1. Gram-Schmidt Orthogonalization: To find orthogonal basis vectors for the given set of vectors and plot the orthonormal vectors.

CODE:

```
% Given set of vectors as columns

V = [1 0 0; 1 1 1; 0 0 1]';

 % Number of vectors

num_vectors = size(V, 2);

 % Initialize orthogonal and orthonormal matrices

U = zeros(size(V));

E = zeros(size(V));

 % Gram-Schmidt Process

U(:,1) = V(:,1);  % First orthogonal vector is the first input vector

E(:,1) = U(:,1) / norm(U(:,1));  % First orthonormal vector

 for i = 2:num_vectors

   U(:,i) = V(:,i);

   for j = 1:i-1

       U(:,i) = U(:,i) - (dot(V(:,i), E(:,j)) * E(:,j)); % Subtract projection onto previous
orthonormal vectors

   end

   E(:,i) = U(:,i) / norm(U(:,i));  % Normalize to get orthonormal vector

end

 % Display the orthonormal vectors

disp('Orthonormal basis vectors:');

disp(E);

 % Plot the original and orthonormal vectors

figure;

hold on;

grid on;

axis equal;
```

```
% Plot original vectors

quiver3(0, 0, 0, V(1,1), V(2,1), V(3,1), 'r', 'LineWidth', 2);

quiver3(0, 0, 0, V(1,2), V(2,2), V(3,2), 'g', 'LineWidth', 2);

quiver3(0, 0, 0, V(1,3), V(2,3), V(3,3), 'b', 'LineWidth', 2);

% Plot orthonormal vectors

quiver3(0, 0, 0, E(1,1), E(2,1), E(3,1), 'r--', 'LineWidth', 2);

quiver3(0, 0, 0, E(1,2), E(2,2), E(3,2), 'g--', 'LineWidth', 2);

quiver3(0, 0, 0, E(1,3), E(2,3), E(3,3), 'b--', 'LineWidth', 2);

xlabel('X');

ylabel('Y');

zlabel('Z');

legend('Original V1', 'Original V2', 'Original V3', 'Orthonormal E1', 'Orthonormal E2',
'Orthonormal E3');

title('Original and Orthonormal Vectors');

hold off;
```

Theory :

The **Gram-Schmidt orthogonalization procedure** is a method used in linear algebra to transform a set of linearly independent vectors into an orthogonal (or orthonormal) set. It is particularly useful when working with vector spaces and helps in creating an orthogonal basis for a subspace.

The procedure works as follows:

1. **Start with a Set of Linearly Independent Vectors**: Let V1,V2,…,VnV_1, V_2, \dots, V_nV1,V2,…,Vn be a set of linearly independent vectors.
2. **First Vector**: The first vector U1U_1U1 is simply the first input vector V1V_1V1. If orthonormalization is desired, normalize U1U_1U1 to obtain E1E_1E1.
3. **Subsequent Vectors**: For each subsequent vector ViV_iVi, subtract the projections of ViV_iVi onto all previously computed orthogonal vectors E1,E2,…,Ei−1E_1, E_2, \dots, E_{i-1}E1,E2,…,Ei−1. This ensures that the new vector UiU_iUi is orthogonal to all previous vectors.

$$U_i = V_i - \sum_{j=1}^{i-1} \text{proj}(U_i, E_j)$$

1. **Normalization**: Each orthogonal vector UiU_iUi is then normalized to obtain an orthonormal vector EiE_iEi.

The final set E1,E2,…,EnE_1, E_2, \dots, E_nE1,E2,…,En is an orthonormal set of vectors, meaning that each vector is orthogonal to others, and each has unit length.

The Gram-Schmidt process is widely used in numerical analysis, signal processing, and machine learning (e.g., principal component analysis) for generating orthonormal bases for vector spaces.

Output:

```
Orthonormal basis vectors:

0.7071   0.4082   -0.5774

0.7071   -0.4082    0.5774

0        0.8165    0.5774
```

Plot:

The plot shows the original vectors and the orthonormal vectors in a 3D coordinate system.

The red, green, and blue arrows represent the original vectors.

The dashed arrows in the same colors represent the orthonormal vectors.

Code Discerption :

This MATLAB code performs the **Gram-Schmidt process** to orthogonalize a set of given vectors and then normalize them to produce orthonormal vectors. Here's a step-by-step breakdown of the code:

1. **Input Matrix of Vectors**: The matrix `V` contains the input vectors as its columns:

   ```
   matlab
   Copy code
   V = [1 0 0; 1 1 1; 0 0 1]';
   ```

The transpose (`'`) is used to make each column of `V` represent a vector. So, `V(:,1) = [1; 1; 0]`, `V(:,2) = [0; 1; 0]`, and `V(:,3) = [0; 1; 1]` are the three 3D vectors.

2. **Initialization**:
   - `num_vectors` determines how many vectors are in the set (3 in this case).
   - `U` is an empty matrix to store the orthogonalized vectors.
   - `E` is an empty matrix to store the orthonormal vectors.
3. **Gram-Schmidt Process**:
   - For the first vector (`V(:,1)`), it directly becomes the first orthogonal vector `U(:,1)`.
   - The first orthonormal vector `E(:,1)` is computed by normalizing `U(:,1)`.
   - For each subsequent vector `V(:,i)`, the code:
     - Subtracts the projections of `V(:,i)` onto all previously orthonormal vectors `E(:,j)`, ensuring `U(:,i)` is orthogonal to the previous vectors.
     - Normalizes `U(:,i)` to obtain the orthonormal vector `E(:,i)`.
4. **Display Results**:
   - The orthonormal vectors are printed using `disp(E)`.
5. **Plotting**:
   - The original vectors `V` and the orthonormal vectors `E` are plotted in 3D space using the `quiver3` function, which visualizes vectors starting from the origin (0, 0, 0).
   - The original vectors are plotted with solid lines in red, green, and blue.
   - The orthonormal vectors are plotted with dashed lines in corresponding colors.

3 Perform the QPSK Modulation and demodulation. Display the signal and its constellation

Code:

```
clc;
clear all;
close all;

% Input bit sequence
bit_seq = [1 1 0 0 0 0 1 1];
N = length(bit_seq);
fc = 1; % Carrier frequency
t = 0:0.001:2; % Time vector
b = []; % Input bit sequence as a waveform
qpsk1 = []; % QPSK signal
bec = []; % Even bit cosine waveform
bes = []; % Odd bit sine waveform
b_o = []; % Odd bit stream
b_e = []; % Even bit stream
```

```
bit_e = []; % Even bit stream (for plotting)
bit_o = []; % Odd bit stream (for plotting)

% Creating input waveform from bit sequence
for i = 1:N
  bx = bit_seq(i) * ones(1, 1000); % Generate pulse for each bit
  b = [b, bx];
end

% Modifying bits for QPSK mapping: 0 -> -1
for i = 1:N
  if bit_seq(i) == 0
    bit_seq(i) = -1;
  end
  if mod(i, 2) == 0
    e_bit = bit_seq(i);
    b_e = [b_e, e_bit];
  else
    o_bit = bit_seq(i);
    b_o = [b_o, o_bit];
  end
end

% Generate QPSK modulated signal
for i = 1:N/2
  % Even bits modulated on cosine wave
  be_c = (b_e(i) * cos(2*pi*fc*t));
  % Odd bits modulated on sine wave
  bo_s = (b_o(i) * sin(2*pi*fc*t));
  q = be_c + bo_s; % Combine both to form QPSK signal

  % Collect even and odd bit streams for plotting
  even = b_e(i) * ones(1, 2000);
  odd = b_o(i) * ones(1, 2000);
  bit_e = [bit_e, even];
  bit_o = [bit_o, odd];
  qpsk1 = [qpsk1, q];
  bec = [bec, be_c];
  bes = [bes, bo_s];
end

% Plotting the QPSK signals and constellations

figure('name', 'QPSK Modulation');
subplot(5, 1, 1);
plot(b, 'o');
grid on;
```

```
axis([0 N*1000 0 1]);
title('Binary Input Sequence');

subplot(5, 1, 2);
plot(bes);
hold on;
plot(bit_o, 'rs:');
grid on;
axis([0 N*1000 -1 1]);
title('Odd Bits (Sine Component)');

subplot(5, 1, 3);
plot(bec);
hold on;
plot(bit_e, 'rs:');
grid on;
axis([0 N*1000 -1 1]);
title('Even Bits (Cosine Component)');

subplot(5, 1, 4);
plot(qpsk1);
axis([0 N*1000 -1.5 1.5]);
title('QPSK Modulated Signal');

% QPSK Constellation Plot
subplot(5, 1, 5);
% QPSK constellation points
constellation = [1 + 1j, -1 + 1j, -1 - 1j, 1 - 1j];
plot(real(constellation), imag(constellation), 'bo', 'MarkerSize', 8, 'LineWidth', 2);
grid on;
axis([-2 2 -2 2]);
title('QPSK Constellation');
xlabel('In-phase (I)');
ylabel('Quadrature (Q)');
```

## Theory:

**Quadrature Phase Shift Keying (QPSK)** is a digital modulation technique used to transmit data efficiently over communication channels. It is a type of phase modulation where data is represented by four distinct phase shifts of a carrier signal. QPSK uses two bits per symbol, meaning each symbol carries two bits of information.

In QPSK, the carrier signal is modulated by varying its phase. The four possible phase shifts (0°, 90°, 180°, and 270°) correspond to different pairs of bits:

- $00 \rightarrow 0°$
- $01 \rightarrow 90°$
- $10 \rightarrow 180°$

- 11 → 270°

This allows QPSK to transmit twice the amount of data compared to Binary Phase Shift Keying (BPSK), where each symbol represents only one bit. Thus, QPSK effectively doubles the bandwidth efficiency without increasing the power requirements.

The signal is formed by combining two carriers: one for the in-phase component (I) and the other for the quadrature component (Q). These components are modulated separately by the two bits in each symbol. The result is a signal that varies in both amplitude and phase, carrying more information in the same time period.

QPSK is widely used in wireless communication systems such as satellite communications, Wi-Fi, and cellular networks, where bandwidth efficiency and power conservation are crucial. The key advantages of QPSK are its improved spectral efficiency and robustness against noise, making it suitable for high-data-rate applications.

## Output

**Resulting Plots:**

1. **Binary Input Sequence**: The first plot shows the binary input sequence as pulses.
2. **Odd and Even Components**: The second and third plots show the sine (odd) and cosine (even) components of the QPSK signal, respectively.
3. **QPSK Modulated Signal**: The fourth plot displays the final QPSK signal.
4. **QPSK Constellation**: The fifth plot shows the QPSK constellation with the four possible constellation points.

**CODE description :**

The MATLAB code you have provided performs **QPSK modulation** for an input binary sequence. The code generates a series of waveforms corresponding to the QPSK modulation scheme and visualizes the different components of the modulation in several subplots.

Let's go through the steps and explain how the code works:

### Key Steps in the Code:

1. **Input Bit Sequence:**

   ```
   matlab
   Copy code
   bit_seq = [1 1 0 0 0 0 1 1];
   ```

   This represents the input binary bit sequence that is going to be modulated using **QPSK**.

2. **Length of the Input Sequence:**

```
matlab
Copy code
N = length(bit_seq);
```

   Here, `N` gives the length of the input bit sequence.

3. **Initialization of Variables:** Several variables are initialized, including `b`, `qpsk1`, `bec`, `bes`, and others which will be used to store intermediate signals.
4. **Creating the Input Waveform:** The code creates a sequence of pulses corresponding to the input binary sequence `bit_seq`. Each bit is stretched over multiple samples (1000 in this case) to create a pulse waveform.

```
matlab
Copy code
for i = 1:N
    bx = bit_seq(i) * ones(1, 1000);
    b = [b, bx];
end
```

   This results in a sequence where each bit in `bit_seq` is mapped to a pulse.

5. **Converting Binary to QPSK Symbol Mapping:** The input bits are divided into **even** and **odd** bits, which will be mapped to the cosine and sine components for QPSK modulation:
   - **Even bits** are mapped to the in-phase (I) component (cosine).
   - **Odd bits** are mapped to the quadrature (Q) component (sine).

   The binary `0` is mapped to `-1` for both even and odd bits.

6. **Generating the QPSK Modulated Signal:** For each pair of bits (even and odd), a **QPSK signal** is created by combining the cosine and sine components:

```
matlab
Copy code
be_c = (b_e(i) * cos(2*pi*fc*t));
bo_s = (b_o(i) * sin(2*pi*fc*t));
q = be_c + bo_s;
```

   The result is the QPSK signal `q`, which is appended to the `qpsk1` array.

7. **Plotting the Results:** The code uses `subplot` to create four different plots:
   - **First Subplot (Binary Sequence)**: It plots the binary waveform (represented by the `b` array) where `0` and `1` are shown as pulses.
   - **Second Subplot (Sine Wave)**: It shows the sine component (`bo_s`) and the `odd` bit stream (represented by `bit_o`).
   - **Third Subplot (Cosine Wave)**: It shows the cosine component (`be_c`) and the `even` bit stream (represented by `bit_e`).
   - **Fourth Subplot (QPSK Signal)**: It shows the complete QPSK modulated signal formed by combining the cosine and sine components.

## Visualization of the Results:

1. **First Plot (Binary Sequence):**
   - The binary bit sequence `bit_seq` is converted to a series of pulses, and this plot shows how the bits are mapped to rectangular pulses.
2. **Second Plot (Sine Wave Component):**
   - This plot shows the sine component of the QPSK signal along with the odd bits (those that are mapped to the sine wave).
3. **Third Plot (Cosine Wave Component):**
   - Similarly, this plot shows the cosine component of the QPSK signal along with the even bits (those that are mapped to the cosine wave).
4. **Fourth Plot (QPSK Signal):**
   - Finally, the modulated QPSK signal is shown, where both the cosine and sine components are combined to form the final signal.

## Potential Modifications or Improvements:

1. **Normalization:** The QPSK signal could be normalized so that it has a unit average power. This ensures that the amplitude of the signal does not vary with different bit sequences.
2. **AWGN (Additive White Gaussian Noise):** To simulate a real-world communication scenario, you could add noise to the modulated QPSK signal and then perform demodulation.
3. **Legend and Titles:** You can add titles, legends, and axis labels to each subplot for better clarity.

## Exp no 2. Simulation of binary baseband signals using a rectangular pulse and estimate the BER for AWGN channel using matched filter receiver.

CODE:

```
% Simplified Parameters

N = 1e4;

SNR_dB = 0:5:20;

pulse_width = 1;

% Number of bits

% SNR values in dB

% Pulse width for rectangular pulse

% Generate random binary data

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

```
% Define the rectangular pulse

t = 0:0.01:pulse_width;

rect_pulse = ones(size(t));

% Initialize BER vector

BER = zeros(length(SNR_dB), 1);

for snr_idx = 1:length(SNR_dB)

% Modulate binary data

tx_signal = [];

for i = 1:N

if data(i) == 1

tx_signal = [tx_signal; rect_pulse'];

else

tx_signal = [tx_signal; zeros(size(rect_pulse'))];

end

end

% Add AWGN

SNR = 10^(SNR_dB(snr_idx) / 10);

noise_power = 1 / (2 * SNR);

noise = sqrt(noise_power) * randn(length(tx_signal), 1);

rx_signal = tx_signal + noise;

% Matched Filter

matched_filter = rect_pulse;

filtered_signal = conv(rx_signal, matched_filter, 'same');

% Sample the output of the matched filter

sample_interval = round(length(filtered_signal) / N);

sampled_signal = filtered_signal(1:sample_interval:end);

% Decision (Threshold = 0.5)

estimated_bits = sampled_signal > 0.5;
```

```
% Compute BER

num_errors = sum(estimated_bits ~= data);

BER(snr_idx) = num_errors / N;

end

% Plot BER vs. SNR

figure;

semilogy(SNR_dB, BER, 'b-o');

grid on;

xlabel('SNR (dB)');

ylabel('Bit Error Rate (BER)');

title('BER vs. SNR for Rectangular Pulse Modulated Binary Data');
```

## Theory :

In digital communication systems, binary baseband signals are used to transmit binary data (0s and 1s) over a communication channel. One of the simplest ways to represent binary data is by using a **rectangular pulse**. A rectangular pulse for a bit ‘1’ or ‘0’ is transmitted over a fixed duration (symbol period), with the amplitude of the pulse being constant during that period.

For a binary signal, the two possible pulse amplitudes are typically:

- **Amplitude A for bit 1**.
- **Amplitude -A for bit 0**.

In baseband transmission, the signal is transmitted at low frequencies, meaning it does not involve modulation of a carrier signal. The rectangular pulses are directly related to the bits being transmitted.

**Bit Error Rate (BER) Estimation for AWGN Channel:**

In a communication system using binary baseband signals, the performance is typically evaluated by the **Bit Error Rate (BER)**, which measures the fraction of bits received incorrectly.

For a system transmitting through an **Additive White Gaussian Noise (AWGN)** channel, the BER can be estimated using a **Matched Filter Receiver**. The matched filter maximizes the signal-to-noise ratio (SNR) and optimally detects the transmitted bit.

The BER for binary signaling with rectangular pulses in an AWGN channel is given by the **Q-function**:

$$\text{BER} = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

where:

- $E_b$ is the energy per bit.
- $N_0$ is the noise power spectral density.

This equation shows that the BER decreases with increasing SNR, and the matched filter helps in minimizing the error rate by maximizing the received signal's energy.

## Output and Interpretation:

- The **BER vs. SNR plot** will show a decreasing trend in the bit error rate as the SNR increases, which is expected. A higher SNR means the signal is stronger compared to noise, leading to fewer errors in detection.
- The use of a **matched filter** maximizes the signal-to-noise ratio (SNR) at the receiver, which improves the detection of the transmitted bits, especially at lower SNRs.

## Example Plot Description:

- **X-axis**: SNR in dB (ranging from 0 dB to 20 dB).
- **Y-axis**: Bit Error Rate (BER), shown on a logarithmic scale.
- As SNR increases, the BER will decrease due to the reduction in noise impact, resulting in a cleaner signal and more accurate bit recovery.

**Code Description :**

The provided MATLAB code simulates the transmission of binary data using **rectangular pulses** and estimates the **Bit Error Rate (BER)** over an **AWGN channel** for various **Signal-to-Noise Ratios (SNRs)**. The key elements of the code are explained step by step below:

**Key Steps in the Code:**

1. **Initialization:**
   - The number of bits N=104N = 10^4N=104 represents the length of the binary data sequence.
   - The SNR values are set from 0 dB to 20 dB in steps of 5 dB (SNR_dB = 0:5:20).

- The pulse width for the rectangular pulse is set to 1.

2. **Generating Binary Data:**
    - A random binary sequence of NNN bits (data = randi([0 1], N, 1)) is generated. This will be the data to be transmitted.
3. **Rectangular Pulse:**
    - A rectangular pulse (rect_pulse = ones(size(t))) is created to represent a binary 1. A pulse for a 0 is simply represented by a sequence of zeros.
4. **Transmission Simulation:**
    - **Modulation**: For each bit in the binary sequence:
        - A 1 is modulated as a rectangular pulse.
        - A 0 is modulated as a zero signal (i.e., no pulse).
    - **AWGN Channel**: Gaussian noise is added to the signal according to the current SNR, using the formula: noise=noise power·randn(L,1)\text{noise} = \sqrt{\text{noise power}} \cdot \text{randn}(L, 1)noise=noise power ·randn(L,1) where the noise power is inversely related to the SNR.
5. **Matched Filter:**
    - A **matched filter** (matched_filter = rect_pulse) is applied to the received signal (rx_signal). This filter is optimal for detecting the rectangular pulses.
    - **Convolution** is used to filter the received signal using the matched filter.
6. **Sampling and Decision:**
    - The output of the matched filter is sampled at intervals (sample_interval = round(length(filtered_signal) / N)).
    - The estimated bits are determined by comparing the sampled values to a threshold of 0.5 (estimated_bits = sampled_signal > 0.5).
7. **BER Calculation:**
    - The **Bit Error Rate (BER)** is calculated by comparing the estimated bits (estimated_bits) with the transmitted bits (data). The number of errors is counted, and the BER is calculated as the ratio of errors to total bits.
8. **Plotting the Results:**
    - The BER is plotted as a function of SNR in dB using a **semilogarithmic scale**. The plot displays how the BER decreases with increasing SNR.

## 12. Encoding and Decoding of Convolution code

**Code:**

```
clc;
clear;
close all;

% Input message to be encoded
msg = [1 0 1 1 0 1 0 0];

% Define constraint length and generator polynomial
constraint_length = 3;
generator_polynomials = [7 5];

% Create trellis structure for the convolutional encoder
trellis = poly2trellis(constraint_length, generator_polynomials);

% Encode the message using convolutional encoder
encoded_msg = convenc(msg, trellis);

% Simulate noise by flipping a bit in the encoded message
encoded_msg_noisy = encoded_msg;
encoded_msg_noisy(4) = ~encoded_msg_noisy(4);  % Flip the 4th bit to simulate noise

% Perform Viterbi decoding on the noisy message
traceback_length = 5;
decoded_msg = vitdec(encoded_msg_noisy, trellis, traceback_length, 'trunc', 'hard');

% Display results
disp('Original Message:');
disp(msg);

disp('Encoded Message:');
disp(encoded_msg);

disp('Noisy Encoded Message (with bit flip):');
disp(encoded_msg_noisy);

disp('Decoded Message:');
disp(decoded_msg);
```

Theory:

**Convolutional Coding** is a method of error correction where each bit of the input message is transformed into multiple output bits based on a set of generator polynomials. It is widely used in communication systems to improve the reliability of data transmission over noisy channels.

In convolutional coding, the message is encoded using a sliding window that operates over a sequence of input bits. The encoder uses memory elements (shift registers) and a set of generator polynomials to produce output bits, where each output bit depends on the current and previous input bits. The number of bits in the output is typically greater than the number of input bits, resulting in an increased redundancy for error correction.

A key feature of convolutional codes is the **constraint length**, which determines the size of the memory used by the encoder. The larger the constraint length, the more reliable the encoding, but at the cost of higher complexity. The encoder is described using a **trellis diagram**, which visually represents the transitions of the encoder's states as input bits are processed.

Convolutional codes are decoded using algorithms like the **Viterbi algorithm**, which efficiently finds the most likely sequence of input bits by evaluating the possible paths in the trellis. This method provides robust error correction, especially in environments with high noise or interference.

Convolutional coding is widely used in modern communication systems, including satellite communications, cellular networks, and digital television.

Sample outputs:

After running the code, you should see output similar to the following:

```
Original Message:
     1     0     1     1     0     1     0     0

Encoded Message:
     1     0     1     1     1     0     1     1     1     0     0

Noisy Encoded Message (with bit flip):
     1     0     1     0     1     0     1     1     1     0     0

Decoded Message:
     1     0     1     1     0     1     0     0
```

### Code Description:

- **Trellis Creation:**
  - You are using the `poly2trellis` function to create a trellis structure for the convolutional code. The `constraint_length` is set to 3, and the `generator_polynomials` are given as `[7 5]`. These are octal representations of the polynomials.
- **Encoding the Message:**
  - The `convenc` function encodes the input message `msg` using the convolutional code defined by `trellis`.

- **Simulating Noise:**
  - In the example, you simulate noise by flipping one bit in the encoded message (`encoded_msg_noisy(4) = ~encoded_msg_noisy(4);`). This simulates a bit error, which can occur in transmission.
- **Viterbi Decoding:**
  - The `vitdec` function is used to decode the noisy encoded message. The decoding uses the Viterbi algorithm with a traceback length of 5 and hard decision decoding (`'hard'`).

## Key Steps in the Code:

1. **Message Encoding**:
   - The message `[1 0 1 1 0 1 0 0]` is encoded using a convolutional code defined by the trellis.
2. **Noise Simulation**:
   - A bit flip (`encoded_msg_noisy(4) = ~encoded_msg_noisy(4);`) is introduced in the encoded message to simulate noise in the transmission.
3. **Viterbi Decoding**:
   - The noisy encoded message is decoded using the Viterbi algorithm with hard decision decoding (`'hard'`). The traceback length of 5 is used to decode the message.
4. **Display**:
   - The original message, encoded message, noisy encoded message, and decoded message are printed to the console.