Minimizing the Gap-to-Capacity of a Rate 1/3 Code via Convolutional Encoding/Decoding

Romeil Sandhu
rsandhu@gatech.edu

Abstract

In this project, we seek to minimize the gap-to-capacity (given by Shannon's theoretical limit) of a rate 1/3 code. This is done via a convolutional encoder/decoder for varying memory elements as well for both soft and hard decoding scheme. We show that the gap-to-capacity can be minimized with respect to the suboptimal un-coded code word or a (3,1) repetition code. Although better schemes are available such as LDPC and turbo codes, we have chosen the convolutional code for its simplicity and generality. That is, a generic framework can be readily developed for which multiple convolutional schemes can be implemented with minimal changes to the overall structure (see Appendix A for MATLAB code). In this paper, we present the basic concepts associated with convolution codes, specific encoding and decoding schemes used in this project, and results comparing the gap-to-capacity of the algorithm implemented with respect to Shannon's optimal code.

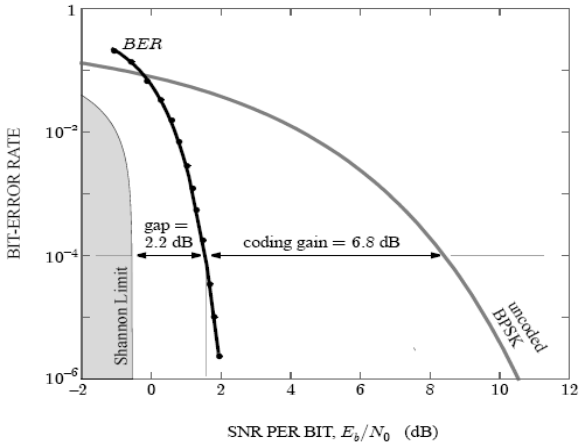

Figure 1. A simulated BER (log scale) versus E_b/N_o (in dB) curve highlighting both the gap to capacity with respect to Shannon Limit curve of a 1/3 system as well as the coding gain with regards to an un-coded code

1. Introduction

Given the code rate contraint of $R = 1/3$ for a binary-input additive white gaussian noise (AWGN) channel, this paper presents several convolutional encoder and decoders of varying element sizes in effort to minimize the gap to capacity of a code with respect to the Shannon limit of any $R = 1/3$ system. This can be seen in Figure 1, where we have also plotted the bit-error rate of an un-coded word. We begin by recalling the model for a binary-input AWGN channel, which is given below as

$$r_l = a_l + n_l \tag{1}$$

where $a_l \in \{-1, 1\}$ is the l-th transmitted symbol, and r_l is the l-th measured or received symbol corrupted by *i.i.d* zero mean Gaussian noise n_l with variance $N_0/2$. Although it is a simple approximation, the AWGN channel presented in Equation (1) has been a powerful instrument in modeling real-life disturbances caused from ambient heat in the transmiiter/receiver hardware and propagation medium. For example, many satellite channels and line-of-sight terrestrial channels can be accurately modeled as an AWGN channel. In this work, we propose to use a 1/n, more specifically, a 1/3 convolution encoder/decoder, to mitigate the disturbance resulting from such a channel. However, before doing so, let us revisit some of classical coding techniques presented in this class, and motivate our reasoning for choosing a convolutional code.

In classical coding theory, Hamming codes, Reed-Muller codes, and Reed-Solomon codes have been popular choices in implementing efficient and reliable coding schemes. However, in this present work, these codes in a standalone fashion can not be directly applied to the problem at hand. For example, there exist no Hamming codes that can produce a binary rate 1/3 code. Similarly, noting that our code length is unconstrained with binary input/output, Reed-Muller codes and Reed-Solomon codes are not well suited. In other words, we seek a coding algorithm that can perform with limited-power. However, if one were to concatenate coding algorithms together (e.g., a (7,4) Hamming Code with a (12,7) Reed-Solomon code), one could

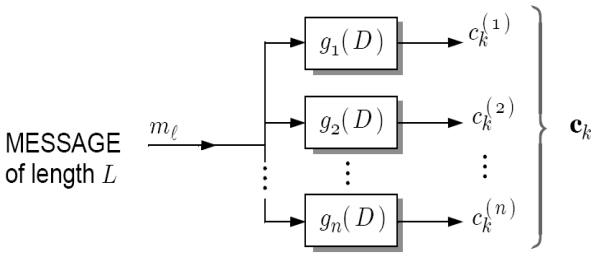

Figure 2. The general form a 1/n convolution encoder.

add flexibility given the constraints of the problem. To this end, we propose to use only a 1/3 convolution code (although one could use a 1/2 convolution code paired with a (3,2) Reed-Solomon code). Lastly, we note that recent work of LDPC code, turbo code, and repeat-accumulate code will offer a better performance gain than the algorithm presented here, but given the limited time of the project, the 1/3 convolution code was chosen.

The remainder of this paper is organized as follows: In the next section, we begin with a review of convolution codes detailing the 1/3 convolution encoder/decoder for a given constraint length. Numerical implementation details are given in Section 3. In Section 4, we present the Bit-Error Rate (BER) rates of our convolution code, with respect to Shannon's limit and the un-coded word, for varying memory element sizes. Finally, we conclude with Section 5

2. Optimal 1/3 Convolution Codes

Binary linear convolution codes, like that of binary linear block codes are useful in the power-limited regime. The general form of a 1/n convolution encoder is given in Figure 2. Here, we see that the encoder is a LTI filter with banks $g_i(D)$ that are both rational and causal. Moreover, the message $m = [m_1, m_2, ..., m_L]$ of length L is passed in bit-by-bit producing n code words c_k^n. From this, we can then form the encoded message as $c_k = [c_1^1, c_1^2, ..., c_1^n, c_2^1, , ..., c_2^n, c_k^1, ..., c_k^n]$. Moreover, if one forms the "Delay Transform" of an impulse response $g_i(D)$, the n-th code word can then be formed as $c_k^n(D) = m(D)g_i(D)$. Together, all possible solutions of an message code forms what is known as a convolution code. Although the formulation of a convolution code assumes a message to be theoretically infinite (as well as the space of acceptable codes), we define block codes of length L.

With this, an encoder can be viewed as a generator matrix G(x) of a linear block code, and hence, multiple encoding schemes can be designed to achieve a rate 1/3 system. In addition, each encoding scheme can contain μ memory elements, adding versatility to design of a particular convolution code. To this end, we seek to implement an optimal

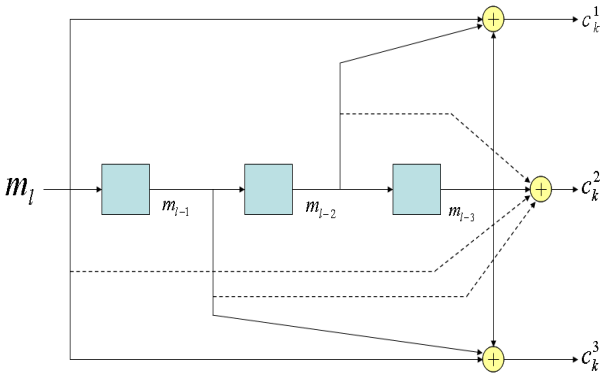

Figure 3. The Optimal Rate 1/3 Convolution Encoder for K = 4.

convolution code of different μ sizes. This is discussed next.

2.1. Encoding Scheme

Using the Table 11-4 presented in [2], we can choose an optimal encoding scheme, dependent on the constraint length $K = 1 + \mu$ for a rate 1/3 convolution. For convenience, we recall three optimal filters with $K = 4, 6, 8$ given below.

K	g_0	g_1	g_2	d_{free}
4	54 101100	64 110100	74 111100	10
6	47 100111	53 101011	75 111101	13
8	452 100101010	662 110110010	756 111101110	16

Table 1. Rate 1/3 convolution codes with minimum distance

Table 1 above shows the optimal filter design for each code generator, where the response is given in octal and binary representation. With this, we can realize the actual encoder via a circuit diagram. This is given in Figure 3 for $K = 4$. Note, for the MATLAB implementation presented in Appendix A, the use of the function `conv.m` is used to do the encoding. After one encodes the message m into a code word c, it is then passed through the channel model given by Equation (1). We then need to be able to recover or decode the code word corrupted by noise.

2.2. Decoding Scheme

We assume now that the code word has been passed through the channel, and now we must decode the (possibly) corrupted message. One popular technique is the Viterbi algorithm, in which one can map the possible solutions to what is known as a trellis map. For the sake of

brevity, we refer the reader for information about how to construct a trellis map [2]. However, we note that through this map, the decoder is able to choose the maximum likelihood (ML) estimate by labeling the nodes with a value denoting the partial branch metric. Then, we seek to find a path with total minimum cost. That is, the decoding scheme can re-expressed as

$$\min_{c \in \mathcal{C}} d_{\mathcal{H}}(r, c) = \sum_{l=0}^{L+\mu} d_H(r_l, c_l) \tag{2}$$

It is important to note that we have yet to define the metric $d_{\mathcal{H}}(.,.)$ in Equation (2). Depending on the chosen metric, one can produce a sub-optimal decoder by choosing the metric to be the Hamming distance. In contrast, if one chooses the L_p norm, specifically the L_2 norm, one can achieve an optimal soft-decoder. Lastly, we also refer the reader to advancement of other chosen metrics that have arisen in prediction theory and have found uses in fields such as computer vision [1]

2.2.1 Sub-optimal Decoder: Hard Decoding

As previously noted, the chosen partial branch metric, $d_{\mathcal{H}}(.,.)$, is crucial for the decoder. In particular, let us denote $\hat{r}_l = [sgn(r_1^l), sgn(r_2^l), ..., sgn(r_k^l)]$, where $sgn(.)$ outputs the sign of value. With our newly formed estimate $\hat{r}_l$, we can then define the partial branch metric using the Hamming distance. This is given as

$$d_H(r_l, c_l) = |\{i | \hat{r}_i^l \neq c_i^l, i = 0, 1, ..., k\}| \tag{3}$$

Given that we first formed the estimate $\hat{r}$ by making "hard" decisions of the received vector r, we denote this procedure as hard decoding.

2.2.2 Optimal Decoder: Soft Decoding

One major drawback of making "hard" decisions in forming the estimate $\hat{r}$, as seen in Section 2.2.1, is a loss of information of the received vector. Instead, if we deal with the received vector r directly, we can then begin to form a measure of similarity via the L_p norm. That is, if define the partial branch metric to be

$$d_H(r_l, c_l) = \big(\sum_{i=1}^{k} |r_i^l - c_i^l|^p \big)^{\frac{1}{p}} \tag{4}$$

where p is chosen to be $p = 2$ or the Euclidean distance, then onen arrives at the optimal soft decoding scheme using the square of the Euclidean distance.

3. Implementation

We have used MATLAB to perform the convolution encoder/decoder algorithm presented in this report. More importantly, we should note that because of the exponential increase in complexity with regards to the number of memory elements μ used and unoptimized MATLAB code, a major drawback is the computational speed. However, from previous experiences that involves a search based type of algorithm, one could invoke a "kd-tree" to perform fast searches.

We also note the generality of the framework and refer the reader to the documented version of the MATLAB code used to implement the convolutional encoder/decoder. This can be found at the end of this report. In particular, the code is written for $K = 4$; however, one can easily change it to incorporate encoders (e.g., $K = 6$ or $K = 8$). These changes will be denoted by a red box.

4. Experiments

We test the robustness of the rate 1/3 convolution code for memory element sizes of $\mu = 3, 5, 7$. Specifically, we measure the coding efficiency of each respective convolution code over 10,000 trials and assume that our message is of $L = 100$ bits. Moreover, this simulation is done over several SNR levels. Although one would ideally like to reach the theoretical coding gain given by Shannon's limit, we deem the "success" of encoder/decoder if it is able to achieve roughly 4 dB using a hard decoding scheme. This base line can then be improved by substituting various branch metrics, such as the L_2 norm. To this end, we present simulation results of the algorithm for both hard and soft decoding, and refer the reader to Appendix A for information of how to switch between the two by trivial changes to the MATLAB code.

We begin with $K = 4$ convolution code (see Figure 3). In Figure 4a, we present the BER simulated over a series trials along with the the Shannon's theoretical limit and the un-coded BPSK algorithm. Figure 4b, shows a zoomed in plot of the value located on the simulated curve at BER = 10^{-4}. The coding gain and gap to capacity at this BER level are 2.242 dB and 6.951 dB, respectively. Using Table 1, we see that the theoretical gain for a hard decoding scheme is $10 \log_{10}(\frac{R * d_{min}}{2}) = 2.1285$ dB, which falls near to what is measured.

Similarly, Figure 4(c)-(d) and Figure 4(e)-(f) show the convolution code results for $K = 6$ and $K = 8$. We again find that the measured coding gain of each curves falls near the expectations of its theoretical coding gains. That is, for $K = 6$ we expect a gain of 3.358 dB, but measured a coding gain of 2.93 dB. Likewise, for $K = 8$, we expect a gain of 4.23 dB, but measured a coding gain of 3.59 dB.

Finally, Figure 5 shows simulated results for the $K = 8$ convolution encoder using the soft decoding scheme dis-

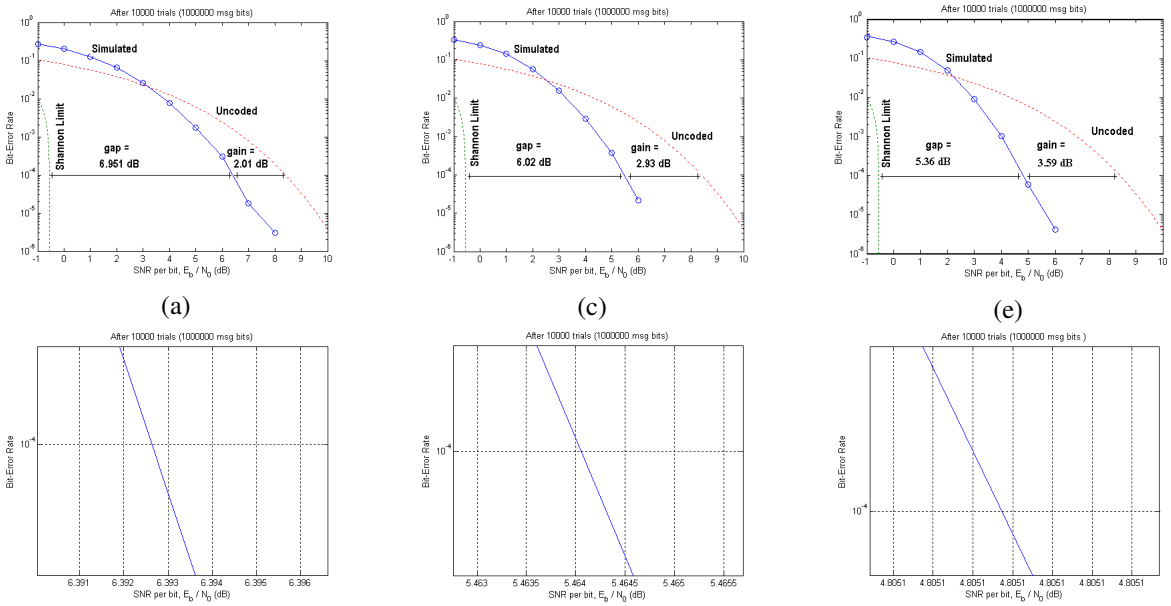

Figure 4. Rate 1/3 Convolution Encoder Simulated Results using a Hard Decoder. (a)-(b) Encoder (Hard Decoding) for $K = 4$. (c)-(d) Encoder (Hard Decoding) for $K = 6$. (e)-(f) Encoder (Hard Decoding) for $K = 8$.

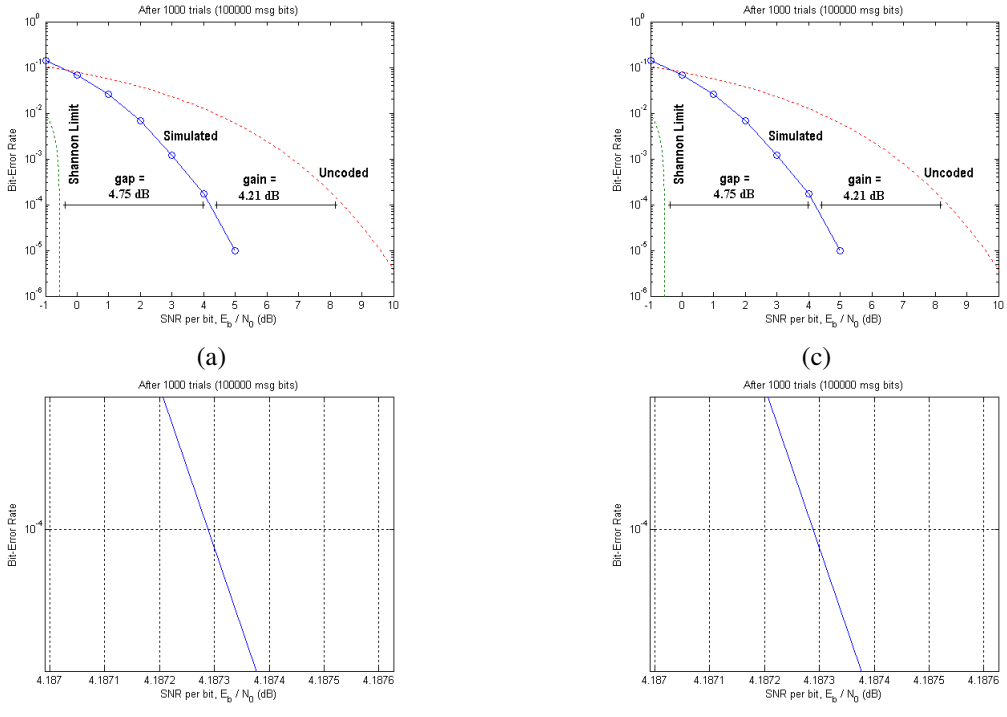

Figure 5. Soft Decoding Results for $K = 4$ and $K = 6$ Rate 1/3 Convolution Encoders. (a)-(b) Encoder (Soft Decoding) for $K = 4$. (c) - (d) Encoder (Soft Decoding) for $K = 6$

cussed in this report. Interestingly, we only were able to measure a coding gain of 4.12 dB, which is far different from what is theoretically expected, i.e., $10 \log_{10}(R * d_{min}) = 7.27$ dB. One particular problem that maybe attributed to such a disparity between the two values could be the small message length of $L = 100$ or the short amount

of trials ($Trials = 10,000$) since each of these contribute to the overall transmitted message bits. Nevertheless, we do achieve a dB gain that is reasonable for objective of this project.

5. Conclusion

In this report, we attempt to mitigate the gap to capacity of Shannon's theoretical limit for a rate 1/3 system. In particular, given the generality and flexibility provided with convolution codes, we present several varying convolution encoders for several varying memory element sizes. Using both soft and hard decoding, we then presented experimental results that for the most part fall within the expected theoretical gains.

References

[1] R.Sandhu, T.Georgiou, and A.Tannenbaum. A new distribution metric for image segmentation. In *SPIE Medical Imaging*, 2008. 3

[2] S. B. Wicker. *Error Control Systems for Digital Communications and Storage*. Prentice Hall, 1995. 2, 3

6. APPENDIX A: MATLAB CODE

Please See Figure 6 through Figure 10 for the detailed MATLAB code. The red boxes highlight regions of code that should be only altered in order to change the design of the encoder (i.e., memory element size or a hard/soft decoding scheme). Current implementation shown is for $K = 4$ with hard decoding.

```
% Main code for ECE6606 project, Spring 2009, Georgia Tech
% Skeleton Written by:  Professor Barry
% Convolution Code by:  Romeil Sandhu

%Initialize Simulation Parameters
L = 100;              % message length
R = 1/3;              % code rate
dbs = -1:10;          % SNR per bit, in dB
trials = 1e4;         % number of trials to perform
%------------

%Define Impulse response for the n generators for a 1/n code - here n=3
g(1) = [1 0 1 1 0 0];                % Impulse Responses _ 1
g(2) = [1 1 0 1 0 0];                % Impulse Responses _ 2
g(3) = [1 1 1 1 0 0];                % Impluse Responses _ 3
n = length(g);                       % Convolution Code (1/n) parameter
memory_els = 3;
%---

%Initialize and compute Shannon Limit/Uncoded Efficiency
errs   = 0*dbs;
EbN0 = 10.^(dbs/10);
sigs = 1./sqrt(2*R*EbN0);
ber0 = logspace(-6,-2.1,81);
ber1 = logspace(-6,-0.99,81);
db0  = 10*log10((2.^(2*R*(1+log2((ber0.^ber0).*(1-ber0).^(1-ber0))))-1)/(2*R));
db1  = 20*log10(erfinv(1-2*ber1));
for trial = 1:trials,

    m = round(rand(1,L));            % message vector

    %--------- ENCODER: 1/3 Convolution Encoder -------%
    c = encode_1_3(m,g,n);
    %--------------------------------------------------%

    %verify code rate!
    if trial==1,disp(['Measured R = ',num2str(length(m)/length(c))]);end;
    noise = randn(1,length(c));
    for i=1:length(dbs),
        r = 2*c - 1 + sigs(i)*noise;

    %--- DECODER:  Convolution Decoder via Trellis Map, ML estimate ---%
    %--- flag = 1 ==> Hard Decoding
    %--- flag = 0 ==> Soft Decoding
    [mhat,node] = decode_1_3(r,n,memory_els,L,1);
    %------------------------------------------%
    errs(i) = errs(i) + sum(mhat~= m);
    end

    %Plot Simulated Result
    ber = errs/(L*trial);
    semilogy(dbs, ber,'o-', db0, ber0,':', db1, ber1,':');
    hold off;
    xlabel('SNR per bit, E_b / N_0 (dB)');
    ylabel('Bit-Error Rate');
    axis([-1 10 1e-6 1])
    title(['After ',num2str(trial),' trials (',num2str(L*trial),' msg bits)', ...
          ' with Mem = ',num2str(memory_els)]);
```

Figure 6. This is the main MATLAB script file that is used to simulate the binary AWGN Channel. To change different Rate 1/3 Convolution Encoders with different memory elements or soft/hard decoding schemes, modify area inside red box

```
function [mhat,node] = decode_1_3(r,n,mem,k,flag)
%Function:  This is the decoder for a generalized convolution encoder.
%This file is independent of the desired rate and memory elements.  It
%first produces a trellis map where we have assigned node states (previous
%and forward) as well as the cost functions associated with received vector
%and acceptable code words.  Note:  Only implements for our binary case.

%r     - codeword
%n     - output (1/n) convolution coder
%m     - number of memory elements
%k     - number of original message bits
%flag - Soft/Hard Decoding ==> 0/1.

%%%%%%%%%%%%%%%%%%%%%%%%%%PSEUDO CODE%%%=%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%1) Initialize next set of nodes, states, stages, etc.

%2) Traverse through nodes that have only been visited (i.e., no need to
%    check on nodes (2,3,4) of a 4 state Trellis Map at time = 1 since we
%    know that we should only visit node 1 given that we begin here.

%3) Given q = 2, we have two inputs = {0,1}.  Input each value and update
%    nodes accordingly
%
%4) Compute acceptable output for each branch of the map, which is
%    formed from the function circuit_logic.  This changes
%    with each encoder.

%5) Compute distance between received vector and acceptable vector for each
%    of the branch at the ith stage.

%6) Update Nodes - a) Next State of the node (Could have 2 Possibilities)
%                  b) Previous State of Node (Could have 2 Possibilities)
%                  c )Node has been visited?
%                  d)Total Cost assigned to Node
%                  e) List/Determine Surviving Branch
%
%7) Form the decoded message by traversing backwards and finding the
%    surviving Branches and Maximum Likelihood (ML) estimate.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%(STEP 1)%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Develop Trellis Map for decoder.  Find # of stages and # of states.
stages = k+mem;
states = 2^mem;
block_st= 1;
%------

%If flag = 1 -> Hard Decoding.  We must first make "hard" decisions of the
%input received vector.
if(flag)
     ind_1 = r>0;
     ind_0 = r<=0;
     r(ind_1) = 1;
     r(ind_0) = -1;
end
%-----

%Initialize State or Memory/Input Elements
for i = 1:mem; c_S.m(i) = 0; end
c_S.st     = 1;
c_S.in     = 0;
%-----

%Initialize Trellis Map, which state we start with etc.
for l = 1:states
     node{1}{l}.p{1}    = NaN;
     node{1}{l}.p{2}    = NaN;
     node{1}{l}.f{1}    = NaN;
     node{1}{l}.f{2}    = NaN;
     node{1}{l}.cost    = -100000;
     node{1}{l}.visit   = 0;
     node{1}{l}.surv    = NaN;
end
node{1}{1}.visit = 1;
node{1}{1}.surv  = 1;
node{1}{1}.cost  = 0;
%-----

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%(STEP 2)%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i = 1:stages
     %Update block status, and initialize next set of nodes
     if(i~=1);block_st = block_st+n; end
     for l = 1:states
          node{i+1}{l}.p{1}    = NaN;
          node{i+1}{l}.p{2}    = NaN;
          node{i+1}{l}.f{1}    = NaN;
          node{i+1}{l}.f{2}    = NaN;
          node{i+1}{l}.cost    = -100000;
          node{i+1}{l}.visit   = 0;
          node{i+1}{l}.surv    = NaN;
     end

```

Figure 7. This is the first half of the generalized decoder of convolution codes. We note that one can perform both soft and hard decoding via a flag input

```
91        %For each state or node, check if we need to do processing on.
92 -      for l = 1:states
93 -          if(node{i}{l}.visit)
94                %If we do process this node, determine its current numerical
95                %state
96 -              c_S.st = l;
97 -              val = l-1;
98 -              for j = mem-1:-1:0
99 -                  if((val - 2^j)>=0)
100 -                     c_S.m{j+1} = 1;
101 -                     val = val-2^j;
102 -                 else
103 -                     c_S.m{j+1} = 0;
104 -                 end
105 -             end
106
107               %%%%%%%%%%%%%%%%%%%%(STEP 3-5)%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
108               %State Input = 0; (Binary, q=2)
109 -             c_S.in    = 0;
110
111               %Determine acceptable output from circuit logic
112 -             [o,n_S] = circuit_logic(c_S,n,mem);
113
114               %Soft or Hard Decoding (e.g., Hard => use Hamming Distance)
115 -             if(flag)
116 -                 dist     = compute_Hamm(o,r,block_st,n);
117 -             else
118 -                 dist     = compute_Lp(o,r,block_st,n);
119 -             end
120
121               %%%%%%%%%%%%%%%%%%%%(STEP 6)%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
122               %Update node's status (e.g., node's status, total cost, is it a
123               %possible survivor?)
124 -             node{i}{c_S.st}.f{1}   = n_S.st;
125 -             node{i}{c_S.st}.visit = 1;
126
127 -             if(isnan(node{i+1}{n_S.st}.p{1}))
128 -                 node{i+1}{n_S.st}.p{1}   = c_S.st;
129 -                 node{i+1}{n_S.st}.visit = 1;
130 -                 node{i+1}{n_S.st}.cost   = node{i}{c_S.st}.cost+dist;
131 -                 node{i+1}{n_S.st}.surv   = c_S.st;
132 -             else
133 -                 node{i+1}{n_S.st}.p{2} = c_S.st;
134 -                 node{i+1}{n_S.st}.visit = 1;
135
136               %Two Possible Survivors: Determine surviving branch
137 -             if(node{i+1}{n_S.st}.cost<=node{i}{c_S.st}.cost+dist)
138 -                 node{i+1}{n_S.st}.surv  =node{i+1}{n_S.st}.p{1};
139 -             else
140 -                 node{i+1}{n_S.st}.surv   = c_S.st;
141 -                 node{i+1}{n_S.st}.cost   = node{i}{c_S.st}.cost+dist;
142 -             end
143 -         end
144
145           %%%%%%%%%%%%%%%%%%%%(STEP 3-5)%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
146           %State Input = 1; (Binary, q=2)
147 -         if(i<=k)
148               %Only process input 1 for first k stages
149 -             c_S.in    = 1;
150
151               %Determine acceptable output from circuit logic
152 -             [o,n_S] = circuit_logic(c_S,n,mem);
153
154               %Update node's status (e.g., node's status, total cost, is it a
155               %possible survivor?)
156 -             if(flag)
157 -                 dist     = compute_Hamm(o,r,block_st,n);
158 -             else
159 -                 dist     = compute_Lp(o,r,block_st,n);
160 -             end;
161
162               %%%%%%%%%%%%%%%%%%%%(STEP 6)%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
163 -             node{i}{c_S.st}.f{2}   = n_S.st;
164 -             node{i}{c_S.st}.visit = 1;
165 -             if(isnan(node{i+1}{n_S.st}.p{1}))
166 -                 node{i+1}{n_S.st}.p{1} = c_S.st;
167 -                 node{i+1}{n_S.st}.visit = 1;
168 -                 node{i+1}{n_S.st}.cost   = node{i}{c_S.st}.cost+dist;
169 -                 node{i+1}{n_S.st}.surv   = c_S.st;
170 -             else
171 -                 node{i+1}{n_S.st}.p{2} = c_S.st;
172 -                 node{i+1}{n_S.st}.visit = 1;
173
174               %Two Possible Survivors: Determine surviving branch
175 -             if(node{i+1}{n_S.st}.cost<=node{i}{c_S.st}.cost+dist)
176 -                 node{i+1}{n_S.st}.surv  =node{i+1}{n_S.st}.p{1};
177 -             else
178 -                 node{i+1}{n_S.st}.surv   = c_S.st;
179 -                 node{i+1}{n_S.st}.cost   = node{i}{c_S.st}.cost+dist;
180 -             end
181 -         end
182 -     end
183 -     end
184 -   end
185 - end
186   %%%%%%%%%%%%%%%%%%%%%%%%%%%(STEP 7)%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
187 - mhat = find_ML_path(node,k);
188 - end
```

Figure 8. This is the second half of the generalized decoder of convolution codes. If one needs to modify the optimal $K = 4$ convolution encoder, then modify circuit logic function

```
function c = encode_1_3(m,g,n)
%Function:  Encodes a Rate 1/3 Convolution Code
%m = message to encode
%g = n generators corresponding to the n outputs
%n = 1/n Convolution Encoder -- # of generators

%First Perform Convolution of Input Message for Each Generator.  This
%produces n outputs... [y{1},y{2},...,y{n}]
for i = 1:n
    y{i} = mod(conv(m,g{i}),2);
end

%Initialize code word to all zeros
c = zeros(1,n*length(y{1}));

%Assemble code word from n outputs
for i = 0:n-1
    c(1+i:n:end) = y{i+1};
end
```

```
function [o,next_State] = circuit_logic(cur_State,n,m)
%Function:  This defines the circuit logic for a specific convolution
%          encoder.  For now, I hand code the n outputs, but this can be
%          easily done automatically via the generator impulse responses.

%cur_State - The current state of the filter
%n         - number of output words [y(1),y(2),...,y(n)]
%m         - number of memory elements

%Define output (n bit) in terms of states
y{1} = mod(cur_State.in + cur_State.m{2} + cur_State.m{3},2);
y{2} = mod(cur_State.in + cur_State.m{1} + cur_State.m{3},2);
y{3} = mod(cur_State.in + cur_State.m{1} + cur_State.m{2} + cur_State.m{3},2);

%Initialize Output Word
o = zeros(1,n*length(y{1}));
for i = 0:n-1; o(1+i:n:end) = y{i+1}; end
next_State.st = 0;
o(o==0) = -1;
%Convert binary vec to state value, and Update State.
for i = 0:m-1
    if(i+1==1); next_State.m{i+1} = cur_State.in;
    else;       next_State.m{i+1} = cur_State.m{i};
    end
    next_State.st = next_State.st+ (2^i)*next_State.m{i+1};
end
next_State.st = next_State.st+1;
end
```

```
function dist = compute_Lp(o,r,block_st,n)
%Function:  Computes the Lp norm... approximate.
%o        - lth branch vector of the trellis map
%r        - total received vector
%block_st - branch currently on
%n        - length of vector block

p = 2;
rhat = r(block_st:block_st+n-1);   %Find r branch vector
dist = (sum(abs(o-rhat).^p));      %Compute Lp distance
```

```
function dist = compute_Hamm(o,r,block_st,n)
%Function:  Computes the Hamming Distance.
%o        - lth branch vector of the trellis map
%r        - total received vector
%block_st - branch currently on
%n        - length of vector block

rhat = r(block_st:block_st+n-1);  %Find r branch vector
dist = sum(o ~= rhat);            %Computes Hamming Distance
```

Figure 9. These are the associated helper files needed to compute the encoding of the convolution code, compute circuit logic for a specific encoder, and compute the appropriate cost functionals for node paths

```
function [mhat] = find_ML_path(node,k)
%Function:  Computes the ML estimate by traversing the Trellis Map, looking
%           for the survivors.

%node - nodes corresponding to the trellis map, contains survivors and
%       state transition
%k    - length of message that we are seeking.

%Initialize survivor and cost list.
p_survivor = zeros(1,length(node)+1);
cost       = zeros(1,length(node)+1);

%Initialize branch output, 1 = top branch taken, 0 = lower branch
branch     = ones(1,length(node));

%Initialize Survivor Trackback
branch(end)       = 0;
p_survivor(end)   = 1;
cost(end)         = node{length(node)}{1}.cost;
p_survivor(end-1) = node{length(node)}{1}.surv;
cost(end-1)       = node{length(node)}{1}.cost;

%Traverse Backwards -- look for surviving branches
for n=length(node)-1:-1:1
    p_survivor(n) = node{n}{p_survivor(n+1)}.surv;
    cost(n) = node{n}{p_survivor(n+1)}.cost;

    %If we take the lower branch, assign a 0.  Otherwise top branch is
    %assigned a 0.
    if(node{n}{p_survivor(n+1)}.f{1} == p_survivor(n+2))
        branch(n) = 0;
    end
end

%The code word is only the first kth bits.  The last length(branch)-k bits
%are by defination, 0.
mhat = branch(1:k);
```

Figure 10. This function traverses the trellis map backwards finding the optimal or “survival” path. It then returns the message that is the ML estimate.