Performance Evaluation of Convolutional Codes : A MATLAB implementation

Surajkumar Harikumar (EE11B075), Manikandan S (EE11B125)

***Abstract*—In this paper, we analyse the performance of a rate 2/3 convolutional code of memory order 6, obtained by puncturing a rate 1/2. We use the soft-decision Viterbi algorithm to decode messages transmitted over noisy channels. By controlling the channel signal-to-noise ratio, we show the bit error rate performance of the code, in comparison to the uncoded bit error rate. We also use the Viterbi algorithm to evaluate the free-distance of the code, the first few terms of the truncated weight enumerating function. We use this to find a union bound on code performance, and compute asymptotic and actual coding gain of the rate 2/3 code.**

***Index Terms*—Convolution Codes, MATLAB, Viterbi Algorithm, Punctured code, Coding Gain, Performance of Code**

I. Introduction

Convolutional codes are a specific class of error-correcting codes with memory. They convert an m-bit input into an n-bit output. Further more, the current information bits also decide the codeword generated by subsequent information bits. Thus, these codes have memory. More specifically, the convolution code can be envisioned using a state diagram, where input bits cause a transition from one state to another.

In this paper, we investigate the performance of a rate 2/3 convolutional code with memory order 6. We create the rate 2/3 code by puncturing a rate 1/2 code, that is by ignoring certain output bits. The code specification is clearly mentioned in subsequent sections.

We use the Viterbi algorithm to evaluate the performance of the convolution code. We add additive-white-gaussian noise to a long codeword, and find the bit-error rates for a given signal-to noise ratio. This is compared with the uncoded wit-error rates of the same code.

We also use a modified version of the algorithm to evaluate the free distance d_{free} of the code, $A_{d_{free}}$, and a few terms of the Weight enumerating function (WEF). We use this to provide a union bound on the performance of the same convolution code, and compare this to the actual performance.

II. Code Specification and Encoding

We are required to design a rate 2/3 convolution code of memory order 6 by puncturing a rate 1/2 code. We obtained the code specifications from Table 12.4 of Lin and Costello textbook [2]. The optimum code of memory order $v = 6$ had $g^{(0)} = 155$ and $g^{(1)} = 117$. Since these were in Octal-representation, we converted these to get the Binary representation and thus the generator polynomial, given by

$$G(D) = [\,1 + D^2 + D^3 + D^5 + D^6,\; 1 + D + D^2 + D^3 + D^6\,] \quad (1)$$

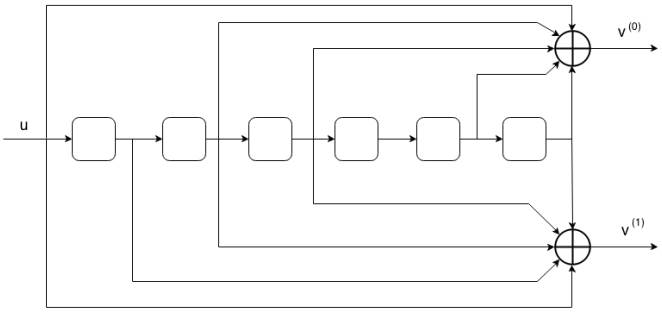

Fig. 1. Systematic feed-forward encoder for the $v = 6$ rate 1/2 convolutional code

We can construct a systematic encoder for this convolution code. The encoder diagram for the same is given in Figure 1. We can easily verify that this has memory order $v = 6$. This encoder diagram specifies a rate 1/2 code, since it takes single input and gives 2 outputs. To make this a rate 2/3 code, we can puncture. Over 2 consecutive information bits, we require 3 codeword bits. So we just ignore one of the codeword bits out of the 4 given by this systematic encoder to get a rate 2/3 code. The table tells us that the optimum choice is to ignore $v^{(1)}$ in every even codeword bit. The code snippet for obtaining the generator polynomial is

```
g0 = de2bi(oct2dec(155));
g1 = de2bi(oct2dec(117));
```

We give a large input size message, and encode it using the convolution code's generator polynomial. For our example, we used an $N = 40,000$ bit message sequence, and encoded using the 2 generator polynomials in 2. This was obtained by using MATLAB's *conv* function on a random sequence of size N. We then take these 2 at a time to find the codeword bits that go into the channel, captured in variable s. The modulation scheme used was BPSK $\{0 \to -1, 1 \to 1\}$

```
ip = rand(1,N)>0.5;
cip1 = mod(conv(double(ip),[ 1 1 0 1 1 0 1 ])
    ,2);
cip2 = mod(conv(double(ip),[ 1 0 0 1 1 1 1 ])
    ,2);
cip = [cip1;cip2];
cip = cip(:).';
s = 2*(cip)-1; % BPSK modulation
```

We then add Additive-White Gaussian Noise to this sequence s, using MATLAB's inbuilt *awgn* function. We iterated over E_b/N_0 in the range of $2 \to 6\ dB$.

III. Trellis Diagram and the Viterbi Decoder

To obtain the code performance, we send the noisy codeword through the Viterbi Decoder. The first step here was to obtain the trellis diagram. As a reference, we used MATLAB's inbuilt poly2trellis function. This returns all the states and outputs leaving a particular state. We flipped this to obtain a matrix of information containing the states entering a particular node in the trellis diagram. Since the Viterbi algorithm is best used with the previous state computation, we generated this. Table I describes completely the transitions from a previous state to current state, and the corresponding input and output bits.

We use a **Soft-Decision Viterbi Decoder**. The Hard-decision decoder classifies the received vector as zero or one (after slicing) and uses these values to compute branch metrics (using Hamming distance). On the other hand, the soft decision decoder uses Euclidian distance as the branch metric.. This means that for each received bit, we compute the deviation from the corresponding branch bit, and find the Root-of-sum-of-square over all bits. The branch metric computation is described below.

$$\begin{aligned} d_i^{(0)} &= r_i^{(0)} - \hat{c}_i^{(0)} \\ d_i^{(1)} &= r_i^{(1)} - \hat{c}_i^{(1)} \\ Br_i &= \sqrt{(d_i^{(0)})^2 + (d_i^{(1)})^2} \end{aligned} \tag{2}$$

The Viterbi decoding algorithm is used with the branch metrics as computed above. There is a weight associated with each node. At each stage, we compute all the branch metrics, and update the node weight with the maximum of the total path metric upto that node from all paths. A small snippet of the code is shown here.

```
for m=1:32
  for m=1:32
  if (M_S(state_diagram(m,1)) + abs(2*
      state_diagram(m,3)-1-cipSoft(2*k-1)) +
            abs(2*state_diagram(m,4)-1-
                                  cipSoft(2*k))
                )<(M_S(state_diagram(m,2)) +
            abs(2*state_diagram(m,5)-1-cipSoft
                (2*k-1)) + abs(2*state_diagram
                (m,6)-1-cipSoft(2*k)))
    phi_S_temp(m,:)= phi_S(state_diagram(m,1)
        ,:);
    phi_S_temp(m,k)=0;
     M_S_temp(m)=M_S(state_diagram(m,1)) +
            abs(2*state_diagram(m,3)-1-cipSoft
                (2*k-1)) +
            abs(2*state_diagram(m,4)-1-cipSoft
                (2*k));
  else
      phi_S_temp(m,:)= phi_S(state_diagram(m
          ,2),:);
      phi_S_temp(m,k)=0;
      M_S_temp(m)=(M_S(state_diagram(m,2)) +
          abs(2*state_diagram(m,5)-1-cipSoft
          (2*k-1)) + abs(2*state_diagram(m,6)
          -1-cipSoft(2*k)));
  end
end end
```

TABLE I
Trellis State Diagram Description

Current State	Previous state for input 0	Previous state for input 1	Output for input 0	Output for input 1
1	1	2	0 0	1 1
2	3	4	0 1	1 0
3	5	6	1 1	0 0
4	7	8	1 0	0 1
5	9	10	1 1	0 0
6	11	12	1 0	0 1
7	13	14	0 0	1 1
8	15	16	0 1	1 0
9	17	18	0 0	1 1
10	19	20	0 1	1 0
11	21	22	1 1	0 0
12	23	24	1 0	0 1
13	25	26	1 1	0 0
14	27	28	1 0	0 1
15	29	30	0 0	1 1
16	31	32	0 1	1 0
17	33	34	1 0	0 1
18	35	36	1 1	0 0
19	37	38	0 1	1 0
20	39	40	0 0	1 1
21	41	42	0 1	1 0
22	43	44	0 0	1 1
23	45	46	1 0	0 1
24	47	48	1 1	0 0
25	49	50	1 0	0 1
26	51	52	1 1	0 0
27	53	54	0 1	1 0
28	55	56	0 0	1 0
29	57	58	0 1	1 0
30	59	60	0 0	1 1
31	61	62	1 0	0 1
32	63	64	1 1	0 0
33	1	2	1 1	0 0
34	3	4	1 0	0 1
35	5	6	0 0	1 1
36	7	8	0 1	1 0
37	9	10	0 0	1 1
38	11	12	0 1	1 0
39	13	14	1 1	0 0
40	15	16	1 0	0 1
41	17	18	1 1	0 0
42	19	20	1 0	0 1
43	21	22	0 0	1 1
44	23	24	0 1	1 0
45	25	26	0 0	1 1
46	27	28	0 1	1 0
47	29	30	1 1	0 0
48	31	32	1 0	0 1
49	33	34	0 1	1 0
50	35	36	0 0	1 1
51	37	38	1 0	0 1
52	39	40	1 1	0 0
53	41	42	1 0	0 1
54	43	44	1 1	0 0
55	45	46	0 1	1 0
56	47	48	0 0	1 1
57	49	50	0 1	1 0
58	51	52	0 0	1 1
59	53	54	1 0	0 1
60	55	56	1 0	0 0
61	57	58	1 0	0 1
62	59	60	1 1	0 0
63	61	62	0 1	1 0
64	63	64	0 0	1 1

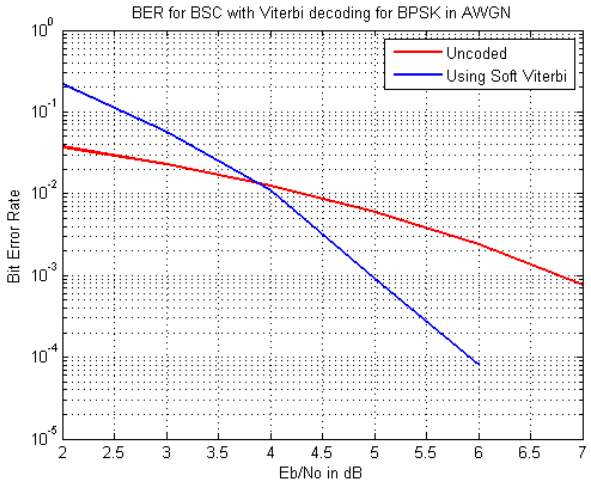

Fig. 2. Comparing the BER code performance of the convolutional code and the uncoded performance

We ran this Viterbi Decoder of a random message sequence of length $N = 40000$, for various values of SNR. We looked at the decoded sequence, and compared it with the input codeword sequence (before noise was added). The number of erroneous bits divided by the total number of bits gives us the **Bit Error Rate** (BER) for the convolutional code. We computed this for varying values of SNR. This was compared against the uncoded BER, which was obtained using the formula

$$\begin{aligned} Uncoded\ BER &= Q\left(\sqrt{\frac{SNR}{2}}\right) \\ &= \frac{1}{2}erfc\left(\sqrt{\frac{SNR}{2}}\right) \\ &= \frac{1}{2}erfc\left(\sqrt{\frac{E_b}{N_0}}\right) \end{aligned} \tag{3}$$

since $SNR = 2 * \frac{E_b}{N_0}$ (we are using BPSK modulation).

We plot this for varying values of $\frac{E_b}{N_0}$. The results are shown in Figure 2. We see that after about 3.6 dB and a Bit error rate of 10^-2, the Soft viterbi decoder performs much better. The coding gain, defined as the difference in E_b/N_0 required to achieve the same SNR, was found to be about 2.7 dB at a BER of 10^{-4}

IV. Calculation of d_{free} and Ad_{free}

The Viterbi algorithm can be modified to calculate the free distance (d_{free}) of a convolutional code. The d_{free} of a convolutional code is defined as the minimum Hamming weight of any non-zero codeword of that code. In order to find that we can use the trellis diagram and start from the all zero state.We will consider paths that diverge from the zero state and merge back to the zero state, with out intermediate passes through the zero state. Since it is not possible to manually find such paths in trellis, we modify our Viterbi algorithm to find such paths.

The idea is very simple. Instead of starting at the zeroth state, we start with state 1 (we assign the path metric as 2 since the transition from state zero to next state has two output bits as 1). We define each branch metric to be weight of the output for the transition through that branch. For example, if there is a transition from state X at time instant t to state Y at time instant $t+1$ and the output of this transition is 11 we assign the branch metric associated with this branch to be 2.

Now comes the idea of Viterbi. Initially assign the weights of all the states to be infinity. Set the state that is connected to zero(here it is 32) to have weight 2. Now run the Viterbi algorithm with branch metric as specified above and take the path with minimum metric to the zero state. The path taken to reach the zero state with minimum metric is the code with the minimum hamming weight

```
Modifying Viterbi algorithm to determine dfree
Bdfree=0;
no_of_paths=1000;
path_metric=Inf(1,64);
path_metric(33)=2;
phi=zeros(64,no_of_paths);
for k=1:no_of_paths
    path_metric_temp=Inf(1,64);
    phi_temp=zeros(64,no_of_paths);
    if(mod(k,2)==1) %odd iteration
        for t=1:64
            if(path_metric(t)+
                state_diagram_weights(t,1)<
                path_metric_temp(state_diagram
                (t,1)+1))
                path_metric_temp(state_diagram
                    (t,1)+1)=path_metric(t)+
                    state_diagram_weights(t,1)
                    ;
                phi_temp(state_diagram(t,1)
                    +1,:)=phi(t,:);
                phi_temp(state_diagram(t,1)+1,
                    k)=state_diagram(t,3);
            end
            if(path_metric(t)+
                state_diagram_weights(t,2)<
                path_metric_temp(state_diagram
                (t,2)+1))
                path_metric_temp(state_diagram
                    (t,2)+1)=path_metric(t)+
                    state_diagram_weights(t,2)
                    ;
                phi_temp(state_diagram(t,2)
                    +1,:)=phi(t,:);
                phi_temp(state_diagram(t,1)+1,
                    k)=state_diagram(t,4);
            end
        end
    else
        for t=1:64
            if(path_metric(t)+
                state_diagram_weights_even(t
                ,1)<path_metric_temp(
                state_diagram(t,1)+1))
                path_metric_temp(state_diagram
                    (t,1)+1)=path_metric(t)+
                    state_diagram_weights_even
```

```
                    (t,1);
                phi_temp(state_diagram(t,1)
                    +1,:)=phi(t,:);
                phi_temp(state_diagram(t,1)+1,
                    k)=state_diagram(t,3);
            end
            if(path_metric(t)+
                state_diagram_weights_even(t
                ,2)<path_metric_temp(
                state_diagram(t,2)+1))
                path_metric_temp(state_diagram
                    (t,2)+1)=path_metric(t)+
                    state_diagram_weights_even
                    (t,2);
                phi_temp(state_diagram(t,2)
                    +1,:)=phi(t,:);
                phi_temp(state_diagram(t,2)+1,
                    k)=state_diagram(t,4);
            end
        end

    end
     path_metric=path_metric_temp;
     phi=phi_temp;
     weight=[weight path_metric(1)];
     path_metric(1)=Inf;
end
weight
disp('Minumum weight');
min(weight)
```

We find that $d_{free} = 6$ and the message associated to be $m(D) = D + D^2 + D^3$. The value of $Ad_{free} = 1$

$$Bd_{free} = 3Ad_{free} = 3$$

V. Finding the first two terms of the BWER

In the previous section we found out the value of d_{free} to be 6 and the value of the Ad_{free} to be 1. The weight of the one input message vector that gives codeword of minimum hamming weight is 3 (Bd_{free}). We find that there exists codewords of hamming weight 7(which is $d_{free} + 1$). We can find out the value of this B_d (i.e the weights of input message vector giving a codeword of hamming weight 7).

$$P_{biterror} \approx Bd_{free}e^{-Rd_{free}E_b/N_0} + Bde^{-RdE_b/N_0}$$

The above formula works for small values of SNR and the value of Bd was found out to be 87 (approximately). Another approach would be to determine all the codewords with weight 7 and found out the input message vector associated with that codeword. The weight of all such codeword multiplied by the Ad corresponding to the codeword with weight 7 gives the value of Bd to be 90.

VI. Union Bound and Coding Gain

The Viterbi decoder gives a very accurate estimate of the performance of the code. However, it takes hours to run and days to get appreciable results during simulation. Thus, we have 2 rough measures on the performance of the convolutional code, Bounds, and Coding Gain.

We used 2 bounds for demonstration, a simpler bound, and a tighter one. The simpler bound we chose was the *Event-Error Probability*, given by

$$P_b(E) < P_b^E(E) = B_{d_{free}} 2^{d_{free}/2} e^{-(Rd_{free}/2)(E_b/N_0)} \quad (4)$$

Since we know $B_{d_{free}} = 3$ and $d_{free} = 6$, we can easily evaluate this bound for a rate $R = 2/3$ code (which this is at th construction level). We easily see from Figure X that is is a fairly loose upper bound, but does track the Convolutional code performance.

The better bound is the *Union Bound*, given by

$$P_b(E) < P_b^U(E) = \sum_{d=d_{free}}^{\infty} B_d e^{-\frac{dRE_b}{N_0}} \quad (5)$$

The most accurate bound would be to sum over all terms in the Bit-Weight-Enumerating-Function. But since we have only the first 2 terms, we create a new bound

$$P_b^{Tr}(E) = B_{d_{free}} e^{-\frac{d_{free}RE_b}{N_0}} + B_{d_{free}+1} e^{-\frac{(d_{free}+1)RE_b}{N_0}} \quad (6)$$

We plot 4 and 6 in Figure 3. We see that the Truncated Union Bound tracks the Convolution code very well, to about 1 order of error magnitude difference.

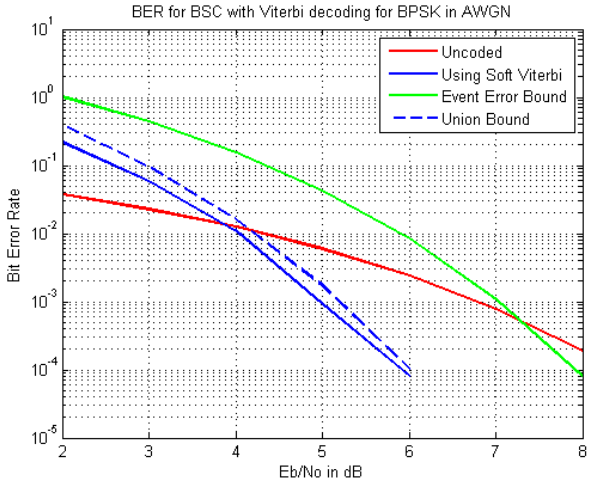

Fig. 3. Comparing the BER code performance of the convolutional code and the uncoded performance

The other measure of code performance is *Coding Gain*. The asymptotic coding gain of a soft-decision covolutional code is given by

$$\gamma = 10\, log_{10}(Rd_{free})\ dB = 6\ dB \quad (7)$$

Using $R = 2/3$ and $d_{free} = 6$, we get the Asymptotic coding gain as 6 , while our actual measured coding gain at $BER = 10^{-4}$ is around 3 dB. At larger values of BER, our coding gain will increase. This concludes the treatment of convolutional code performance.

References

[1] Todd. K. Moon, *Error Correction Coding*.
[2] Shu Lin, Daniel J. Costello, *Error Control Coding-Fundamentals and Applications, 2ed*
[3] http://en.wikipedia.org/

d