Experiment No. 5: Gram-Schmidt Orthogonalization

Aim:
To find orthogonal basis vectors for a given set of vectors and plot the orthonormal vectors.

Software Required:
MATLAB

Theory:
The Gram-Schmidt process orthonormalizes a set of vectors in an inner product space by generating an orthogonal set of vectors spanning the same subspace. It involves subtracting projections of vectors onto previously orthonormalized vectors and normalizing the resulting vectors.

Code:

```
% Given set of vectors as columns
V = [1 1 0; 1 0 1; 0 1 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;
```

Output:

Result:
The Gram-Schmidt process successfully generates an orthonormal basis for the input vectors. The orthonormal vectors are displayed, and their graphical representation confirms the process.

Experiment No. 6: Simulation of Binary Baseband Signal

Aim: To simulate binary baseband signals using a rectangular pulse and estimate the Bit Error Rate (BER) for an AWGN channel using a matched filter receiver.

Software Required:
MATLAB

Theory:
In digital communication, the Bit Error Rate (BER) is a key performance metric that indicates the number of received bits that are incorrect. For a system using rectangular pulse modulation, the SNR impacts the quality of the received signal, and a higher SNR typically results in fewer errors. The relationship between BER and SNR is derived using a matched filter in the presence of Gaussian noise.

Code:

```
N = 1e4;
SNR_dB = 0:5:20;
pulse_width = 1;
data = randi([0 1], N, 1);
t = 0:0.01:pulse_width;
rect_pulse = ones(size(t));
BER = zeros(length(SNR_dB), 1);

for snr_idx = 1:length(SNR_dB)
    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
    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 = rect_pulse;
    filtered_signal = conv(rx_signal, matched_filter, 'same');
    sample_interval = round(length(filtered_signal) / N);
    sampled_signal = filtered_signal(1:sample_interval:end);
    estimated_bits = sampled_signal > 0.5;
    num_errors = sum(estimated_bits ~= data);
    BER(snr_idx) = num_errors / N;
end
```

```
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');
```

Output:

Result:
As expected, the BER decreases with increasing SNR, indicating better performance (fewer errors) at higher SNR values.

Experiment No. 7: QPSK Modulation and Demodulation

Aim:
To implement Quadrature Phase Shift Keying (QPSK) modulation and demodulation.

Software Required:
MATLAB

Theory:
Quadrature Phase Shift Keying (QPSK) is a modulation scheme that conveys two bits per symbol by varying the phase of the carrier signal. The signal is modulated using two carriers, one in phase (cosine) and the other in quadrature phase (sine). The demodulation process involves matching the received signal with the reference carrier signals and decoding the transmitted data.

Code:

```
clc;
clear all;
close all;
tb = 1;
t = 0:(tb/100):tb;
fc = 1;
c1 = sqrt(2/tb)*cos(2*pi*fc*t);
c2 = sqrt(2/tb)*sin(2*pi*fc*t);
n = 8;
m = rand(1, n);
t1 = 0;
t2 = tb;
for i = 1:2:(n-1)
   t = [t1:(tb/100):t2];
   if m(i) > 0.5
      m(i) = 1;
      m_s = ones(1, length(t));
   else
      m(i) = 0;
      m_s = -1 * ones(1, length(t));
   end
   odd_sig(i, :) = c1 .* m_s;
   if m(i+1) > 0.5
      m(i+1) = 1;
      m_s = ones(1, length(t));
   else
      m(i+1) = 0;
      m_s = -1 * ones(1, length(t));
   end
   even_sig(i, :) = c2 .* m_s;
   qpsk1 = odd_sig + even_sig;
```

```
    subplot(3, 2, 1);
    plot(t, qpsk1(i, :));
    title('QPSK Signal');
    xlabel('t---->');
    ylabel('s(f)');
    grid on;
    hold on;
    t1 = t1 + (tb + 0.01);
    t2 = t2 + (tb + 0.01);
end
hold off;
subplot(3, 2, 2);
stem(m);
title('Binary Data Bits');
xlabel('n---->');
ylabel('b(n)');
grid on;
subplot(3, 2, 3);
plot(t, c1);
title('Carrier Signal 1');
xlabel('t---->');
ylabel('c1(t)');
grid on;
subplot(3, 2, 4);
plot(t, c2);
title('Carrier Signal 2');
xlabel('t---->');
ylabel('c2(t)');
grid on;
t1 = 0;
t2 = tb;
for i = 1:n-1
    t = [t1:(tb/100):t2];
    x1 = sum(c1 .* qpsk1(i, :));
    x2 = sum(c2 .* qpsk1(i, :));
    if(x1 > 0 & x2 > 0)
        demod(i) = 1;
        demod(i+1) = 1;
    elseif(x1 > 0 & x2 < 0)
        demod(i) = 1;
        demod(i+1) = 0;
    elseif(x1 < 0 & x2 < 0)
        demod(i) = 0;
        demod(i+1) = 0;
    elseif(x1 < 0 & x2 > 0)
        demod(i) = 0;
        demod(i+1) = 1;
    end
    t1 = t1 + (tb + 0.01);
    t2 = t2 + (tb + 0.01);
end
subplot(3, 2, 5);
stem(demod);
title('QPSK Demodulated Bits');
xlabel('n---->');
```

```
ylabel('b(n)');
grid on;
```

Output:

Result:
The QPSK modulation and demodulation processes correctly map binary data to QPSK symbols and successfully recover the transmitted bits at the receiver.

Experiment No. 8: 16-QAM Modulation

Aim: To implement 16-QAM modulation and observe the transmitted and received signal constellations.

Software Required:
MATLAB

Theory:
16-QAM (Quadrature Amplitude Modulation) is a modulation technique that uses both amplitude and phase to encode data. It combines the concepts of both Amplitude Shift Keying (ASK) and Phase Shift Keying (PSK) to transmit 4 bits per symbol, resulting in 16 different signal points in the constellation diagram.

Code:

```
close all;
m = 16;
k = log2(m);
n = 9e3;
nsamp = 1;
x = randi([0, 1], n, 1);
stem(x(1:20), 'filled');
title('Bit Sequence');
xlabel('Bit Index');
ylabel('Bit Amplitude');
x = reshape(x, k, length(x) / k).';
xsym = bi2de(x, 'left-msb');
figure;
stem(xsym(1:10));
title('Symbol Plot');
xlabel('Symbol Index');
ylabel('Symbol Amplitude');
y = qammod(xsym, m);
ytx = y;
ebno = 10;
snr = ebno + 10 * log10(k) - 10 * log10(nsamp);
yn = awgn(ytx, snr);
figure;
scatter(real(y), imag(y));
title('Transmitted Signal Constellation');
figure;
scatter(real(yn), imag(yn));
title('Received Signal Constellation');
```

Output:

Result:
The 16-QAM modulation scheme successfully maps bits to symbols, and the received signal shows the effects of noise, visible in the change of positions of the received constellation points.

Experiment No. 9: Huffman Coding and Decoding

Aim:
To implement Huffman coding and decoding to compress a set of data based on given probabilities.

Software Required:
MATLAB

Theory:
Huffman coding is a popular algorithm used for lossless data compression. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters. The algorithm builds an optimal prefix code based on the frequency of symbols. After encoding the data, the decoder uses the Huffman tree to reverse the process and recover the original data.

Code:

```
x = input('Enter the number of symbols: ');
N = 1:x;
disp('The number of symbols are N:');
disp(N);

P = input('Enter the probabilities: ');
disp(P);

S = sort(P, 'descend');
disp('The sorted probabilities are:');
disp(S);

[dict, avglen] = huffmandict(N, S);
disp('The average length of the code is:');
disp(avglen);

H = 0;
for i = 1:x
   H = H + (P(i) * log2(1/P(i)));
end
disp('Entropy is:');
disp(H);
disp('bits/msg');

E = (H / avglen) * 100;
disp('Efficiency is:');
disp(E);

codeword = huffmanenco(N, dict);
disp('The codewords are:');
```

```
disp(codeword);

decode = huffmandeco(codeword, dict);
disp('Decoded output is:');
disp(decode);
```

Output:

Result:
The Huffman encoding and decoding process was successfully implemented. The output codewords are efficient, and the decoded data matches the original input, confirming the correctness of the algorithm.

Experiment No. 10: Hamming Code Encoding and Decoding

Aim:
To encode and decode binary data using a Hamming code.

Software Required:
MATLAB

Theory:
Hamming codes are linear error-correcting codes that add redundant bits to data, allowing detection and correction of single-bit errors.

Code:

```
clc
clear all
close all

parity_matrix = [0, 1, 1; 1, 0, 1; 1, 1, 0; 1, 1, 1];
generator_matrix = [eye(4), parity_matrix];

data = input('Enter the data bits: ');
Encoded_Data = mod(data * generator_matrix, 2);

Received_Data = input('Enter the received bits: ');

parity_check_matrix = [parity_matrix', eye(3)];
syndrome = mod(Received_Data * parity_check_matrix', 2);

if syndrome == zeros(1, length(syndrome))
   error = 0;
else
   error = 1;
end

if error == 0
   decoded_data = Received_Data(1:4);
else
   parity_check_matrix_transpose = parity_check_matrix';
   for i = 1:7
      if parity_check_matrix_transpose(i, :) == syndrome
         error_row = i;
         break;
      end
   end
   Received_Data(error_row) = xor(Received_Data(error_row), 1);
   decoded_data = Received_Data(1:4);
```

```
end
```

Output:

Result:
The implementation of CRC successfully detected and corrected errors in the transmitted data, ensuring error-free data recovery.

Experiment No. 11: CRC for Error Detection and correction in C

Aim:
To implement Cyclic Redundancy Check (CRC) error detection in C programming.

Software Required:
dev c ++

Theory:
In this experiment, the Cyclic Redundancy Check (CRC) is implemented in C to simulate error detection for a given message. The message is encoded with a polynomial-based CRC code, transmitted, and then checked for errors at the receiver end. If no errors are detected, the receiver can safely assume that the data is error-free; otherwise, the program identifies and reports the error.

Code:

```
#include <stdio.h>

int a[100], b[100], i, j, len, k, count = 0;
int gp[] = {1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1};

int main()
{
   void div();
   printf("\n enter the length of data frame:");
   scanf("%d", &len);
   printf("\n enter the message:");
   for (i = 0; i < len; i++)
      scanf("%d", &a[i]);
   for (i = 0; i < 16; i++)
      a[len++] = 0;
   for (i = 0; i < len; i++)
      b[i] = a[i];
   k = len - 16;
   div();
   for (i = 0; i < len; i++)
      b[i] = b[i] ^ a[i];
   printf("\n data to be transmitted:");
   for (i = 0; i < len; i++)
      printf("%2d", b[i]);
   printf("\n\nenter the received data:");
   for (i = 0; i < len; i++)
      scanf("%d", &a[i]);
```

```
    div();
    for (i = 0; i < len; i++)
        if (a[i] != 0)
        {
            printf("\n ERROR in received data");
            return 0;
        }
    printf("\n Data received is ERROR FREE");
}

void div()
{
    for (i = 0; i < k; i++)
    {
        if (a[i] == gp[0])
        {
            for (j = i; j < 17 + i; j++)
                a[j] = a[j] ^ gp[count++];
        }
        count = 0;
    }
}
```

Output:

Result:
The CRC implementation in C correctly encodes the message, checks for errors in transmission, and handles error correction successfully.

Experiment No. 12: Convolutional Code

Aim:
To encode and decode data using convolutional coding.

Software Required:
MATLAB

Theory:
The Viterbi algorithm is a dynamic programming algorithm used for decoding convolutionally encoded data. It is widely used in error correction, particularly in communications systems that employ convolutional coding. The algorithm operates by finding the most likely sequence of states that could have generated the received sequence, thus minimizing the error rate.

Code:

```
clc;
clear all;
close all;

K = 3;
G1 = 7; % first generator polynomial: 111
G2 = 5; % second generator polynomial: 101
disp('Message input');
msg = [1 0 0 1 1];
trel = poly2trellis(K, [G1 G2]);

coded = convenc(msg, trel);
disp('Encoded Output');
disp(coded);

tblen = length(msg);
decoded = vitdec(coded, trel, tblen, 'trunc', 'hard');
disp('Decoded Output');
disp(decoded);
```

Output:

Result:
The Viterbi decoder successfully decoded the convolutionally encoded data, with the decoded output matching the original message, demonstrating the effectiveness of the Viterbi algorithm.