Viterbi Decoding for Convolutional Codes

Name:Sai Bhargav Dasari UnityId:sdasari

***Abstract*—The objective of this project is to analyze the performance of a 1/2 Convolutional Code with 3 memory elements using 1, 2 and 3-bit quantization of the received sequence transmitted over noisy channels using Viterbi algorithm to decode the message. By controlling the channel Signal-to-Noise ratio ,compare the BER performance of the code for the specified quantization schemes in comparison to Uncoded BER. The following report talks about the algorithmic details and own implementation of Viterbi decoding and presents the results in comparison to Uncoded BER.**

I. Introduction

Convolutional Codes are typically represented with three parameters n,k, and K, where n and k together represent the rate of the code while K represents the number of shift registers used in the encoding part. In Convolutional Codes, the input data is not typically divided into blocks of k bits but instead a continuous stream of data is used at the encoder's input. The coded sequence of n-bits obtained after encoding not only depends on the corresponding k-bits information message but also the past information bits.In other words, these codes have memory.

This report investigates the performance of a rate(k/n) 1/2 code with Constraint length(K)= 3. The details of the encoding are provided in the following section. Section 3 talks about Viterbi Decoding and the Trellis Structure. Section 4 presents the results of the simulation and section 5 presents the code used in the simulations.

II. Convolutional Encoder

Since, the complexity of the Viterbi Algorithm is tied to the complexity of the Convolutional Code used , a rather simple non-systematic encoder mentioned in [1] of the following form $G(x) = (1 + x + x^2, 1 + x^2)$ has been used.The Massey-sain condition can be used easily to prove that the mentioned code is not catastrophic. Since $1 + x + x^2$ is a prime polynomial of degree 2 in GF(2) the gcd of $1 + x + x^2$ and $1 + x^2$ is 1.The following picture depicts the encoding circuit [1] for the specified code:

As can be seen from the generating function G(x), the coded data depends on the current input bit and the two past message bits. This represents the memory part of a Convolutional Code.

After generating the codeword, it was modulated using BPSK and noise was added to it using MATLAB's *awgn* function.

III. Viterbi Decoding

Given a noisy codeword, it is inefficient in Convolutional Codes to sift through all the possible codewords and find the most likely codeword. This is because unlike block codes, convolutional codes aren't implemented for a specified block length. Usually the length of this continuous message might span several thousands of bits and as such to find the most likely codeword one might need to look at 2^{1000} combination of codewords or more and as such the memory requirements for such decoding is huge.Viterbi Decoding is a classic algorithm to recursively find the codeword closest to the received data.

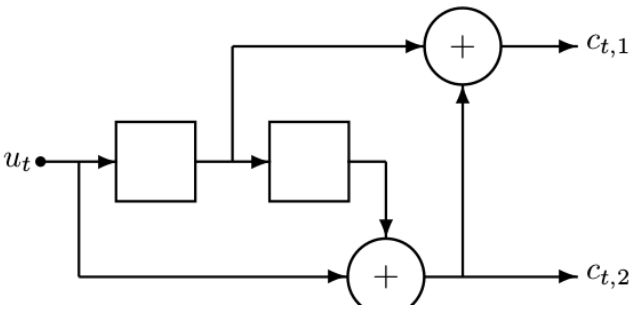

Fig. 1. Encoding Circuit with Shift Registers

The noisy codeword is demodulated and then decoded using my implementation of Viterbi Algorithm. The first step in Viterbi Decoding is to construct the trellis diagram [2] [3]. It gives information about the state transition and output information at each of the states.Given that the size of the code considered is small, instead of constructing the trellis state diagram, the state transition table and the output table are implemented independently. The state of the system is represented by the entries in the shift registers. Given that we have two shift registers, the total possible number of states is $2^2 = 4$.Also, since 1-bit quantization is considered we have two additional columns.One each for input bit=0,1. The following tables show the state transition table and output table for the encoder mentioned in section 2:

Previous State	Input 0	Input 1
00	00	10
01	00	10
10	01	11
11	01	11

TABLE I: State Transition Table

To ease the implementation, the states $(00, 01, 10, 11)$ are represented in the code as $(0, 1, 2, 3)$.The Viterbi-Decoding uses the Hamming Distance between the received codeword and the set of possible outputs at each instance to compute the error metric for each state and stores for each state it's

Previous State	Input 0	Input 1
00	00	11
01	11	00
10	10	01
11	01	10

TABLE II: Output Table

preceding state with the least error metric. For a codeword of length L, Viterbi Decoding needs memory on the order of $L * 2^{K-1} = 4L$ as opposed to the 2^L needed by standard decoding.

A. Soft- Decision Decoding

In the previous section we considered 1-bit quantization of the output followed by Viterbi Decoding.The branch metrics at each iteration were computed by initially thresholding the received sequence and using hamming distance between the thresholded received sequence and the possible outputs.The thresholding operation results in a loss of information and as such isn't optimal.In this section, the case with no quantization is considered.This is called Soft-Decision Decoding.

Given that the channel is AWGN , the probability of modulated code bit given input code bit is given by :

$$P(y/x) = k \exp(- \|y - x\|^2 / N)$$

where y- received modulated code bit
x- actual modulated code bit
c- input code bit
k=constant
From the above equation we can infer the following:

$$-\log(P(y/x)) = \frac{1}{N0} \|y - x\|^2 + k1$$

Therefore a valid branch metric for soft-decision decoding is given by:

$$m(y/x) = \|y - x\|^2$$

The modulation scheme considered in the simulations is BPSK which outputs a -1 if the input data bit is 0 and outputs 1 if the input bit is 1.Therefore x can either be -1 or 1.

$$m(y/1) = y^2 - 2y + 1$$
$$m(y/-1) = y^2 + 2y + 1$$

The terms y^2,1 are constant between both the cases and as such can be discarded. Therefore the metric can be written as:

$$m(y/1) = -y$$
$$m(y/-1) = y$$

As the Viterbi algorithm takes two received coded bits at a time for processing, we need to find the Euclidean distance from both the bits.

$$m_{00} = (y_1 + 1)^2 + (y_2 + 1)^2 = y_1 + y_2$$
$$m_{01} = (y_1 + 1)^2 + (y_2 - 1)^2 = y_1 - y_2$$
$$m_{10} = (y_1 - 1)^2 + (y_2 + 1)^2 = -y_1 + y_2$$
$$m_{11} = (y_1 - 1)^2 + (y_2 + 1)^2 = -y_1 - y_2$$

The Hamming distance in the branch metric calculation for hard-decision decoding is replaced by the euclidean distance as specified above.The performance of this scheme compared to the hard decision scheme is presented in the next section.

IV. Quantization

In reality, the no quantization case is not feasible and we need to have some level of quantization in the final output received from the channel. The performance of the code heavily relies on the type of quantization implemented.

Quantizers can be broadly classified into two types-Scalar Quantizers and Vector Quantizers.Scalar Quantizers sample one output sample at a time whereas Vector quantizers(m-dimensional) sample m samples at a time.While Vector Quantizers are claimed to achieve better performance over their Scalar counterparts in decoding of Convolutional Codes with high number of states,this project only considers scalar quantization schemes as the Convolutional Code being considered is simple and has only 4 states. A straightforward way to achieve Scalar Quantization would be to quantize the sample space uniformly.But in most cases, uniform quantization will be sub optimal and there's a need for better techniques. Optimal fixed rate scalar quantizers introduced by Max [4] and Lloyd [5] minimize the average distortion for a given number of quantization levels and are known as Lloyd-Max Quantizers.

While the number of quantization bits can be set high so as to achieve high performance it is to be noted that the hardware complexity of the decoder grows linearly with the number of bits.In this project,output quantization of 2 and 3-bits is considered.

A. Lloyd-Max Quantizer

For a signal X with pdf given by $f_x(x)$ the Lloyd-Max Quantizer obtains the quantized signal $X^{'}$ in such a way that the mean square error between X and $X^{'}$ is minimized.

$$\underset{min}{d} = E[(X - X^{'})^2]$$

A quantizer is defined by it's representative levels and the decision thresholds. An output value is assigned the representative value x_q if it falls between the decision thresholds t_q and t_{q+1}.The iterative method for obtaining the partitions in a M-bit Lloyd-Max quantizer are given as follows:

- Guess initial set of representative levels x_q where q= 1, 2,3 ... 2^M
- Calculate Decision thresholds

$$t_q = \frac{1}{2}(x_{q-1} + x_q)$$

- Calculate new representative levels for all q = 1,2,3...2^M

$$x_q = \frac{\int_{t_q}^{t_{q+1}} x.f_x(x)dx}{\int_{t_q}^{t_q+1} f_x(x)dx}$$

The algorithm is iterated until the relative change of MSE between successive iterations reaches a value less than 10^{-7}. The MATLAB function *lloyds* [7] is used to implement the lloyd-max quantization part. The probability distribution of the signal can be approximated by utilizing a few output samples.In my simulations, I've used 1000 samples to approximate the distrbution $f_x(x)$.

V. Results

This section discusses results of simulations done on MATLAB. Analysis of performance is done with respect to BER. BPSK is the modulation scheme utilized and the noise added is additive AWGN noise. The following plot depicts the performance plot for Coded 1-bit quantization , Coded no quantization , Coded 2-bit quantization ,Coded 3-bit quabtization, Uncoded code of length 10^5 bits. The code for hard decision viterbi and soft-decision viterbi is implemented as different functions *viterbidechard* and *viterbidecsoft* respectively.The difference between the code files only lies in a few lines corresponding to the branch metric.

During my simulations I realized that the initialization of representative levels in Lloyd's quantization played a significant role in the performance of the quantization [6]. In the initial case the representative levels have been randomly initialized while in the second case the initialization is done using the levels of a uniform quantization.

A. Case 1- Random Initialization

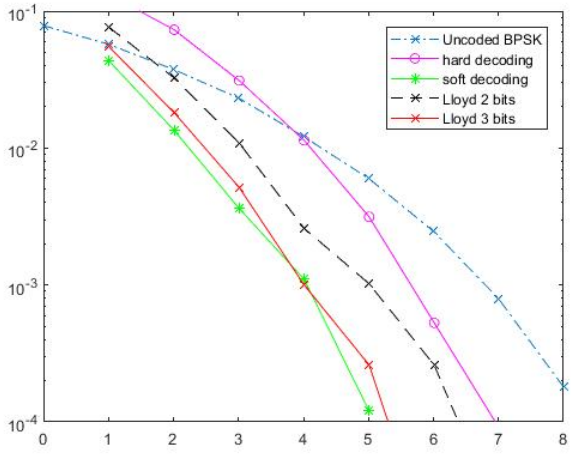

Fig. 2. Performance Comparison Plot

The below table lists the Energy per bit required to achieve an error rate of 1.8 * 10^{-4}

Algorithm	Eb/N0(dB)
Uncoded	8
Hard Decoding	6.6
Quantized 4-bit	6.15
Quantized 8-bit	5.1
Soft Decoding	4.8

B. Case 2-Initialization using Uniform Quantization

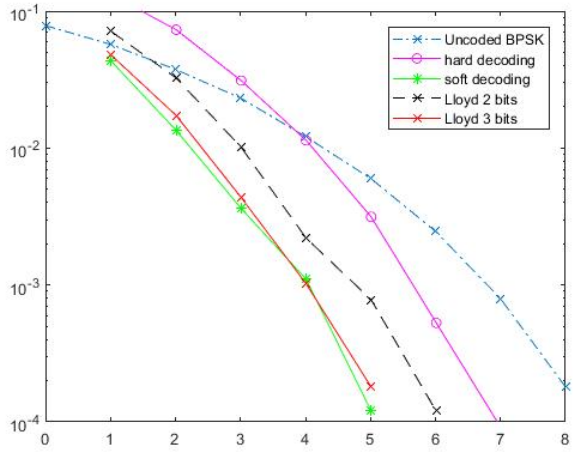

Fig. 3. Performance Comparison Plot

The following table illustrates the Energy per bit needed to achieve an error-rate of 1.8 * 10^{-4} in this case.

Algorithm	Eb/N0(dB)
Uncoded	8
Hard Decoding	6.6
Quantized 4-bit	5.8
Quantized 8-bit	5.01
Soft Decoding	4.8

VI. Conclusion

The final report talks about the implementation of quantized soft-decoding and compares it's performance to hard decoding and soft-decoding

It can be seen from the above results that while the hard-decision decoding improves the performance by around 1.4 dB compared to Uncoded BPSK it still pales in comparison to the 3.2 dB performance increment of the Soft-decision decoding. Since soft-decision decoding isn't actually feasible and there is a need for digitization ,the performance of 3-bit and 2-bit output quantization is evaluated. The performance of the quantized soft-decoding depends quite a bit on the initialization of Lloyd-max Algorithm. In the second case it can be seen that the performance of 3-bit decoding is just

0.2dB away from the no quantized soft-decoding and the performance of 2-bit quantization is approximately 1 dB away from soft-decision decoding.

VII. References

[1]Roth, Ron. Introduction to coding theory. Cambridge University Press, 2006.

[2]Tutorial on Convolutional Coding with Viterbi Decoding. home.netcom.com/~chip.f/viterbi/tutorial.html

[3]Shu Lin, Daniel J. Costello, Error Control Coding-Fundamentals and Applications, 2ed

[4] Max, J. (1960). Quantizing for minimum distortion. IRE Transactions on Information Theory, 6(1), 7-12.

[5]Lloyd, S. (1982). Least squares quantization in PCM. IEEE transactions on information theory, 28(2), 129-137.

[6] Vicory, J., Celebi, M. E. (2012, February). An adaptive and deterministic method for initializing the Lloyd-Max algorithm. In Image Processing: Algorithms and Systems X; and Parallel Processing for Imaging Applications II (Vol. 8295, p. 82951I). International Society for Optics and Photonics.

[7]Lloyd Algorithm. (n.d.). Retrieved from https://www.mathworks.com/help/comm/ref/lloyds.html

```
function [predecessor_states,error_Table,optimum_path,data] =
 viterbidecsoft(K,coded)

%Function implements the soft decision viterbi decoding
% K denotes the constraint length and coded is the noisy received
% sequence

%Intializing arrays essential for the state decoding process
%The State Transition table: the next state given the current state
 and
%input
transition_Table=[0 0 2;1 0 2;2 1 3;3 1 3];
%The output table:determins the output given the current state and the
%input
output_Table=[0 0 3;1 3 0;2 2 1;3 1 2];
%Error Metric Array-Track of accumulated error
error_Table=zeros(2^(K-1),1);
%Initialize the current state to be always 0
current_state=[0];
%State predecessor history-local optimum at each time instant
predecessor_states=zeros(2^(K-1),length(coded)/2);

%Main section where magic happens
for i=0:length(predecessor_states)-1
    %possible states the algorithm might end up in given it's current
 state
    possible_states=transition_Table(current_state+1,2:3);
    %possible outputs the algorithm might end up in given it's current
 state
    possible_outputs=output_Table(current_state+1,2:3);
    possible_states=reshape(possible_states',1,[]);
    possible_outputs=reshape(possible_outputs',1,[]);
    possible_outputs=de2bi(possible_outputs,2,'left-msb');
    possible_outputs(possible_outputs~=0)=-1;
    possible_outputs(possible_outputs==0)=1;
    %Distance between received sequence and possible outputs
    %ham_distance= sum(abs(dec2bin(coded(i),2)-
dec2bin(possible_outputs,2)),2);
    euc_distance= sum([coded(2*i+1) coded(2*i
+2)].*possible_outputs,2);
    %A variable to keep track if a state has aleady been updated
 during the loop
    flag=zeros(2^(K-1),1);
    %A duplicate error metric variable to keep track of error in the
 for loop
    error_Table2=nan(2^(K-1),1);
    %loop through current states
    for count=1:length(current_state)
        count2=transition_Table(current_state(count)+1,2:3);
        %update error metric for the possible states
        if(sum(flag(count2+1))==0)
            error_Table2(count2+1,1)=
 error_Table(current_state(count)+1,1)+euc_distance(2*count-1:2*count,1);
```

```
            flag(count2+1)=1;
            predecessor_states(count2+1,i+1)= current_state(count);
        else

 error_temp=error_Table(current_state(count)+1,1)+euc_distance(2*count-1:2*count,1

 [error_Table2(count2+1,1),ind]=min([error_Table2(count2+1,1)';error_temp']);
            change_index= ind==2;
            if(~isempty(ind==2))
                predecessor_states(count2(change_index)+1,i
+1)=current_state(count);
            end


        end

    end
    error_Table2(isnan(error_Table2))=0;
    error_Table=error_Table2;
    error_table(:,i+1)=error_Table;
    %update current state to possible state for next iteration
    current_state=unique(sort(possible_states));
end

%Traceback
%End of block is zero since it's just flush bits
optimum_path=predecessor_states(1,length(predecessor_states));
for count3=length(predecessor_states)-1:-1:2
    path= predecessor_states(optimum_path(1)+1,count3);
    optimum_path=[path optimum_path];
end

%Decode the optimum path
current_state=0;
data=[];
for count4=1:length(optimum_path)
    if(transition_Table(current_state+1,2)==optimum_path(count4))
        data=[data 0];
    else
        data=[data 1];
    end
    current_state=optimum_path(count4);
end
%last bit is 0.(The impact of flush bits)
data=[data 0];

end
```

Published with MATLAB® R2016b

```
function [predecessor_states,error_Table,optimum_path,data] =
 viterbidechard(K,coded)

%Function implements the hard decision viterbi decoding
% K denotes the constraint length and coded is the demodulated binary
% sequence

%Intializing arrays essential for the state decoding process
%The State Transition table: the next state given the current state
 and
%input
transition_Table=[0 0 2;1 0 2;2 1 3;3 1 3];
%The output table:determins the output given the current state and the
%input
output_Table=[0 0 3;1 3 0;2 2 1;3 1 2];
%Error Metric Array-Track of accumulated error
error_Table=zeros(2^(K-1),1);
%Initialize the current state to be always 0
current_state=[0];
%State predecessor history-local optimum at each time instant
predecessor_states=zeros(2^(K-1),length(coded));

for i=1:length(coded)
    %possible states the algorithm might end up in given it's current
 state
    possible_states=transition_Table(current_state+1,2:3);
    %possible outputs the algorithm might end up in given it's current
 state
    possible_outputs=output_Table(current_state+1,2:3);
    possible_states=reshape(possible_states',1,[]);
    possible_outputs=reshape(possible_outputs',1,[]);
    %Distance between received sequence and possible outputs
    ham_distance= sum(abs(dec2bin(coded(i),2)-
dec2bin(possible_outputs,2)),2);
    %A variable to keep track if a state has aleady been updated
 during the loop
    flag=zeros(2^(K-1),1);
    %A duplicate error metric variable to keep track of error in the
 for loop
    error_Table2=nan(2^(K-1),1);
    %loop through current states
    for count=1:length(current_state)
        count2=transition_Table(current_state(count)+1,2:3);
        %update error metric for the possible states
        if(sum(flag(count2+1))==0)
            error_Table2(count2+1,1)=
 error_Table(current_state(count)+1,1)+ham_distance(2*count-1:2*count,1);
            flag(count2+1)=1;
            predecessor_states(count2+1,i)= current_state(count);
        else

 error_temp=error_Table(current_state(count)+1,1)+ham_distance(2*count-1:2*count,1
```

```
 [error_Table2(count2+1,1),ind]=min([error_Table2(count2+1,1)';error_temp']);
            change_index=find(ind==2);
            if(~isempty(ind==2))

 predecessor_states(count2(change_index)+1,i)=current_state(count);
            end


        end

    end
    error_Table2(isnan(error_Table2))=0;
    error_Table=error_Table2;
    %update current state to possible state for next iteration
    current_state=unique(sort(possible_states));
end

%Traceback
%End of block is zero since it's just flush bits
optimum_path=predecessor_states(1,length(coded));
for count3=length(coded)-1:-1:2
    path= predecessor_states(optimum_path(1)+1,count3);
    optimum_path=[path optimum_path];
end

%Decode the optimum path
current_state=0;
data=[];
for count4=1:length(optimum_path)
    if(transition_Table(current_state+1,2)==optimum_path(count4))
        data=[data 0];
    else
        data=[data 1];
    end
    current_state=optimum_path(count4);
end
%last bit is 0.(The impact of flush bits)
data=[data 0];

end
```

Published with MATLAB® R2016b

```
%%
%main.m
clear all
close all
%Convolutional Encoding
K=3;  %Constraint Length
t = poly2trellis(K,[7 5]);
%size of input data msg
%IT SHOULD BE EVEN
N=100000;
%generate random sequence
msg=randi([0,1],1,N);
%Flush bits-Add Two zeros at the end
msg=[msg 0 0];
code = convenc(msg,t);%Encode
transmitted=pskmod(code,2,pi); %BPSK modulation

%%
%AWGN Noise.Comparison with Uncoded vs Convolutional Coding
snr=-2:0;
for i=1:length(snr)
    snr(i)
    noisy_signal=awgn(transmitted,snr(i),'measured');
    demod=pskdemod(noisy_signal,2,pi);
    [~,ber(i)]=biterr(code,demod);
    %Convert message from binary into states for input to viterbi decoding
    transmitted_states=bin_to_states(demod);
    %Decoding using my Viterbi Algorithm
    [~,~,~,decoded_data_soft_l4] = viterbidecsoft(K,lloyd_quantization(real↙
(noisy_signal),4));
    [~,~,~,decoded_data_hard] = viterbidechard(K,transmitted_states);
    [~,~,~,decoded_data_soft_l8] = viterbidecsoft(K,lloyd_quantization(real↙
(noisy_signal),8));
    [~,~,~,decoded_data_soft] = viterbidecsoft(K,real(noisy_signal));

    [~,codedbersoftl4(i)]=biterr(msg,decoded_data_soft_l4);
    [~,codedbersoftl8(i)]=biterr(msg,decoded_data_soft_l8);
    [~,codedbersoft(i)]=biterr(msg,decoded_data_soft);
    [~,codedberhard(i)]=biterr(msg,decoded_data_hard);
end
ebn0=snr-10*log10(1/2);%Rate-1/2 convolutional code
%%
%Plotting
semilogy(snr,ber,'x-.',ebn0,codedbersoftl4,'xk--',ebn0,codedbersoftl8,'xr-',ebn0,↙
codedbersoft,'*g-',ebn0,codedberhard,'om-')
axis([0 7 0.0001 0.1])
xlabel('E_b/N_0 (dB)');
ylabel('BER');
title('BER of rate 1/2 K=3 Convolutional Code')
legend('Uncoded BPSK','Quantized 2-bit','Quantized 3-bit','Soft-Decision','Hard↙
Decision')
```

```
function y = lloyd_quantization(input_sig,num_levels)
    %Compute the optimum Lloyd Quantization values given an input signal
    %Inputs
    %input_sig-Input Signal
    %num_levels-Numberr of Quantization levels
    %Outputs
    %y-Quantized final output
    y=zeros(size(input_sig));
    if(length(input_sig)<10000)
        [~,levels]=lloyds(input_sig,num_levels);
    else
        [~,levels]=lloyds(input_sig(1:1000),num_levels);
    end
    for i=1:length(input_sig)
        [~,ind]=min(abs(input_sig(i)-levels));
        y(i)=levels(ind);
    end
end
```

```
%%Auxiliary function needed for main to run
function msg2= bin_to_states(msg)
msg2=[];
for i=0:(length(msg)/2)-1
    msb=2*i+1;
    lsb=msb+1;
    msg2=[msg2 2*msg(msb)+msg(lsb)];
end
end
```

Published with MATLAB® R2016b