Convolutional Encoding and Viterbi Decoding

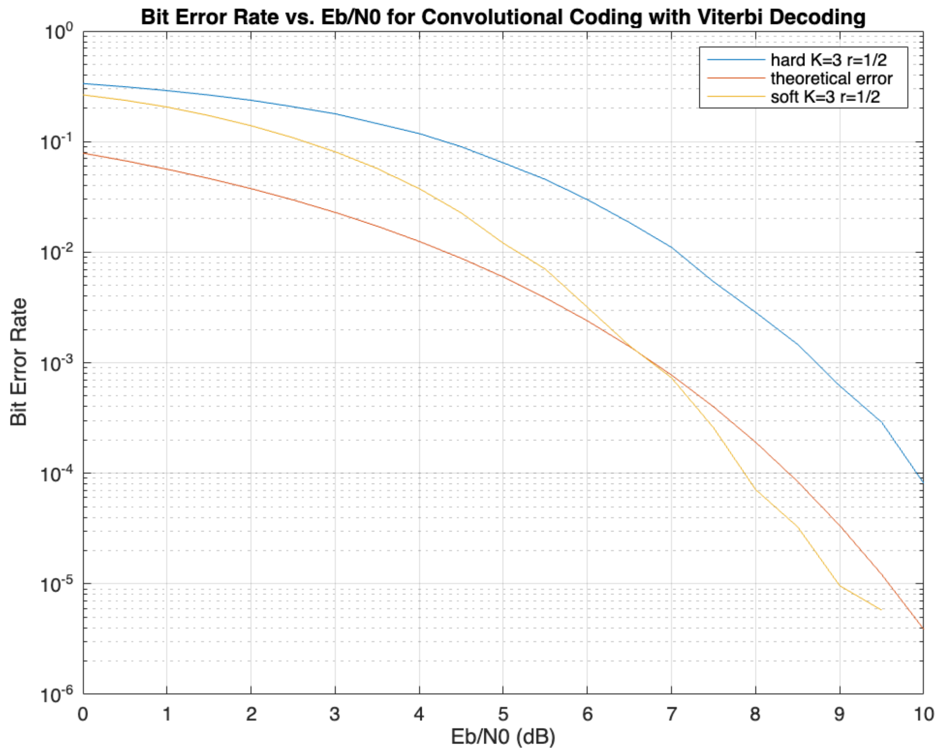
```
EbNodB = 0:0.5:10; % SNR values in dB
gamma = 10.^(EbNodB / 10); % Convert SNR from dB to linear scale
ES = 1; % Signal energys
nsim = 5000;
inputsz =50;
input = rand(1,inputsz) > 0.5;
```

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]];

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

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

%obtaining encoded sequence
encodedseq = encoding(G,Kc,inputseq);

% BPSK modulation
modulatedsig = 1 - 2*encodedseq;

% for storing BER
harderrorrate1 = zeros(1,length(EbNodB));
softerrorrate1 = zeros(1,length(EbNodB));
theerrorrate1 = zeros(1,length(EbNodB));
hardnumerrors = zeros(size(EbNodB));
softnumerrors = zeros(size(EbNodB));


%Simulating errors for all values of EbNodB

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

```
            theerrorrate1(i) = BER_th;
      for j=1:nsim
            % SNR and noise simulation
            noise = (noisepow)*randn(size(modulatedsig)); % AWGN noise

            % Add noise to modulated signal
            recsignal = modulatedsig + noise;

            %For BPSK demodulation
            threshold = 0;


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

            % Hard Decoding
            harddecseq = harddec(s,Kc,n,demodulatedsig,length(inputseq));

            %Soft Decoding
            softdecseq = softdec(s,Kc,n,recsignal,length(inputseq));

            %Computing the number of error bits in the decoded sequence
            for k=1:length(inputseq)
                if(harddecseq(k)~=inputseq(k))
                    hardnumerrors(i) = hardnumerrors(i) + 1;
                end
                if(softdecseq(k)~=inputseq(k))
                    softnumerrors(i) = softnumerrors(i) + 1;
                end
            end
      end


end

%Computing BER
harderrorrate1 = hardnumerrors / (nsim*length(inputseq));
softerrorrate1 = softnumerrors / (nsim*length(inputseq));

% Plot error rates vs. Eb/N0
semilogy(EbNodB, harderrorrate1);
hold on;
semilogy(EbNodB,theerrorrate1);
semilogy(EbNodB,softerrorrate1);
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;
```

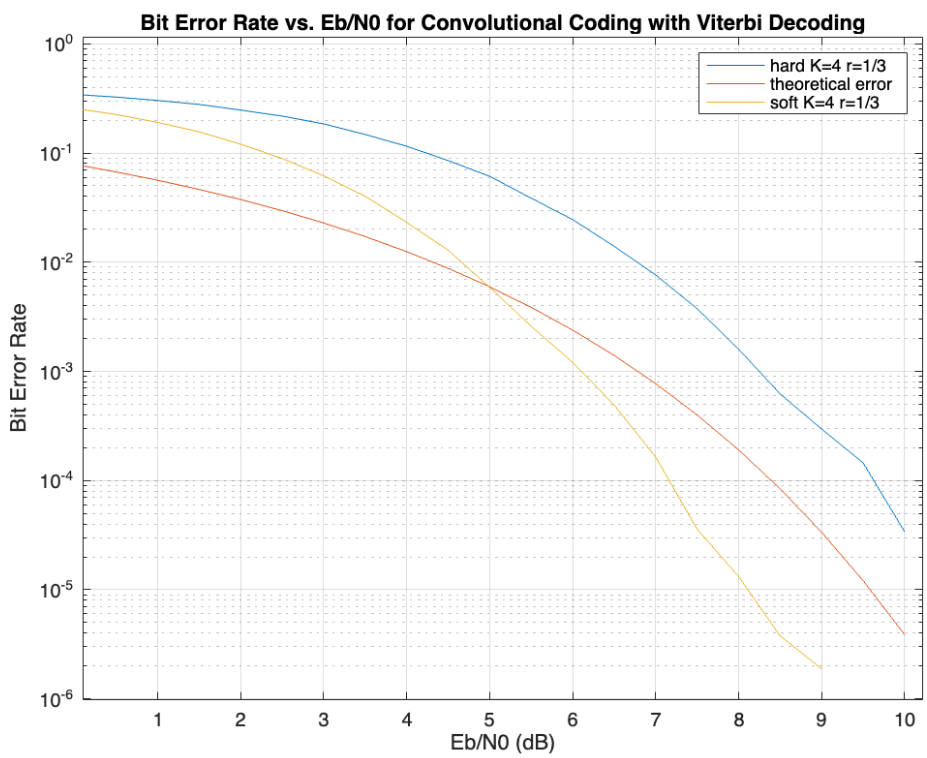


ii) For k = 1 , n = 3 , Kc = 4 and r = 1/3

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

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

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

%obtaining encoded sequence
encodedseq = encoding(G,Kc,inputseq);
```

```
% BPSK modulation
modulatedsig = 1 - 2*encodedseq;

% for storing BER
harderrorrate2 = zeros(1,length(EbNodB));
softerrorrate2 = zeros(1,length(EbNodB));
theerrorrate2 = zeros(1,length(EbNodB));
hardnumerrors = zeros(size(EbNodB));
softnumerrors = zeros(size(EbNodB));


%Simulating errors for all values of EbNodB

for i = 1:length(EbNodB)
             %Generating noise
             noisepow=sqrt(1/(r*gamma(i)));
             BER_th = 0.5 * erfc(sqrt(1*gamma(i)));
             theerrorrate2(i) = BER_th;
      for j=1:nsim
             % SNR and noise simulation
             noise = (noisepow)*randn(size(modulatedsig)); % AWGN noise

             % Add noise to modulated signal
             recsignal = modulatedsig + noise;

             %For BPSK demodulation
             threshold = 0;


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

             % Hard Decoding
             harddecseq = harddec(s,Kc,n,demodulatedsig,length(inputseq));

             %Soft Decoding
             softdecseq = softdec(s,Kc,n,recsignal,length(inputseq));

             %Computing the number of error bits in the decoded sequence
             for k=1:length(inputseq)
                 if(harddecseq(k)~=inputseq(k))
                     hardnumerrors(i) = hardnumerrors(i) + 1;
                 end
                 if(softdecseq(k)~=inputseq(k))
```

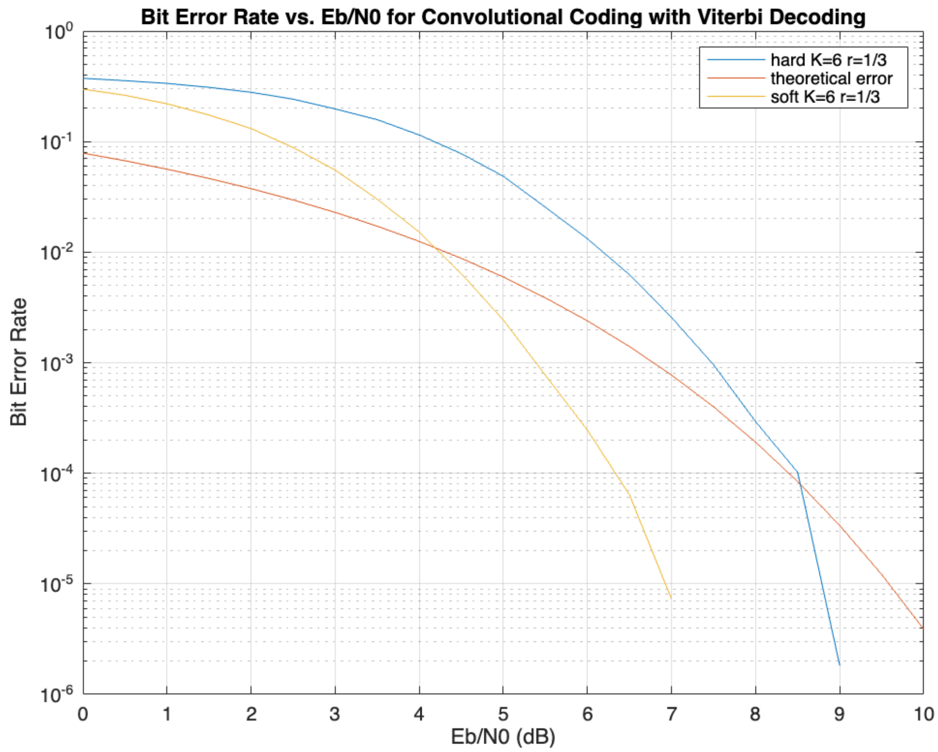
```
                    softnumerrors(i) = softnumerrors(i) + 1;
                end
            end
        end


end

%Computing BER
harderrorrate2 = hardnumerrors / (nsim*length(inputseq));
softerrorrate2 = softnumerrors / (nsim*length(inputseq));

% Plot error rates vs. Eb/N0
semilogy(EbNodB, harderrorrate2);
hold on;
semilogy(EbNodB,theerrorrate2);
semilogy(EbNodB,softerrorrate2);
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

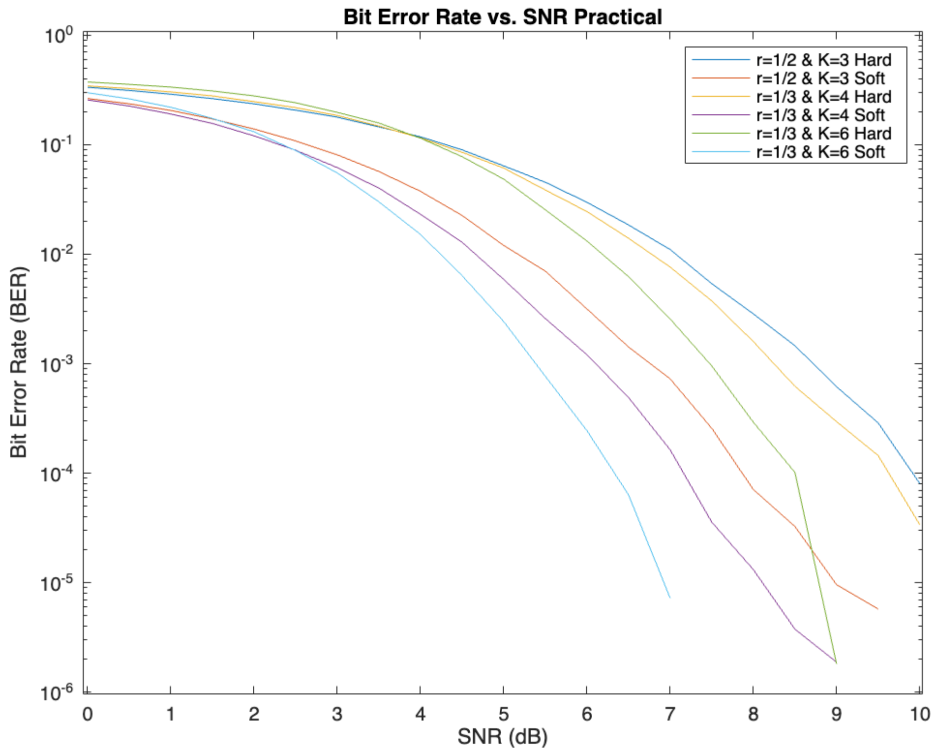
```
k = 1;
n = 3;
r = k/n;
Kc = 6;
G = [ [1 0 0 1 1 1];[1,0,1,0,1,1];[1,1,1,1,0,1]];

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

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

%obtaining encoded sequence
encodedseq = encoding(G,Kc,inputseq);

% BPSK modulation
modulatedsig = 1 - 2*encodedseq;

% for storing BER
harderrorrate3 = zeros(1,length(EbNodB));
softerrorrate3 = zeros(1,length(EbNodB));
theerrorrate3 = zeros(1,length(EbNodB));
hardnumerrors = zeros(size(EbNodB));
softnumerrors = zeros(size(EbNodB));


%Simulating errors for all values of EbNodB

for i = 1:length(EbNodB)
             %Generating noise
             noisepow=sqrt(1/(r*gamma(i)));
             BER_th = 0.5 * erfc(sqrt(1*gamma(i)));
             theerrorrate3(i) = BER_th;
      for j=1:nsim
             % SNR and noise simulation
             noise = (noisepow)*randn(size(modulatedsig)); % AWGN noise

             % Add noise to modulated signal
             recsignal = modulatedsig + noise;

             %For BPSK demodulation
             threshold = 0;


             %Calculating Demodulated signal
             demodulatedsig = zeros(size(recsignal));
             for k=1:length(recsignal)
               if(recsignal(k) < threshold)
```

```
                    demodulatedsig(k) = 1;
                end
            end

            % Hard Decoding
            harddecseq = harddec(s,Kc,n,demodulatedsig,length(inputseq));

            %Soft Decoding
            softdecseq = softdec(s,Kc,n,recsignal,length(inputseq));

            %Computing the number of error bits in the decoded sequence
            for k=1:length(inputseq)
                if(harddecseq(k)~=inputseq(k))
                    hardnumerrors(i) = hardnumerrors(i) + 1;
                end
                if(softdecseq(k)~=inputseq(k))
                    softnumerrors(i) = softnumerrors(i) + 1;
                end
            end
    end


end

%Computing BER
harderrorrate3 = hardnumerrors / (nsim*length(inputseq));
softerrorrate3 = softnumerrors / (nsim*length(inputseq));

% Plot error rates vs. Eb/N0
semilogy(EbNodB, harderrorrate3);
hold on;
semilogy(EbNodB,theerrorrate3);
semilogy(EbNodB,softerrorrate3);
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,harderrorrate1,EbNodB, softerrorrate1, EbNodB, harderrorrate2,
EbNodB, softerrorrate2, EbNodB, harderrorrate3, EbNodB, softerrorrate3);
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] = statediag(G,Kc,n)

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

    % Initialize an array to hold the binary representation of each state
    arr = zeros(totnoofstate,Kc-1);
    for i=0:totnoofstate-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(totnoofstate,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:totnoofstate
        % 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
        nextstate0 = [];
        for j=1:Kc-1
            nextstate0 = [nextstate0 arr0(j)]; % Compute the next state
        end
        x = bit2int(nextstate0',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
        nextstate1 = [];
        for j=1:Kc-1
            nextstate1 = [nextstate1 arr1(j)]; % Compute the next state
        end
        outputArg(i,4)=bit2int(nextstate1',Kc-1); % Store the next state for input 1
    end
end
```

Function For Encoding sequence

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

    encodmsg = [];

    % Initialize the shift register with the first element of the input
    % sequence
    arr = [inputseq(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(inputseq)
        arr1 = arr';
        arr2 = G*arr1;
```

```
        arr2 = mod(arr2,2);
        encodmsg = [encodmsg 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(inputseq))
            arr(1)=inputseq(i+1);
        end
    end

    outputArg = encodmsg;
end
```

Function For Viterbi Hard Decoding

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

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

    % Setting the branch metric and previous state(path metric) for the initial
state as 0 and -1, respectively
    valarr(1,1)=0;
    prevstate(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(valarr(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+valarr(i,j)<valarr(ns0,j+1))
                    valarr(ns0,j+1) = hd0+valarr(i,j);
                    prevstate(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+valarr(i,j)<valarr(ns1,j+1))
                    valarr(ns1,j+1) = hd1+valarr(i,j);
                    prevstate(ns1,j+1) = i;
                    prev_inp(ns1,j+1) = 1;
                end
            end
        end
    end
    i = 1;
    decodseq = [];
```

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

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

Function For Viterbi Soft Decoding

```
function [outputArg] = softdec(s,Kc,n,demod_seq,inplen)

    % 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 = inplen+1;

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

    % Setting the path metric and previous state for the initial state as 0 and -1,
respectively
    valarr(1,1)=0;
    prevstate(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(valarr(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+valarr(i,j)<valarr(ns0,j+1))
                valarr(ns0,j+1) = path_metric0+valarr(i,j);
                prevstate(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+valarr(i,j)<valarr(ns1,j+1))
                valarr(ns1,j+1) = path_metric1+valarr(i,j);
                prevstate(ns1,j+1) = i;
                prev_inp(ns1,j+1) = 1;
            end
        end
    end
end
i = 1;
decodseq = [];

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

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