Project Report on Convolutional Codes, Trellis Structure and Viterbi Decoding

Yue Fei 1003944146

I. Introduction

This project is a brief investigation on the practical implementations of binary linear convolutional codes, which are useful in the power-limited (lower SNR and low distortion) region [1]. Prior to Turbo code, convolutional codes concatenated with hard-decision Reed-Solomon codes are the most efficient and approaching closest to Shannon's limit [**?**]. Thus, it was largely used in the deep space applications and in wireless communication systems, namely Physical Broadcast Channel (PBCH) and the Physical Downlink Control Channel (PDCCH) for the Long Term Evolution (LTE) system [2, 3]. In this project, the emphasis is on rate $1/2$ binary linear time-invariant convolutional codes, which is the simplest family of convolutional code to understand and construct.

II. Preliminary Concepts

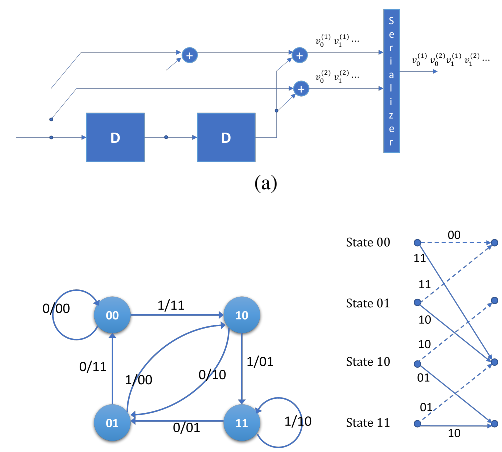
There are fundamental structures to define convolutional codes: 1) algebraic structure, arising from the fact that convolutional encoders are linear systems; 2) dynamic structure, treating convolutional encoders as finite-state systems. The finite-state systems are the foundations to apply Viterbi algorithm [1]. Therefore, in this work we will focus on the second interpretation by representing the convolutional code as a finite-state machine.

Definition A convolutional code is characterized by three integers, n, k, K. K denotes the constraint length, defined as $K = m + 1$ [4], where m is the maximum number of stages (memory size) in any shift register. Note that in some literature [1, 5], constraint length v_i for the ith input of a polynomial convolutional generator matrix G is also defined to be controllable indices,

$$v_i = \max_{1 \leq j \leq c} \{\deg g_{ij}(D)\},$$

which is equivalent to the number of delay elements (shift registers). Here we consider the first definition. The constraint length typically ranges between 3 and 9 and the path memory of the Viterbi decoder is usually a few constraint lengths [4, 6]. k denotes the number of input bits. n denotes the number of output bits. The performance of a convolutional code depends on the coding rate and the constraint length. Longer constraint length K means more coding gain, since each output bit will be influenced by a larger number of message bits. For convolutional codes, the output bits are parity bits, and only parity bits are sent over channels. Thus, a larger constraint length generally implies a greater resilience to bit errors [7].

Code Representation A convolutional encoder can be represented in three graphical forms, including state transition diagram, tree diagram, and trellis diagram. For the purpose of encoding with trellis structure and decoding with Viterbi algorithm, state transition diagram and trellis diagram are sufficient. Figure. 1(a) shows the schematic of a convolutional encoder, with $k = 1$, $n = 2$, and $K = 3$. The D blocks can be considered as the delay elements in LTI (linear time invariant) systems, or equivalently the shift registers in circuit design. The corresponding Mealy-style finite state machine and trellis diagram are shown in Fig. 1(b) and Fig. 1(c) respectively.



(a)

(b) (c)

Fig. 1: (a) An encoder for a binary rate R = 1/2 convolutional code. (b) Mealy finite state machine for binary rate R = 1/2 convolutional code. The edges are labeled as *[input-bit]/[output-bits]*. (c) Trellis structure for R = 1/2 convolutional code.

Code Property One single most important parameter for determining the error-correcting capability of the code is the *free distance* denoted as d_{free}. The most likely error event is that the transmitted codeword is changed by the BSC (binary symmetric channel) so that it is decoded as its closest (in Hamming distance) neighbor. This minimum distance is called the *free distance* of the convolutional code [5, 7]. For conventional non-tail-biting convolutional code, the registers are initialized with 0, and sequences of 0 are fed in the end to signify the end of codeword transmission. In addition, convolutional codes are linear codes (can be proved using Laurent series [1]). The smallest Hamming distance is also the minimum Hamming weight. Specifically, we can compute the free distance of a convolutional code through computing the difference in *path metrics* between the all-zeroes output and the path with the smallest non-zero path metric going from the initial 00 state to some future 00 state. The *path metric* is value associated with each state (node) in the trellis. At each timestamp in the sequential decoding process, it is updated to be the smallest accumulated sum amongst all the incoming edges of the state (node).

From the state transition diagram shown in Fig. 1(b), we can easily compute that the minimum path metric is 5. The path $00 \rightarrow 10 \rightarrow 01 \rightarrow 00$. The accumulated Hamming distances are computed from output bits on the arcs $11 \rightarrow 10 \rightarrow\rightarrow 11$. In total, it takes 6 outputs to get back the initial state. Thus, it is able to correct up $\lfloor(5-1)/2\rfloor = 2$ bits error in 6 bits.

III. Experiment

A. Trellis Generation

Given Rate $1/2$ convolutional code is the most commonly used in practice, the trellis generation module only supports the trellis generation for such rate, but user defined constraint length.

The inputs are K (constraint length) and the coefficients of two polynomials, g_1 and g_2 corresponding the generator polynomials for output parity *bit* 0 and *bit* 1. Since the code is binary, coefficient 1 indicates the connection to the processing stage, and coefficient 0 indicates otherwise.

The output is a *trellis* structure containing the same fields as the one defined in MATLAB. Its validity can be validified using the built-in toolbox `istrellis`. Two of the most fields are *next state* table and *output* table. For both tables, the number of rows is equal to $2^{(K-1)}$. When $K=3$, there are $2^{(3-1)} = 4$ states. There are only two columns, each for one possible input symbol: 0 or 1. For the example demonstrate in Fig. 1, the next-state table and output table look as follows:

Next-State Table

	Next State if	
Current State	Input = 0	Input = 1
00	00	10
01	00	10
10	01	11
11	01	11

(a)

Output Table

	Output Symbols if	
Current State	Input = 0	Input = 1
00	00	11
01	11	00
10	10	01
11	01	10

(b)

Fig. 2: (a) Next state table; (b) Output table.

The algorithm in Appendix A is essentially performing XOR operations on state values and current inputs [8].

B. Convolutional Encoder

The convolutional encoder is trivial to implement using *trellis*. Iterating over each message bit, we compute the current output through indexing into the *Output* table with *current state* as row index and current message bit as the column index. Then the *current state* is updated through the *Next state* table with the same set of indices as above. Detailed code is included in Appendix B.

C. Viterbi Decoder

In this project, we applied **Hard-decision decoding**, which means a quantizer is applied before the decoder. An early decision regarding whether a "0" or "1" is made by comparing to a threshold voltage. It throws away information in this "demapping" (digitizing) process, especially when the "analog" value is close to threshold.

There are two metrics involved: the **branch metric** (BM) and the **path metric** (PM). The branch metric is instantaneous measure of the Hamming distance computed at each time stamp. It computes the "distance" between what was transmitted and what was received, and is defined for each arc in the trellis. The path metric is an accumulated measure of Hamming distance between the expected bit sequence and the actual received stream from the initial state (time $t=0$) to the current time t_i. The path metric is associated with the states in the trellis. The dimension of the path metric is then *Num. of States* $\times$ *Num. of Input Symbols*= $2^{(K-1)} \times 2$.

For the implementation details, there are *seven* data structures required in total [9].

1. A copy of the convolutional encoder `next state` table. The dimensions are $2^{(K-1)} \times 2^k$.
2. A copy of the convolutional encoder `output` table. The dimensions are $2^{(K-1)} \times 2^k$.
3. An `input` table. The row index is the `current state`, and the column index is the `next state`. The entry gives what input value 0 or 1 would produce the next state, given the current state. The dimensions are $2^{(K-1)} \times 2^{(K-1)}$.
4. An array called `state history` table to hold the predecessor state of the current optimal state. By optimal, we mean the state that corresponds the least accumulated path metric. The dimensions of this array are $2^{(K-1)}x(Kx5+1)$. The column dimension is termed as *traceback depth* for practical chip design. For our simulation purpose only, we can assume this *traceback depth* to be equal to $L/2+1$, which is half of the input sequence length.
5. An array to store the path metric or *accumulated error metrics* (`ACM`). The dimensions of this array are $2^{(K-1)} \times 2$. Note that since states can be transitioned from multiple predecessor states. The error metric is only updated if the new path metric is smaller. Thus, one helper array is needed to hold the *best* path metric from the previous time stamp (dimensions $= 2^{(K-1)} \times 1$)
6. An array to store a list of states determined during traceback stage. The dimensions are $K \times 5 + 1$, i.e. $L/2+1$. The last and the initial states are both 0 by construction. `state history`$[L/2+1]$ contains the optimal predecessor state with up-to-date minimum path metric. Then, we use the state value in `state history`$[L/2+1]$ to obtain the second last optimal predecessor value from entry `state history`$[L/2]$, and so on.

A detailed path metric update process is show in Fig.3.

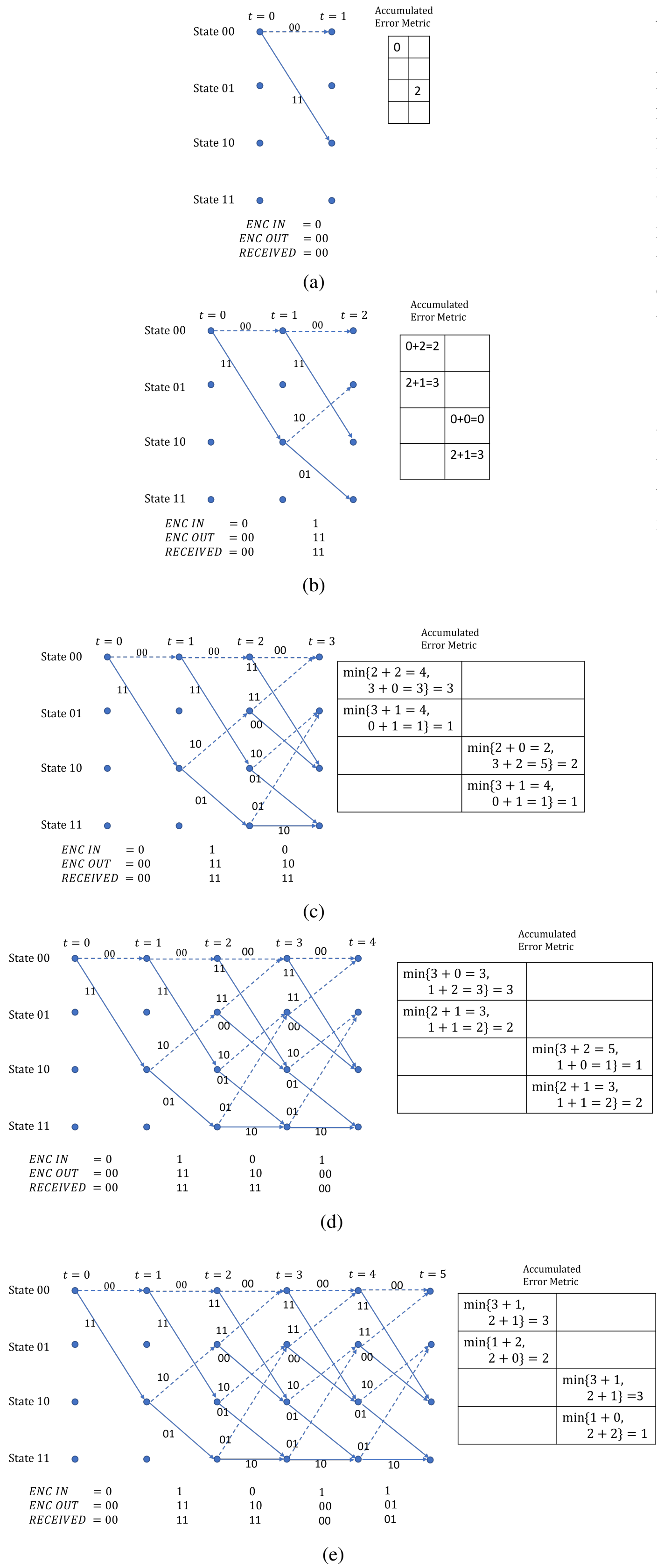


Fig. 3: Decoder Trellis status and the corresponding path metric table at each time stamp from $t = 0$ to $t = 5$.

D. BSC Channel Simulation

To evaluate the correctness of the decoder, a memoryless BSC channel with crossing probability $p = 0.2$ is implemented. The message sequences are randomly generated. The message bit length is 10 and the last two bits forced to be 00 to indicate the end of current transmission. Through simulation, we can observe that when the number of bit errors present is 3, if the errors are scattered in the codeword, then the decoder has the chance to correct the errors. However, if the bit-flippings occur as burst errors, then the decoder will fail to correct all the errors. The demo code is included in Appendix D

IV. Conclusion

The implementation in this project is an initial stab at the convolutional code. A number of further improvements worth investigating in depth, such as *soft-decision* decoding, tail-biting trellises and the performance comparison between iterative and list decoding of convolutional codes.

References

[1] G. D. Forney, "Chapter 9 - introduction to convolutional codes," in *MIT6.451 Lecture Notes*, 2003.

[2] R. A. Baby, "Convolution coding and applications: A performance analysis under awgn channel," in *2015 International Conference on Communication Networks (ICCN)*, 2015, pp. 84–88.

[3] M. H. Omar, A. El-Mahmoudy, K. G. Seddik, and A. Elezabi, "On the tail-biting convolutional code decoder for the lte and lte-a standards'," in *2013 Asilomar Conference on Signals, Systems and Computers*, 2013, pp. 510–514.

[4] F. Huang, "Chapter 2 convolutional codes," 1997.

[5] R. Johannesson and K. S. Zigangirov, *Fundamentals of Convolutional Coding*. Wiley-IEEE Press, 1999.

[6] A. Grami, "Chapter 10-error-control coding," in *Introduction to Digital Communications*, A. Grami, Ed. Boston: Academic Press, 2016, pp. 409–455.

[7] "6.02 convolutional coding." MIT Staff, 2010.

[8] A. Muehlfeld, "Matlab central file exchange," Natick, Massachusetts, United States, 2008. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/22427-viterbi-trellis-generator

[9] A. Gerstlauer, "Viterbi decoder," in *EE 382V - System-on-a-Chip (SoC) Design*, 2009.

Appendix

A. Trellis Generation

This section includes MATLAB Code for trellis generation for rate $\frac{1}{2}$ with arbitrary constraint length $K \geq 3$.

```
function trellis = create_trellis(K, g1_taps, g2_taps)
% Create trellis struct for Rate 1/2 Convolutional Codes
% Inputs:
%       K = constraint length => Num. of Shift Registers = K - 1
%       g1_taps: Indices of taps => Output C0
%       g2_taps: Indices of taps => Output C1
% Outputs:
%       trellis:={
%                 numInputSymbols     = 2^1
%                 numOutputSymbols    = 2^2
%                 numStates           = 2^(K-1)
%                 nextStates          = [numStates x numInputSymbols] matrix
%                 outputs             = [numStates x numInputSymbols] matrix
%                 }
%
if nargin < 3
    % input is always a tap for both outputs
    K = 3;
    g1_taps =[1 2];
    g2_taps= [2];

end
nIBit = 1; nOBit = 2; numInputSym = 2^nIBit; numOutputSym = 2^nOBit;
num_regs = K-1; num_states = 2^num_regs;


% Next State Table
next_state = zeros(num_states,numInputSym); %0,1,2,... num_states-1

% Output Table
g = zeros(num_states,numInputSym);

% Loop through each state
% => generate the corresponding next states + outputs
for i=1:num_states
    %% Initialize Shift Register
    state_dec = i-1; % decimal
    % convert state into tap-line
    state_bin = decimalToBinaryVector(state_dec,num_regs);

    %% Determine Next States
    % Next states for input 0
    next_state_in0_bin = [0 state_bin(1:K-2)];
    next_state_in0_dec = binaryVectorToDecimal(next_state_in0_bin);

    % Next states for input 1
    next_state_in1_bin = [1 state_bin(1:K-2)];
    next_state_in1_dec = binaryVectorToDecimal(next_state_in1_bin);

    next_state(i,:) = [next_state_in0_dec next_state_in1_dec];

    %% Determine Outputs
    % Conversion to facilitate XOR:
    %       binary 0 -> decimal 1
    %       binary 1 -> decimal -1
    state_bin = (-2*state_bin) + 1;

    g1_in0 = prod(state_bin(g1_taps)); % depends on the evenness of -1
    g1_in1 = g1_in0 * (-1);

    g2_in0 = prod(state_bin(g2_taps));
    g2_in1 = g2_in0 * (-1);

    % Convert back to binary 0, 1 notation
    g1_in0 = (-g1_in0 +1)/2;
    g1_in1 = (-g1_in1 +1)/2;
    g2_in0 = (-g2_in0 +1)/2;
    g2_in1 = (-g2_in1 +1)/2;

    % Store as decimal number
    y_in0 = binaryVectorToDecimal([g1_in0 g2_in0]);
    y_in1 = binaryVectorToDecimal([g1_in1 g2_in1]);

    % Store
    g(i,:) = [y_in0 y_in1];
end

% Create trellis struct
trellis = struct('numInputSymbols',numInputSym,'numOutputSymbols',numOutputSym,...
```

```
    'numStates',num_states,'nextStates',next_state,...
    'outputs',g);

%Check if a valid trellis was created
    if(istrellis(trellis) == 1)
        disp('Trellis generated successfully');
    else
        disp('Trellis generation failed');
    end
end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Some good convolutional codes [2]
% Constraint    |              |                |
% length        |      G1      |       G2       |
% ---------------------------------------------------------
%  3            |  110         |  111     |
%  4            |  1101        |  1110    |
%  5            |  11010       |  11101   |
%  6            |  110101      |  111011  |
%  7            |  110101      |  110101  |
%
%
%
% _Reference:_
% [1]   Andrew Muehlfeld (2023). Viterbi Trellis Generator (https://www.mathworks.com/matlabcentral/fileexchange/22427-viterbi-trellis-generator),
%       MATLAB Central File Exchange. Retrieved April 26, 2023.
%
% [2] J. Bussgang, "Some properties of binary convolutional code generators,"
% in IEEE Transactions on Information Theory,
% vol. 11, no. 1, pp. 90-100, January 1965, doi: 10.1109/TIT.1965.1053723.
```

B. Convolutional Code Encoder for Binary Message

The following code is analogous to the builtin functdion `convenc(msg,trellis)` in MATLAB Communication toolbox .

```
function encoded = conv_enc_trellis(msg, trellis)
% ==========================================
% Inputs:
%   1. msg = binary vector
%   2. trellis = rate 1/2 convolutional code  trellis
% Outputs:
%   encoded = codewords
% ==========================================
L = length(msg);
stateTable = trellis.nextStates;
outTable = trellis.outputs;
encoded = zeros(1,2*L);

curr_state = 0;
% Encode msg bit by bit
for i=1:L
    inSym = msg(i)+1;
    next_state = stateTable(curr_state+1, inSym); %+1 for index
    curr_output = outTable(curr_state+1, inSym);
    curr_state = next_state;
    encoded((2*i-1):(2*i)) = decimalToBinaryVector(curr_output,2); %output: MSB->LSB
end

if isequal(encoded,convenc(msg,trellis))
    disp('Code generated successfully');
else
    disp('Code Generation Failure');
end
end
```

C. Viterbi Decoder for Binary Message

The following code is analogous to the built The following code is a naive implementation of Viterbi Decoder assuming infinite traceback buffer depth. The inputs are a truncated version of the builtin function: `decoded-out=vitdec(codedin,trellis)` in MATLAB Communication toolbox .

```
function decoded = viterbi_dec(codedin, trellis)
% % Rate 1/2 Viterbi decoder
```

```
% % Received msg=> BPSK modulated: -1 +1
% % Hard Decision Metric: Hamming distance
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%-------
%              Some good convolutional codes [2]
% Constraint     |                    |               |
% length         |        G1          |       G2      |
% ---------------------------------------------------------
%   3            |   110              |   111         |
%   4            |   1101             |   1110        |
%   5            |   11010            |   11101       |
%   6            |   110101           |   111011      |
%   7            |   110101           |   110101      |
% trellis3=create_trellis(3,[1 2], [2]);
% trellis4=create_trellis(4,[1 3],    [1 2]);
% trellis5=create_trellis(5,[1 3],    [1 2 4]);
% trellis6=create_trellis(6,[1 3 5],  [1 2 4 5]);
% trellis7=create_trellis(7,[1 3 5],  [1 3 5]);

L = length(codedin); M = L/2;
numStates = trellis.numStates;
K = log2(numStates) + 1; %constraint length
numReg = K-1;
k=1; numSym = 2^k;

nextStateT = trellis.nextStates; % nextstate = T1(curr_state,curr_input)
outputT = trellis.outputs;        % output = T2(curr_state,curr_input)
inputT = NaN(numStates,2^k);  % input = T3(curr_state,next_state)

%-----------------
% Construct output binvec table:
%---------------


%-----------------
% Construct input table:
%   binary 0 -> 0
%   binary 1 -> 1
%   impossible transition -> nan
%%%%%%%%%%-------
for cS=1:numStates
    for n=1:numSym
        nS = nextStateT(cS,n);
        inputT(cS,nS+1) = n-1;
    end
end

% Quantizer
codedin = real(codedin) > 0;
% Decoder Parameter
state_history = zeros(numStates,M);
accum_err = zeros(numStates,2);
curr_cost = zeros(numStates);
bkkeeper = zeros(numStates,2);
for t=1:M
    rbits = codedin(2*t-1:2*t);
    %% ===== Decode First two bits =====
    % when t= 0 or t=1, we dont need to update state_history
    if t==1
        initS = 0;
        currO = decimalToBinaryVector(outputT(initS+1,:));
        hammingDist = sum(xor(rbits,currO),2);
        nxtS = nextStateT(initS+1,:);
        accum_err(nxtS(1)+1,1) = hammingDist(1);
        accum_err(nxtS(2)+1,2) = hammingDist(2);
        continue;
    end
    if t==2
        %only two possible states:0 0 and 1 0
        possibleS = [0 nextStateT(1,2)];
        prev_accum_err = accum_err;
        for g=1:2
            currS = possibleS(g);
            currO = decimalToBinaryVector(outputT(currS+1,:));
            hammingDist = sum(xor(rbits,currO),2);
            nxtS = nextStateT(currS+1,:);
            tmp1 = prev_accum_err(currS+1,g) + hammingDist(1);
            accum_err(nxtS(1)+1,1) = tmp1;
            tmp2 = prev_accum_err(currS+1,g) + hammingDist(2);
```

```
                accum_err(nxtS(2)+1,2) = tmp2;

                state_history(nxtS(1)+1,t) = currS;
                state_history(nxtS(2)+1,t) = currS;
            end
            curr_cost = sum(accum_err,2);
            continue;
        end

        for d=1:numStates
            currO = decimalToBinaryVector(outputT(d,:));
            hammingDist = sum(xor(rbits,currO),2);
            nxtS = nextStateT(d,:);
            tmp1 = curr_cost(d) + hammingDist(1);
            if (bkkeeper(nxtS(1)+1,1) == 0)|| (bkkeeper(nxtS(1)+1,1) == 1 && (tmp1 < accum_err(nxtS(1)+1,1)))
                accum_err(nxtS(1)+1,1) = tmp1;
                bkkeeper(nxtS(1)+1,1) = 1;
                state_history(nxtS(1)+1,t) = d-1;
            end

            tmp2 = curr_cost(d) + hammingDist(2);
            if (bkkeeper(nxtS(2)+1,2) == 0) || (bkkeeper(nxtS(2)+1,2) == 1 && (tmp2 < accum_err(nxtS(2)+1,2)))
                accum_err(nxtS(2)+1,2) = tmp2;
                bkkeeper(nxtS(2)+1,2) = 1;
                state_history(nxtS(2)+1,t) = d-1;
            end
        end
        % reset
        bkkeeper = zeros(numStates,2);
        curr_cost = sum(accum_err,2);
end

%% Now we use state_history to trace back
[min_dist,opt_state] = min(curr_cost);
selected_states = zeros(1,M);
selected_states(M) = state_history(opt_state,M);
val = selected_states(M);
for i=1:M-1
    selected_states(M-i) = state_history(val+1,M-i);
    val = selected_states(M-i);
end

decoded = zeros(1,M);
for m=1:M-1
    curr_input = inputT(selected_states(m)+1,selected_states(m+1)+1);
    decoded(m) = curr_input;
end


end
```

D. Demo on Memoryless BSC Channel Simulation

```
% function decoded = viterbi_dec(codedin,trellis)
% % Rate 1/2 viterbi decoder
% % Received msg=> BPSK modulated: -1 +1
% % metric: Euclidean distance
clc;clear all; close all;
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           Some good convolutional codes [2]
% Constraint     |                    |                    |
% length         |        G1          |        G2          |
% ------------------------------------------------------------------
%   3            |   110              |   111              |
%   4            |   1101             |   1110             |
%   5            |   11010            |   11101            |
%   6            |   110101           |   111011           |
%   7            |   110101           |   110101           |

nbits=10;
msg = [rand(1,nbits-2)>0.5 0 0];
trellis3=create_trellis(3,[1 2],     [2]);
trellis4=create_trellis(4,[1 3],     [1 2]);
trellis5=create_trellis(5,[1 3],     [1 2 4]);
trellis6=create_trellis(6,[1 3 5],   [1 2 4 5]);
```

```
trellis7=create_trellis(7,[1 3 5], [1 3 5]);
p = 0.2; nTrials=100;

encoded = conv_enc_trellis(msg,trellis3);
signal = 2*encoded - 1; % BPSK modulation

perr = 0;
for iTrial=1:nTrials
    noise = 2.*(rand(size(signal)) < p); % AWGN ~ CN(0,1)
    y_channel = signal + noise; % channel output

    codedin = y_channel;
    decodedout = viterbi_dec(codedin, trellis3);
perr = perr + size(find(msg - decodedout(1:nbits)),2) / nbits;
end

BER = perr/nTrials;
fprintf("Bit error rate (BER) is: %f\n", BER);
```