Convolutional Encoding and Viterbi Decoding

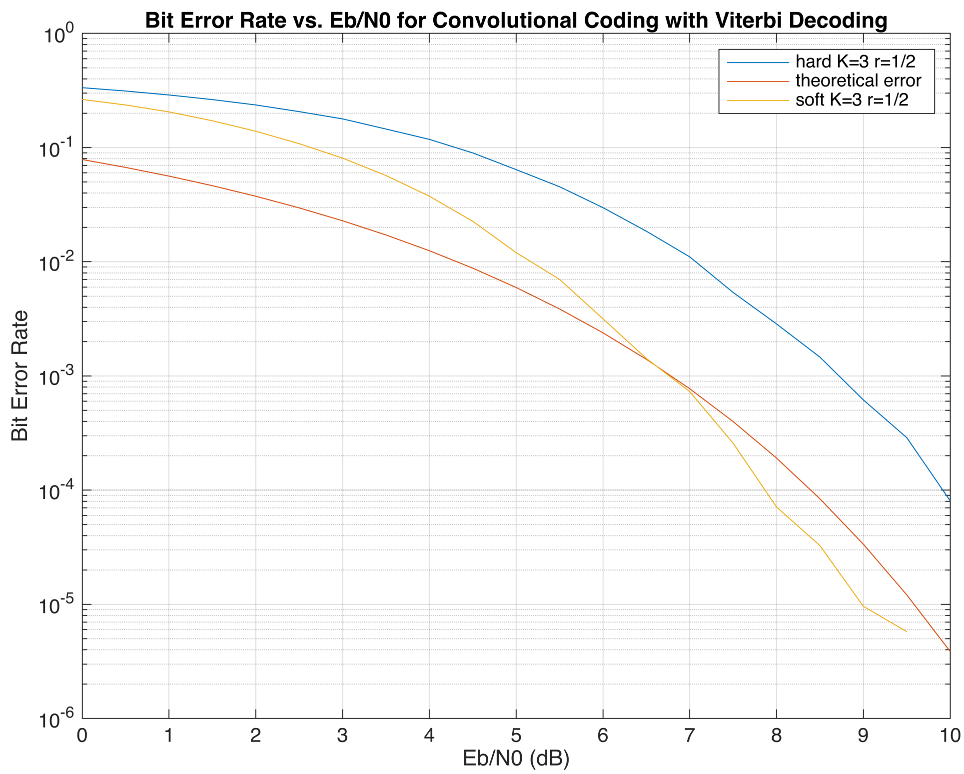
```
EbNodB = 0:0.5:10; % SNR values in dB
gamma = 10.^(EbNodB / 10); % Convert SNR from dB to linear scale
ES = 1; % Signal energys
nsim = 10000;
input_length =50;
input = rand(1,input_length) > 0.5;
```

i) For k = 1 , n = 2 , Kc = 3 and r = 1/2

```
k = 1;
n = 2;
r = k/n;
Kc = 3;
G = [[1 0 1]; [1 1 1]];

input_seq = input;
for i =1 : Kc-1
      input_seq = [input_seq 0]; %padding zeros
end

%obtaining state diagram
s = state_diag(G,Kc,n);

%obtaining encoded sequence
encoded_seq = encoding(G,Kc,input_seq);

% BPSK modulation
modulated_signal = 1 - 2*encoded_seq;

% for storing BER
error_rates_hard_1 = zeros(1,length(EbNodB));
error_rates_soft_1 = zeros(1,length(EbNodB));
theoratical_error_1 = zeros(1,length(EbNodB));
num_errors_hard = zeros(size(EbNodB));
num_errors_soft = zeros(size(EbNodB));


%Simulating errors for all values of EbNodB

for i = 1:length(EbNodB)
             %Generating noise
             noise_power=sqrt(1/(r*gamma(i)));
             BER_th = 0.5 * erfc(sqrt(1*gamma(i)));
```

```
            theoratical_error_1(i) = BER_th;
      for j=1:nsim
            % SNR and noise simulation
            noise = (noise_power)*randn(size(modulated_signal)); % AWGN
noise

            % Add noise to modulated signal
            received_signal = modulated_signal + noise;

            %For BPSK demodulation
            threshold = 0;


            %Calculating Demodulated signal
            demodulated_signal = zeros(size(received_signal));
            for k=1:length(received_signal)
              if(received_signal(k) < threshold)
                  demodulated_signal(k) = 1;
              end
            end

            % Hard Decoding
            decoded_seq_hard =
decoding_hard(s,Kc,n,demodulated_signal,length(input_seq));

            %Soft Decoding
            decoded_seq_soft =
decoding_soft(s,Kc,n,received_signal,length(input_seq));

            %Computing the number of error bits in the decoded sequence
            for k=1:length(input_seq)
                if(decoded_seq_hard(k)~=input_seq(k))
                    num_errors_hard(i) = num_errors_hard(i) + 1;
                end
                if(decoded_seq_soft(k)~=input_seq(k))
                    num_errors_soft(i) = num_errors_soft(i) + 1;
                end
            end
      end


end

%Computing BER
error_rates_hard_1 = num_errors_hard / (nsim*length(input_seq));
error_rates_soft_1 = num_errors_soft / (nsim*length(input_seq));

% Plot error rates vs. Eb/N0
semilogy(EbNodB, error_rates_hard_1);
hold on;
```

```
semilogy(EbNodB,theoratical_error_1);
semilogy(EbNodB,error_rates_soft_1);
hold off;
xlabel('Eb/N0 (dB)');
ylabel('Bit Error Rate');
title('Bit Error Rate vs. Eb/N0 for Convolutional Coding with Viterbi
Decoding');
legend('hard K=3 r=1/2', 'theoretical error', 'soft K=3 r=1/2');
grid on;
```

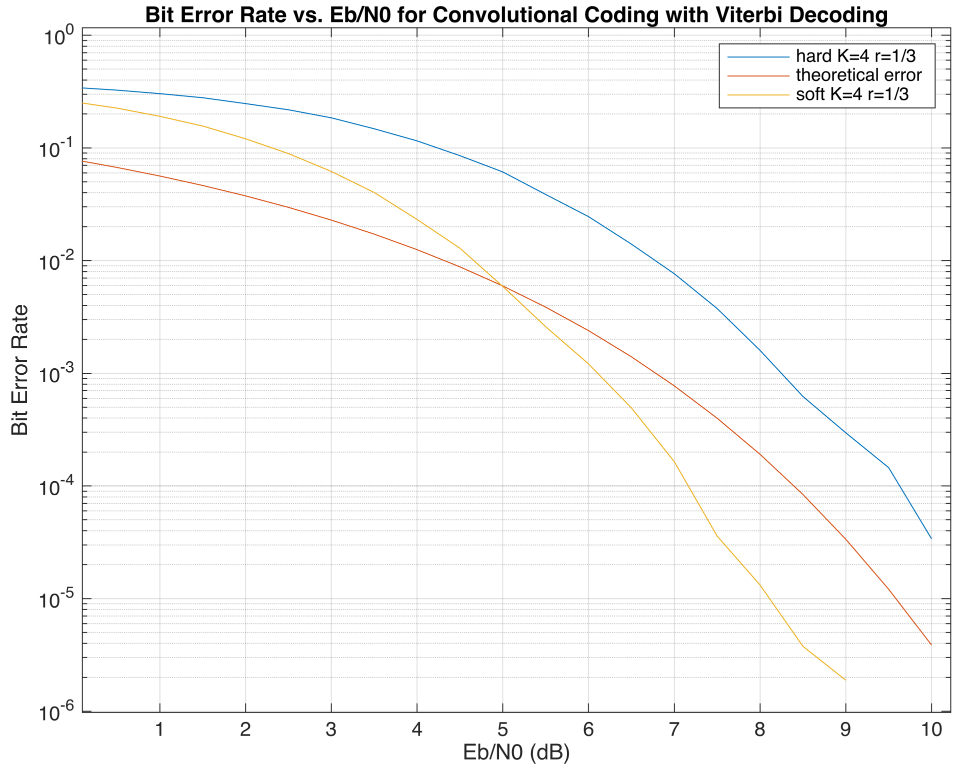


ii) For k = 1 , n = 3 , Kc = 4 and r = 1/3

```
k = 1;
n = 3;
r = k/n;
Kc = 4;
G = [[1 0 1 1];[1 1 0 1];[1 1 1 1]];

input_seq = input;
for i =1 : Kc-1
      input_seq = [input_seq 0]; %padding zeros
end
```

```
%obtaining state diagram
s = state_diag(G,Kc,n);

%obtaining encoded sequence
encoded_seq = encoding(G,Kc,input_seq);

% BPSK modulation
modulated_signal = 1 - 2*encoded_seq;

% for storing BER
error_rates_hard_2 = zeros(1,length(EbNodB));
error_rates_soft_2 = zeros(1,length(EbNodB));
theoratical_error_2 = zeros(1,length(EbNodB));
num_errors_hard = zeros(size(EbNodB));
num_errors_soft = zeros(size(EbNodB));


%Simulating errors for all values of EbNodB

for i = 1:length(EbNodB)
             %Generating noise
             noise_power=sqrt(1/(r*gamma(i)));
             BER_th = 0.5 * erfc(sqrt(1*gamma(i)));
             theoratical_error_2(i) = BER_th;
      for j=1:nsim
             % SNR and noise simulation
             noise = (noise_power)*randn(size(modulated_signal)); % AWGN
noise

             % Add noise to modulated signal
             received_signal = modulated_signal + noise;

             %For BPSK demodulation
             threshold = 0;


             %Calculating Demodulated signal
             demodulated_signal = zeros(size(received_signal));
             for k=1:length(received_signal)
               if(received_signal(k) < threshold)
                     demodulated_signal(k) = 1;
               end
             end

             % Hard Decoding
             decoded_seq_hard =
decoding_hard(s,Kc,n,demodulated_signal,length(input_seq));

             %Soft Decoding
```

```
            decoded_seq_soft =
decoding_soft(s,Kc,n,received_signal,length(input_seq));

            %Computing the number of error bits in the decoded sequence
            for k=1:length(input_seq)
                if(decoded_seq_hard(k)~=input_seq(k))
                    num_errors_hard(i) = num_errors_hard(i) + 1;
                end
                if(decoded_seq_soft(k)~=input_seq(k))
                    num_errors_soft(i) = num_errors_soft(i) + 1;
                end
            end
      end


end

%Computing BER
error_rates_hard_2 = num_errors_hard / (nsim*length(input_seq));
error_rates_soft_2 = num_errors_soft / (nsim*length(input_seq));

% Plot error rates vs. Eb/N0
semilogy(EbNodB, error_rates_hard_2);
hold on;
semilogy(EbNodB,theoratical_error_2);
semilogy(EbNodB,error_rates_soft_2);
hold off;
xlabel('Eb/N0 (dB)');
ylabel('Bit Error Rate');
title('Bit Error Rate vs. Eb/N0 for Convolutional Coding with Viterbi
Decoding');
legend('hard K=4 r=1/3', 'theoretical error', 'soft K=4 r=1/3');
grid on;
```

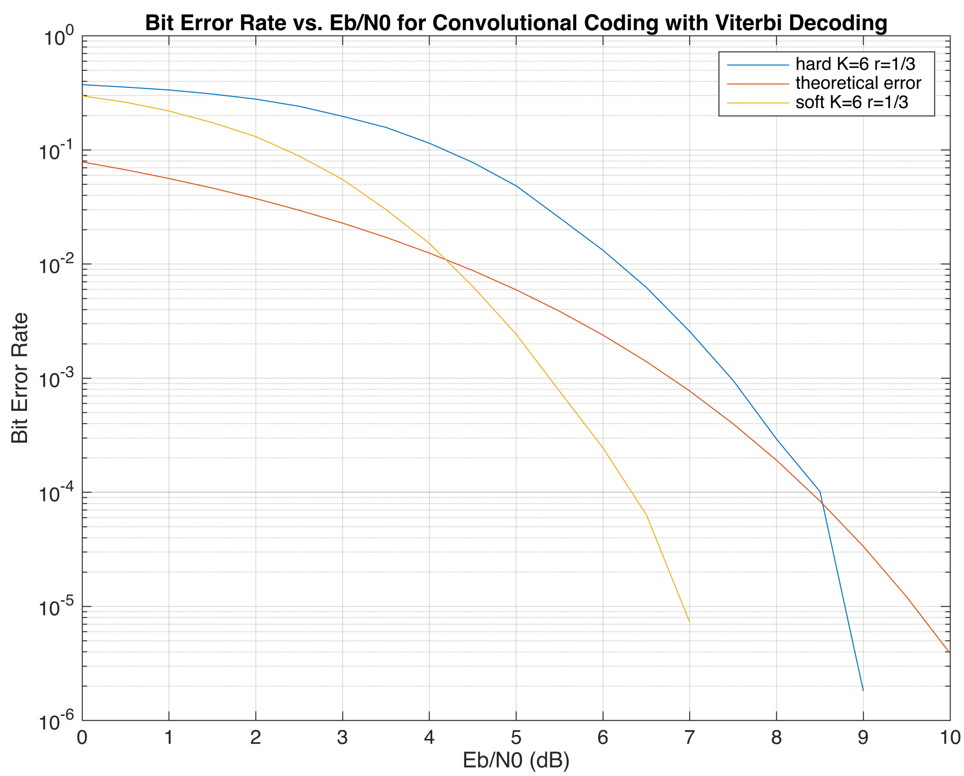


iii) For k = 1 , n = 3 , Kc = 6 and r = 1/3

```
k = 1;
n = 3;
r = k/n;
Kc = 6;
G = [ [1 0 0 1 1 1];[1,0,1,0,1,1];[1,1,1,1,0,1]];

input_seq = input;
for i =1 : Kc-1
      input_seq = [input_seq 0]; %padding zeros
end

%obtaining state diagram
s = state_diag(G,Kc,n);

%obtaining encoded sequence
encoded_seq = encoding(G,Kc,input_seq);

% BPSK modulation
modulated_signal = 1 - 2*encoded_seq;

% for storing BER
error_rates_hard_3 = zeros(1,length(EbNodB));
```

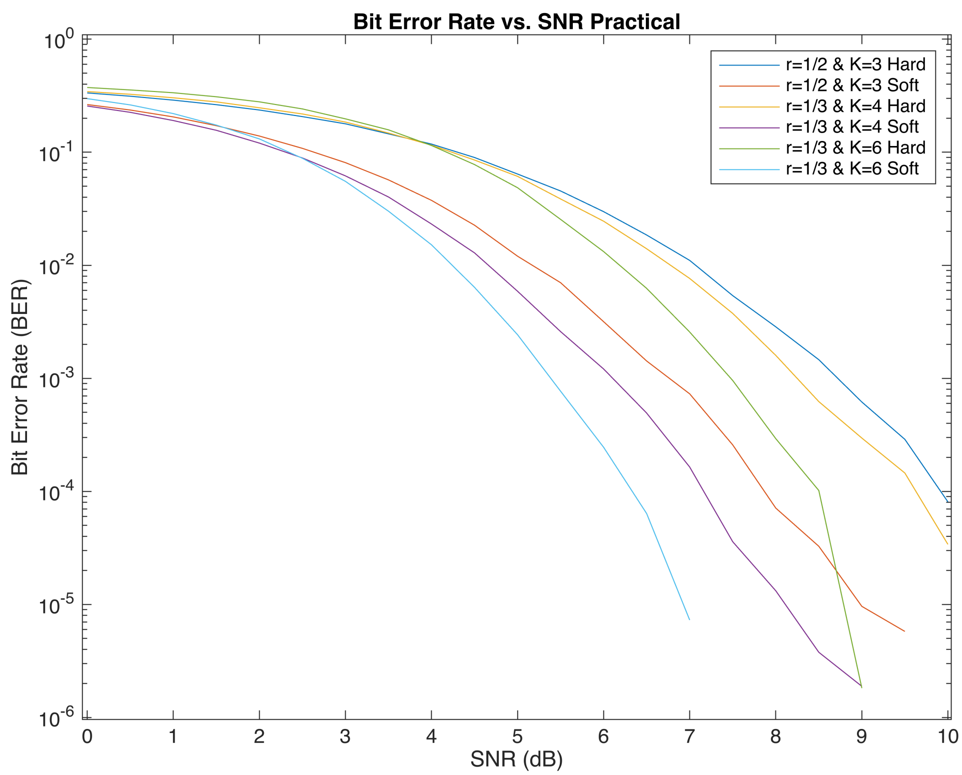
```
error_rates_soft_3 = zeros(1,length(EbNodB));
theoratical_error_3 = zeros(1,length(EbNodB));
num_errors_hard = zeros(size(EbNodB));
num_errors_soft = zeros(size(EbNodB));


%Simulating errors for all values of EbNodB

for i = 1:length(EbNodB)
             %Generating noise
             noise_power=sqrt(1/(r*gamma(i)));
             BER_th = 0.5 * erfc(sqrt(1*gamma(i)));
             theoratical_error_3(i) = BER_th;
      for j=1:nsim
             % SNR and noise simulation
             noise = (noise_power)*randn(size(modulated_signal)); % AWGN
noise

             % Add noise to modulated signal
             received_signal = modulated_signal + noise;

             %For BPSK demodulation
             threshold = 0;


             %Calculating Demodulated signal
             demodulated_signal = zeros(size(received_signal));
             for k=1:length(received_signal)
               if(received_signal(k) < threshold)
                     demodulated_signal(k) = 1;
               end
             end

             % Hard Decoding
             decoded_seq_hard =
decoding_hard(s,Kc,n,demodulated_signal,length(input_seq));

             %Soft Decoding
             decoded_seq_soft =
decoding_soft(s,Kc,n,received_signal,length(input_seq));

             %Computing the number of error bits in the decoded sequence
             for k=1:length(input_seq)
                 if(decoded_seq_hard(k)~=input_seq(k))
                     num_errors_hard(i) = num_errors_hard(i) + 1;
                 end
                 if(decoded_seq_soft(k)~=input_seq(k))
                     num_errors_soft(i) = num_errors_soft(i) + 1;
                 end
             end
```

```
        end


end

%Computing BER
error_rates_hard_3 = num_errors_hard / (nsim*length(input_seq));
error_rates_soft_3 = num_errors_soft / (nsim*length(input_seq));

% Plot error rates vs. Eb/N0
semilogy(EbNodB, error_rates_hard_3);
hold on;
semilogy(EbNodB,theoratical_error_3);
semilogy(EbNodB,error_rates_soft_3);
hold off;
xlabel('Eb/N0 (dB)');
ylabel('Bit Error Rate');
title('Bit Error Rate vs. Eb/N0 for Convolutional Coding with Viterbi
Decoding');
legend('hard K=6 r=1/3', 'theoretical error', 'soft K=6 r=1/3');
grid on;
```



Analysis Of Hard And Soft Decoding For All Rates

```
figure(1);
semilogy(EbNodB,error_rates_hard_1,EbNodB, error_rates_soft_1, EbNodB,
error_rates_hard_2, EbNodB, error_rates_soft_2, EbNodB, error_rates_hard_3,
EbNodB, error_rates_soft_3);
xlabel('SNR (dB)');
ylabel('Bit Error Rate (BER)');
legend('r=1/2 & K=3 Hard','r=1/2 & K=3 Soft','r=1/3 & K=4 Hard','r=1/3 &
K=4 Soft','r=1/3 & K=6 Hard','r=1/3 & K=6 Soft');
title('Bit Error Rate vs. SNR Practical');
```



Function For State Diagram

```
function [outputArg] = state_diag(G,Kc,n)

    % Calculate the number of states based on the constraint length Kc
    no_of_states = 2^(Kc-1);

    % Initialize an array to hold the binary representation of each state
    arr = zeros(no_of_states,Kc-1);
    for i=0:no_of_states-1
        % Convert the state index to binary representation
        x = int2bit(i,Kc-1);
        x1 = x';
        arr(i+1,:)=x1;
    end
```

```
    % Initialize the output array to hold the state diagram

    outputArg = zeros(no_of_states,4);

    % In the first and the second columns, we are storing the output from
the corresponding state
    % when the input bit is 0 and 1, respectively
    % And in third and fourth columns, we are storing the next states,
    % when the input bits are 0 and 1, respectively

    for i=1:no_of_states
        % Compute the output and next states for input 0
        arr0 = arr(i,:);
        arr0 = [0 arr0]; % Add input 0 to the state
        op0 = mod(G*arr0',2); % Compute the output
        outputArg(i,1)=bit2int(op0,n); % Store the output
        next_state0 = [];
        for j=1:Kc-1
            next_state0 = [next_state0 arr0(j)]; % Compute the next state
        end
        x = bit2int(next_state0',Kc-1); % Convert next state to integer
        outputArg(i,3)=x; % Store the next state for input 0

        % Compute the output and next states for input 1
        arr1 = arr(i,:);
        arr1 = [1 arr1]; % Add input 1 to the state
        op1 = mod(G*arr1',2); % Compute the output
        outputArg(i,2)=bit2int(op1,n); % Store the output
        next_state1 = [];
        for j=1:Kc-1
            next_state1 = [next_state1 arr1(j)]; % Compute the next state
        end
        outputArg(i,4)=bit2int(next_state1',Kc-1); % Store the next state
for input 1
    end
end
```

Function For Encoding sequence

```
function [outputArg] = encoding(G,Kc,input_seq)

    encoded_msg = [];

    % Initialize the shift register with the first element of the input
    % sequence
    arr = [input_seq(1)];
```

```
    % Pad the shift register array with Kc-1 zeros
    for i=1:Kc-1
        arr = [arr 0];
    end

    % Obtaining the encoded sequence for each input bit by multiplying
shift register array with G matrix
    for i=1:length(input_seq)
        arr1 = arr';
        arr2 = G*arr1;
        arr2 = mod(arr2,2);
        encoded_msg = [encoded_msg arr2'];

        % Shift the shift register to the right by one position
        for j=Kc:-1:2
            arr(j) = arr(j-1);
        end

        % Update the first element of the shift register with the next input
        if(i~=length(input_seq))
            arr(1)=input_seq(i+1);
        end
    end

    outputArg = encoded_msg;
end
```

Function For Viterbi Hard Decoding

```
function [outputArg] = decoding_hard(s,Kc,n,demod_seq,inp_len)
    % Calculating the number of rows and columns in the trellis
    rows = 2^(Kc-1);
    cols = inp_len+1;

    % Initialize 2-D arrays for storing the path metric, previous states and
    % previous inputs
    val_arr = 1000*ones(rows,cols);
    prev_state = -1*ones(rows,cols);
    prev_inp = -1*ones(rows,cols);

    % Setting the branch metric and previous state(path metric) for the
initial state as 0 and -1, respectively
    val_arr(1,1)=0;
    prev_state(1,1)=-1;

    % Iterate over each column of the trellis
    for j=1:cols-1
        x=[];
```

```
        % Extracting n bits from the demodulated sequence corresponding to
the current column
        for i=n*j-(n-1):n*j
            x = [x demod_seq(i)];
        end

        % Iterate over each state of the trellis
        for i=1:rows

            % Check whether the state has a valid path metric
            if(val_arr(i,j)~=1000)

                % Calculation for input bit 0
                op0 = s(i,1);
                ns0 = s(i,3)+1;
                op0_bin = int2bit(op0,n);
                op0_bin = op0_bin';

                % Calculating the hamming distance for transition 0
                hd0 = 0;
                for k=1:length(x)
                    if(x(k)~=op0_bin(k))
                        hd0=hd0+1;
                    end
                end

                % Update values in branch matric if the transition improves
the metric
                if(hd0+val_arr(i,j)<val_arr(ns0,j+1))
                    val_arr(ns0,j+1) = hd0+val_arr(i,j);
                    prev_state(ns0,j+1) = i;
                    prev_inp(ns0,j+1) = 0;
                end


                % Calculation for input bit 0
                op1 = s(i,2);
                ns1 = s(i,4)+1;
                op1_bin = int2bit(op1,n);
                op1_bin = op1_bin';

                % Calculating the hamming distance for transition 0
                hd1 = 0;
                for k=1:length(x)
                    if(x(k)~=op1_bin(k))
                        hd1=hd1+1;
                    end
                end
```

```
                % Update values if the transition improves the metric
                if(hd1+val_arr(i,j)<val_arr(ns1,j+1))
                    val_arr(ns1,j+1) = hd1+val_arr(i,j);
                    prev_state(ns1,j+1) = i;
                    prev_inp(ns1,j+1) = 1;
                end
            end
        end
    end
    i = 1;
    decoded_seq = [];

    % Backtrack through the trellis to find the most likely sequence
    for j=cols:-1:2
        decoded_seq = [decoded_seq prev_inp(i,j)];
        i = prev_state(i,j);
    end

    % Return the decoded sequence
    outputArg = fliplr(decoded_seq);
end
```

Function For Viterbi Soft Decoding

```
function [outputArg] = decoding_soft(s,Kc,n,demod_seq,inp_len)

    % Soft decoding code almost same as Hard decoding, instead of
    % demodulated signal, we pass the received signal and instead of
    % hamming distance we use euclidean distance

    % Calculating the number of rows and columns in the trellis
    rows = 2^(Kc-1);
    cols = inp_len+1;

    % Initialize 2-D arrays for storing the path metric, previous states and
    % previous inputs
    val_arr = 1000*ones(rows,cols);
    prev_state = -1*ones(rows,cols);
    prev_inp = -1*ones(rows,cols);

    % Setting the path metric and previous state for the initial state as 0
and -1, respectively
    val_arr(1,1)=0;
    prev_state(1,1)=-1;

    % Iterate over each column of the trellis
    for j=1:cols-1
        x=[];
```

```
        % Extracting n bits from the demodulated sequence corresponding to
the current column
        for i=n*j-(n-1):n*j
            x = [x demod_seq(i)];
        end

        % Iterate over each state of the trellis
        for i=1:rows

            % Check whether the state has a valid path metric
            if(val_arr(i,j)~=1000)

                % Calculation for input bit 0
                op0 = s(i,1);
                ns0 = s(i,3)+1;
                op0_bin = int2bit(op0,n);
                op0_bin = op0_bin';
                op0_bin = 1 - 2*op0_bin;

                % Calculating the euclidean distance for transition 0
                path_metric0 = sum((x-op0_bin).^2);

                % Update values if the transition improves the metric
                if(path_metric0+val_arr(i,j)<val_arr(ns0,j+1))
                    val_arr(ns0,j+1) = path_metric0+val_arr(i,j);
                    prev_state(ns0,j+1) = i;
                    prev_inp(ns0,j+1) = 0;
                end


                % Calculation for input bit 0
                op1 = s(i,2);
                ns1 = s(i,4)+1;
                op1_bin = int2bit(op1,n);
                op1_bin = op1_bin';
                op1_bin = 1 - 2*op1_bin;

                % Calculating the euclidean distance for transition 0
                path_metric1 = sum((x-op1_bin).^2);

                % Update values if the transition improves the metric
                if(path_metric1+val_arr(i,j)<val_arr(ns1,j+1))
                    val_arr(ns1,j+1) = path_metric1+val_arr(i,j);
                    prev_state(ns1,j+1) = i;
                    prev_inp(ns1,j+1) = 1;
                end
            end
        end
    end
    i = 1;
```

```
    decoded_seq = [];

    % Backtrack through the trellis to find the most likely sequence
    for j=cols:-1:2
        decoded_seq = [decoded_seq prev_inp(i,j)];
        i = prev_state(i,j);
    end

    % Return the decoded  sequence
    outputArg = fliplr(decoded_seq);
end
```