**8 Generate 16-QAM Modulation and obtain the QAM constellation**

**CODE:**

clc;

clear;

close all;

% Parameters

N = 1000; % Number of bits

M = 16; % 16-QAM (16 symbols, 4 bits per symbol)

k = log2(M); % Bits per symbol for 16-QAM

SNR\_dB = 20; % Signal to noise ratio in dB

% Generate random binary data

data\_bits = randi([0 1], 1, N); % Reshape bits into groups of 4 (for 16-QAM)

symbols\_bits = reshape(data\_bits, k, []).'; % Map bits to 16-QAM symbols using Gray coding

% Define the constellation points for 16-QAM (normalized)

% QAM constellation (-3,-1,1,3) for real and imaginary parts

qam\_levels = [-3 -1 1 3];

qam\_symbols = zeros(1, size(symbols\_bits, 1));

for i = 1:size(symbols\_bits, 1)

% Convert first 2 bits to in-phase (I) value

I\_bits = bi2de(symbols\_bits(i, 1:2), 'left-msb');

I\_val = qam\_levels(I\_bits + 1);

% Convert last 2 bits to quadrature (Q) value

Q\_bits = bi2de(symbols\_bits(i, 3:4), 'left-msb');

Q\_val = qam\_levels(Q\_bits + 1);

% 16-QAM symbol is the combination of I and Q components

qam\_symbols(i) = I\_val + 1i\*Q\_val;

end

% Normalize the power of the QAM symbols to 1

qam\_symbols = qam\_symbols / sqrt(mean(abs(qam\_symbols).^2));

% Display the 16-QAM constellation

figure;

plot(real(qam\_symbols), imag(qam\_symbols), 'bo', 'MarkerSize', 6, 'LineWidth', 2);

xlabel('In-phase (I)');

ylabel('Quadrature (Q)');

title('16-QAM Constellation Diagram');

grid on;

axis([-2 2 -2 2]);

% Simulate AWGN Channel (optional)

rx\_signal = awgn(qam\_symbols, SNR\_dB, 'measured'); % Add noise with SNR

% Display the received constellation with noise

figure;

plot(real(rx\_signal), imag(rx\_signal), 'ro', 'MarkerSize', 6, 'LineWidth', 2);

xlabel('In-phase (I)');

ylabel('Quadrature (Q)');

title(['16-QAM Constellation Diagram with Noise (SNR = ' num2str(SNR\_dB) ' dB)']);

grid on;

axis([-2 2 -2 2]);

**Theory:**

Quadrature Amplitude Modulation (QAM) is a modulation technique that combines both amplitude modulation (AM) and phase modulation (PM) to encode data in digital communication systems. It uses two carrier signals that are 90° out of phase with each other, known as the in-phase (I) and quadrature (Q) components. By varying the amplitude of both components simultaneously, multiple distinct symbols can be created, each representing a unique combination of I and Q values.

QAM is highly efficient in terms of spectral utilization, allowing the transmission of multiple bits per symbol. The number of bits per symbol depends on the modulation order (M), where M=2kM = 2^kM=2k, and kkk is the number of bits per symbol. For example, 16-QAM transmits 4 bits per symbol, while 64-QAM transmits 6 bits per symbol. The higher the modulation order, the more bits can be transmitted in the same bandwidth, improving data rates.

However, higher-order QAM schemes, while increasing data rate, also make the system more sensitive to noise. As the number of symbols increases, the constellation points (symbol locations) become closer together, making it harder to distinguish between them in noisy environments.

QAM is widely used in modern communication systems, including digital television, wireless networks (Wi-Fi, LTE, 5G), and cable modems, due to its high data transmission efficiency.

**Explanation :**

This MATLAB code simulates a 16-QAM (Quadrature Amplitude Modulation) system, which is widely used in communication systems for high data rates. Let's break down the code and its key components:

**1. Parameters Initialization:**

N = 1000; % Number of bits

M = 16; % 16-QAM (16 symbols, 4 bits per symbol)

k = log2(M); % Bits per symbol for 16-QAM (log2(16) = 4 bits)

SNR\_dB = 20; % Signal-to-noise ratio in dB

* N = 1000: The number of bits to be transmitted.
* M = 16: This specifies that 16-QAM is used, meaning there are 16 possible symbols.
* k = log2(M): This calculates the number of bits per symbol (for 16-QAM, this is 4 bits).
* SNR\_dB = 20: The signal-to-noise ratio in decibels for the simulation.

**2. Generating Random Binary Data:**

data\_bits = randi([0 1], 1, N);

* This generates a random binary sequence of N bits (1000 bits in this case).

**3. Reshape the Bits into Symbol Groups:**

symbols\_bits = reshape(data\_bits, k, []).';

* This reshapes the data into k-bit groups (4 bits per symbol), which is necessary for mapping the bits to the 16-QAM symbols.

**4. Mapping Bits to 16-QAM Symbols:**

qam\_levels = [-3 -1 1 3]; % QAM levels for real and imaginary parts

qam\_symbols = zeros(1, size(symbols\_bits, 1)); % Initialize array for symbols

* qam\_levels = [-3 -1 1 3]: These are the possible values for the real (I) and imaginary (Q) components of the QAM symbols.
* The qam\_symbols array is initialized to store the resulting symbols after mapping.

For each symbol, the 4 bits are split into 2 bits for the I (in-phase) component and 2 bits for the Q (quadrature) component:

for i = 1:size(symbols\_bits, 1)

% Convert first 2 bits to in-phase (I) value

I\_bits = bi2de(symbols\_bits(i, 1:2), 'left-msb');

I\_val = qam\_levels(I\_bits + 1); % Map to QAM level (I)

% Convert last 2 bits to quadrature (Q) value

Q\_bits = bi2de(symbols\_bits(i, 3:4), 'left-msb');

Q\_val = qam\_levels(Q\_bits + 1); % Map to QAM level (Q)

% 16-QAM symbol is the combination of I and Q components

qam\_symbols(i) = I\_val + 1i\*Q\_val; % Construct the complex QAM symbol

end

* For each symbol, the first two bits are mapped to an I value, and the last two bits are mapped to a Q value.
* These I and Q values are then combined to form a complex number representing the QAM symbol.

**5. Normalizing the QAM Symbols:**

qam\_symbols = qam\_symbols / sqrt(mean(abs(qam\_symbols).^2));

* The QAM symbols are normalized so that their average power is 1. This is done by dividing by the square root of the mean squared magnitude of the symbols. This ensures that the symbols' average power is unity, which is important for proper signal-to-noise ratio (SNR) calculation.

**6. Display the 16-QAM Constellation Diagram:**

figure;

plot(real(qam\_symbols), imag(qam\_symbols), 'bo', 'MarkerSize', 6, 'LineWidth', 2);

xlabel('In-phase (I)');

ylabel('Quadrature (Q)');

title('16-QAM Constellation Diagram');

grid on;

axis([-2 2 -2 2]);

* This plot shows the constellation of the 16-QAM symbols, where the real and imaginary parts of the complex symbols are plotted on the x and y axes, respectively.
* The symbols are displayed as blue circles ('bo'), and the plot is labeled accordingly.

**7. Simulate an AWGN Channel:**

rx\_signal = awgn(qam\_symbols, SNR\_dB, 'measured'); % Add noise with SNR

* This line simulates the transmission of the QAM symbols through an Additive White Gaussian Noise (AWGN) channel. Noise is added to the signal with the specified SNR in dB.
* The 'measured' option ensures that the noise is scaled so that the SNR is applied correctly.

**8. Display the Received Constellation with Noise:**

figure;

plot(real(rx\_signal), imag(rx\_signal), 'ro', 'MarkerSize', 6, 'LineWidth', 2);

xlabel('In-phase (I)');

ylabel('Quadrature (Q)');

title(['16-QAM Constellation Diagram with Noise (SNR = ' num2str(SNR\_dB) ' dB)']);

grid on;

axis([-2 2 -2 2]);

* This plot shows the received QAM symbols after noise is added.
* The received symbols are shown as red circles ('ro'), with the SNR value displayed in the title.
* You can observe the effect of noise by comparing this plot with the previous one (without noise).

**Full MATLAB Code Overview:**

* The code first generates random binary data and then maps it to 16-QAM symbols.
* It normalizes the symbols for unit power and displays the constellation before noise is added.
* The received signal is simulated by adding AWGN noise, and the resulting noisy constellation is plotted.
* This simulates the behavior of a 16-QAM system in a noisy environment, which is typical in real-world communication systems.

**Potential Improvements and Extensions:**

1. **Demodulation**: To fully simulate the system, you can add a demodulation step that decodes the received noisy symbols back into binary data. This typically involves finding the closest constellation points to the received symbols.
2. **Error Rate Calculation**: After demodulation, you can compare the decoded data with the original transmitted data and compute the Bit Error Rate (BER) to evaluate the performance of the system.
3. **Varying SNR**: You could vary the SNR over a range of values to see how the performance of the system changes under different noise levels.

Sample outputs:

**Output Plots:**

1. **16-QAM Constellation Diagram (Before Noise)**:
   * This plot shows the ideal constellation of the 16-QAM symbols with the points at their expected locations. The real part (I) is plotted on the x-axis, and the imaginary part (Q) is plotted on the y-axis.

**Plot Description**:

* + Blue circles represent the 16-QAM symbols.
  + The symbols are placed at positions like (-3, -3), (-3, -1), (3, 3), etc., as defined by the qam\_levels.

**Expected Plot Output**:

* + 16 well-separated points forming the 16-QAM constellation.

1. **16-QAM Constellation Diagram (After Noise with SNR = 20 dB)**:
   * This plot shows how the constellation looks after noise is added with an SNR of 20 dB.

**Plot Description**:

* + Red circles represent the received symbols.
  + The points in this plot are spread out from the ideal positions due to the noise, but they should still be roughly around the expected constellation points, with some scatter.

**Expected Plot Output**:

* + The symbols are still clustered around the ideal positions but with noticeable deviation due to the noise. The SNR of 20 dB means that the noise is moderate but not overwhelming.

**9. Encoding and Decoding of Huffman code**.

CODE

clc;

p=input('Enter the probabilities:');

n=length(p);

symbols=[1:n];

[dict,avglen]=huffmandict(symbols,p);

temp=dict;

t=dict(:,2);

for i=1:length(temp)

temp{i,2}=num2str(temp{i,2});

end

disp('The huffman code dict:');

disp(temp)

fprintf('Enter the symbols between 1 to %d in[]',n);

sym=input(':')

encod=huffmanenco(sym,dict);

disp('The encoded output:');

disp(encod);

bits=input('Enter the bit stream in[];');

decod=huffmandeco(bits,dict);

disp('The symbols are:');

disp(decod);

H=0;

Z=0;

for(k=1:n)

H=H+(p(k)\*log2(1/p(k)));

end

fprintf(1,'Entropy is %f bits',H);

N=H/avglen;

fprintf('\n Efficiency is:%f',N);

for(r=1:n)

l(r)=length(t{r});

end

m=max(l)

s=min(l)

v=m-s;

fprintf('the variance is:%d',v);

Theory:

**Huffman Coding** is a widely used algorithm for data compression, designed to minimize the average length of codes assigned to characters based on their frequencies of occurrence. It is a type of **prefix coding**, where no code is a prefix of another, ensuring that the encoding is uniquely decodable.

The process involves the following steps:

1. **Frequency Analysis**: First, the frequency of each character in the input data is calculated.
2. **Building a Huffman Tree**: A binary tree is constructed where each leaf node represents a character, and the tree is built by repeatedly combining the two least frequent characters into a new node until only one node remains, which becomes the root of the tree.
3. **Assigning Codes**: Once the tree is built, the characters are assigned binary codes based on the path from the root to the leaf node. Moving left adds a '0' and moving right adds a '1' to the code for that character.

The key advantage of Huffman coding is its **variable-length code**, where more frequent characters are assigned shorter codes, and less frequent characters get longer codes, leading to efficient data representation.

Huffman coding is optimal for **lossless data compression**, as it guarantees the minimum possible average code length for a given set of symbol frequencies. It is widely used in file compression formats like ZIP, GZIP, and in image formats like JPEG.

huffman coding description:

1. **Input Probabilities:**

matlab

p = input('Enter the probabilities:');

n = length(p);

symbols = [1:n];

* + The code first asks the user to input a set of probabilities (p), where each probability corresponds to a symbol in a source. The number of symbols is stored in n.

1. **Generate Huffman Dictionary:**

[dict, avglen] = huffmandict(symbols, p);

temp = dict;

t = dict(:, 2);

for i = 1:length(temp)

temp{i, 2} = num2str(temp{i, 2});

end

* + huffmandict is used to generate the Huffman dictionary (dict) for the given symbols and their probabilities (p).
  + The dictionary dict is a cell array, where the first column contains the symbols, and the second column contains the corresponding Huffman codes.
  + The code converts the numeric Huffman codes into strings for display purposes.

1. **Display Huffman Dictionary:**

disp('The huffman code dict:');

disp(temp);

* + Displays the Huffman code dictionary (temp), where the first column is the symbol and the second column is the corresponding Huffman code in string format.

1. **Encode Symbols:**

matlab

Copy code

fprintf('Enter the symbols between 1 to %d in[]', n);

sym = input(':');

encod = huffmanenco(sym, dict);

disp('The encoded output:');

disp(encod);

* + The user is asked to input a sequence of symbols (between 1 and n).
  + huffmanenco is used to encode the sequence of symbols based on the Huffman dictionary, and the encoded output is displayed.

1. **Decode Bitstream:**

bits = input('Enter the bit stream in[];');

decod = huffmandeco(bits, dict);

disp('The symbols are:');

disp(decod);

* + The user is asked to input a bitstream (a sequence of 0s and 1s).
  + huffmandeco is used to decode the bitstream back to the original symbols, and the decoded symbols are displayed.

1. **Calculate Entropy:**

H = 0;

for k = 1:n

H = H + (p(k) \* log2(1 / p(k)));

end

fprintf(1, 'Entropy is %f bits', H);

* + The code calculates the entropy of the source using the formula: H=−k=1∑np(k)log2p(k) where p(k) is the probability of symbol k. This represents the average amount of information per symbol.

1. **Calculate Efficiency:**

N = H / avglen;

fprintf('\n Efficiency is:%f', N);

* + The efficiency is computed as the ratio of the entropy H to the average code length avglen (from the Huffman dictionary). The efficiency measures how effective the Huffman code is in terms of representing the information content of the source.

1. **Calculate Code Length Variance:**

for r = 1:n

l(r) = length(t{r});

end

m = max(l);

s = min(l);

v = m - s;

fprintf('The variance is:%d', v);

* + The code calculates the variance of the Huffman codeword lengths. It finds the maximum and minimum code lengths (m and s), and the variance v is the difference between them.

**Sample 1: Simple Case (4 symbols)**

**Input:**

Enter the probabilities: [0.4 0.3 0.2 0.1]

* This represents a simple source with 4 symbols where:
  + Symbol 1 has a probability of 0.4,
  + Symbol 2 has a probability of 0.3,
  + Symbol 3 has a probability of 0.2,
  + Symbol 4 has a probability of 0.1.

Enter the symbols between 1 to 4 in[]: [1 2 3 4]

* This represents the sequence of symbols [1, 2, 3, 4] that you want to encode.

Enter the bit stream in[]: [1 0 1 1 0 0 1 0]

* This is a sample bitstream (encoded output) you want to decode.

**Expected Output:**

1. **Huffman Code Dictionary:**

The huffman code dict:

{

[1] '0'

[2] '10'

[3] '11'

[4] '010'

}

The Huffman codes will be:

* + Symbol 1: 0
  + Symbol 2: 10
  + Symbol 3: 11
  + Symbol 4: 010

1. **Encoded Output:**

The encoded output:

0 10 11 010

The symbols [1 2 3 4] will be encoded into the bitstream 0 10 11 010.

1. **Decoded Symbols:**

The symbols are:

[1 2 3 4]

The bitstream [1 0 1 1 0 0 1 0] will be decoded back to the original sequence of symbols [1, 2, 3, 4].

1. **Entropy Calculation:**

Entropy is 1.846 bits

The entropy for the given probabilities is approximately 1.846 bits.

1. **Efficiency:**

Efficiency is: 0.921 bits

Efficiency is the ratio of entropy to average code length. The average code length is calculated by the Huffman algorithm and the efficiency tells you how well the coding scheme is performing.

1. **Variance of Codeword Lengths:**

The variance is: 2

The variance of codeword lengths (using max-min difference) is 2 in this example. (In a more rigorous approach, you could compute the true variance using the squared differences from the mean length.)

**10.Encoding and Decoding of binary data using a Hamming code.**

CODE:

% Hamming (7,4) Code: Encoding and Decoding

% This code provides both encoding and decoding functions

% for the Hamming (7,4) code, with error detection and correction.

% Main function to encode, simulate error, and decode

function hamming74()

% Example 4-bit data (You can change this to test other data)

data = [1 0 1 0]; % Example data: 1010

% Encode the data

encoded\_data = hamming74\_encode(data);

disp('Encoded Data:');

disp(encoded\_data);

% Simulate a bit error (flip the 5th bit for testing)

received\_data = encoded\_data;

received\_data(5) = mod(received\_data(5) + 1, 2); % Flip the 5th bit

disp('Received Data (with error):');

disp(received\_data);

% Decode the received data (with error correction)

decoded\_data = hamming74\_decode(received\_data);

disp('Decoded Data:');

disp(decoded\_data);

end

% Function to encode data using Hamming (7,4) code

function encoded\_data = hamming74\_encode(data)

% Check if the input data is 4 bits long

if length(data) ~= 4

error('Input data must be 4 bits long');

end

% Define the generator matrix G for Hamming (7,4)

G = [1 0 0 0 1 1 1;

0 1 0 0 1 0 1;

0 0 1 0 0 1 1;

0 0 0 1 1 1 0];

% Encode the data by multiplying it with the generator matrix

encoded\_data = mod(data \* G, 2); % Mod 2 for binary addition

end

% Function to decode data using Hamming (7,4) code

function decoded\_data = hamming74\_decode(received\_data)

% Check if the input received data is 7 bits long

if length(received\_data) ~= 7

error('Received data must be 7 bits long');

end

% Define the parity-check matrix H for Hamming (7,4)

H = [1 1 1 0 1 0 0;

1 1 0 1 0 1 0;

1 0 1 1 0 0 1];

% Calculate the syndrome vector (S = H \* received\_data')

syndrome = mod(received\_data \* H', 2); % Mod 2 for binary addition

% Check if there is an error

if any(syndrome) == 1

% If syndrome is non-zero, error exists

error\_position = bi2de(syndrome, 'left-msb') + 1; % Convert binary to decimal and adjust for MATLAB indexing

disp(['Error detected at position: ', num2str(error\_position)]);

% Correct the error by flipping the erroneous bit

received\_data(error\_position) = mod(received\_data(error\_position) + 1, 2);

else

disp('No error detected');

end

% Extract the data bits (the first 4 bits of the received data)

decoded\_data = received\_data(1:4);

end

Theory:

he 7,4 Hamming code is an error-detecting and error-correcting code that encodes 4 data bits into 7 bits by adding 3 parity bits. It is designed to detect and correct single-bit errors in the transmitted data.

The process involves the following steps:

Data Bits: The 4 data bits are transmitted along with 3 parity bits. The parity bits are calculated based on the data bits to ensure that the overall code has specific parity properties.

Parity Bit Calculation: The parity bits are placed at positions 1, 2, and 4 (using 1-based indexing). Each parity bit checks a subset of the codeword bits. Specifically:

P1 checks bits 1, 3, 5, 7.

P2 checks bits 2, 3, 6, 7.

P3 checks bits 4, 5, 6, 7.

Error Detection and Correction: At the receiver end, the parity bits are recalculated. If the recalculated parity does not match the received parity, the receiver can detect and correct a single-bit error.

**Program Description:**

The 7,4 Hamming code is widely used in computer networks and digital communication for error correction, ensuring reliable data transmission. It can correct single-bit errors and detect two-bit errors.

The code implements encoding and decoding using the **Hamming (7,4) code**. This is a linear error-correcting code that maps 4-bit data words into 7-bit codewords, adding 3 parity bits to ensure that the code can detect and correct single-bit errors.

The Hamming (7,4) code has the following key components:

* **4 information bits** (data bits)
* **3 parity bits** (check bits)
* **Error detection and correction**: It can detect and correct single-bit errors.

**Breakdown of the code:**

**Main Function (hamming74):**

This function simulates the encoding, transmission (with an error introduced), and decoding process of a Hamming (7,4) code.

1. **Data to Encode**:
2. data = [1 0 0 1]; % Example 4-bit data (1010)

Here, the data being encoded is a 4-bit binary vector: [1 0 0 1].

1. **Encoding**: The function hamming74\_encode(data) is called to encode the 4-bit data using the Hamming (7,4) code.
2. **Simulate Error**:
3. received\_data(5) = mod(received\_data(5) + 1, 2); % Flip the 5th bit

The 5th bit of the encoded data is flipped to simulate an error during transmission.

1. **Decoding**: The function hamming74\_decode(received\_data) is called to decode the received (possibly corrupted) data and attempt to correct any errors.
2. **Displaying Results**: The encoded data, the received data (with the error), and the decoded data are displayed.

**Encoding Function (hamming74\_encode):**

1. **Input Check**:
2. if length(data) ~= 4
3. error('Input data must be 4 bits long');
4. end

This checks if the input data is exactly 4 bits long. If not, it throws an error.

1. **Generator Matrix (G)**: The generator matrix GG is defined as:
2. G = [1 0 0 0 0 1 1;
3. 0 1 0 0 1 0 1;
4. 0 0 1 0 1 1 0;
5. 0 0 0 1 1 1 1];

This is a 4×74 \times 7 matrix used to encode the 4-bit data. Each row of this matrix corresponds to one of the data bits, with additional parity bits (the last 3 columns) added for error detection and correction.

1. **Encoding**:
2. encoded\_data = mod(data \* G, 2);

The data is multiplied by the generator matrix GG in binary (mod 2), which results in the 7-bit encoded data.

* + The multiplication here is in binary (i.e., modulo 2 addition), which is why the mod(..., 2) is used.
  + The result is a 7-bit codeword that contains both the original 4 data bits and the 3 parity bits.

**Decoding Function (hamming74\_decode):**

1. **Input Check**:
2. if length(received\_data) ~= 7
3. error('Received data must be 7 bits long');
4. end

This checks if the received data is exactly 7 bits long, as it should be after encoding.

1. **Parity-Check Matrix (H)**: The parity-check matrix HH is defined as:
2. H = [0 1 1 1 1 0 0;
3. 1 0 1 1 0 1 0;
4. 1 1 0 1 0 0 1];

This is a 3×73 \times 7 matrix used to check if any errors have occurred during transmission.

1. **Syndrome Calculation**:
2. syndrome = mod(received\_data \* H', 2);

The syndrome is calculated by multiplying the received data by the transpose of the parity-check matrix HH (denoted as H′H') and taking the result modulo 2. The syndrome vector indicates if and where an error has occurred:

* + If the syndrome is a zero vector, the received data is error-free.
  + If the syndrome is non-zero, an error has occurred at a specific position.

1. **Error Detection and Correction**:
2. if any(syndrome) == 1
3. error\_position = bi2de(syndrome, 'left-msb') + 1;
4. received\_data(error\_position) = mod(received\_data(error\_position) + 1, 2);

If the syndrome is non-zero (indicating an error), it converts the syndrome from binary to decimal to determine the position of the error. The error is then corrected by flipping the erroneous bit at that position.

* + bi2de(syndrome, 'left-msb') converts the syndrome (a binary vector) to a decimal number, representing the position of the error in the 7-bit received data.
  + The bit at the error position is flipped using mod(received\_data(error\_position) + 1, 2).

1. **Extracting the Data**:
2. decoded\_data = received\_data(1:4);

After correcting any errors, the first 4 bits of the received data are extracted as the decoded data (the original message).

**Error Handling:**

* **Syndrome**: The syndrome vector tells us if there is an error in the received data. If the syndrome is all zeros, there is no error. If the syndrome is non-zero, it indicates where the error is.
* **Error Correction**: If an error is detected, the bit at the error position is flipped to correct it.

**Sample input and outputs:**

1. **Input Data**: [1 0 0 1] (4-bit)
2. **Encoded Data**: After encoding with the generator matrix, the result will be a 7-bit string.
3. **Simulated Error**: The 5th bit is flipped to simulate an error.
4. **Decoded Data**: The received data is decoded, errors are detected and corrected, and the original 4-bit data [1 0 0 1] is recovered.