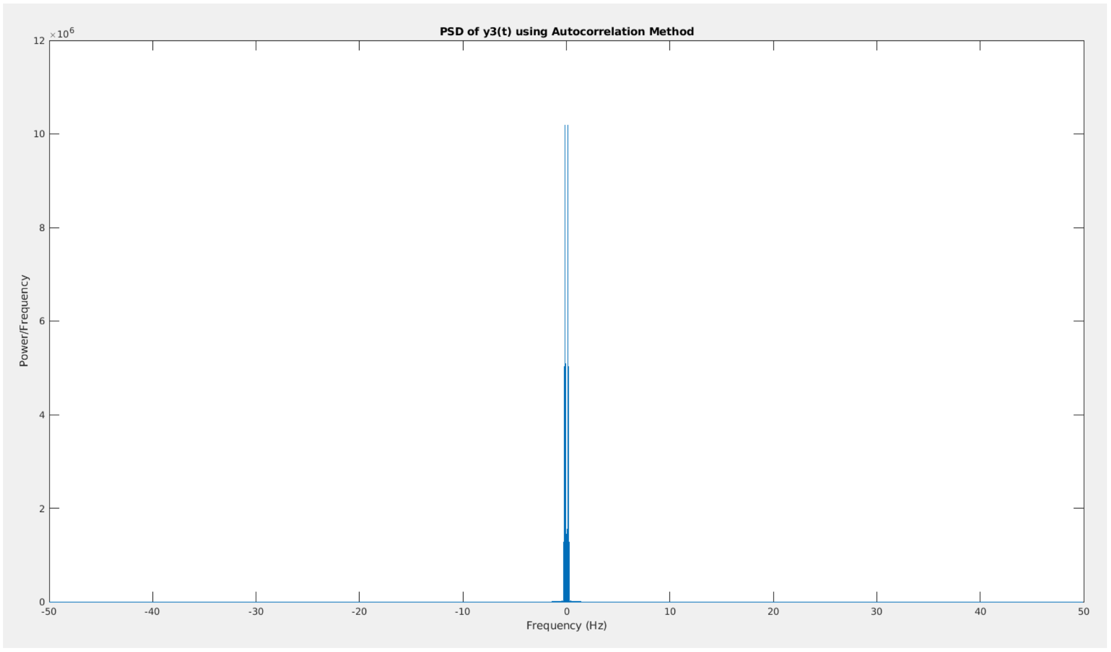


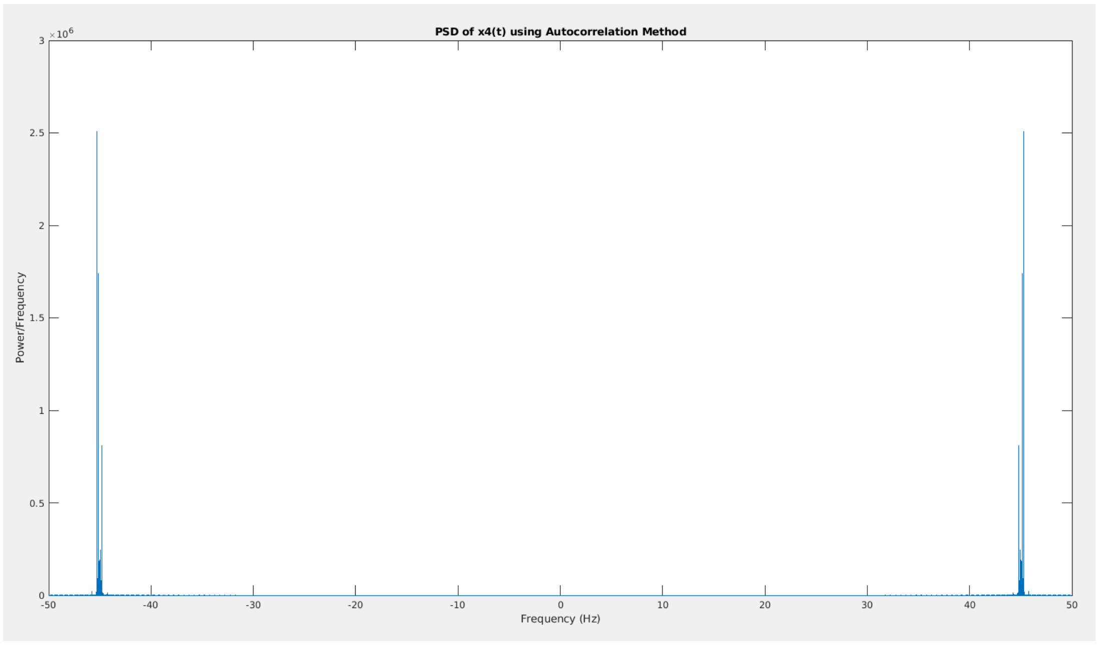

PSD of x4(t) using Autocorrelation Method
$\times 10^6$
2.5
2
1.5
1
0.5
0
Power/Frequency
-50
-40
-30
-20
-10
0
10
20
30
40
50
Frequency (Hz)

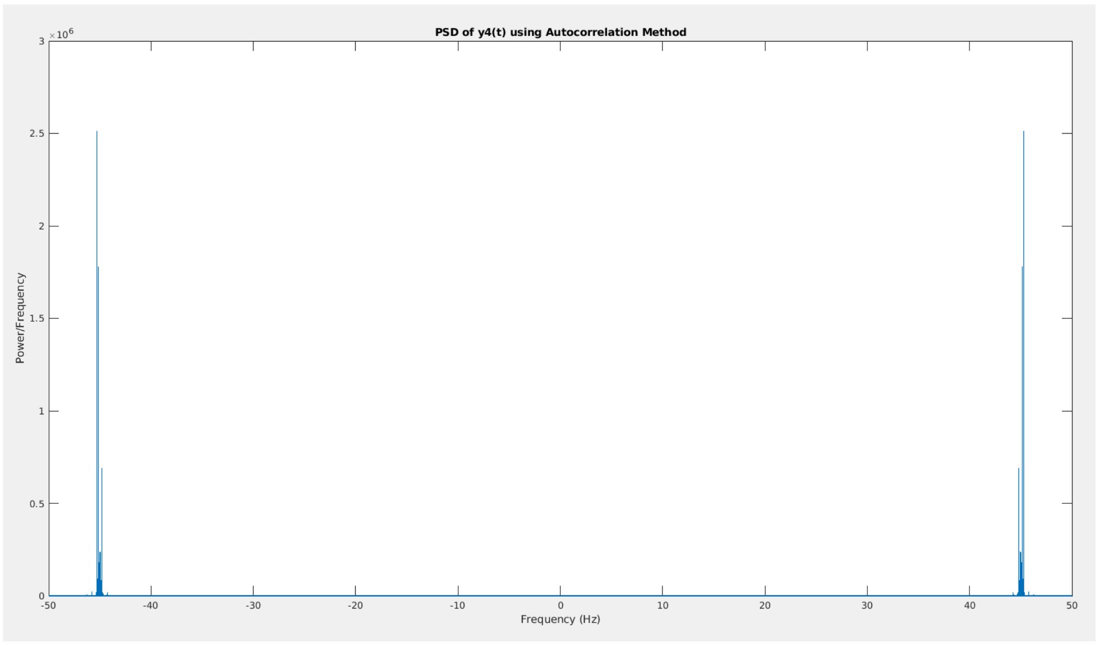

PSD of y4(t) using Autocorrelation Method
$\times 10^6$
2.5
2
1.5
1
0.5
0
Power/Frequency
-50
-40
-30
-20
-10
0
10
20
30
40
50
Frequency (Hz)

2. Memoryless channel & Rectangular pulse

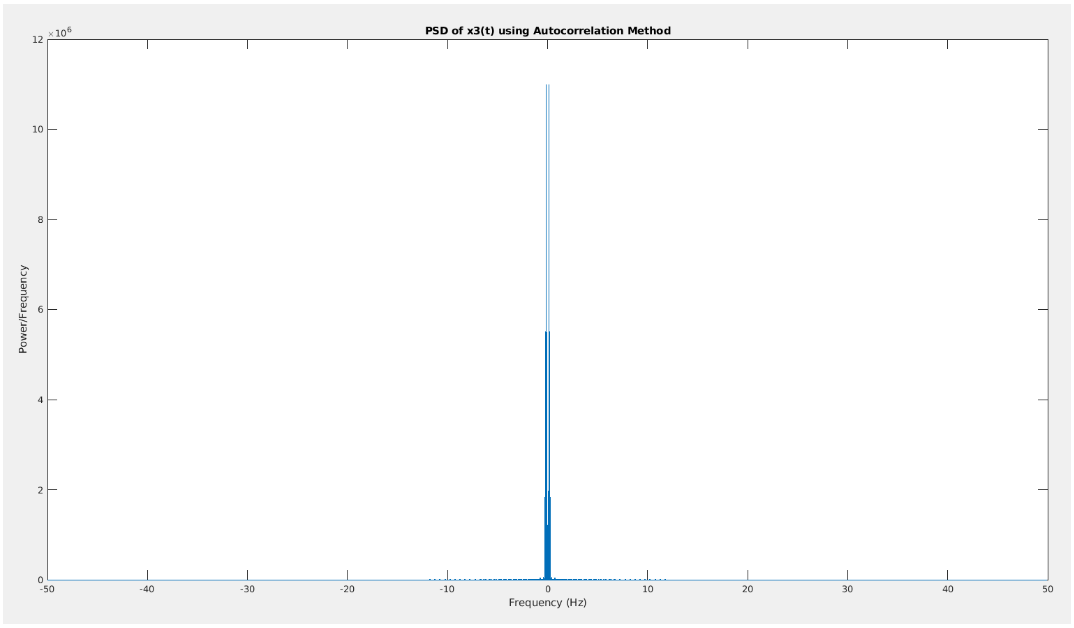


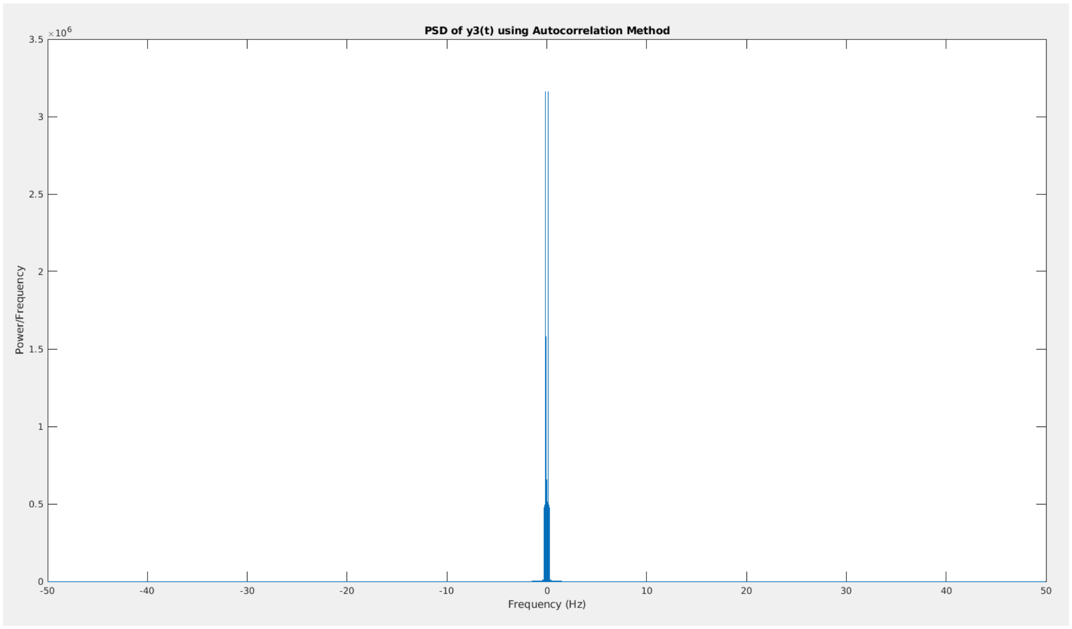


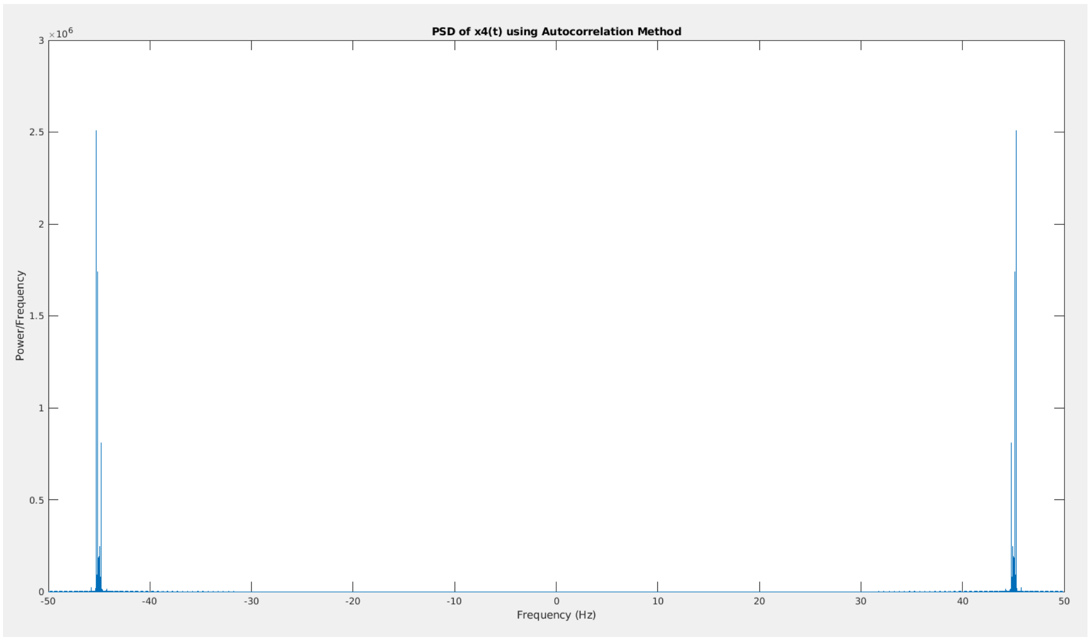

PSD of x4(t) using Autocorrelation Method
×10^6
Power/Frequency
Frequency (Hz)
-50
-40
-30
-20
-10
0
10
20
30
40
50
0
0.5
1
1.5
2
2.5
3


PSD of y4(t) using Autocorrelation Method
×10^6
Power/Frequency
Frequency (Hz)
-50
-40
-30
-20
-10
0
10
20
30
40
50
0
0.5
1
1.5
2
2.5
3

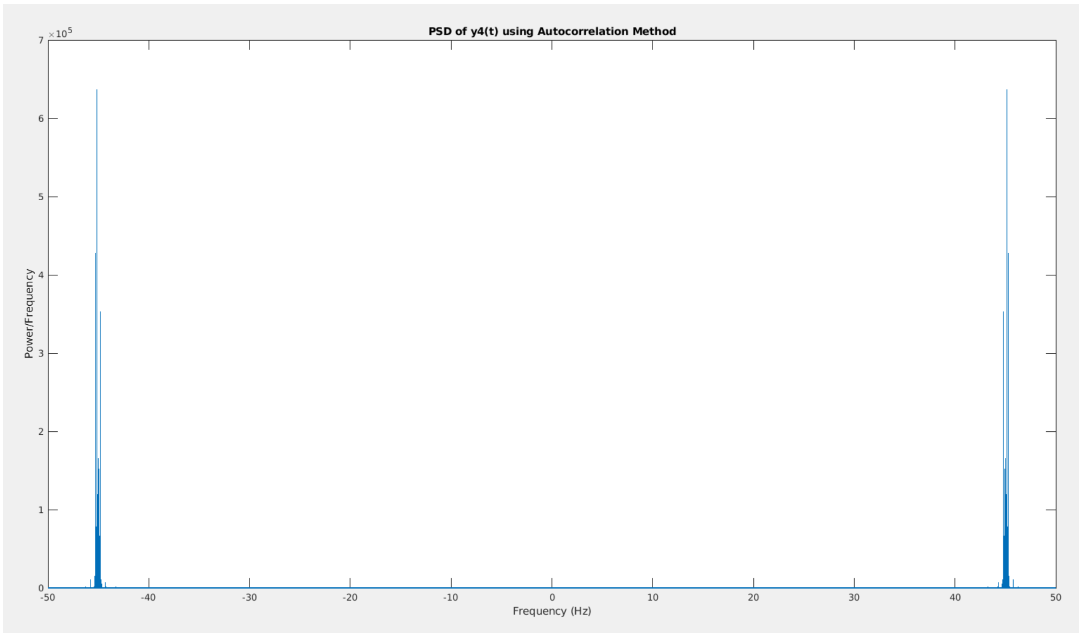
3. Memory channel & Rectangular pulse





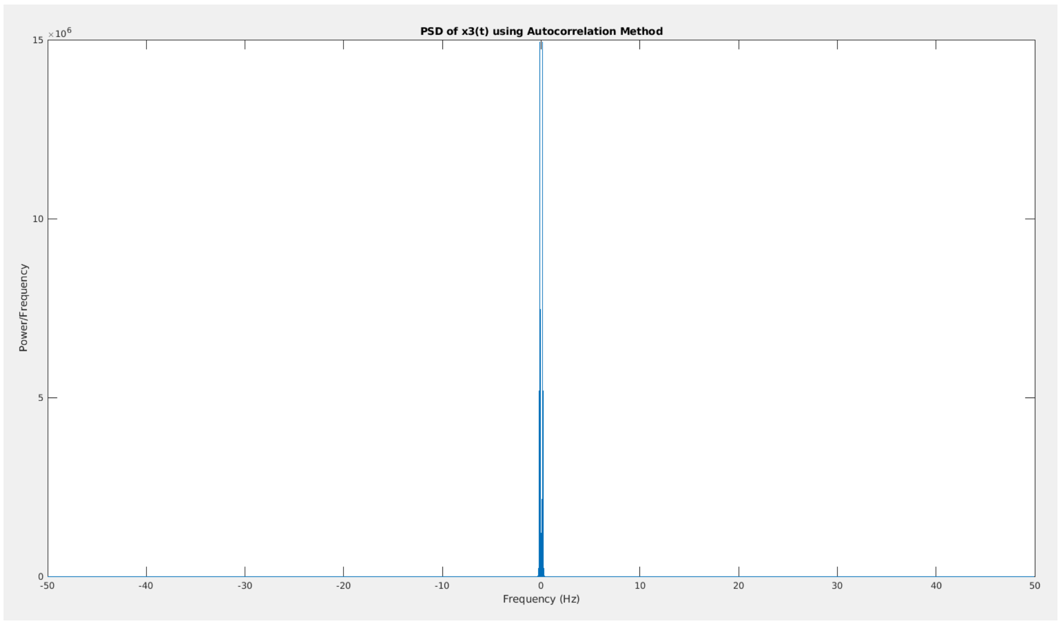


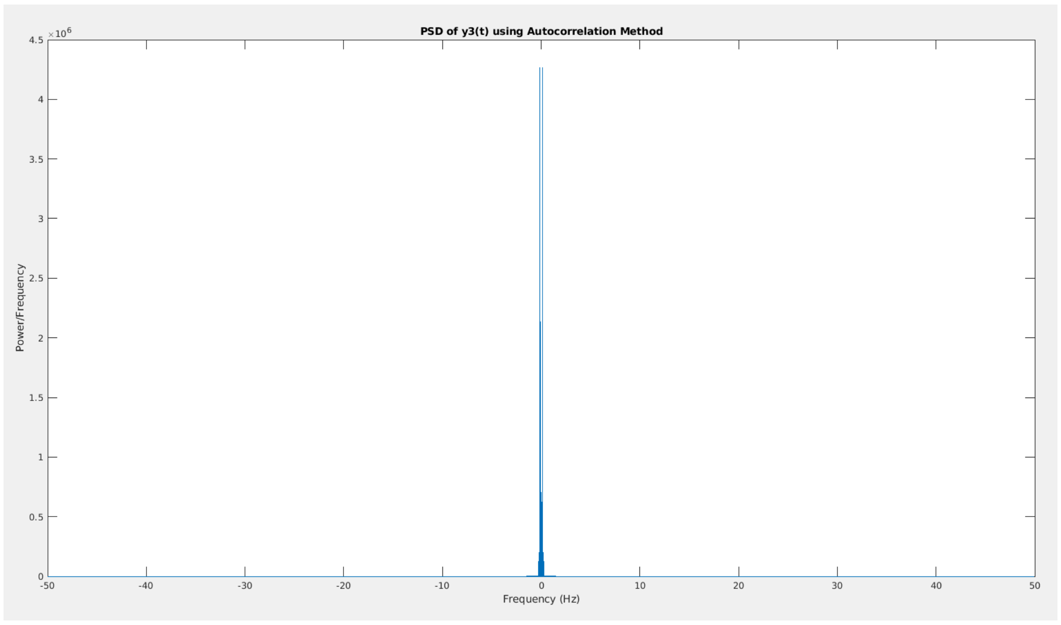
PSD of x4(t) using Autocorrelation Method
$\times 10^6$
Power/Frequency
Frequency (Hz)

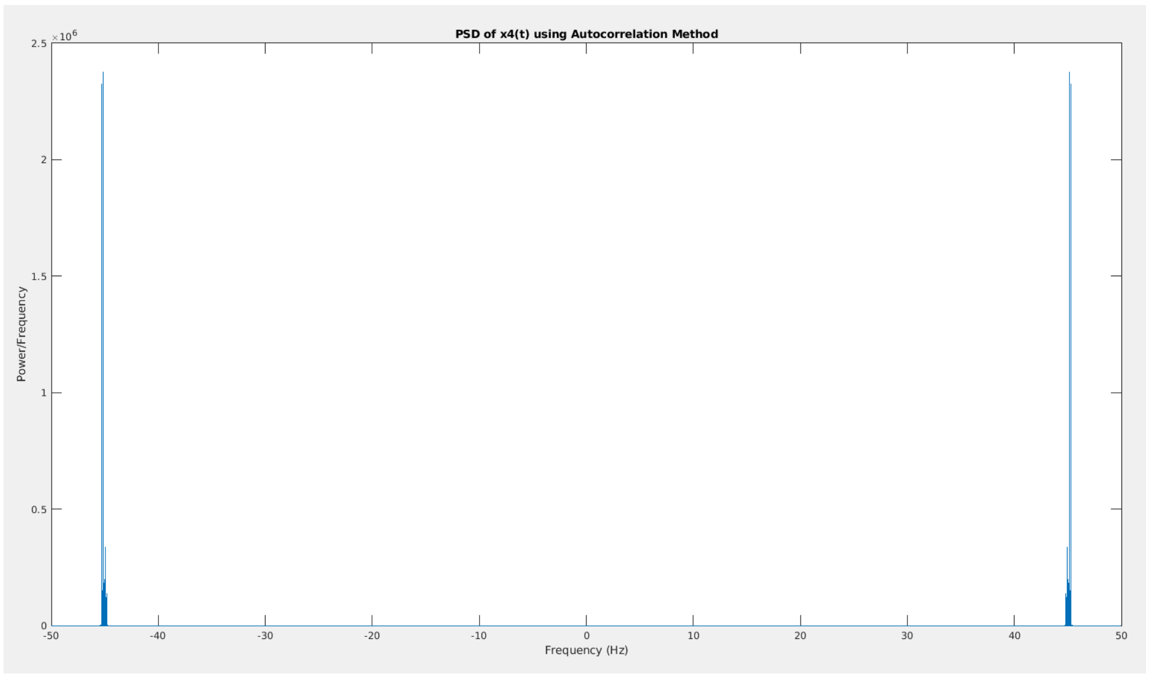


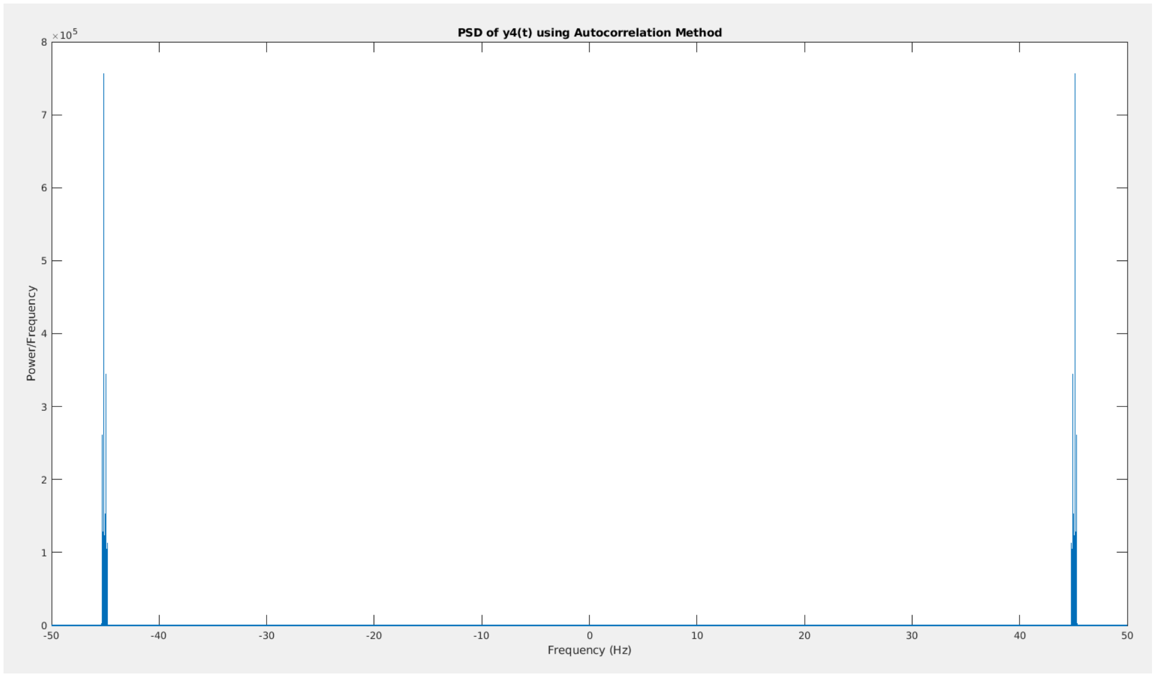
PSD of y4(t) using Autocorrelation Method
$\times 10^5$
Power/Frequency
Frequency (Hz)

4. Memory channel & Raised Cosine Pulse





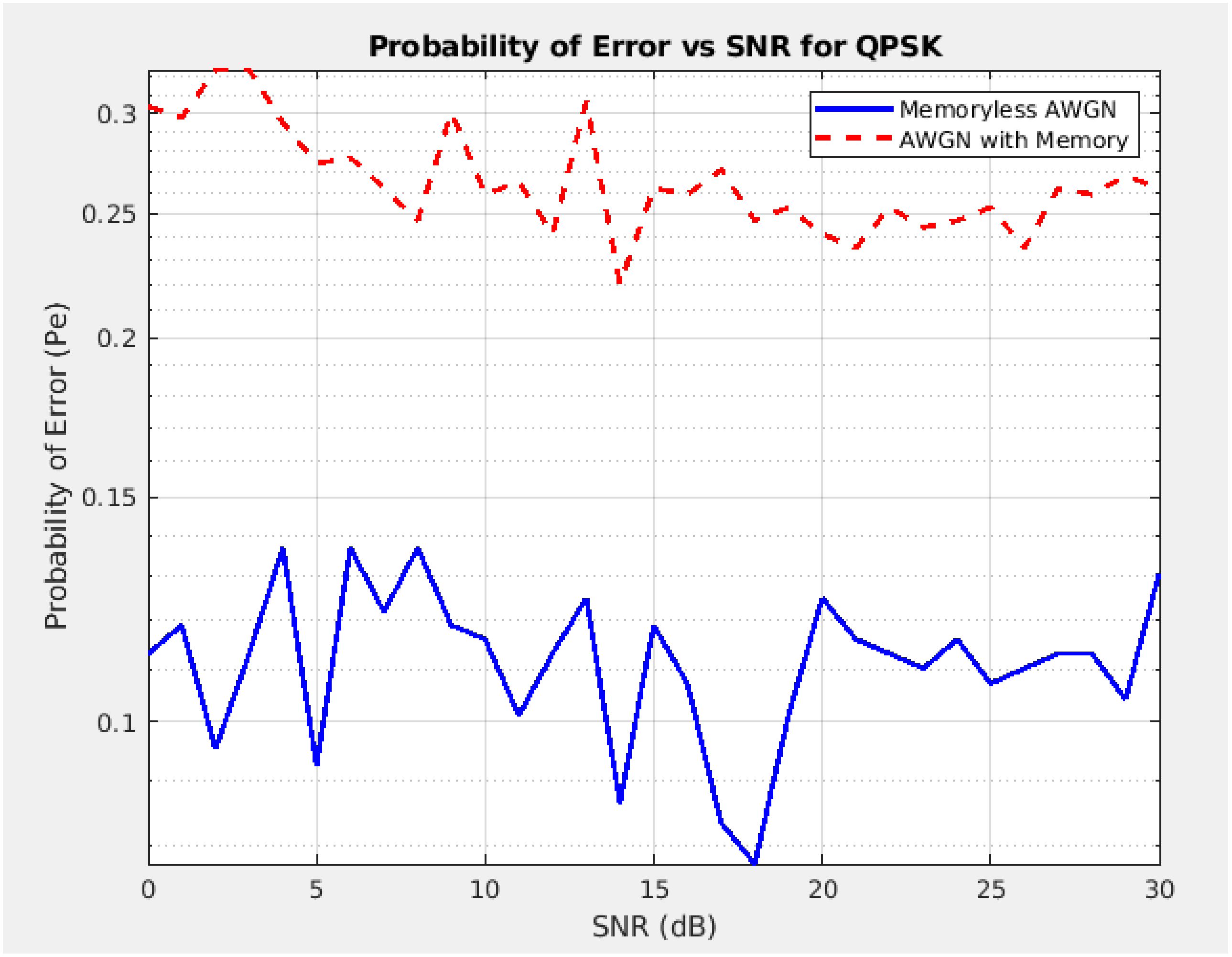

PSD of x4(t) using Autocorrelation Method
×10^6
2.5
2
1.5
1
0.5
0
Power/Frequency
-50
-40
-30
-20
-10
0
10
20
30
40
50
Frequency (Hz)


PSD of y4(t) using Autocorrelation Method
×10^5
8
7
6
5
4
3
2
1
0
Power/Frequency
-50
-40
-30
-20
-10
0
10
20
30
40
50
Frequency (Hz)

Observations from PSD's:

We observe before modulation, x3(t), is baseband (frequencies at w=0) based on raised cosine or rectangular pulse. But here, since our sampling frequency is 100 (due to high number of bits - more computations - more time), the fft is limited to 0 to Fs/2 (frequency resolution of fft inbuilt function in matlab).

BER Plots for both noises:



We can observe that BER for memory AWGN is higher.

Overlap at Decision Points: In the AWGN channel with memory, the impulse response introduces overlap between consecutive symbols. This overlap can lead to interference at decision points, where the receiver must determine the transmitted symbol based on the received signal. The presence of overlapping symbols can increase the likelihood of incorrect symbol decisions, especially if the overlap is significant or if the channel impulse response is poorly matched to the symbol duration.

Integration Over Symbol Duration: When integrating the received signal over the symbol duration (from 0 to T_s), as commonly done in symbol detection, the AWGN channel with memory accumulates not only the energy from the current symbol but also energy from previous symbols due to the filtering effect of the channel impulse response. This accumulation of energy from multiple symbols can make it more challenging to distinguish between symbols, leading to higher BER.

Interference from Consecutive Signals: The filtering effect of the channel impulse response can cause interference from consecutive symbols, especially if the delay introduced by the impulse response is comparable to or greater than the symbol duration. This interference can distort the received signal and make it more difficult for the receiver to accurately detect the transmitted symbols, resulting in a higher BER.

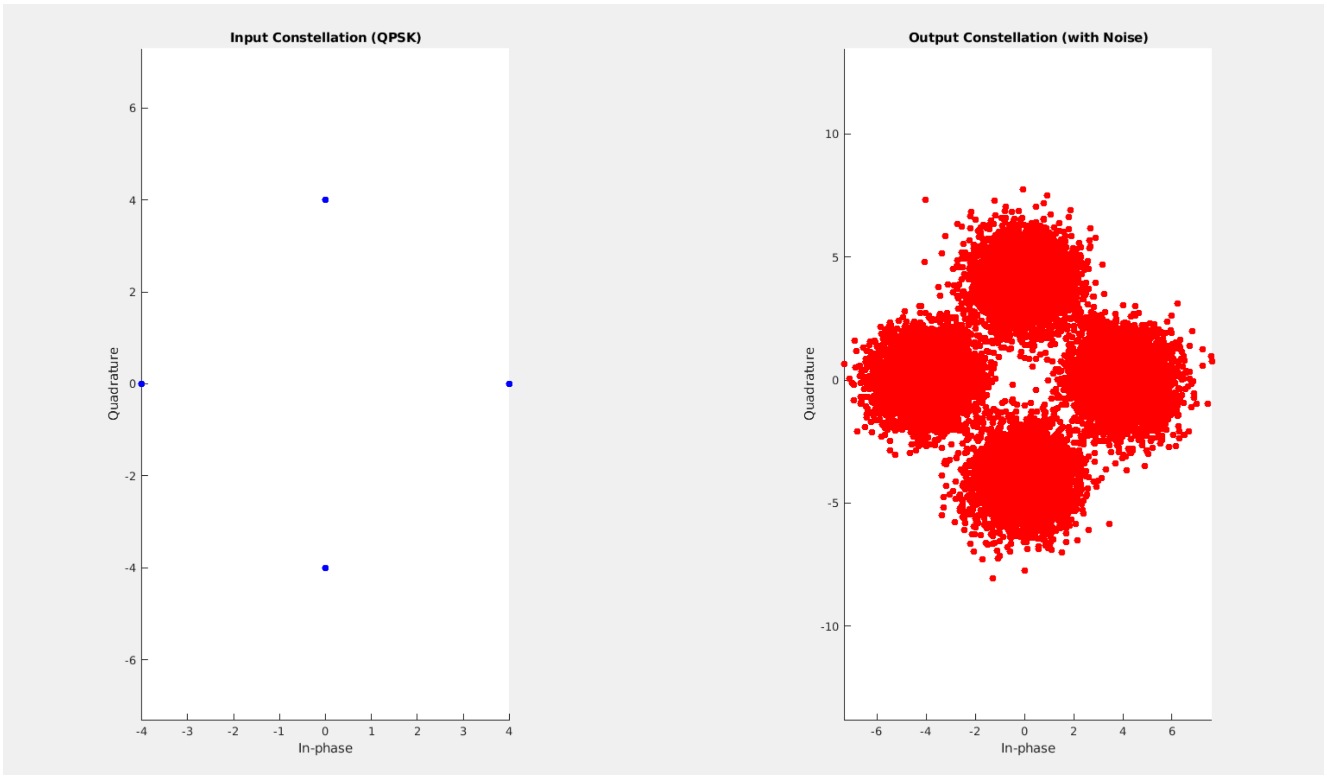
Non-Ideal Channel Response: The channel impulse response may not perfectly match the desired filtering characteristics, leading to non-ideal behavior such as amplitude variations, phase distortion, or time dispersion. These deviations from the ideal response can introduce additional uncertainty and variability in the received signal, contributing to errors in symbol detection and decoding.

Increased Sensitivity to Noise: The presence of memory in the channel can make the communication system more sensitive to noise and disturbances. Noise that is correlated over time, such as colored noise or noise with long-term dependencies, can interact more strongly with the transmitted signal in the presence of channel memory, potentially leading to higher error rates compared to memoryless channels.

By considering these factors, it becomes evident that the presence of memory in the channel introduces additional complexities and challenges in signal detection and decoding, which can contribute to a higher BER compared to memoryless channels.

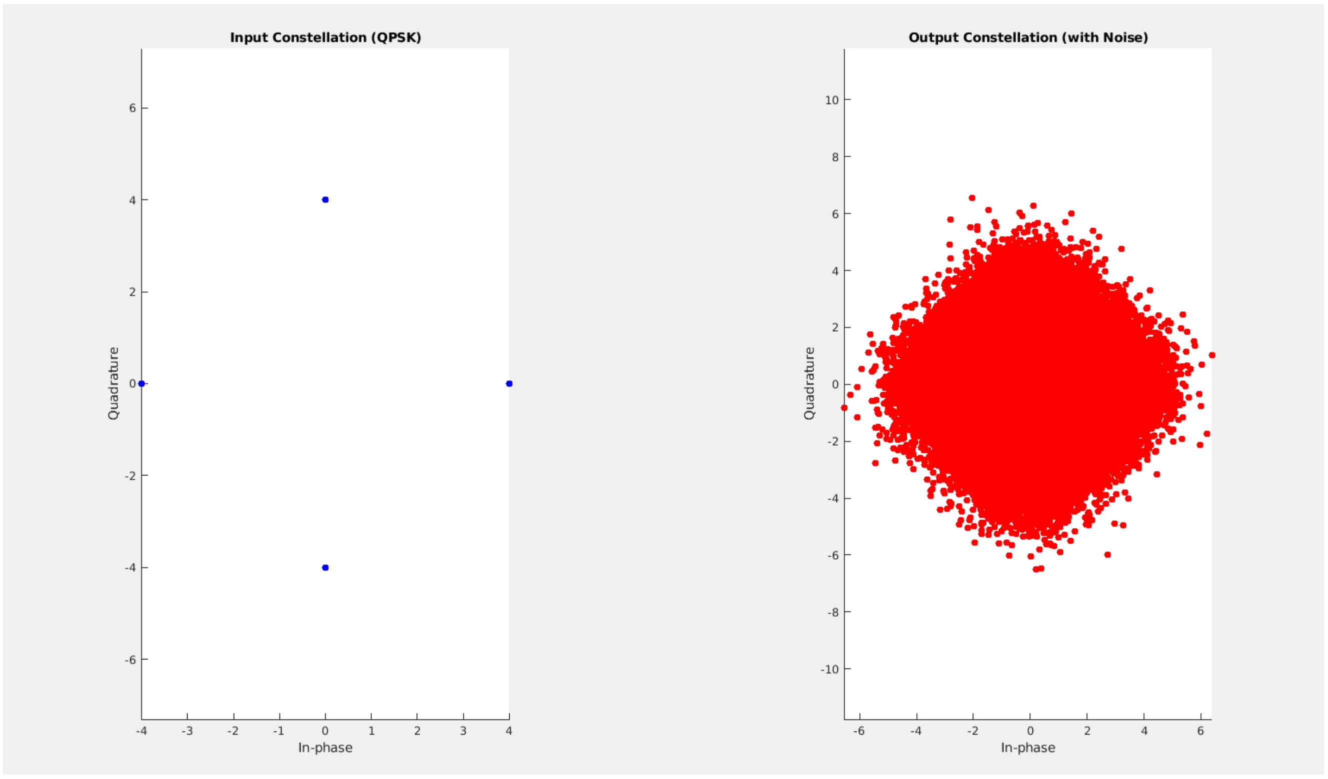
Plots of Input and Output Constellations:

For memoryless AWGN:



In the Memoryless channel, the noise affects the QPSK symbols uniformly, causing a spread or dispersion around each symbol point. However, the spread is relatively uniform across all symbols. The Memoryless channel tends to distort symbols uniformly, spreading them out in all directions.

For memory AWGN:



In contrast, the Memory channel introduces more structured noise, leading to clusters or groups of points around each symbol. This structured noise can distort the symbol points and introduce more errors in symbol detection.

Due to the added noise and distortion, the Memory channel typically results in a higher error probability compared to the Memoryless channel. The Memory channel exhibits memory effects, which means that the noise and distortion characteristics can vary over time and depend on previous signal samples. This can lead to time-varying channel conditions and varying levels of distortion for different symbol sequences.

The Memoryless channel may offer better overall symbol detection performance due to its uniform noise distribution, but it may not account for realistic channel behaviors.

The Memory channel, while introducing more distortion and noise, better simulates real-world channel conditions with memory effects and non-uniform noise characteristics.

Also, it depends highly on the ‘a’ and ‘b’.