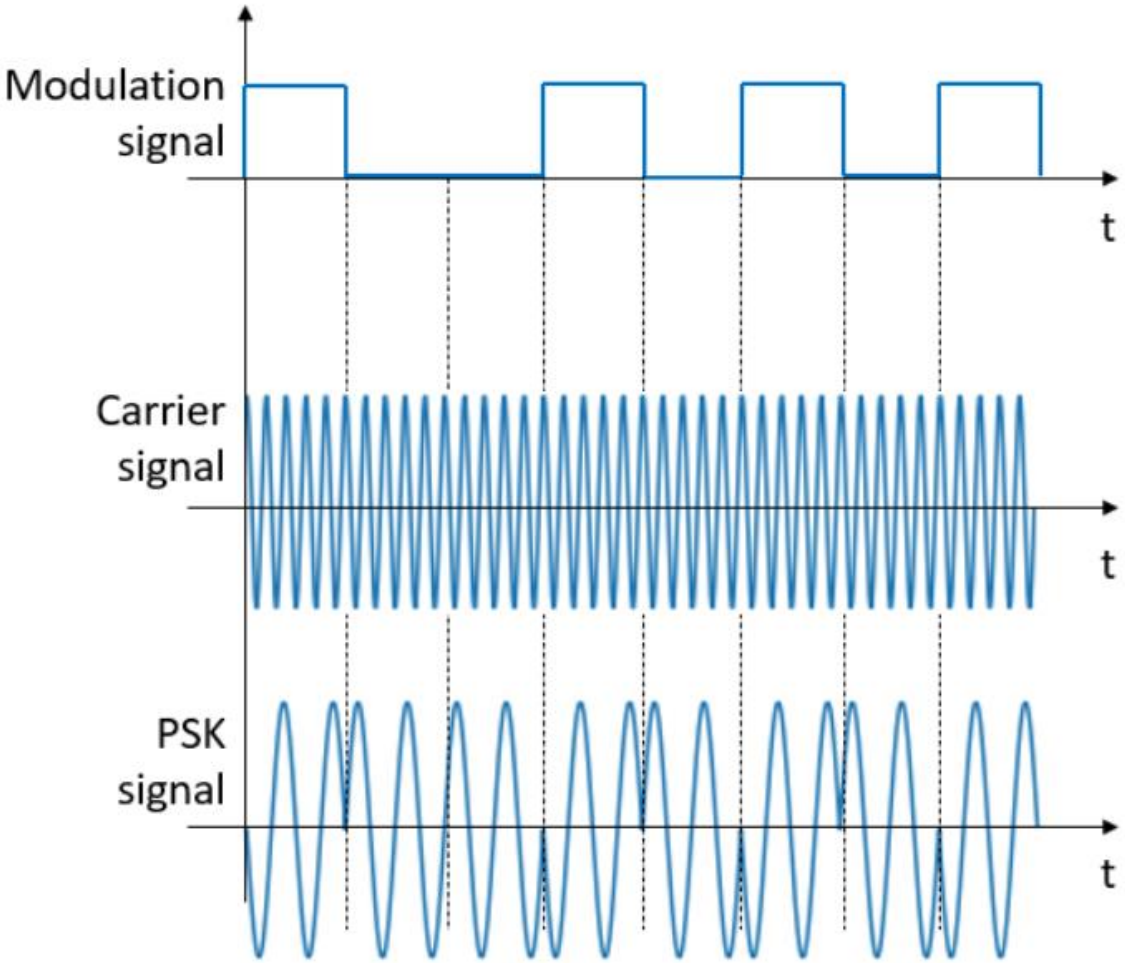
Introduction

Initially, PSK modulation is a technique where the phase of a carrier analog signal is modulated to encode a discrete signal, allowing it to be transmitted through analog channels.



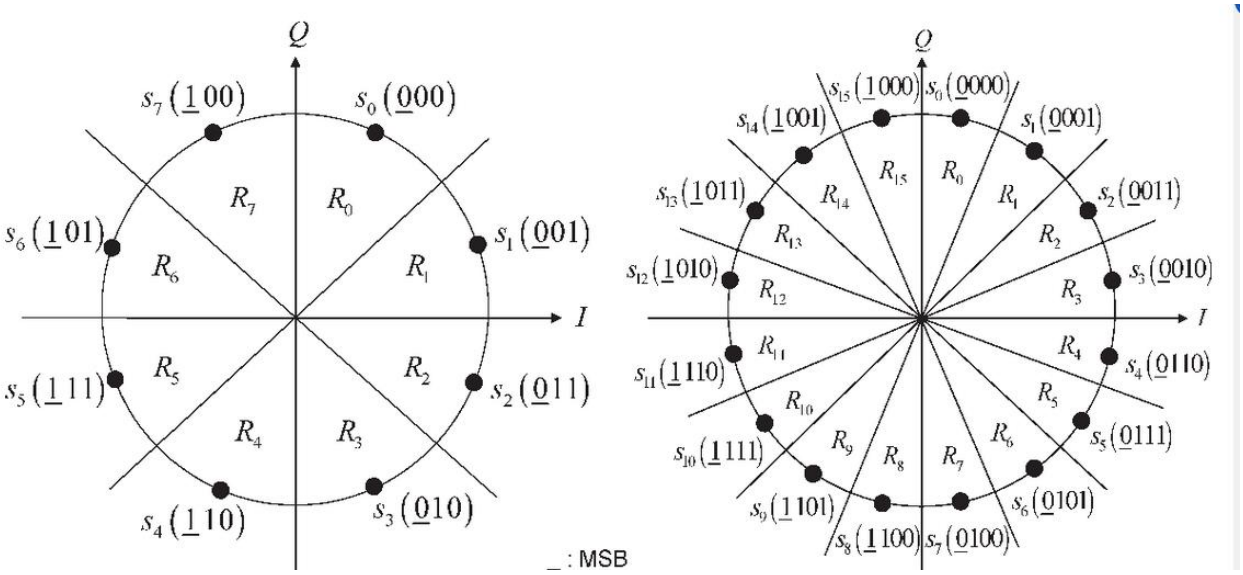
Εικόνα 1. Κλασσικό σύστημα PSK

We observe that the classic method has 2 phase levels, either 0 or 180 degrees. This means the decision-making process is relatively simple, as it has a large margin for potential error due to the wide phase range.

Continuing, when we talk about M-PSK, the "M" represents the number of different available phase shifts or symbols. The use of the so-called constellation diagram becomes necessary to separate the phase angles based on the number of different phases. Additionally, M indicates how many bits are used to encode the symbols (i.e., 2K2^K2K bits = M symbols). For example, 8-PSK has eight possible phase states and 3 bits per symbol, while 16-PSK has sixteen phase states and 4 bits per symbol. The number

of bits per symbol increases with the number of phase shifts, with each phase representing a specific combination of bits.

The M-PSK system operates on the principle of dividing the phase circle into M distinct phase angles. Each angle corresponds to a unique symbol. For instance, in 8-PSK, the phase circle is divided into 8 equally spaced segments of 45 degrees each, while in 16-PSK, it is divided into 16 segments of 22.5 degrees each. During each symbol period, a specific phase angle is selected, corresponding to the group of bits being transmitted.



Εικόνα 2. Κύκλος αστερισμών M-PSK

The advantages of M-PSK over the classic method include better data transmission efficiency, as we have more bits per symbol (i.e., it makes better use of the available bandwidth). However, as we increase M, the BER (bit error rate) also increases because the symbols (i.e., phases) are closer to each other, making it more likely that noise will shift the signal to a different region of the phase circle, leading the detector to make an incorrect decision.

Question 1

A) Without gray coding

Binary entrance sequence

First, we need to generate the input bit sequence as requested. This sequence will be random, with each symbol (0 or 1) being equally likely to appear.

```
%% Generate Binary Input Sequence
bits = randi([0, 1], Lb, 1);
```

Specifically, we define that 100,000 bits will be generated.

```
Lb = 1e5;
```

Additionally, we ensure that the number of bits is divisible by the number of bits per symbol, so that no extra bits remain and there is no shortage for encoding all the symbols. The value kkk (bits per symbol) is explained below on how it is calculated.

```
if mod(Lb, k) ~= 0
    Lb_adjusted = floor(Lb / k) * k;
    Lb = Lb_adjusted;
end
```

Mapping bits to symbols

We are required to group the generated bits into sets of k bits in order to encode the symbols. This is done after calculating k based on M (by finding the logarithm of M with base 2), and then we use the reshape function to create the groups.

```
M = 16;                         % Modulation order (8-PSK or 16-PSK)
k = log2(M);
```

Below, while it would be quite simple to directly take the numbers from the sequence and assign them to the correct symbols (e.g., for 4-PSK, we take 00 and map it directly to 00 and 11 to 11), we prefer to first convert the bit groups into decimal numbers (e.g., for 16-PSK, we get values from 0 to 15) and then, based on these, assign the group to the correct symbol.

```
binary_symbols = bi2de(bit_groups, 'left-msb');
theta_m = 2 * pi * m / M;
```

Orthogonal pulse

We initialize all the elements of the exercise exactly as they are given.

```
Rs = 250e3;                 % Symbol rate (symbols per second)
Tsym = 1 / Rs;              % Symbol period (seconds)
Fs = 10e6;                  % Sampling frequency (Hz)
Tsample = 1 / Fs;           % Sampling period (seconds)
Ns = Tsym / Tsample;        % Number of samples per symbol
Ns = round(Ns);             % Ensure Ns is an integer
```

We generate the rectangular pulse, which is applied to the in-phase (I) and quadrature (Q) components of the signal. These components represent the two orthogonal parts of the signal that carry the phase and amplitude information of the symbols.

```
pulse_shape = sqrt(2 / Ns) * ones(1, Ns);  % Rectangular pulse with
unit energy
for idx = 1:num_symbols
    idx_range = (idx-1)*Ns + 1 : idx*Ns;
    m = binary_symbols(idx);     % No Gray coding applied
    theta_m = 2 * pi * m / M;
    I_baseband_total(idx_range) = cos(theta_m) * pulse_shape;
    Q_baseband_total(idx_range) = sin(theta_m) * pulse_shape;
end
```

Modulation M-PSK

After constructing the I/Q components of the baseband signal, the next step is to modulate them onto a high-frequency carrier wave. This is necessary because the baseband signal itself is low-frequency and needs to be shifted to a higher frequency for transmission. The carrier modulation is performed using the cosine and sine waves of the carrier frequency Fc.

```
carrier_cos_total = cos(2 * pi * Fc * t_total);
carrier_sin_total = sin(2 * pi * Fc * t_total);
```

Once the carrier signals are defined, the in-phase (I) and quadrature (Q) components are modulated onto the carrier.

```
s_t = I_baseband_total .* carrier_cos_total + Q_baseband_total .*
carrier_sin_total;
```

AWGN Channel

We set the required SNR value

```
SNR_dB = 20;
SNR = 10^(SNR_dB / 10);
```

Next, we proceed to calculate the noise. The noise spectral density (N_0) is the ratio of energy per bit (E_b) to the SNR, meaning it represents how much noise power is present in the channel relative to the bit energy. Additionally, the noise variance (σ^2) is calculated by dividing the noise spectral density by 2. This is because the noise affects both the real and imaginary components of the signal (as the signal is complex in this case due to the modulation).

```
N0 = Eb / SNR;  % Noise spectral density
sigma2 = N0 / 2;  % Noise variance (for real and imaginary components)
```

The real noise samples are generated using a normal (Gaussian) distribution, where the noise variance determines the spread of the values.

```
noise = sqrt(sigma2) * randn(1, length(s_t));  % Generate noise
r_t = s_t + noise;  % Add noise to the transmitted signal
```

Demodulator M-PSK

The demodulation begins with separating the I (in-phase) and Q (quadrature) components of the received signal, using the same carrier frequencies that were used in the modulation process.

```
r_I = r_t .* carrier_cos_total;
r_Q = r_t .* carrier_sin_total;
```

After separating the I and Q components, the next step is to compute the average of the samples over each symbol period. The transmitted signal was sampled multiple times per symbol (with Ns samples per symbol), and now the demodulator calculates the average of these samples to obtain the final I and Q values that represent each symbol.

```
for idx = 1:num_symbols
    idx_range = (idx-1)*Ns + 1 : idx*Ns;
    I_r = sum(r_I(idx_range)) / Ns;  % Normalize by Ns to average the
values
    Q_r = sum(r_Q(idx_range)) / Ns;  % Normalize by Ns to average the
values
    received_symbols(idx, :) = [I_r, Q_r];  % Store the received I/Q
```

```
symbols
end
```

Decider M-PSK

The demodulator compares the received I/Q values with the constellation points (the ideal I/Q values for each symbol) and selects the closest constellation point. This process is done using a minimum distance criterion.

```
for idx = 1:num_symbols
    r = received_symbols(idx, :);
    distances = sum((constellation - r).^2, 2);
    [~, min_idx] = min(distances);
    detected_symbols(idx) = min_idx - 1;
end
```

Once the symbols are detected, the final step is to map them back to their corresponding bit sequences. This is the reverse process of what was done during modulation.

```
binary_symbols_detected = detected_symbols;  % No Gray-to-binary
conversion needed
detected_bits_matrix = de2bi(binary_symbols_detected, k, 'left-msb');
estimated_bits = reshape(detected_bits_matrix.', [], 1);
```

B) With gray coding

The only parts that change are the bit-to-symbol mapping process.

```
binary_symbols = bi2de(bit_groups, 'left-msb');  % Convert to decimal
symbols
gray_symbols = bitxor(binary_symbols, floor(binary_symbols / 2));  %
Gray code
```

Similarly, we change all sections that contain the symbols with the Gray-coded symbols.