Contents

Chapter 9

Sequence Detection

Chapter 8 studied the concept of coding and sequences, tacitly assuming that a maximum likelihood receiver could be implemented simply by comparing the channel-output sequences with all possible sequential-encoder-produced sequences and then selecting the most likely sequence to have occured. With long-length sequences, the complexity of implementation of such a decoder could be enormous, approaching infinite complexity exponentially fast with sequence length for most codes. Fortunately, a number of optimum and suboptimum decoding methods exist that manage the complexity. Even infinite-length-codeword codes can often be decoded with a complexity that is finite per symbol period. This chapter presents several decoding algorithms that exhibit such finite real-time complexity.

Chapter 9 begins in Section 9.1 with examples and a general description of the **Viterbi Algorithm**, a method that allows exact implementation of **Maximum Likelihood Sequence Detection (MLSD)** if $\nu < \infty$ for any sequential encoder (including partial response). MLSD has finite complexity per symbol period and a reasonable delay. The Viterbi Algorithm applies equally well to the BSC and to the AWGN. Chapter 9 more formally applies to both hard and soft decoding: **Hard decoding** simply means that an inner system has already made decisions about individual symbol values – for convolutional codes, specifically 1 or 0 is already decided. Thus any outer code is for a BSC, and a decoder would use Hamming distance, as in Chapter 8. **Soft decoding** means that the inner system has made no decision (so an AWGN remains in many cases or possibily a conditional probability distribution). The analysis of MLSD can be complicated, but is executed for two examples in Section 9.2, the $1+D$ soft-decoded partial-response channel and a hard-decoded use of 4-state convolutional code example of Section 8.1. Section 9.2 then also includes a general analysis of MLSD for any sequential encoder, and introduces Ginis' code analysis program. This program computes, at perhaps great complexity, the essential parameters for any trellis so that its probability of symbol error and bit error can be accurately estimated. Section 9.3 proceeds to the **"à posteriori probability " (APP) or Bahl-Cooke-Jelinek-Ravin (BCJR)** algorithm for implementation of a MAP detector for a code with finite block length (or segmented into packets). This MAP detector will minimize the probability of a symbol or bit error given observation of the entire corresponding channel-output sequence. The MAP detector minimizes the probability of a symbol error, not the probability of a sequence error like MLSD, and thus the MAP detector can perform yet better than MLSD. This MAP detector can provide hard- or soft-decision output and is of great interest in the concatenated coding systems of Chapter 11. An ad-hoc approximation of the MAP detector using the Viterbi Detector is known as the **Soft-Output Viterbi Algorithm (SOVA)** and appears in Section 9.4. Section 9.5 looks at soft information from the perspective of channel or code constraints. Section 9.6 then proceeds with sub-optimum **iterative decoding** methods that can be applied to situations in which multiple codes have been concatenated as in Chapter 11, but is sufficiently general to apply to a number of different situations with just a single, or even no, code. Section 9.6 addresses the conversion of symbol-based system into binary likelihood-ratio-based coding, which can greatly simplify decoder implementation without loss in performance.

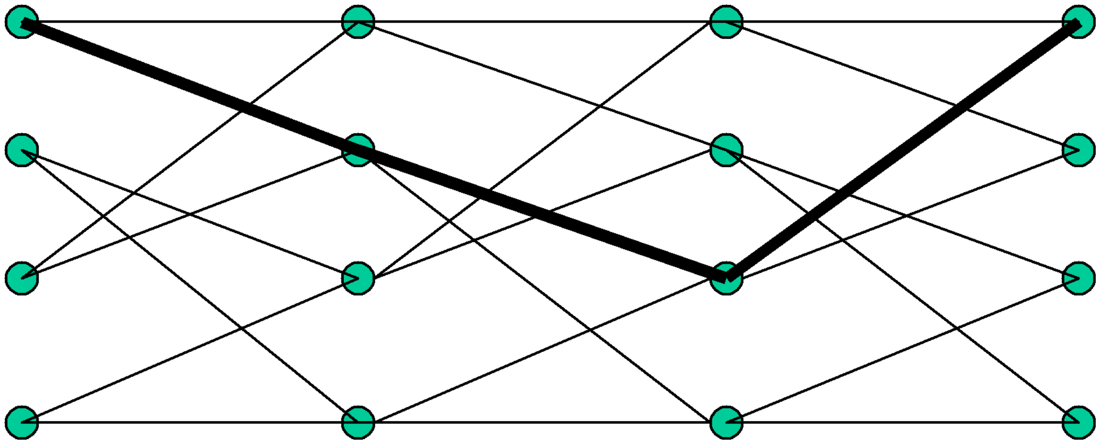


Figure 9.1: Example of ML detection in trellis.

9.1 Trellis-based MLSD with the Viterbi Algorithm

The **maximum-likelihood sequence detector (MLSD)** maximizes the function

$$\hat{\boldsymbol{x}}(D) = \arg\left\{\max_{\hat{\boldsymbol{x}}(D)} p_{\boldsymbol{y}(D)/\boldsymbol{x}(D)}\right\} \quad . \tag{9.1}$$

Simply stated, MLSD finds that sequence through the trellis that looks most like the received output sequence $\boldsymbol{y}(D)$, where "looks most like" varies with the exact conditional channel probability description.

As a simple example, consider Figure 9.1. The highlighted path is the MLSD decision for the sequence that best matches the received channel outputs. For each channel output, there will be one path[1] that is best. The receiver need only wait until the entire sequence is received and then compare it against all possible encoder sequences. The concept is simple, but the complexity grows exponentially with k in terms of the number of possible sequences that are to be compared to the output sequence. The Viterbi Algorithm reduces this complexity through a recursive elimination procedure where trellis paths that are worse than other paths are eliminated early from further consideration. The reader may note that at each stage of the trellis, the decoder could retain only the best "surviving" path into each state, eliminating all the rest into that state at that time.

Thus, the complexity of the optimum MLSD receiver need not be infinite (per unit time) for a sequential encoder's decoding, as long as ν is finite, even for a semi-infinite input data sequence. The next subsection introduces the recursive Viterbi MLSD procedure using the example of a $1 + D$ binary partial response channel.

9.1.1 The Additive White Gaussian Noise Channel

The MLSD finds for the AWGN the sequence through the trellis that satisfies

$$\min_{\hat{x}(D)} \|\boldsymbol{y}(D) - H(D)\hat{\boldsymbol{x}}(D)\|^2 \tag{9.2}$$

where usually $H(D) = I$ (or 1 for PAM or QAM symbols in the sequence). For partial response channels, $H(D)$ represents the controlled intersymbol interference effect. For coded systems on the

[1] In the case of ties, one selects randomly and it is likely an error is made by the detector, but ties are rare on most practical channels, as for example in simple symbol-by-symbol detection on the AWGN channel.

AWGN, $H(D) = I$. This chapter generalizes the notation for the noiseless sequence through the trellis on the AWGN to be called $\tilde{\boldsymbol{y}}(D)$ where

$$\tilde{\boldsymbol{y}}(D) = \begin{cases} \boldsymbol{x}(D) & \text{coded systems} \\ H(D)X(D) & \text{partial-response systems} \end{cases} \quad . \tag{9.3}$$

The term "closest path" in the AWGN case literally means the trellis sequence that is at smallest squared distance from the channel output in a possibly infinite-dimensional space. The minimum distance is then (as always) the distance between the two closest trellis sequences. Sometimes finding the two closest sequences is easy, and sometimes it is extremely difficult and requires the search program of Section 9.2. However, d_{min} always exists. Furthermore, this second volume of this textbook refines the definition of nearest neighbors to be ONLY those sequences at distance d_{min}, so

$$N_e \stackrel{\Delta}{=} \text{the number of symbol errors at any time } k \text{ that could arise from any} \tag{9.4}$$

$$\text{past "first" error event with distance } d_{min} \; . \tag{9.5}$$

This definition of N_e differs from that of Chapter 1 where any neighbor (even at larger distance) with a common decision boundary was included in N_e. It is simply too hard to find N_e by searching in an infinite-dimensional space for all neighbors who might have a common decision boundary, thus the refined definition that includes only those at distance d_{min}. With this refined definition, the NNUB becomes an approximation of, and not necessarily an upper bound on, error probability for MLSD:

$$P_e \approx N_e Q\left(\frac{d_{min}}{2\sigma}\right) \quad . \tag{9.6}$$

Often (9.6) is a very accurate approximation, but sometimes those neighbors not at minimum distance can be so numerous in multiple dimensions that they may dominate the probability of error. Thus, coded systems investigate the set of distances

$$d_{min} = d_{min}(0) < d_{min}(1) < d_{min}(2) < .... < d_{min}(i) \quad , \tag{9.7}$$

in other words the set of distances starting with the smallest and then the next smallest $d_{min}(1)$, the second-next smallest $d_{min}(2)$ and so on with $d_{min}(i)$ being the notation for the i^{th}-next smallest distance for the code. Each of these distances has a corresponding number of occurences on average for all sequences diverging at the same point in time, N_1, N_2, ... N_i (ordered in terms of the index i of the corresponding $d_{min}(i)$).

Thus,

$$P_e \leq \sum_{i=0}^{\infty} N_i \cdot Q\left(\frac{d_{min}(i)}{2\sigma}\right) \quad , \tag{9.8}$$

which in most cases is dominated by the first few terms at reasonably high SNR, because of the Q-function's rapid decrease with increasing argument.

An MLSD decoder's sequence-selection error is often called an **error event**, which occurs with minimum probability P_e. In many cases, this minimum probability tends to 1 if the sequence is infinite in length (stating only that eventually an error is made and not necessarily poor performance.[2]). For practical evaluation, performance is measured as a per-symbol error probability in that it applies to the common point in time that starts all the evaluated sequences in computing the distance spectrum $\{d_{min}(i)\}$ and N_i. Such an error could commence at any time, so this is a symbol-error probability corresponding to choosing the best sequence. For a block code, the designer would need to evaluate error events beginning at each and every symbol time instant within the block to get the correct probability of sequence error, and this probability might vary over the block, and so it would need to be averaged for a measure of symbol-error probability. In the case of the semi-infinite trellis implied by any trellis with $\nu > 0$, this number is the same for all time instants as long as the sequential encoder is time invariant (and no previous sequence errors have been made).

[2] For a capacity-achieving code or good code of course, the probability of a sequence error can be made arbitrarily small, but most practical designs do not operate at a target sequence or symbol error rate that is zero.

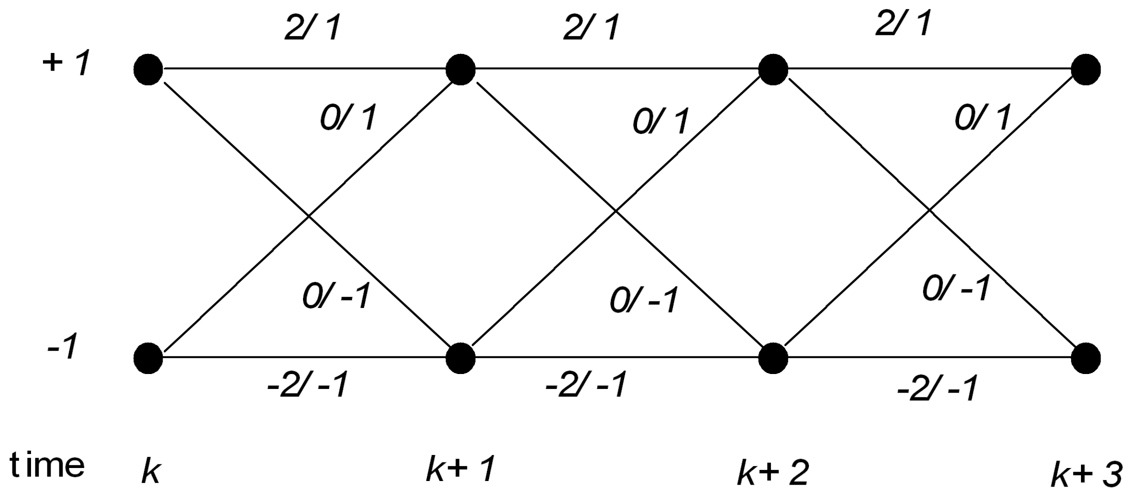


Figure 9.2: Trellis Diagram for $1 + D$ partial-response/sequential-encoder with $b = 1$ PAM.

9.1.2 Example of $H(D) = 1 + D$ Partial Response for Viterbi MLSD

Figure 9.2 illustrates again the trellis for the $1+D$ partial-response channel example of Chapter 8, which had AWGN distortion. The process of MLSD on this channel amounts to finding the sequence through the trellis that is closest to the received sequence $y(D)$ in terms of Euclidean distance, which is simply the sum of squared differences from y_k over all times that the sequence exists.

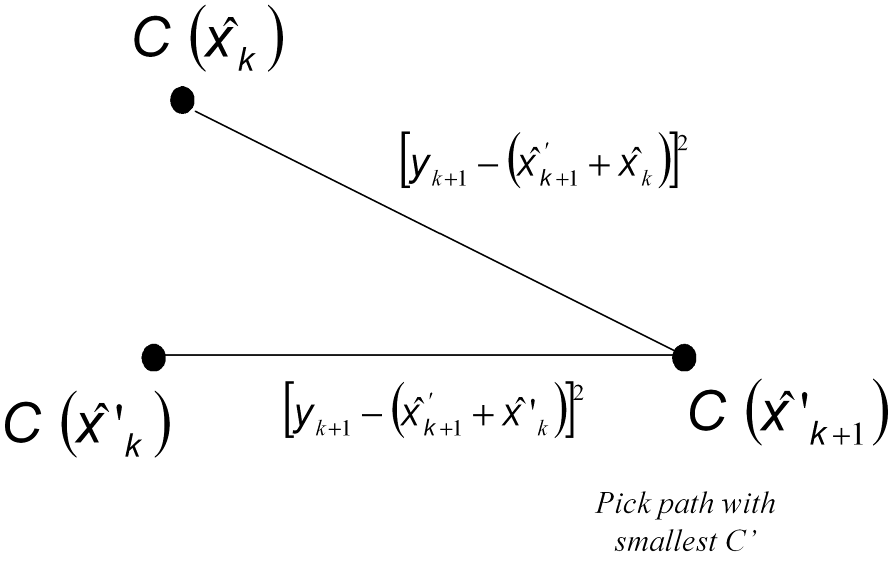
More formally. the ML detector for any sequence of inputs of length $k + 1$, starting at time 0 and ending at time k, would select that input sequence $\hat{x}_0$ ... $\hat{x}_k$ that minimizes the “cost”

$$\mathcal{C}_k \triangleq \sum_{m=0}^{k} |y_m - (\hat{x}_m + \hat{x}_{m-1})|^2 = \|y - h * x\|^2 \quad , \tag{9.9}$$

for a $1 + D$ channel. If the MLSD-selected sequence is correct the cost will essentially be the sum of squared channel noise over $k + 1$ successive symbols. In any case, this MLSD-selected sequence is the one that is closest in squared Euclidean distance to the channel-output sequence. The MLSD-selected sequence of length $k + 2$ minimizes

$$\begin{aligned} \sum_{m=0}^{k+1} |y_m - (\hat{x}_m + \hat{x}_{m-1})|^2 &= \left[\sum_{m=0}^{k} |y_m - (\hat{x}_m + \hat{x}_{m-1})|^2\right] + |y_{k+1} - (\hat{x}_{k+1} + \hat{x}_k)|^2 & (9.10) \\ \mathcal{C}_{k+1} &= \mathcal{C}_k + |y_{k+1} - (\hat{x}_{k+1} + \hat{x}_k)|^2 \quad . & (9.11) \end{aligned}$$

The values of $\hat{x}_0$ ...$\hat{x}_{k-1}$ that minimize the first term on the right in (9.11) **for each of the M values of $\hat{x}_k$** are the same as those that minimize (9.9) for each value of $\hat{x}_k$ because the second term on the right in (9.11) does not depend on $\hat{x}_0$, ..., $\hat{x}_{k-1}$. Figure 9.3 illustrates this concept. For each state at time k, there is a unique minimum cost $\mathcal{C}(\hat{x}_k)$ that is the smallest sum of squared differences between the channel output and any trellis sequence up to and including sample time k that contains the specific value of $\hat{x}_k$ corresponding to that state. Another state, with corresponding second cost $\mathcal{C}(\hat{x'_k})$ has a different smallest cost for all paths up to time k that terminate in state x'_k. Both the paths in Figure 9.3 merge into a common state at time $k + 1$. The smallest cost associated with that state at time $k + 1$ is found by first adding to each state's cost at time k the corresponding single squared error term corresponding to the path into this common state, and then selecting the path with the smaller cost as the new **survivor** for this common state. For each state, the sequence detector need only find the


$C\left(\hat{x}_k\right)$
$\left[y_{k+1}-\left(\hat{x}'_{k+1}+\hat{x}_k\right)\right]^2$
$C\left(\hat{x}'_k\right)$
$\left[y_{k+1}-\left(\hat{x}'_{k+1}+\hat{x}'_k\right)\right]^2$
$C\left(\hat{x}'_{k+1}\right)$

Pick path with smallest C'

Figure 9.3: Iterative cost minimization for Duobinary channel.

path into that sate that minimizes the cost in (9.11). This recursive detection process can then repeat itself exactly for each new sample time, always maintaining M possible costs and candidate previous sequences (often called **survivors**) at each time sample. In terms of the trellis diagram for $M = 2$ in Figure 9.2, detection is then a recursive process that corresponds to finding that sequence that is closest to the received sequence at each time k. The extension to time $k + 1$ requires knowledge of only the best path and associated cost into each of the 2 states at time k. MLSD decides which of the 4 possible extensions of these two paths is closest to the output sequence when the channel output sequence is augmented by y_{k+1}. This recursive detection process could continue ad infinitum, and only needs to find the best path at any time into the two possible states to be ready to incorporate the next channel output sample. This recursive method of detection is known as the **Viterbi Algorithm** (A. Viterbi, 1967) in communication theory. In general, there are 2^ν states for any trellis (actually M^ν for M-ary PAM with partial response) and a candidate **survivor** path into each. So, there are in general 2^ν survivor paths (M^ν partial response). In short, there are many paths into a state at time k, but the decoder only needs to remember one of them - the one with the lowest cost.

The following example illustrates the concept of sequence detection with the trellis:

EXAMPLE 9.1.1 Suppose that differential encoding is used with the binary $1+D$ channel, and that the following sequence was input to the precoder, 1 0 1 0 1. The encoder sets the initial channel state according to the last $\bar{m}$ being zero (but the receiver is not presumed to know this). The following table summarizes the channel signals for precoding and symbol-by-symbol detection:

time	k	$k+1$	$k+2$	$k+3$	$k+4$	$k+5$
m	-	1	0	1	0	1
$\bar{m}$	0	1	1	0	0	1
x_k	-1	1	1	-1	-1	1
$\tilde{y}_k$=x_k+x_{k-1}	-	0	2	0	-2	0
y_k	-	.05	2.05	-1.05	-2	-.05
Quantized y_k ($\hat{y}_k$)	-	0	2	-2	-2	0
$\left(\left[\frac{\hat{y}_k}{d}+(M-1)\right]\right)_{M=2}$	-	1	0	0	0	1

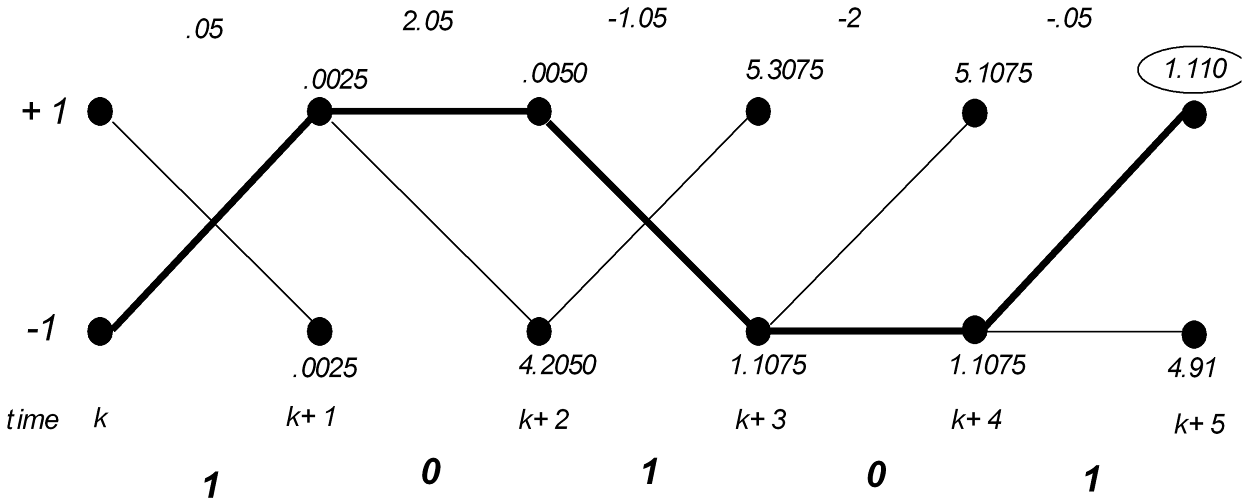


Figure 9.4: Sequence detection example.

> The symbol-by-symbol detection described in Section 3.7 detects $\hat{m}$= 1 0 0 0 1, which has one error in the middle position, $m_{k+3} \neq 0$.
>
> The sequence detection result is shown in Figure 9.4, where no errors are made in decoding the data. The precoder may appear unnecessary with the sequence detection system. However, without precoding, long strings of errors in the trellis (i.e., picking a path that deviates for a while from the correct path) will result in long strings of bit errors, whereas with the precoding, only 2 bit errors are associated with any such **error event** in sequence detection. In fact, it is possible without precoding for a noise sample, with component along the error vector between two nearest neighbor sequences exceeding $d_{min}/2$, to produce an infinite number of input bit errors for the two corresponding infinite sequences in the trellis – this is called **quasi-catastrophic error propagation** in partial-response channels.

Two Viterbi Algorithm matlab programs are provided here for the interested reader. The first is for partial-response channels by former Ph.D. student Dr. Ghazi Al-Rawi (see also the EE379B website at Stanford):

```
% EE379B
% MLSD using VA for partial response channels
% Written by Ghazi Al-Rawi,April 17, 2001
%   Modified by J. Cioffi, March 29, 2003
%
%  Usage: mh = mlsd1D(y, v, b, Sk, Yk, Xk);
%
% y is the channel output sequence
%   v is constraint length (log2 of the number of states
%   b is the number of bits per symbol
%   Sk is the previous-state description matrix and is 2^v x 2^b
%       for instance for the EPR4 channel H=[1 1 -1 -1] the nextstate trellis is the vector
%       nextState = [1 2; 3 4; 5 6; 7 8; 1 2; 3 4; 5 6; 7 8];
%       and the corresponding previous state is
%       Sk = [1 5; 1 5; 2 6; 2 6; 3 7; 3 7; 4 8; 4 8];
%   Yk contains the noiseless 1D trellis output corresponding to Sk or for the EPR4
%       Yk = [0 -2; 2 0; 2 0; 4 2; -2 -4; 0 -2; 0 -2; 2 0];  2^v x 2^b
%   Xk is the b x 2^v vector of single input corresponding to all the transitions
%       into each of the 2^v states
%       Xk = [0 1 0 1 0 1 0 1];
% mh is the detected message sequence

function mh = MLSD(y,v,b,Sk,Yk,Xk)

M = 2^b;
C = [0 1e12*ones(1, M^v-1)]; % State metric, initialized so as to start at state 0
D=[];
Paths=[];
C_next=[];
for k=1:length(y),
   i=1;
   for j=1:M^v,
      for b=1:M
         D(i) = (Yk(j, b)-y(k))^2;
         i=i+1;
      end
   end
   i=1;
   for j=1:M^v,
      minCost=1e12;
      for b=1:M
         cost = C(Sk(j, b))+D(i);
         i=i+1;
         if (cost<minCost)
            minCost = cost;
            Paths(j, k) = Sk(j, b);
         end
      end
      C_next(j)=minCost;
   end
   if (mod(k,1000)==0)
      C_next = C_next - min(C_next); % Normalization
   end
   C = C_next;
end

% Find the best survivor path at k=length(y)
[minCost, I] = min(C);
if (length(I)>1)
   warning('Warning: There are more than one path with equal cost at state k=length(y)!');
end

% We will pick the first
sp=I(1);

xh=[];
for k=length(y):-1:1,
   xh(k) = Xk(sp);
   sp = Paths(sp, k);
```

```
end
mh = xh
```

The second C program set is for convolutional codes by former Ph.D. student Dr. Kee-Bong Song **simple test file2.m**

```
N_STATE=64;
G = [bin2dec('1101101') bin2dec('1001111')]; %Note that G takes decimal numbers
[next_state, output]=trellis(G);
%Note that my program use the state numbering from 0 unlike the Ginis dmin progam
%(which starts from 1)

% Viberbi Algorithm initilization
[prev_state_index, input_index, trellis_out, A] = siva_init
              ( next_state, output, N_IN , N_OUT );

% Data packet
total_packet_bit = 100;
packet = (sign(rand(1,total_packet_bit)-0.5)+1)*0.5;

% Convolutional Encoder
encoded_output = conv_enc(packet,next_state,output,N_IN,N_OUT,N_STATE);

noise = sigma*randn(1,length(encoded_output)) ;

% Channel output (Noiseless for test purpose)
channel_output = 2 * encoded_output -1 + noise ;


% Soft-input Viterbi decoding

decoded_output=siva( channel_output, prev_state_index , input_index , trellis_out,
         A, N_IN , N_OUT, N_STATE, TRACE_LEN );
```

conv enc.m

```
function code_out=conv_enc( input_bit, next_state, output, N_IN, N_OUT, N_STATE)

% Convolutional Encoder
%
% code_out = conv_enc( input_bit, next_state, output )
%
% input_bit : binary input bit stream (message)
% next_state : Next state matrix  (can be generated from trellis function) (N_STATE by 2^N_IN)
% output : Output codeword matrix (can be generated from trellis function) (N_STATE by 2^N_IN)
% N_IN  : number of input bits of the encoder
% N_OUT : number of output bits of the encoder
% N_STATE : number of state of the convolutional code
%
% input_bit and code_ouput orderings are as follows
%
% input_bit=
% [ input_bit1 intput_bit2 input_bit3 ... input_bitN_IN   input_bitN_IN+1 ... ]
%    1st        2nd           3rd     ...    N_IN th         1st
% : Encoder input feeding order
%
% code_out=
% [ output_bit1 output_bit2 output_bit3 ... output_bitN_OUT output_bitN_OUT+1 ...]
%    1st          2nd          3rd           N_OUT th             1st
% : Encoder output order
%
% Assumes the zero initial state (achieved by zero tail bits in the previous message)
% The length of input_bit should be the multiple of N_IN

if nargin~=6
    error('Wrong number of input arguments');
    return;
elseif prod(size(next_state)==size(output))==0
    error('Dimension of next_state should be the same as that of output');
    return;
elseif prod(size(next_state)==[N_STATE 2^N_IN])==0
    error('N_STATE or N_IN is not compatible with the size of next_state');
    return;
end

L_MESSAGE=length(input_bit);
N_CODE=L_MESSAGE/N_IN;

if N_CODE ~= round (L_MESSAGE/N_IN)
    error('Length of input bit stream should be the integer multiple of the
           number of input bits of the encoder!');
    return;
end

%Output initialization
code_out=zeros(1,N_CODE*N_OUT);

for i=1:N_CODE

    start_index=(i-1)*N_IN+1;
    input_dec=2.^(0:N_IN-1)*(input_bit(start_index:start_index+N_IN-1))';
%input codeword (convert from binary to decimal)

    if i==1
        cur_state=0;
    end

    %Convert the output codeword into binary vector

    code_out((i-1)*N_OUT+1:i*N_OUT)=bitget(output(cur_state+1,input_dec+1),1:N_OUT);
    cur_state = next_state(cur_state +1 , input_dec +1 );

end
```

siva.m

```
function decode_output = siva( soft_rcv_bit, prev_state_index , input_index ,
          trellis_out, A, N_IN , N_OUT, N_STATE, trace_length)

% Updated Sliding Window Vectorized Soft-Input Viterbi Algorithm
%
% decode_output = siva( soft_rcv_bit, prev_state_index , input_index ,
%              trellis_out, A, N_IN , N_OUT, N_STATE, trace_lengtha)
%
% soft_rcv_bit : (possibly de-interleaved and de-punctured) soft received bit stream (row
%                vector) The length of soft_rcv_bit should be the multiple of N_OUT
%                The sign of bit metric indicates the value of the received bit
%                with the magnitude equal to the reliability of the decision.
%                For hard-input decoding, simple +-1 sequence can be used
%
% prev_state_index : Previous state index associated with the state transition
%      ( see siva_init function ) ( N_IN by num_trans_max matrix )
% input_index : Input index associated with the state transition
%      ( see siva_init function ) ( N_IN by num_trans_max matrix )
% trellis_out : Bipolar trellis output matrix for soft-input decoding
%      ( see siva_init function )
% A : Additional matrix to be added for irregular trellis ( see siva_init_function )
% N_IN   : The number of input bits of the convolutional code
% N_OUT  : The number of output bits of the convolutional code
% N_STATE : The number of states of the encoder
% trace_length : Viterbi detection trace length (by default, traces back from the
%                best metric at the last stage) if the trace_length is greater than the
%                actual number of codeword received, the default option is used
%
% Assumes that the encoder started from the zero state
%
% Dimension relations are as follows
% num_trans_max = Maximum number of transition to the state (usually should
%  be the same as 2^N_IN );
% prev_state_index : N_STATE by num_trans_max
% input_index : N_STATE by num_trans_max
% trellis_out : num_trans_max*N_STATE by N_OUT
%
% 8/31/2002

BRANCH_MAX=1e5; %Infinite (Maximum branch metric)
L_MESSAGE=length(soft_rcv_bit); %Length of received bits
N_CODEWORD= L_MESSAGE/N_OUT;    %Number of output codeword received


if N_CODEWORD~=round(L_MESSAGE/N_OUT)
    error('The length of receive_vector should be the multiple of N_OUT');
    return;
end

switch nargin
case 8
    trace_length=N_CODEWORD;
case 9
    trace_length=min(N_CODEWORD,trace_length);
otherwise
```

```
    error('Wrong number of input arguments');
    return;
end

state_index=1;                           %Best path's state index initialization

num_trans_max = size( prev_state_index, 2 );
row_trellis_out = size ( trellis_out , 1);

init_state_index = state_index;     %Initial state index
cost = zeros( N_STATE , 1 );            %cost metric matrix

% Index matrix that indicates the location of the previous state that causes the
% minimum cost transition to each state

index_matrix = zeros(N_STATE , N_CODEWORD);

%Initial decoding stage

%Taking the received soft bits of length N_OUT

soft_input = soft_rcv_bit( 1 : N_OUT);
soft_input = soft_input ( ones(1,row_trellis_out ), :);

%Calculate the branch metric
delta = trellis_out - soft_input;
branch_metric_vec = sum ( delta.^2 , 2 );

%Filter out the states whose transitions are not from the initial state
impossible_index = find( prev_state_index' ~= init_state_index );

%Set the cost the infinite
branch_vec(impossible_index) = BRANCH_MAX;

cost_temp_matrix = transpose ( reshape( branch_vec, num_trans_max, N_STATE ) ) + A;

%Find the state with the minimum cost
[ cost index ]= min( cost_temp_matrix , [] , 2 );

%Find the index associated with the minimum-cost transition

index_matrix( : , 1 ) = index ;

%Sliding window (pipelined) decoding stage

for n= 2 : N_CODEWORD        %Decoding index

    %Taking the received soft bits of length N_OUT sequentially
    soft_input = soft_rcv_bit( (n-1)*N_OUT+1 : n*N_OUT );
    soft_input = soft_input( ones( 1 , row_trellis_out ) , : ) ;

    %Calculate the path metric
    delta = trellis_out - soft_input;
    branch_metric_vec = sum ( delta.^2 , 2);
    cost_temp_matrix = ...
        transpose( reshape( branch_metric_vec , num_trans_max , N_STATE) ) +
             cost( prev_state_index ) + A;

    [ cost index ]= min( cost_temp_matrix , [] , 2 );

    % Normalize the cost for numerical resolution
    cost = cost - min(cost);

    %Index update

    index_matrix( : , n ) = index ;

    if n >= trace_length %Sliding window decoding

        [min_cost_metric state_index] = min( cost );

        %Trace back to trace_length

        cur_state = state_index ;

        for i = 1 : trace_length - 1

            cur_state = ...
                prev_state_index( cur_state , index_matrix( cur_state , n - i + 1) ) ;

        end

        x = input_index( cur_state , index_matrix( cur_state , 1 ) ) - 1;

        decode_output( ( n - trace_length )*N_IN + 1 : ( n - trace_length + 1 )*N_IN )
                 = ...
            bitget( x , 1 : N_IN );
     end

    %End of one of of N_STATE parallel update stages

end

%Last output buffer update
cur_state = state_index;

for i = 1 : trace_length - 1

    x = input_index( cur_state , index_matrix( cur_state , n - i + 1 ) ) - 1;

    cur_state = prev_state_index( cur_state , index_matrix( cur_state , n - i + 1) );

    decode_output ( (n - i )*N_IN + 1 : (n - i + 1) *N_IN ) = ...
        bitget( x , 1 : N_IN );

end
```

trellis.m

```
function [next_state, output]=trellis(G)

% Trellis diagram generator function
% [next_state, output]=trellis(G)
% G: Generator polynomial matrix (N_IN by N_OUT real matrix)
%   G(i,j) is the polynomial associated with j th output bit and i th input bit (decimal unit)
% next_state : next state matrix (N_STATE by 2^N_IN)
%              state is numbered from 0 to N_STATE-1
% output : output codeword matrix (N_STATE by 2^N_IN)
%       output(i,j) means output codeword correponding to input value of (j-1) at (i-1) state
%
% This version of trellis function assumes the feedforward implementation
% (i.e G should always be the interger matrix)

[N_IN N_OUT]=size(G);   %N_IN : Number of input bit of the convolutional encoder
                        %N_OUT : Number of output bit of the convolutional encoder

for i=1:N_IN
    n_tap(i)=size(dec2bin(G(i,:)),2)-1; % Maximum number of delay elements for i th input bit
end

N_TOTAL_TAP=sum(n_tap);    % Total number of delay elements
N_STATE=2^sum(n_tap)  ;    % Number of states

current_state=0:N_STATE-1;

%Next state and output codeword initialization
next_state=zeros(N_STATE,2^N_IN);
output=zeros(N_STATE,2^N_IN);

%State update and output codeword calculation

%State loop
for i=1:N_STATE

    %Input loop
    for j=1:2^(N_IN)

        %Output bit vector initialization (bit_out is a binary representation of output(i,j)
        %in the vector form)
        bit_out=zeros(1,N_OUT);

        %Specific update for each input bit
        for k=1:N_IN

            %start_index and end_index tell the beginning and the end of state bits associated
            %with delay elements for k-th input bit
            %   start_index corresponds to the unit-delayed element
            %   end_index corresponds to the most delayed (i.e n_tap(k)) element

            if k==1
                start_index=1;
            else
```

```
                start_index=(k-1)*n_tap(k-1)+1;
            end

            end_index=start_index+n_tap(k)-1;
            left_shift=N_TOTAL_TAP-sum(n_tap(1:k));

            %Contents in delay elements associated with k-th input bit
            small_state=bitand(bitshift(current_state(i),-(start_index-1),N_STATE),
                2^(end_index+1)-1);

            %Output calcuation
            bit_in=bitget(j-1,k); %k-th input bit for j-th input
            state_and_input=2*small_state+bit_in; %delayed and the current bit
              of k-th input bit

            for l=1:N_OUT
               %Contribution of k-th input bit to l th output bit
                contr=bitand(G(k,l),state_and_input);
                %output bit vector is the sum of each contribution from k-th input bit
                bit_out(l)=mod(bit_out(l)+sum(dec2bin(contr)),2);
            end

            %State update
            next_small_state(k)=bitshift(small_state,1,n_tap(k))+bit_in;
            next_state(i,j)=next_state(i,j)+next_small_state(k)*2^(start_index-1);
        end

        output(i,j)=sum(2.^(0:N_OUT-1).*bit_out);
    end
end
```

siva init.m

```
function [prev_state_index, input_index, trellis_out, A] = siva_init
          ( next_state, output, N_IN , N_OUT )

% Vectorized Soft-Input Viterbi Algorithm Initialization
% [prev_state_index, input_index, trellis_out, A] = siva_init ( next_state, output,
%                                                                N_IN , N_OUT )
%
% prev_state_index : N_STATE by num_trans_max matrix. prev_state_index( i , : )
%                    indicates the indexes of states whose next
%                    state transition is i. If num_trans(i) < num_trans_max,
%                    prev_state_index( i, num_trans(i)+ 1 : num_trans_max ) = 1;
%
% input_index : N_STATE by num_trans_max matrix which indicates the associated inputs with
%                    the state transition
%
%
% trellis_out : N_STATE*num_trans_max by N_OUT matrix which indicates the asociated trellis
%                   outputs in bipolar forms.
%
%                 [ O(1,1) O(1,2) .. O(1,N_OUT)   ----------------
%                 O(1,1) O(1,2) .. O(1,N_OUT)   |               |
%                                 ..            |  num_trans(1) |
%                 O(1,1) O(1,2) .. O(1,N_OUT)   ---             | num_trans_max
%                   0      0         0                          |
%                                 ..                            |
%                   0      0         0          ------------------
%                 O(2,1) O(2,2) .. O(2,N_OUT)   ---
%                 O(2,1) O(2,2) .. O(2,N_OUT)   |
%                               ..              |  num_trans(2)
%                 O(2,1) O(2,2) .. O(2,N_OUT)   ---
%                               ..
%                 O(N_STATE,1)  .. O(N_STATE,N_OUT) ]
%
%               Output associated with the state tansition
%
%  A : Additional matrix to be added to make irrelevant cost infinite
%     A returns 0 for the simple case of  num_trans = constant
%
% Sep 3, 2002
%

N_STATE=size(next_state,1);     %Number of states

if prod(size(next_state)==size(output))==0
    error('Dimensions of next_state and output should be the same');
    return;
end

for i = 1 : N_STATE

    [ row_index , col_index ] = find( next_state == i - 1 );
    num_trans( i ) = length(col_index ) ;

end

num_trans_max = max(num_trans);

trellis_out = zeros( N_STATE * num_trans_max , N_OUT );
prev_state_index = zeros( N_STATE , num_trans_max );

for i=1:N_STATE

    [ row_index ,col_index] = find( next_state == i-1 );

    K = num_trans(i);

    for j=1:N_OUT
        trellis_out( (i-1)*K +1 : i*K , j ) = 2*bitget( diag(output( row_index,
                                                        col_index)), j)-1;
    end

    prev_state_index ( i , 1 : K ) = row_index';
    input_index ( i , 1 : K)  = col_index';

end
```

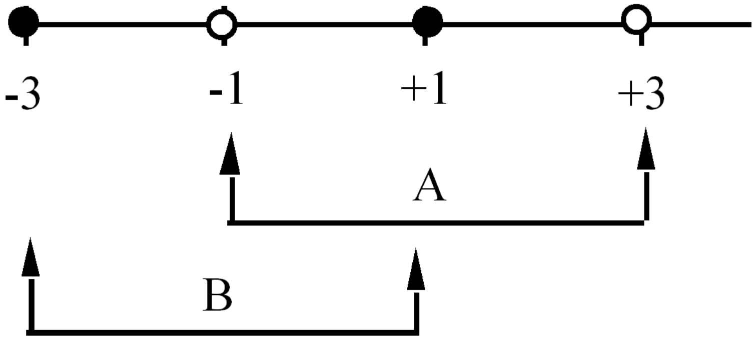


Figure 9.5: Partitioning for Reduced State Sequence Detection.

```
[ I , J ] = find(prev_state_index == 0 );

if isempty(I)
    %Simple case : the number of transition to each state is the same

    A = 0;

else

    BRANCH_MAX = 1e5;

    A = zeros( N_STATE , num_trans_max );
    % Additional matrix to be added to make cost infinite

    for i = 1 : length(I)

        prev_state_index(I(i),J(i)) = 1;
        B( I(i) , J(i) ) = BRANCH_MAX;

    end

end

num_trans_max = max(num_trans);
total_trans = sum (num_trans);
row_trellis_out = size ( trellis_out , 1);
cum_num_trans = cumsum (num_trans) - num_trans;
```

9.1.3 Reduced State Sequence Detection Example

It is possible to reduce the number of states to 2 for the $1+D$ channel in implementing (nearly optimum) sequence detection for $M > 2$. Figure 9.5 illustrates a constellation labeling for the $M = 4$ case. The constellation partitions into two groups (A and B) with twice the minimum distance within each group as in the entire signal set. Figure 9.6 shows a new two-state trellis for the $M = 4$ $1+D$ channel. This

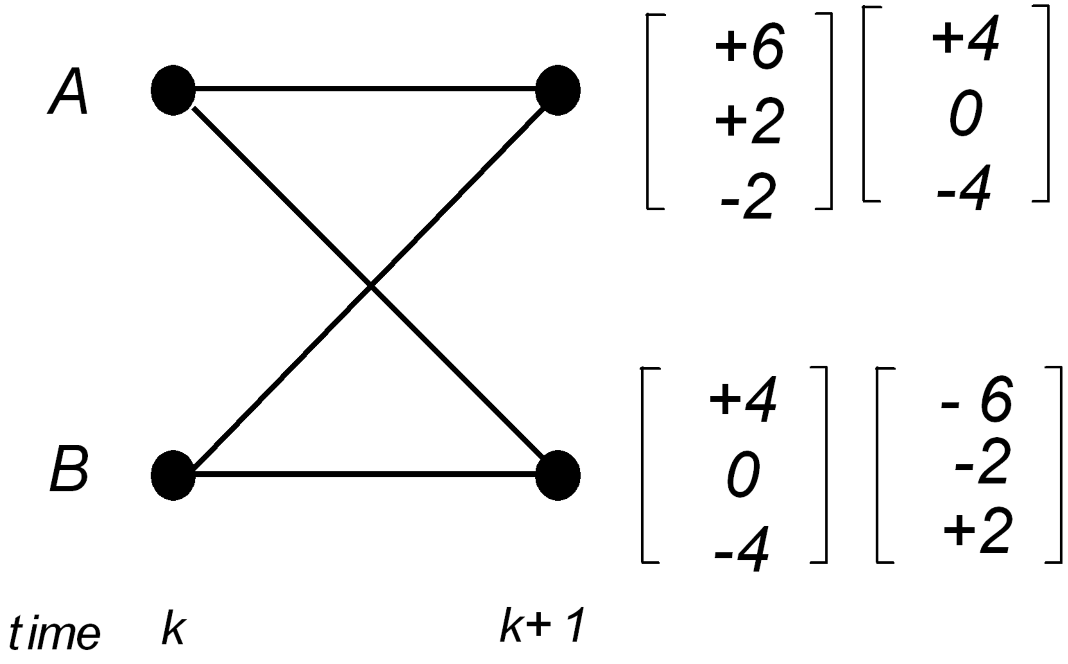


Figure 9.6: Reduced State Sequence Detection trellis.

new trellis only records the partition, A or B, that was last transmitted over the channel. The minimum distance between any two sequences through this trellis ia still $d_{min} = 2\sqrt{2}$. The distinction between any two points, by symbol-by-symbol detection, in either set A or in set B, once selected, has an even larger minimum distance, $d = 4$. Thus, the probability of error will be approximately the same as the M-state detector.

In general for PAM or QAM, it is possible to partition the signal constellation into subsets of increasing minimum distance (see the mapping-by-set-partitioning principles of Chapter 8 for a specific procedure for performing this partitioning). In reduced state sequence detection (RSSD, Eyuboglue, 1989), one partitions the constellation to a depth such that the increased intra-partition minimum distance equals or exceeds the minimum distance that can occur in sequence detection. The number of partitioned sets is then denoted M'. Obviously, $M' \leq M$ and most often $M' = 2$ making the number of states consistently 2^{ν} as in the sequential encoders without partial response. Then RSSD can be applied to the corresponding $(M')^{\nu}$-state trellis, potentially resulting in a significant complexity reduction with negligible performance loss. If the signal set cannot be partitioned to a level at which the intra-partition distance equals or exceeds the minimum distance, then simple RSSD cannot be used.

9.1.4 Sequence Detection with the Viterbi Algorithm

The $1 + D$ sequence-detection example of the previous section (see Figure 9.4) is a specific use of the Viterbi Algorithm. This section generalizes this algorithm. The Viterbi Algorithm was introduced in 1967 by Viterbi for decoding of convolutional codes - its application to partial-response channels was first published by Forney in 1972. The algorithm, itself, in its purest mathematical form, was developed by mathematicians, unknown to Viterbi at the time of his publication, and is a specific example of what is more generally called “dynamic programming.” However, its applications to detection in communication are so prevalent that the use of the term “Viterbi Algorithm” is ubiquitous in communication, and in recognition of Viterbi’s independent insight that the method could be used to great advantage in simplifying optimal decoding in data communication. The Viterbi Algorithm can be applied to any trellis.

Several quantities help describe the Viterbi Algorithm;

- state index - i, $i = 0, 1, ..., M^{\nu} - 1$
- state metric for state i at sampling time $k \triangleq \mathcal{C}_{i,k}$ (sometimes called the "path metric")
- set of previous states to state $i \triangleq J_i$ (that is, states that have a transition into state i)
- $\tilde{y}_k(j \to i)$ noiseless output in going from state j to state i. (i.e., the value of the trellis branch, which is just x_k when $H(D) = 1$ for coded systems)
- branch metric in going from state j to state i at time k, $\Delta_{j,i,k} \triangleq |z_k - \tilde{y}_k(j \to i)|^2$
- $\bar{j}_i$ = "survivor" path - the path that has minimum metric coming into state i.

Definition 9.1.1 (Viterbi Detection) *Set $\mathcal{C}_{-1} = \mathcal{C}_{i,-1} = 0$ and recursively minimize for each state $i = 0, ..., 2^{\nu} - 1$ (or $M^{\nu} - 1$ for partial response) at each time $k \geq 0$*

$$\mathcal{C}_{i,k} = \min_{j \in J_i} \left[\mathcal{C}_{j,k-1} + \Delta_{j,i,k}\right] \quad . \tag{9.12}$$

There are 2^{ν} (M^{ν} pr) new $\mathcal{C}_{i,k}$'s updated and $2^{\nu} \cdot 2^{b}$ ($M^{\nu+1}$ pr) branch metrics, $\Delta_{j,i,k}$, computed at each sampling time k. There are also 2^{ν} (M^{ν} pr) surviving paths.

The survivor paths saved, $\bar{j}_i$, can be infinite length. In practice, they are truncated (typically length 5ν). There is very little loss associated with such truncation in practice. Furthermore, to prevent overflow of the accumulated metrics, the same (negative) amount can be added, periodically, to the all the metrics to keep the accumulated metrics within precision boundaries. This constant could be equal in magnitude to the smallest metric, thus always forcing this smallest metric to zero. Equivalently, circular overflow may be used as long as the maximum metric difference is less than one-half full dynamic range.

The basic operation in the Viterbi Decoder is to **add** a branch metric to a previous state metric, to **compare** the result with other such results for all other paths into the same state, and to **select** the path with the lowest metric. Selection replaces the metric for that state with the new metric for subsequent iterations, and augments the survivor path by the corresponding symbol on the selected path. This operation is often abbreviated **Add-Compare-Select (ACS)**.

9.1.5 Viterbi Example for Convolutional Code

Convolutional codes (Chapter 10) such as the example studied in Section 8.1 can be applied to the BSC, in which case the sequences to be compared in MLSD are binary and the Hamming distance or number of bits that differ replaces the squared distance as the cost function in the Viterbi Algorithm. If the convolutional code does use bipolar transmission over an AWGN (with $0 \to -1$ and $1 \to +1$), then the cost remains the squared distance. The smallest distance for the BSC is the minimum number of bits different between any two different code sequences and is sometimes called d_{free} to distinguish it from d_{min} for the AWGN. However, the two are essentially the same and related by

$$d_{min}^2 = 4 \cdot d_{free} \bar{\mathcal{E}}_{\boldsymbol{x}} \quad . \tag{9.13}$$

This chapter will sometimes simplify notation and refer to d_{free} as d_{min} with the context-dependent distinction being the BSC or the AWGN.

This section returns to the 4-state convolutional code of Section 8.1 and illustrates decoding on the BSC channel with the Viterbi algorithm. Figure 9.7 illustrates an output sequence above a trellis and the corresponding costs and path selected by a Viterbi detector. Readers may find this second example useful in understanding Viterbi detection, and are encouraged to trace the calculations within the trellis to see how the highlighted path is selected. The costs are integers for the BSC. It is assumed that the encoder started in state 0.

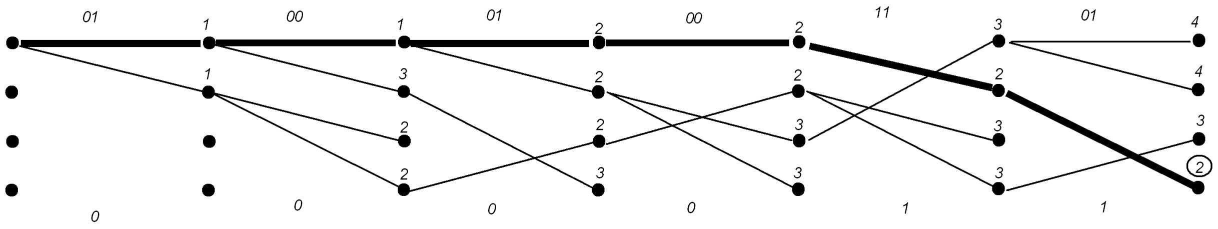


Figure 9.7: Viterbi decoder example.

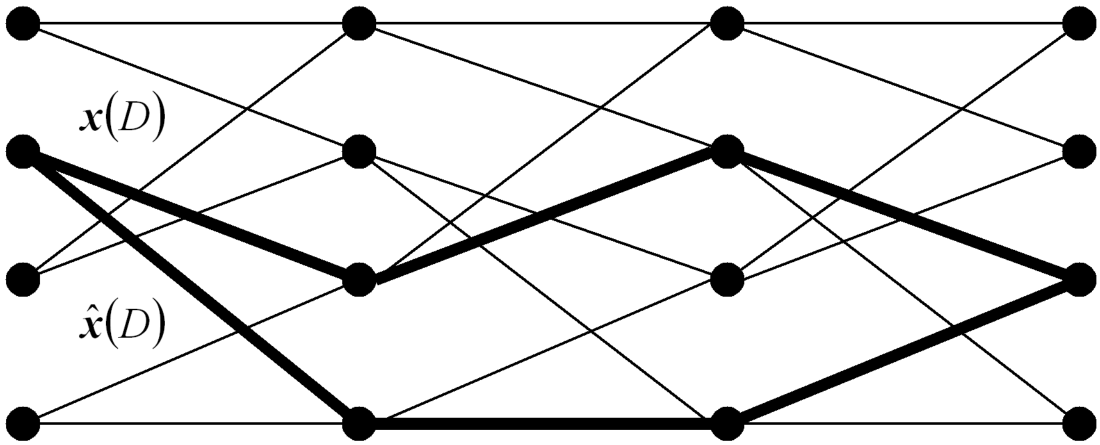


Figure 9.8: Example of trellis error event.

9.2 Analysis of MLSD

Analysis of MLSD conceptually reduces to the computation of the distance spectrum $d_{min}(i), i = 0, 1, ...$ with $d_{min}(0) \stackrel{\Delta}{=} d_{min}$ and the corresponding multiplicities N_i with $N_0 \stackrel{\Delta}{=} N_e$. Determination of $d_{min}(i)$ and N_i can be complex. Several methods exist for deriving the distances, the nearest-neighbor counts, and even the corresponding bit-error probabilities. Early versions of this text enumerated special trellis-searching procedures, an alternative method of transfer functions, and various input/event enumeration procedures. The author found them of great theoretical interest, but these methods required considerable effort on the part of the communication engineering student to learn and master. After expending such effort, the engineer learns that a wealth of good codes have been found and analyzed without need for all the analysis theory. The engineer can use these codes, design encoders and decoders, and understand these code's performance improvement without need for excessive theory.

Thus experience of the professor/author was to take a different approach in this text that focuses instead on the use of known codes and less on the creation of new codes and analysis. We thus trust the many fine coding theorists to have done good work with their searching procedures and to make use of their effort rather than to understand it in detail.

This section begins with a discussion of error events, an alternative to trying to enumerate every codeword and its nearest neighbors. With error events, analysis investigates directly the type of errors that can occur. After some discussion of error events and how to analyze them, as well as the corresponding code performance, this section proceeds to provide a couple of examples that are among the simplest ($1 + D$ partial response and 4-state convolutional code) to analyze in detail as an example of what is useful and the amount of work necessary to get that useful information. Finally, this section briefly mentions transfer functions and their use, but quickly proceeds to Ginis's sequential encoder analysis program, which can be fed any trellis for a specific sequential encoder and will produce/search the trellis for the distance spectrum, the average number of nearest neighbors, and even the average number of bit errors. The mechanization in a program allows the communications engineer to analyze the system in terms of salient parameters without having to investigate the detailed code structure.

9.2.1 Error Events

Error events are used by communication theorists to enumerate the various ways in which a MLSD receiver can incorrectly decide which message sequence has been transmitted. Prior to formal definition, Figure 9.8 illustrates two sequences corresponding to one possible error event through a 4-state trellis. The decoder decides the incorrect sequence, which differs only in 3 symbol periods from the correct sequence because the channel output was such that it looked more like the incorrect sequence.

Definition 9.2.1 (Error Event) *An* **error event** *occurs when MLSD does not detect the correct sequential-encoder trellis sequence $\hat{\boldsymbol{x}}(D) \neq \boldsymbol{x}(D)$ and the trellis paths $\hat{\boldsymbol{x}}(D)$ and $\boldsymbol{x}(D)$ first diverge at some specific time point k where $\hat{\boldsymbol{x}}_k \neq \boldsymbol{x}_k$. For the time-invariant trellis with $\nu \geq 1$ of this text, k can always be assumed to be 0 without loss of generality.*

The two paths are assumed to merge again at some later time $k + L_y$ where $L_y < \infty$ is the **length of the error event in the trellis** *and may tend towards infinity in some codes for some error events, but the error event must always have trellis paths merge at some later finite time. The* **length of the error event** *is $L_x < \infty$ where $\hat{\boldsymbol{x}}_m = \boldsymbol{x}_m \; \forall \; m \geq k + L_x$, and may also tend towards infinity. $L_x \leq L_y$ for such finite-length error events.*

The **trellis error event sequence** *between the noiseless sequence, $\tilde{\boldsymbol{y}}_k$ (=$\boldsymbol{x}_k$ when $H(D) = I$) and the MLSD sequence, $\hat{y}_k$, is*

$$\boldsymbol{\epsilon}_y(D) \stackrel{\Delta}{=} \tilde{\boldsymbol{y}}(D) - \hat{\boldsymbol{y}}(D) = H(D)\boldsymbol{x}(D) - H(D)\hat{\boldsymbol{x}}(D). \quad . \tag{9.14}$$

With the definition of the **error event sequence** *$\boldsymbol{\epsilon}_x(D) \stackrel{\Delta}{=} \boldsymbol{x}(D) - \hat{\boldsymbol{x}}(D)$, then (9.14) becomes*

$$\boldsymbol{\epsilon}_y(D) = H(D)\boldsymbol{\epsilon}_x(D) \quad . \tag{9.15}$$

When there is no partial response and $H(D) = I$, then $\boldsymbol{\epsilon}_y(D) = \boldsymbol{\epsilon}_x(D)$ and $L_y = L_x$. For partial response $L_y = L_x + \nu$. One may also define an encoder input or message error event sequence $\epsilon_m(D) \stackrel{\Delta}{=} m(D) - \hat{m}(D)$, where the subtraction[3] is modulo M. For the BSC with binary encoder output sequence of 0's and 1's for $\boldsymbol{x}(D)$, which is sometimes called $\boldsymbol{v}(D) = \boldsymbol{x}(D)$ for convolutional codes or block codes with $\bar{b} < 1$, the subtraction in the error event $\boldsymbol{\epsilon}_v(D) = \boldsymbol{v}(D) - \hat{\boldsymbol{v}}(D)$ is vector modulo $M = 2$.

The inclusion of the possibility of partial-response channels, which are like codes and described by a trellis and sequences, leads to the slight complication in the definition of the error event above. This complication forces the distinction between channel-input and channel-output error events above and admits the possibility that $L_y > L_x$. For instance for the binary $1 + D$ channel, an input error event of $\epsilon_x(D) = 2$ and $L_x = 1$ produces the channel output error event $\epsilon_y(D) = 2 + 2D$ with $L_y = 2$. These two descriptions, $\epsilon_x(D) = 2$ and $\epsilon_y(D) = 2 + 2D$, correspond to the same input error of the sequence $\hat{x}(D) = -1$ being the decoder output when the correct sequence was $x(D) = 1$.

9.2.2 Performance Analysis with Error Events

The probability of sequence error is

$$P_e = Pr\{\hat{\boldsymbol{x}}(D) \neq \boldsymbol{x}(D)\} \tag{9.16}$$

and is minimized by MLSD when all sequences are equally likely. P_e is also the overall probability that any of the error events in the set of all possible error events for a specific code can occur. The designer can upper bound the error probability by enumerating all possible error events, then upper bounding the probability of each, and finally summing all these probabilities. Clearly not all error events may have the same probability of occurrence. Some events have a smaller distance between them and thus have greater probability of occurrence. Figure 9.9 illustrates two error events for a binary $1 + D$ channel that have the same distance $d^2_{min} = 8$, but have different probabilities. The two error events in Figure 9.9 are $\epsilon_x = 2$ and $\epsilon_x = 2 - 2D$. The first has probability of occuring $.5Q(\sqrt{2}/\sigma)$ because the value of $\epsilon_{x,0} = 2$ can only occur half the time when $x_0 = +1$, and never when $x_0 = -1$. That is, the probability

[3] It is possible for the input error event to have infinite length, but corresponds to a situation that is known as a catastrophic encoder and is not a code to be used in practice.

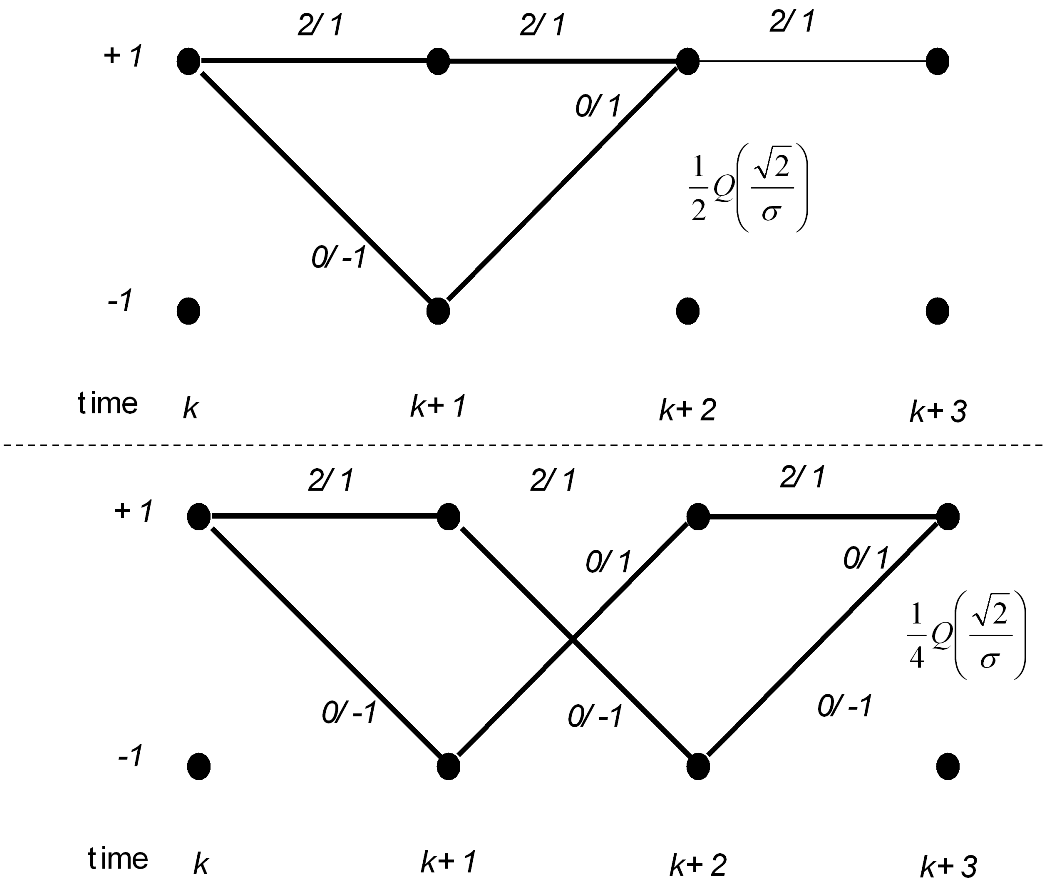

2/ 1
2/ 1
2/ 1
+ 1
0/ 1
$\frac{1}{2}Q\left(\frac{\sqrt{2}}{\sigma}\right)$
0/ -1
-1
time
k
k+ 1
k+ 2
k+ 3
2/ 1
2/ 1
2/ 1
+ 1
0/ 1
0/ 1
$\frac{1}{4}Q\left(\frac{\sqrt{2}}{\sigma}\right)$
0/ -1
0/ -1
0/ -1
-1
time
k
k+ 1
k+ 2
k+ 3

Figure 9.9: Illustration of error events with same d_{min} and different probabilities.

that 2 is allowed is $1/2$ at any given point in time for the $1+D$ channel. Similarly, the input error event $2-2D$ is allowed (.5)(.5)=.25 of the time, corresponding only to the error event sequence of length (L_x) two $1-D$, which is one of four equally likely messages of length two. The number of nearest neighbors N_e for this trellis will include terms like these two in determining the multiplier of the Q-function for probability of error bounding.

In general, the probability of error for any MLSD on the AWGN channel was previously bounded in Equation (9.8). For the BSC, a similar expression exists, which is developed here.

An error event confusing binary codeword m with binary codeword $\tilde{m}$ with Hamming distance between them d_h has probability

$$P\{\varepsilon_{m\tilde{m}}\} = \left(\begin{array}{c} d_h \\ \lceil \frac{d_h}{2} \rceil \end{array} \right) \cdot p^{\lceil \frac{d_h}{2} \rceil} \cdot (1-p)^{n-\lceil \frac{d_h}{2} \rceil} \quad . \tag{9.17}$$

Lemma 9.2.1 (Upper bound) *The following relationship holds for* $p \leq \frac{1}{2}$

$$\left(\begin{array}{c} d_h \\ \lceil \frac{d_h}{2} \rceil \end{array} \right) \cdot p^{\lceil \frac{d_h}{2} \rceil} \cdot (1-p)^{n-\lceil \frac{d_h}{2} \rceil} \leq \left[4p(1-p)\right]^{d_h/2} \quad . \tag{9.18}$$

proof: (By induction)
For even values of d_h, the "ceiling function" notation is not necessary and it is clear that

$$\left(\begin{array}{c} 2 \\ 1 \end{array} \right) < 4 \quad . \tag{9.19}$$

For even numbers, it suffices then to show that (recalling $n \geq d_h$) if $\left(\begin{array}{c} 2k \\ k \end{array} \right) < 4^k$, then this same relation holds for $k \rightarrow k+1$.

$$\begin{aligned} \left(\begin{array}{c} 2k+2 \\ k+1 \end{array} \right) &= \frac{(2k+2)\cdot(2k+1)\cdots k+2}{(k+1)\cdots 1} && (9.20) \\ &= \frac{(2k+2)\cdot(2k+1)}{(k+1)^2} \cdot frac(2k)\cdot(2k-1)\cdots k+1k\cdots 1 && (9.21) \\ &< \frac{(2k+2)\cdot(2k+1)}{(k+1)^2} \cdot 4^k && (9.22) \\ &< 4\cdot 4^k = 4^{k+1} \quad . && (9.23) \end{aligned}$$

For odd $d_h = 2k+1$, the induction proof first notes

$$\left(\begin{array}{c} 3 \\ 2 \end{array} \right) \sqrt{p} < \frac{3\cdot\sqrt{2}}{\sqrt{2}} = 3 < 4^{1.5} = 8 \quad . \tag{9.24}$$

For odd numbers, it suffices then to show that (recalling $n \geq d_h$) if $\left(\begin{array}{c} 2k-1 \\ k \end{array} \right) < 4^k$, then this same relation holds for $k \rightarrow k+1$.

$$\begin{aligned} \left(\begin{array}{c} 2k+1 \\ k+1 \end{array} \right) &= \frac{(2k+1)\cdot(2k)\cdots k+2}{(k+1)\cdots 1} && (9.25) \\ &= \frac{(2k+1)\cdot(2k)}{((k+1)^2} \cdot frac(2k-1)\cdot(2k-2)\cdots(k+1)k\cdots 1 && (9.26) \\ &< \frac{(2k+1)\cdot(2k)}{(k+1)^2} \cdot 4^k && (9.27) \\ &< 4\cdot 4^k = 4^{k+1} \quad . && (9.28) \end{aligned}$$

QED.

The expression

$$P\{\varepsilon_{m\tilde{m}}\} < [4p(1-p)]^{d_h/2} \tag{9.29}$$

is a special case of what is known as the Bhattacharya Bound (or B-Bound), which is not necessary nor used in this text. [4]

The probability of error for the BSC can then be bounded with in a fashion nearly identical to the AWGN using the BSC recognizing that all error events for the BSC have integer distances, counting from the smalles d_{free} upward:

$$P_e \leq \sum_{d=d_{free}}^{\infty} a(d) \left[\sqrt{4p(1-p)}\right]^d \quad , \tag{9.31}$$

with $a(d)$ enumerating the number of error events of weight d. Equation (9.31) can often also be approximated by just the first, or perhaps the first few terms, leading to

$$P_e \approx N_e \cdot \left(\begin{array}{c} d_{free} \\ \lceil \frac{d_{free}}{2} \rceil \end{array} \right) \cdot p^{\lceil \frac{d_{free}}{2} \rceil} \approx N_e \left[4p(1-p)\right]^{d_{free}/2} \quad , \tag{9.32}$$

effectively a nearest-neighbor union bound for the BSC. This probability is the probability of a sequence error and minimized. For convolutional codes, the probability of individual bit errors (rather than of sequence errors) may be of more interest.

Probability of bit error for either the AWGN or the BSC requires more information about the input message bit to codeword mappings of the sequential encoder. Essentially, the designer can determine the function

$$a(d,b) \stackrel{\Delta}{=} \text{number of error events of distance } d \text{ that correspond to } b \text{ input bit errors on average.} \tag{9.33}$$

The on-average part of the definition only needs interpretation when partial-response channels are used, and otherwise $a(d,b)$ is a pure enumeration of input to output mappings and associated input bit errors. The probability of error can actually be bounded according to

$$\bar{P}_b < \frac{1}{b} \sum_{d=d_{free}}^{\infty} \sum_{b=1}^{\infty} b \cdot a(d,b) \cdot \left[\sqrt{4p(1-p)}\right]^d \tag{9.34}$$

(the b divider in front is the number of bits per symbol, and the sum index b is just a variable). One then determines that the average total number of bit errors per error event (when all input messages are equally likely and independent from symbol to symbol) is

$$N_b = \sum_{b=1}^{\infty} b \cdot a(d_{free}, b) \quad . \tag{9.35}$$

and

$$\bar{P}_b \approx \frac{N_b}{b} \cdot \left(\begin{array}{c} d_{free} \\ \lceil \frac{d_{free}}{2} \rceil \end{array} \right) \cdot p^{\lceil \frac{d_{free}}{2} \rceil} \approx \frac{N_b}{b} \left[4p(1-p)\right]^{d_{free}/2} \quad . \tag{9.36}$$

For the AWGN, again,

$$\bar{P}_b \approx \frac{N_b}{b} \cdot Q\left[\frac{d_{min}}{2\sigma}\right] \quad . \tag{9.37}$$

[4] The B-Bound states more generally that

$$P\{\varepsilon_{m\tilde{m}}\} \leq \sum_{\boldsymbol{z}} \sqrt{\mathrm{p}_{\boldsymbol{y}/\boldsymbol{x}}(\boldsymbol{z}, \boldsymbol{x}_{\tilde{m}}) \mathrm{p}_{\boldsymbol{y}/\boldsymbol{x}}(\boldsymbol{z}, \boldsymbol{x}_m)} \quad . \tag{9.30}$$

9.2.3 Example Exact Analysis of the $1 + D$ partial response channel

For the partial response channel with $H(D) = 1 + D$, it is trivial to determine by inspection (even for $M > 2$) that the minimum distance is thus $d_{min}^2 = d^2 + d^2 = 2^2 + 2^2 = 8$, or $d_{min} = \sqrt{2} \cdot d = 2\sqrt{2}$. Then $\left(\frac{d_{min}}{2\sigma_{pr}}\right)^2 = \left(\frac{\sqrt{2}d}{2\sigma_{pr}}\right)^2 =$ MFB. Thus, with MLSD, the MFB can be attained, with finite real-time complexity and without equalization! Contrast this detector with the ZFE, which suffers infinite noise enhancement on this channel, and the ZF-DFE which is even with precoding 3 dB inferior to MLSD.[5]

Analysis by enumeration of input strings

For the $1 + D$ channel with binary ($x_k = \pm 1$) inputs, the NNUB $\bar{P}_e$ expression is

$$\bar{P}_e \leq \bar{N}_e \cdot Q\left[\frac{d_{min}}{2\sigma_{pr}}\right] = \bar{N}_e \cdot Q\left[\frac{\sqrt{2}}{\sigma_{pr}}\right] \quad , \tag{9.38}$$

since the matched-filter bound performance level is achieved on this channel. $\bar{N}_e$ is

$$\bar{N}_e = \sum_i p(i) \cdot N_i \quad , \tag{9.39}$$

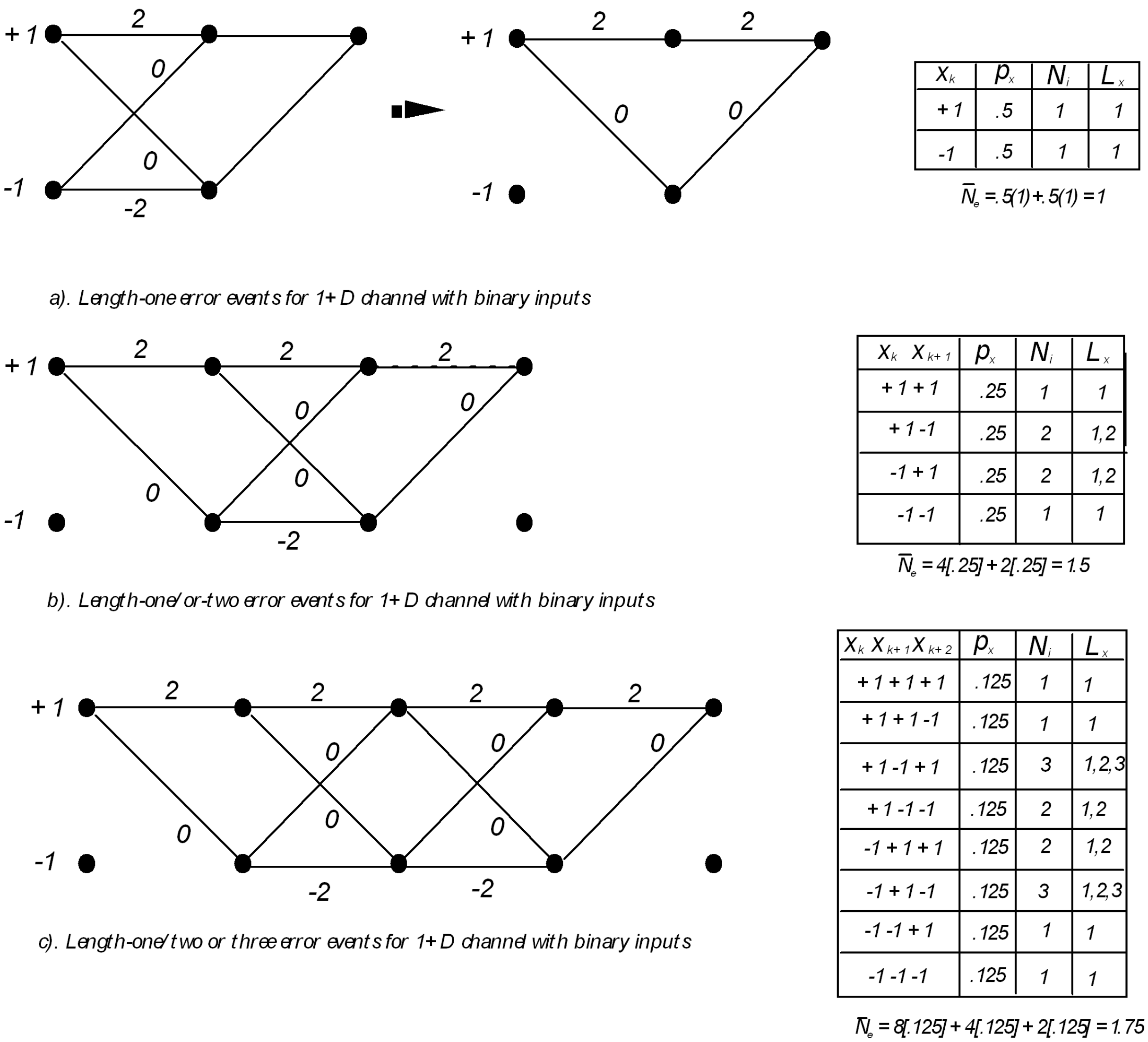
where $p(i)$ is the probability of the i^{th} input. However, with MLSD, there are an infinite number of input sequences. Figure 9.10 illustrates error events of lengths 1,2, and 3 with a trellis. For an input error event to be of length L_x, it must correspond to two paths that diverge (i.e., have a nonzero first entry) in the input in the first trellis stage, and merge exactly $L_y = L_x + \nu = L + 1$ stages later in the trellis. (If it merges sooner or diverges later, it is not of length L_x.) [6] Symmetry permits consideration of only those error events beginning in one of the two states (+1 is used in Figure 9.10), because the error events for the other state will be of identical length, number, and distribution. The number of nearest neighbors and their associated probabilities will be the same for either terminating state of the output error-event sequence. This is because the input error-event sequence sample is 0 in the last ν stages (In Figure 9.10, $\nu = 1$). Analysis need only consider merges into the top state (+1) in Figure 9.10: more generally, analysis need only consider the first L_x stages and any final states into which a merge occurs, since all further stages in the trellis correspond to no differences on the inputs or $\epsilon_{x,k} = 0$ values. This analysis of sequence detection will assume that only those error events corresponding to minimum distance are included in $\bar{N}_e$ because it is difficult to include those of greater distance (even if they have a common decision boundary).

For length $L_x = 1$, the input +1 has only one neighbor at (channel-output) distance $d_{min} = 2\sqrt{2}$ and that is the input sequence -1. The input error event sequence is thus $\epsilon_x(D) = 2$ and the output error sequence is $\epsilon_y(D) = 2 + 2D$. The situation for the other input sequence (-1) is identical, so that there is only one nearest neighbor of length $L_x = 1$, on the average. Thus, as $\bar{N}_e(1) = .5(1) + .5(1)$, where the argument of $\bar{N}_e(L_x)$ is the length of the input error event sequence. For lengths $L_x \leq 2$, there are four possible sequences that begin with a nonzero input error event sequence sample at time (trellis stage) 0. (This analysis can consider only error events that begin at the sample time 0 in computing error probabilities because this is the time at which the decoder is presumed to be in error.) From the table included in Figure 9.10(b), there are 2 error events of length 2 and 4 of length 1. The input sequences $X(D) = \pm(1 - D)$ have two nearest neighbors each (one of length 1, $\epsilon_x(D) = \pm 2$ and one of length 2, $\epsilon_x(D) = \pm(2 - 2D)$), while $X(D) = \pm(1 + D)$ have only one nearest neighbor, each, of length 1, $\epsilon_x(D) = \pm 2$. Thus, the number of nearest neighbors is $\bar{N}_e(1, 2) = .25(2) + .25(2) + .25(1) + .25(1) = 1.5$. This computation reorganizes as

$$\bar{N}_e(1, 2) = \bar{N}_e(1) + \bar{N}_e(2) = 4(.25) + 2(.25) = 1.5 \quad . \tag{9.40}$$

[5] A higher error coefficient will occur for the Q-function with MLSD than in the strict MFB, which we will see later can be significant at higher values of M.

[6] This is sometimes called the analysis of the **first error** event, and tacitly assumes that all previous inputs have been correctly decoded. In practice, once an error has been made in sequence detection, it may lead to other future errors being more likely (error propagation) because the survivor metric coming into a particular state where the event merged is no longer the same as what it would have been had no previous error events occurred.



x_k	p_x	N_i	L_x
+1	.5	1	1
-1	.5	1	1

$\overline{N}_e = .5(1) + .5(1) = 1$

a). Length-one error events for 1+D channel with binary inputs

$x_k \; x_{k+1}$	p_x	N_i	L_x
+1 +1	.25	1	1
+1 -1	.25	2	1,2
-1 +1	.25	2	1,2
-1 -1	.25	1	1

$\overline{N}_e = 4[.25] + 2[.25] = 1.5$

b). Length-one/or-two error events for 1+D channel with binary inputs

$x_k \; x_{k+1} x_{k+2}$	p_x	N_i	L_x
+1 +1 +1	.125	1	1
+1 +1 -1	.125	1	1
+1 -1 +1	.125	3	1,2,3
+1 -1 -1	.125	2	1,2
-1 +1 +1	.125	2	1,2
-1 +1 -1	.125	3	1,2,3
-1 -1 +1	.125	1	1
-1 -1 -1	.125	1	1

c). Length-one/two or three error events for 1+D channel with binary inputs

$\overline{N}_e = 8[.125] + 4[.125] + 2[.125] = 1.75$

Figure 9.10: Binary error events of length 1, 2, and 3 for the $1 + D$ channel.

Error events of length 3 or less include only two new error-event sequences, $\epsilon_x(D) = \pm(2 - 2D + 2D^2)$ of length 3, and the rest of the error event events are just those that occurred for length 2 or less. Thus,

$$\bar{N}_e(1,2,3) = \bar{N}_e(1) + \bar{N}_e(2) + \bar{N}_e(3) = 8(.125) + 4(.125) + 2(.125) = 1.75 \quad . \tag{9.41}$$

In general, input error events (corresponding to output minimum distance) of length L_x are then given by

$$\epsilon_x(D) = \begin{cases} \pm(2 - 2D + 2D^2 - .... + 2D^{L_x-1}) & L_x \text{ odd} \\ \pm(2 - 2D + 2D^2 - .... - 2D^{L_x-1}) & L_x \text{ even} \end{cases} \tag{9.42}$$

with the corresponding channel-output error events as

$$\epsilon_y(D) = (1+D)\epsilon_x(D) = \begin{cases} \pm(2 + 2D^{L_x}) & L_x \text{ odd} \\ \pm(2 - 2D^{L_x}) & L_x \text{ even} \end{cases} \quad . \tag{9.43}$$

In general, the two error events of length L_x will contribute $\bar{N}_e(L_x) = 2 \cdot 2^{-L_x}$ to $\bar{N}_e$. Then

$$\bar{N}_e = 2 \sum_{L_x=1}^{\infty} 2^{-L_x} = 2(\frac{1}{1-.5} - 1) = 2 \quad , \tag{9.44}$$

so

$$\bar{P}_e \approx 2Q\left(\frac{\sqrt{2}}{\sigma_{pr}}\right) \quad . \tag{9.45}$$

Analysis by enumeration of error events (*d*=2)

It can be tedious to enumerate all input sequences for a partial-response channel to compute $\bar{N}_e$. Instead, a reduced form is the enumeration for linear partial response channels of the input error events, as in Equation (9.42). Again for the $1 + D$ partial-response channel, for any $M \geq 2$, these error events are the only ones that can produce the minimum distance of $d_{min} = 2\sqrt{2}$. The probability that an input error event sequence value at any sample time k can occur is just $\frac{M - \frac{|\epsilon_k|}{2}}{M} = \frac{M-1}{M}$. The error $\epsilon_k \neq 0$ or the error event would have ended earlier. Then, for any $M \geq 2$, the number of nearest neighbors is

$$\bar{N}_e = 2 \sum_{L_x=1}^{\infty} \left(\frac{M-1}{M}\right)^{L_x} = 2(M-1) \quad . \tag{9.46}$$

While the argument of the Q-function is the same as that in the MFB, the nearest neighbor count is at least a factor of $2(M-1)/[2(1-1/M)] = M$ larger. For large M, the increase in P_e can be significant, so much so, that there is very little improvement with respect to symbol-by-symbol detection with precoding.

To determine the average number of bit errors occurring in each error event, let us first assume that no precoding is used with sequence detection, but that adjacent input levels differ in at most one bit position when encoded. The average number of bit errors per error event was defined in Chapter 1 as

$$N_b = \sum_b b a(d_{min}, b) \quad . \tag{9.47}$$

which is equivalent to the expression

$$N_b = \sum_i n_{b\epsilon}(i) \mathrm{p}_\epsilon(i) \tag{9.48}$$

where $n_{b\epsilon}(i)$ is the number of bit errors corresponding to error event i and $\mathrm{p}_\epsilon(i)$ is the probability that this error event can occur. The average number of bit errors per error event for the duobinary partial-response channel can then be computed as

$$N_b = 2 \sum_{L_x=1}^{\infty} (L_x) \left(\frac{M-1}{M}\right)^{L_x} = 2M(M-1) \quad , \tag{9.49}$$

Then, the bit error rate is accurately approximated by

$$\bar{P}_b(\text{no precode}) \approx \frac{N_b}{b} Q\left[\frac{\sqrt{2}}{\sigma_{pr}}\right] = \frac{2M(M-1)}{\log_2(M)} Q\left[\frac{\sqrt{2}}{\sigma_{pr}}\right] \quad . \tag{9.50}$$

The coefficient of the Q function in this expression is unacceptably high. It can be reduced by using precoding, which leads to a maximum of 2 input bit errors, no matter how long the error event. Then $N_b = 2\bar{N}_e$, and

$$\bar{P}_b(\text{precode}) \approx \frac{4(M-1)}{\log_2(M)} Q\left[\frac{\sqrt{2}}{\sigma_{pr}}\right] \quad . \tag{9.51}$$

When $M = 2$, precoding does not reduce the bit error rate, but it does prevent a long string of bit errors from potentially occurring when a long error event does occur. Precoding is almost universally used with sequence detection on partial-response channels because of the practical reason of avoiding this "catastrophe." Also, as M increases above 2, the bit error probability is also significantly reduced with precoding. These results are summarized in the following formal definition and theorem.

> **Definition 9.2.2 (Quasi-Catastrophic Error Propagation)** *A controlled ISI channel, and associated input symbol encoding, is said to exhibit* **quasi-catastrophic error propagation** *if it is possible for an error event that produces minimum distance at the channel output to produce an infinite number of input symbol errors. With M-ary (power-limited) inputs to the controlled-ISI channel, necessarily, the probability of such a "catastrophic" occurrence is infinitesimally small.*

While the probability of an infinite number of input bit errors is essentially zero, the probability that a large finite number of bit errors will be associated with a single minimum-distance error event is not. This usually occurs with channels that exhibit quasi-catastrophic error propagation. This effect is undesirable and can be eliminated by changing the input encoding rule.

> **Theorem 9.2.1 (Precoding for Sequence Detection)** *By using the precoder, $\mathcal{P}(D)$, for a partial-response channel that permits symbol-by-symbol detection (i.e., no memory of previous decisions is required with this precoder of Section 3.7), the controlled-ISI channel with MLSD cannot exhibit quasi-catastrophic error propagation*
>
> **Proof:** Since the partial-response precoder of Section 3.7 makes the channel appear memoryless, then the input symbols corresponding to $\epsilon_{y,m} = 0$ in the interior of an infinite-length error event must correspond to $\epsilon_{x,m} = 0$ (because the outputs are the same and the receiver otherwise could not have made a memoryless decision, or in other words $\epsilon_{y,m} = \epsilon_{x,m} = 0$). Thus, no input symbol errors can occur when $\epsilon_{y,m} = 0$, as must occur over nearly every (but a finite number) of sample periods for partial-response channels with their integer coefficients, leaving a (small) finite number of input symbol errors. **QED.**

That is, good designers use precoding even with MLSD on partial-response channels – it limits long strings of error bursts.

9.2.4 Example Analysis of 4-state Convolutional code

For the 4-state convolutional code example that has been used several times in Chapter 8 and Chapter 9, the error events may be written as

$$\boldsymbol{\epsilon}_v(D) = \epsilon_u(D) \cdot G(D) \tag{9.52}$$

where all multiplication and addition is modulo-2 and $G(D) = [1 + D + D^2 \quad 1 + D^2]$ because the code is modulo-2 linear in combining inputs to get output bits. Thus, one notes in this special case of the convolutional code, the set of all possible trellis error events is the same as the set of nonzero codewords, because the set of possible message error events modulo 2 is trivially the same as the set of all input sequences. This greatly simplifies analysis. Essentially, the distance distribution is simply the numbers of ones in the nonzero codewords, and the values for N_d are simply determined by counting codewords with the same number of ones.

In addition to the D-Transform transfer function of the matrix generator, $G(D)$, there is a second type of scalar "transfer function" that can be used for bookkeeping purposes in the analysis of convolutional codes. This transfer function $T(W)$ is constructed as shown for the 4-state example in Figure 9.11. In this figure, the underlying state-transition diagram corresponding to the trellis has been redrawn in a convenient manner with all inputs, outputs, and states in binary. Any codeword is formed by tracing a path through the state transition diagram. The all zeros codeword is the self loop at state 00. Comparison to this codeword determines the other possible codewords, and their distances from this all-zeros codeword. Further, this set of distances is the same with respect to any other codeword. The lower half of the same figure redraws the diagram using a place keeper for the Hamming weight of the codeword segment along each branch. The Hamming weight on each branch is an exponent to the place-keeping variable W. This place keeping variable (with exponent) is construed as the argument of a transfer function. The Hamming weight of any codeword is the exponent of W in transfer function along the corresponding path between the input and output state. The output state is just the input state repeated. For this example, the minimum weight codeword, or equivalently d_{min}, comes from the W^5 path, so that $d_{min} = 5$.

However, there is much more information contained in this transfer function. Computing the transfer function can be done in several ways, but we use Mason's Gain formula here:

$$T(\mathrm{W}) = \frac{\sum T_k \Delta_k}{\Delta} \quad , \tag{9.53}$$

where

$$\Delta = 1 - (\text{sum loop gains}) + (\text{sum products of two non-touching loop gains}) - ... \quad , \tag{9.54}$$

$$\Delta_k = \Delta(\text{with the } k^{th} \text{ forward path removed}) \tag{9.55}$$

and

$$T_k = \text{gain of } k^{th} \text{ forward path} \quad , \tag{9.56}$$

for this example,

$$T(\mathrm{W}) = \frac{\mathrm{W}^5(1-\mathrm{W}) + \mathrm{W}^6}{1 - (\mathrm{W} + \mathrm{W} + \mathrm{W}^2) + \mathrm{W} \cdot \mathrm{W}} = \frac{\mathrm{W}^5}{1 - 2\mathrm{W}} = \mathrm{W}^5 \sum_{k=0}^{\infty} (2\mathrm{W})^k \quad . \tag{9.57}$$

Thus,

$$T(\mathrm{W}) = \mathrm{W}^5 \left[1 + 2\mathrm{W} + 4\mathrm{W}^2 + 8\mathrm{W}^3 + ...\right] \quad . \tag{9.58}$$

There is one nearest neighbor at distance 5, 2 nearest neighbors at distance 6, 4 at distance 7, and so on. Thus, the laborious trellis search could have been avoided by using the transfer function approach. (A similar approach could also have been used for partial response channel, with (squared) Euclidean distance replacing Hamming distance as the exponent of W.) Any permutation of the order of the output bits in the codeword symbols will not affect $T(\mathrm{W})$. Thus, a permutation or relabeling of outputs does not really affect the essential properties of a code, although the codewords will be different. One can also easily see that N_e (as the number of error event sequences) is the coefficient of $\mathrm{W}^{d_{min}}$ in $T(\mathrm{W})$ or

$$N_e = \frac{1}{d_{min}!} \frac{\partial T(\mathrm{W})}{\partial \mathrm{W}^{d_{min}}} |_{\mathrm{W}=0} \quad . \tag{9.59}$$

While the transfer function reveals many properties of a given convolutional code, it can be very difficult to compute for codes with more states, perhaps more difficult than just searching the trellis. This N_e is not the total number of symbol errors that occur at any particular symbol period, which is sometimes confused by other authors. The next subsection produces the desired N_e-like coefficient for probability of symbol error expressions.

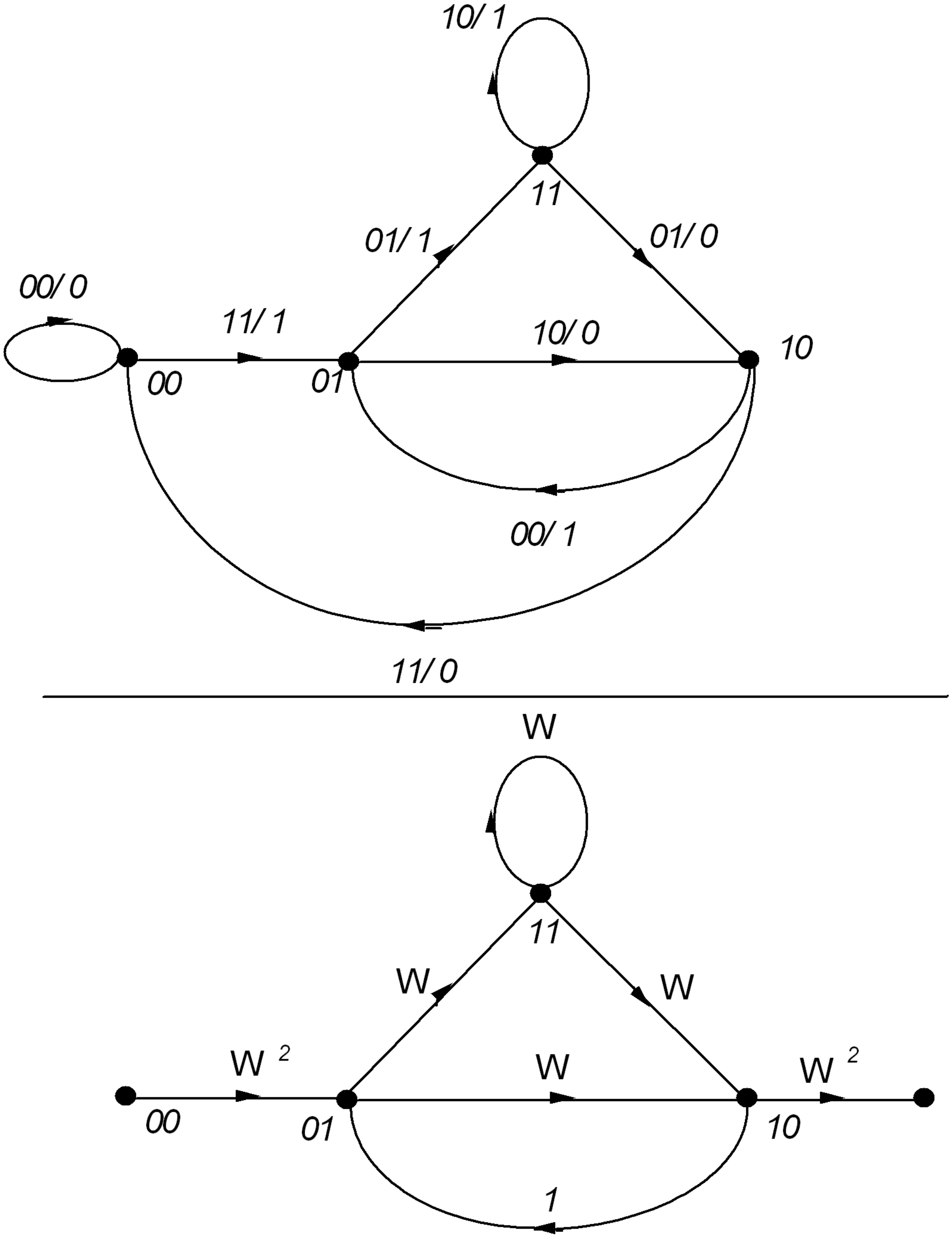


Figure 9.11: Transfer Function Construction for 4-State Example

Expanded Use of Transfer Functions

Analogous to the use of the variable W as a place holder for distance, transfer-function analysis can use L as a placeholder variable for the "length" of the error event (or codeword) under question, I as the corresponding placeholder for the number of nonzero bits on the corresponding input, and J as the number of symbol errors. Then each branch will have its gain multiplied by L, and by the power of I that corresponds to the number of input bits that were ones, and by the power of J that corresponds to the number of symbol errors. Figure 9.12 repeats Figure 9.11 with L, I, and J included. Again, using Mason's Gain formula,

$$T(\mathrm{W}, L, I, J) = \frac{\mathrm{W}^5 L^3 I J^3 (1 - \mathrm{W}LIJ) + \mathrm{W}^6 L^4 I^2 J^4}{1 - (\mathrm{W}L^2 IJ + \mathrm{W}LIJ + \mathrm{W}^2 L^3 I^2 J^2) + \mathrm{W}^2 L^3 I^2 J^2} \tag{9.60}$$

$$= \frac{\mathrm{W}^5 L^3 I J^3}{1 - \mathrm{W}L^2 IJ - \mathrm{W}LIJ} = \frac{\mathrm{W}^5 L^3 I J^3}{1 - \mathrm{W}LIJ(1+L)} \tag{9.61}$$

$$= \mathrm{W}^5 L^3 I J^3 \left(1 + \mathrm{W}LIJ(1+L) + (\mathrm{W}LIJ)^2 (1+L)^2 + ...\right) \tag{9.62}$$

which shows there is one error event of length 3, with $d_{min} = 5$, 1 corresponding input bit error, and 3 symbol errors; one error event of length 4 (and also one of length 5), $d = 6$, 2 input bit errors, and 4 input symbol errors; and so on. Also, with a little thought,

$$N_b = \frac{1}{d_{min}!} \frac{\partial T(\mathrm{W}, 1, I, 1)}{\partial \mathrm{W}^{d_{min}} \partial I} |_{\substack{\mathrm{W}=0 \\ I=1}} \tag{9.63}$$

and

$$N_e' \text{ symbol errors } = \frac{1}{d_{min}!} \frac{\partial T(\mathrm{W}, 1, 1, J)}{\partial \mathrm{W}^{d_{min}} \partial J} |_{\substack{\mathrm{W}=0 \\ J=1}} \tag{9.64}$$

Note that for partial-response systems earlier in this chapter, the N_e computed was a number of symbol errors directly, even though sequence error was minimized by the MLSD. Typically, this symbol-error N_e for partial-response is much more of interest than the number of possible sequence errors (which is usually infinite in partial response in any case). For convolutional codes, the number of symbol errors is probably less of interest than the probability of bit error, but Equation (9.64) illustrates the utility of transfer function analysis.

9.2.5 Ginis Code Analysis Program

This section describes an algorithm for the computation of the minimum distance between two paths in a trellis. An early version of this algorithm was suggested in the dissertation of Dr. Sanjay Kasturia, a former Ph.D. student at Stanford matriculating in 1989, but this particular program with many enhancements is the product of former Stanford Ph.D. student Dr. George Ginis in 2001. This algorithm has been programmed in MATLAB, and can be downloaded from the EE379B web page.

Algorithm Description

The algorithm can be described in general terms as a "Breadth-First-Search" method applied to finding converging paths. The trellis is "extended" by one state at each iteration, until the minimum distance is found.

More specifically, the algorithm maintains a list of pairs (s_1, s_2) of states together with a corresponding value. This value is the minimum distance, or cost Δ, between two paths of equal length starting at some common state and ending at s_1 and s_2 (see Fig. 9.13). Equivalently, this list of pairs is a list of partially constructed (or completed) error events for the trellis, for the common starting state. These two states s_1 and s_2 may differ, but can also be the same state, in which latter case cost is that associated with an error event. The algorithm consists of two stages: Initializing the list and then iterating this list once initialized.

Initializing the list:
Initially, the list (denoted as L) is set to the null set. For each state of the trellis, the list of pairs of states

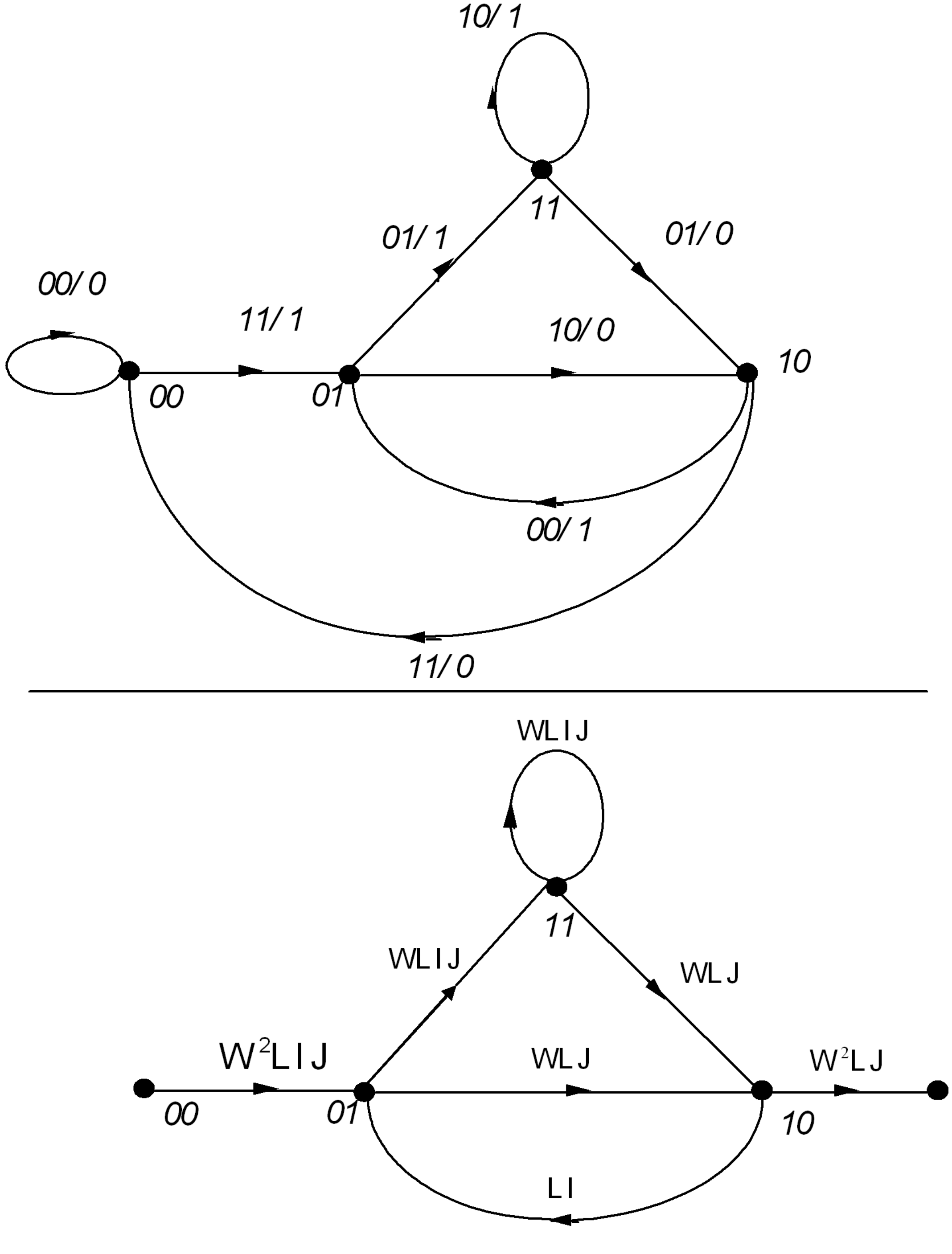


Figure 9.12: Expanded Transfer Function Computation

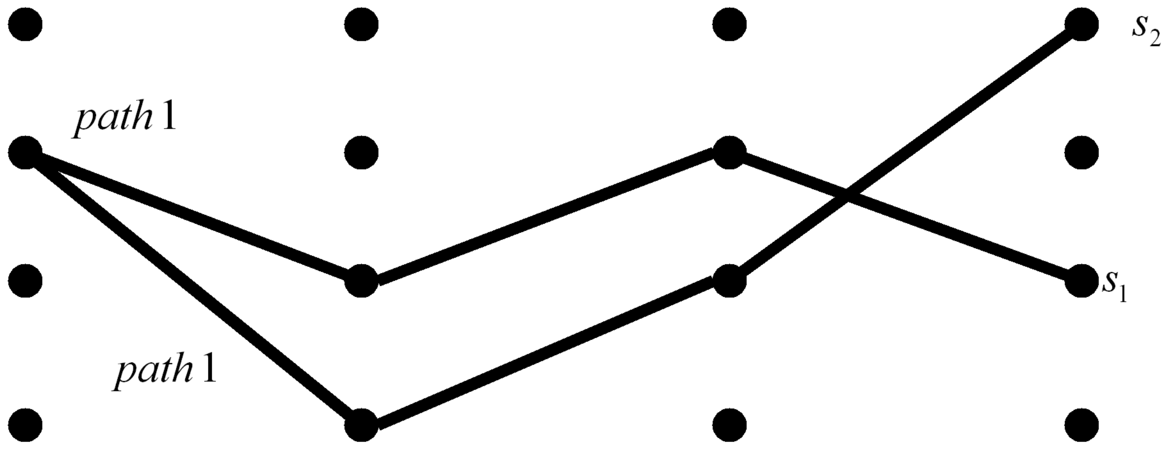


Figure 9.13: Illustration of Ginis's searching program.

that can be reached in one step are found. (A pair of states (s_1, s_2) is reachable from a state s_0 in one step, if there is one trellis branch/transition from s_0 to s_1, and another from s_0 to s_2.) For each pair the cost is called "d" in the program (Hamming distance for BSC or distance squared for AWGN) between the corresponding branches is computed. Each and every state pair not already in L are inserted into L together with their associated costs, d. If a state pair is already in L, but the existing associated cost is larger than the new cost d, then the cost is updated to the new d. At the end of the initialization, L contains all pairs of states that can be reached in one symbol period, or one iteration of the trellis, from any single state, together with their corresponding costs.

Before proceeding to the second stage, an upper bound on the minimum cost is defined (e.g. some large value, which is certain to exceed the minimum distance). Each time that an error event with lower cost than the previous lowest cost (which is initialized at this upper bound) is found, the upper-bound/lowest cost is replaced with this new smaller possible minimum distance. Also, a new list D is initialized to the null set. The list D eventually contains all the state-pairs that were searched.

Iterating the list:

In this second stage of the program, the pair of states in L with the smallest associated cost is picked, e.g. (s_1, s_2). If this cost is found to be larger than the upper bound, the program exits and the upper bound value is returned as the minimum cost. This is one of two possible exits from the algorithm. Otherwise, the pair is "extended" as described in the following: The set of all pairs of states reachable by extending s_1 and s_2 is first found, and for each new pair in this set the associated cost d is computed by adding this candidate cost to the old cost between s_1 and s_2. If any such pair has the new states $s_1 = s_2$, then there is a *termination of an error event.* In the case of termination of the error event, and if the upper bound is larger than d, this upper bound is updated to d. If the pair does not have same states, $s_1 \neq s_2$, then this new pair is checked whether it is already in L or in D. If it is neither in L nor in D, then it has not yet been extended, and it is inserted in L together with its associated d. If the pair is already in L, but with an old associated distance that is larger than d, then the distance is updated to new smaller cost d. In call cases, the original (s_1, s_2) that was extended is deleted from L (further extensions only research the same cost possibilities) and added to the set of searched pairs in D. The above procedure is repeated, until L becomes empty, which is the second exit referred to above, or until the upper bound on minimum cost is now below the d associated with all the remaining partial error events to be searched in L.

Using the MATLAB Program

Ginis' d_{min} program consists of several simple MATLAB m-files implementing the algorithm. The user only needs to be aware of few of these m-files: The list L is initialized (dmin_init.m), and the iteration steps are performed (dmin_iter.m). The output of the program is the minimum distance.

The program has 4 inputs in the following order (s,b,st-matrix,out-matrix):

- s = the number of states
- b = the number of input bits per symbol
- st-matrix = a matrix with next states
- out-matrix = a matrix with outputs

The elements of the matrices have the column number specifying the input, and the row number specifying the current state.

The distributed version of the program defines distance in the sense of the Hamming distance (see bdistance.m). Of course, in case a different cost metric is needed, this function can be easily modified to include that cost instead, see problem 9.11. The function bdistance included here is binary or Hamming distance – it could be replaced by Euclidean distance for other searches with the AWGN channel. Changing bdistance allows searching of both partial response trellises and the "trellis-code" trellises of Chapter 10 – see Problem 9.11.

The matlab programs are listed below:

dmin_init

```
% L=dmin_init(nstates,ninput,trellis_transition,trellis_output)
% Initialize list L. nstates are the number of trellis states,
% ninput are the number of input bits, and trellis_transition,
% trellis_output are
% determined from the trellis initialization. The list is modeled
% as a matrix, where each row represents an entry. The columns
% correspondingly represent state1, state2, distance, and
% membership in L (0) or D (1).
%
% George Ginis, April 2001

function L=dmin_init(nstates,ninput,trellis_transition,trellis_output)
  L=zeros(0,0);      % initialize to empty
  for k=1:nstates    % iterate for each state of the trellis
    for m=1:2^ninput   % iterate for all possible inputs
      [sd(m,1) sd(m,2)] = trellis_fn(k,m-1,trellis_transition,trellis_output);
      % produce next states and corresponding outputs
    end

    for l1=1:2^ninput-1        % form cross product
      for l2=l1+1:2^ninput
pointer = in_list(sd(l1,1),sd(l2,1),L);
dist=bdistance(sd(l1,2),sd(l2,2));  % compute distance between paths
if (pointer > 0)
  [dummy1, dummy2, distprev, mdummy] = extract_list(pointer,L);
  if (dist < distprev)   % if new distance is smaller,
                                 % update the entry
    L=setdist_list(pointer,dist,L);
  end
        else
  L=add_list(sd(l1,1),sd(l2,1),dist,L); % add pair and
                                                % distance to list
end
      end
    end
  end
```

trellis_fn.m

```
% [state_out,output] = ...
%    trellis_fn(state_in,input,trellis_transition,trellis_output)
% Produces next state and output of trellis. state_out is the next
% state (1 through the number of states) and output is the
% corresponding output. state_in is the previous state and input
% is the trellis input. trellis_transition and trellis_output are
% arguments of dmin_main.
%
% George Ginis, April 2001

function [state_out,output] = ...
         trellis_fn(state_in,input,trellis_transition,trellis_output)

  % produce next state of trellis
  state_out = trellis_transition(state_in,input+1);

  % produce trellis output
  output = trellis_output(state_in,input+1);

  % In the above, 1 is added to the input so that the matrix
  % indexing begins at 1 and not at 0.
```

setdist_list.m

```
% L=setdist_list(pointer,d,L)
% Set distance of entry of L pointed by pointer to d.
%
% George Ginis, April 2001

function L=setdist_list(pointer,d,L)

  L(pointer,3)=d;   % set distance
```

add_list.m

```
% L=add_list(s1,s2,d,L)
% Adds the pair (s1,s2) to the list L together with the distance d.
%
% George Ginis, April 2001

function L=add_list(s1,s2,d,L)

  ls=size(L,1);     % find number of entries of L

  L(ls+1,:)=[s1, s2, d, 0];   % add entry to the end of the table
```

delete_list.m

```
% L=delete_list(pointer,L)
% Deletes the entry of L indicated by pointer and adds it to the list D.
%
% George Ginis, April 2001

function L=delete_list(pointer,L)

  L(pointer,4)=1;   % set flag to 1 to indicate that element has been
                    % deleted from L and added to D.
```

dmin_iter

```
% [dmin,L]=...
%     dmin_iter(L,nstates,ninput,trellis_transition,trellis_output,ub)
% Returns minimum distance dmin of trellis. L is the list
% initialized by dmin_init, ub is some known upper bound on the
% minimum distance (e.g. parallel transitions), and nstates,
% ninput, trellis_transition, trellis_output are produced
% by dmin_init.
%
% George Ginis, April 2001

function [dmin,L]=...
 dmin_iter(L,nstates,ninput,trellis_transition,trellis_output,ub)
  pointer=findmin_list(L);  % find entry of L with minimum distance
  while (pointer > 0)     % loop while L is nonempty
    [s1,s2,d,m]=extract_list(pointer,L);
    if (d > ub)  % min. distance in L exceeds upper bound
      break;
    end

    % extend the pair of states by one step
    for m=1:2^ninput    % iterate for all possible inputs
      [sd1(m,1) sd1(m,2)]=trellis_fn(s1,m-1,trellis_transition,trellis_output);
      % produce next states and corresponding outputs from s1
    end
    for m=1:2^ninput    % iterate for all possible inputs
      [sd2(m,1) sd2(m,2)]=trellis_fn(s2,m-1,trellis_transition,trellis_output);
      % produce next states and corresponding outputs from s2
    end
    for l1=1:2^ninput        % find cross product
      for l2=1:2^ninput
newdistance = d + bdistance(sd1(l1,2),sd2(l2,2));
        % update distance
if (continueflag == 0)

  if (sd1(l1,1) == sd2(l2,1))  % error termination event
    if (newdistance < ub)
      ub = newdistance;   % update upper bound
    end
  else
    pointer_new=in_list(sd1(l1,1),sd2(l2,1),L);
    % check whether the new pair is an entry in the list
    if (pointer_new == 0) % if not in the list, then add entry
      L=add_list(sd1(l1,1),sd2(l2,1),newdistance,L);
    else
      [s1dummy,s2dummy,dp,m]=extract_list(pointer_new,L);
      % extract distance and list membership information
      if (m == 0)   % pair is in L
if (newdistance < dp)  % if new distance is smaller, update
  L=setdist_list(pointer_new,newdistance,L);
end
      end
    end
  end
      end
    end
    L=delete_list(pointer,L);  % delete entry from L, add to D
    pointer=findmin_list(L);  % find entry of L with minimum distance
  end
  dmin = ub;   % minimum distance equals upper bound
```

dmin_main

```
% [dmin,L] = dmin_main(nstates,ninput,trellis_transition,trellis_output)
% Function finding the minimum distance in a general trellis. The
% outputs include the minimum distance and a list L (useful for
% debugging purposes). The arguments are nstates (the number of
% states), ninput (the number of input bits), trellis_transition (a
% matrix describing the trellis state transitions) and trellis_output (a
% matrix describing the trellis outputs).
% The number of columns of the matrices equal the number of input
% bits, and the number of rows equal the number of trellis states.
%
% e.g. the 4-state convolutional code of fig. 8.6 has:
% nstates = 4
% ninput = 1
% trellis_transition=[1 2; 3 4; 1 2; 3 4]
% trellis_output=[0 3; 2 1; 3 0; 1 2]
%
% George Ginis, April 2001
function [dmin,L] = dmin_main(nstates,ninput,trellis_transition,trellis_output)
% Initialize the list
L=dmin_init(nstates,ninput,trellis_transition,trellis_output);

ub=100;          % initialize upper bound
% Iterate on the list
[dmin,L]=...
   dmin_iter(L,nstates,ninput,trellis_transition,trellis_output,ub);
```

extract_list

```
% [s1,s2,d]=extract_list(pointer,L)
% Extract from list L the entry pointed by pointer. Returns pair
% (s1,s2) and corresponding distance d.
%
% George Ginis, April 2001
function [s1,s2,d,m]=extract_list(pointer,L)
```

```
  s1=L(pointer,1);   % access elements
  s2=L(pointer,2);
  d=L(pointer,3);
  m=L(pointer,4);
```

findmin_list

```
% pointer=findmin_list(L)
% Searches L to find the pair (s1,s2) having the minimum distance. If
% found, it returns a pointer to the pair. If the list is empty, then 0
% is returned.
%
% George Ginis, April 2001
function pointer=findmin_list(L)
  ls=size(L,1);     % find number of entries of L
  min_d=inf;        % initialize distance
  pointer=0;        % initialize pointer
  for k=1:ls
    if (L(k,4) == 0)
      if (L(k,3) <= min_d)
min_d = L(k,3);   % update minimum distance
pointer = k;      % update pointer
      end
    end
  end;
  return;
```

in-list

```
% pointer=in_list(s1,s2,L)
% Searches L and D to find the pair (s1,s2). If found, it returns a
% pointer to the pair, otherwise 0 is returned.
%
% George Ginis, April 2001
function pointer=in_list(s1,s2,L)
  ls=size(L,1);     % find number of entries of L
  for k=1:ls
    if ( L(k,1) == s1 & L(k,2) == s2 )
      pointer=k;          % if found, return pointer
      return;
    end
  end;
  pointer=0;
  return;
```

bdistance

```
% d=bdistance(dec1,dec2)
% Returns binary distance between decimal numbers dec1 and dec2.
%
% George Ginis
function d = bdistance(dec1,dec2)
  xor_d = bitxor(dec1,dec2);   % compute binary xor
  bin_d = dec2bin(xor_d);      % convert xor to binary representation
  d = 0;                       % initialize distance
```

```
for k=1:length(bin_d)
  if (bin_d(k) ~= '0')
    d=d+1;                   % increase distance when 1 is encountered
  end
end
```

9.2.6 Rules for Partial-Response Channels

The following rules can be easily identified for attaining the MFB performance level with MLSD for partial response channels:

1. If $\nu = 1$, then MFB performance is always obtained with MLSD
2. If $\nu = 2$, then MFB performance is obtained if $\text{sign}(h_0 = -\text{sign}(h_2)$.

9.2.7 Decision Feedback Sequence Detection

In Decision Feedback Sequence Detection, the feedback section input used in decision-feedback calculations is determined from the survivor in the trellis. That is, the calculation of the branch metric $\Delta_{i,j,k}$ uses the survivor for the starting state as the input to the feedback section. This input to the feedback section thus will vary with the input state to the branch metric.

For example if a trellis code with a trellis were the input to the $1 + .9D^{-1}$ channel that appears throughout this book with 8.4 dB of SNR and MFB=10dB for uncoded transmission, then the branch metric calculations for Viterbi decoding of the code itself would use the last symbol in the survivor path into the initial state, so $.633x_{survivor,k-1}$ is the feedback section output. Then, the coding gain may be added to the 8.4 dB directly to estimate the performance of the system.

9.3 The APP Algorithm

Decoders need not initially make a hard decision about the transmitted symbol values. Instead, the decoder may compute a measure of the relative likelihood or probability of the various input message values. This measure is called the "soft information." The receiver makes a hard decision by selecting the input message value with the largest soft information. When two concatenated codes are used, the soft information may be useful in the "other code's" decoder. Successive passing of soft information between decoders in a convergent manner is called "iterative decoding" and examined in Section 9.6. This section describes a method for computing soft information.

This method is also an implementation of the exact MAP detector for decoding, although seldom used alone for just this MAP purpose.

9.3.1 MAP with the APP Algorithm

Maximum likelihood (or MAP) sequence detection chooses the sequence that has maximum likelihood ($p_{\boldsymbol{y}(D)/\boldsymbol{x}(D)}$) or maximum a posteriori ($p_{\boldsymbol{x}(D)/\boldsymbol{y}(D)}$) probability. Such a best detected sequence may not produce the minimum probability of symbol error for each of its constituent symbols, nor does it correspondingly produce minimum probability of bit or message error for each individual message. That is, while Section 9.2 computed a symbol (not sequence) probability of error P_e for MLSD, this particular quantity had not been minimized. A receiver could instead directly maximize $p_{m_k/\boldsymbol{y}(D)}$ for each symbol instant k. This is yet more complicated than MLSD for exact implementation. Such maximization may also lead to better system performance if symbol error probability is more important as a system measure than sequence error probability.

This subsection describes the **à posteriori probability (APP)** algorithm that computes the AP probabilities when a code can be described by a trellis diagram over any finite block of K symbol outputs of a code. These probabilities of course can then be searched if desired for the maximum soft value to produce the hard MAP estimate of m_k. Since m_k is probably unrelated to very distant symbols in most codes, K can be chosen finite without serious performance loss. The APP algorithm is sometimes also called the "forward-backward" algorithm or the **Bahl-Cocke-Jelinek-Ravin (BCJR)** algorithm.

The block of receiver samples to which the APP algorithm is applied is denoted $\boldsymbol{Y}_{0:K-1}$. K corresponding MAP values of m_k , $k = 0, ..., K-1$ and the corresponding values of $p_{m_k/\boldsymbol{Y}_{0:K-1}}$ $k = 0, ..., K-1$ are determined. The latter are the soft information.

A trellis description of the code at any time k within the block has a set of states $\mathcal{S}_k = \{0, ..., |S_k|-1 = 2^{\nu}-1\}$, with individual state denoted $s_k = j$ where $j = 0, ..., |S_k|-1$. Each branch between state $s_{k-1} = i$ and state $s_k = j$ in the trellis has a conditional probability of occuring that is a function of the code and denoted by

$$p_k(i,j) = p(s_{k+1} = j/s_k = i) \quad . \tag{9.65}$$

Further an individual branch has associated with it a probability distribution for the input data message that can occur along it

$$q_k(i,j,m_k) = p(m_k/s_k = i, s_{k+1} = j) \quad , \tag{9.66}$$

which usually trivializes to a value of 1 for the assigned message for that transition and 0 for all other symbols not assigned to that transition, but could for instance be $1/M'$ for each of M' equally likely parallel transitions, or generally some non-trivial distribution for the parallel transitions. A particularly useful probability in executing the APP is $p_k(s_k = i, s_{k+1} = j/\boldsymbol{Y}_{0:K-1})$ because each branch at time k usually corresponds to a unique value of m_k (that is, when there are no parallel transitions). The set of ordered pairs $B(m_k)$ contains all pairs of beginning and ending trellis branch states (denoted by the branch endpoint states i and j in ordered pair (i,j)) on which m_k can occur. The desired APP can be written as in terms of the beginning and ending states on which the specific input m_k occurs)

$$p_k(m_k/\boldsymbol{Y}_{0:K-1}) = \sum_{(i,j)\in B(m_k)} p_k(s_k = i, s_{k+1} = j/\boldsymbol{Y}_{0:K-1}) \tag{9.67}$$

$p_k(s_k = i, s_{k+1} = j/\boldsymbol{Y}_{0:K-1})$ can be computed by dividing $p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:K-1})$ by $p(\boldsymbol{Y}_{0:K-1})$. Since the values of $\boldsymbol{Y}_{0:K-1}$ are not a function of the *estimate* of $\boldsymbol{x}_k$, the normalization does not change the value that would be selected as the MAP estimate for any particular received $\boldsymbol{Y}_{0:K-1}$. Thus, this normalization is often ignored in the APP algorithm, which simply computes the distribution $p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:K-1})$ and then proceeds to the MAP estimate that maximizes the sum in (9.67). Individual decisions can subsequently be made for each m_k for $k = 0, ..., K-1$. Such individual decisions have minimum probability of symbol-message (not sequence) error.

When there are parallel transitions (as often occurs in the trellis codes of Chapter 10), the expression in (9.67) generalizes to

$$\begin{aligned}
p_k(m_k/\boldsymbol{Y}_{0:K-1}) &= \sum_{(i,j)\in B(m_k)} p_k(m_k/s_k = i, s_{k+1} = j, \boldsymbol{y}_k) \cdot p_k(s_k = i, s_{k+1} = j/\boldsymbol{Y}_{0:K-1}) && (9.68)\\
&= \sum_{(i,j)\in B(m_k)} \frac{p(\boldsymbol{y}_k, m_k/s_k = i, s_{k+1} = j) \cdot p(s_k = i, s_{k+1} = j)}{p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j) \cdot p(s_k = i, s_{k+1} = j)} \cdot p_k(s_{k-1} = i, s_k = j/\boldsymbol{Y}_{0:K-1})\\
&= \sum_{(i,j)\in B(m_k)} \frac{p(\boldsymbol{y}_k, m_k/s_k = i, s_{k+1} = j)}{p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j)} \cdot p_k(s_{k-1} = i, s_k = j/\boldsymbol{Y}_{0:K-1})\\
&= \sum_{(i,j)\in B(m_k)} \frac{p(\boldsymbol{y}_k/m_k, s_k = i, s_{k+1} = j) \cdot p(m_k/s_k = i, s_{k+1} = j)}{p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j)} \cdot p_k(s_{k-1} = i, s_k = j/\boldsymbol{Y}_{0:K-1})\\
&= \sum_{(i,j)\in B(m_k)} \frac{p(\boldsymbol{y}_k/m_k, s_k = i, s_{k+1} = j) \cdot p(m_k/s_k = i, s_{k+1} = j)}{p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j)} \cdot p_k(s_{k-1} = i, s_k = j/\boldsymbol{Y}_{0:K-1}) && (9.69)\\
&= \sum_{(i,j)\in B(m_k)} \frac{p(\boldsymbol{y}_k/m_k, s_k = i, s_{k+1} = j) \cdot q_k(i, j, m_k)}{p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j)} \cdot p_k(s_{k-1} = i, s_k = j/\boldsymbol{Y}_{0:K-1}) \quad . && (9.70)
\end{aligned}$$

where

$$\begin{aligned}
p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j) &= \sum_{m'_k} p(\boldsymbol{y}_k/m'_k, s_k = i, s_{k+1} = j) \cdot p(m'_k/s_k = i, s_{k+1} = j) && (9.71)\\
&= \sum_{\boldsymbol{x}'_k} p(\boldsymbol{y}_k/m'_k, s_k = i, s_{k+1} = j) \cdot q_k(i, j, m'_k) \quad . && (9.72)
\end{aligned}$$

To test the case where there are no parallel transitions, $q(i, j, \boldsymbol{x}_k)$ simplifies to equal to one for the particular value of $m'_k = m_k$ corresponding to the $i \to j$ branch, and is zero for all other values, and (9.72) simplifies on the right to $p(\boldsymbol{y}_k/m_k)$. Then this term is common to numerator and denominator in (9.70), and thus cancels, leaving (9.67).

The APP algorithm computes $p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:K-1})$ for either (9.67) or (9.70) as a function of 2 recursively updated state-dependent quantities generated by progressing forward and backward through the code trellis, and of a third non-recursive branch-dependent quantity that is a function that corresponds only to the time index k of interest.

The forward recursively computed quantity is the joint state and past output probability (tacitly a function of $\boldsymbol{Y}_{0:k}$)

$$\alpha_k(j) = p(s_{k+1} = j, \boldsymbol{Y}_{0:k}) \quad j = 0, ..., |S_{k+1}| - 1 \quad . \qquad (9.73)$$

The backward recursively computed quantity is the state-conditional future output distribution (tacitly a function of $\boldsymbol{Y}_{k+1:K-1}$

$$\beta_k(j) = p(\boldsymbol{Y}_{k+1:K-1}/s_{k+1} = j) \quad j = 0, ..., |S_{k+1}| - 1 \quad . \qquad (9.74)$$

The current branch probability and third quantity is (tacitly a function of $\boldsymbol{y}_k$)

$$\gamma_k(i, j) = p(s_{k+1} = j, \boldsymbol{y}_k/s_k = i) \quad , i = 0, ..., |S_k| - 1 \ , \ j = 0, ..., |S_{k+1}| - 1 \quad . \qquad (9.75)$$

If the detector has these 3 quantities available at any time k, then

$$p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:K-1}) = p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}, \boldsymbol{y}_k, \boldsymbol{Y}_{k+1,K-1}) \qquad (9.76)$$

$$
\begin{aligned}
&= p_k(\boldsymbol{Y}_{k+1,K-1}/s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}, \boldsymbol{y}_k) \cdot p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}, \boldsymbol{y}_k) \\
&= p_k(\boldsymbol{Y}_{k+1,K-1}/s_{k+1} = j) \cdot p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}, \boldsymbol{y}_k) \\
&= \beta_k(j) \cdot p_k(s_{k+1} = j, \boldsymbol{y}_k/s_k = i, \boldsymbol{Y}_{0:k-1}) \cdot p_k(s_k = i, \boldsymbol{Y}_{0:k-1}) \\
&= \beta_k(j) \cdot \gamma_k(i,j) \cdot \alpha_{k-1}(i) \qquad (9.77)
\end{aligned}
$$

Thus the quantity in (9.67) and (9.70) necessary for a MAP detector is the product of the $\alpha_{k-1}(i)$ on the state starting the branch, the $\gamma_k(i,j)$ on the branch, and the $\beta_k(j)$ at the end of that same branch. The APP first computes all γ, then executes the α recursions, and finally the β recursions.

γ **computation** The quantity $\gamma_k(i,j)$ is computed first for all branches according to

$$
\begin{aligned}
\gamma_k(i,j) &= p(s_{k+1} = j, \boldsymbol{y}_k/s_k = i) && (9.78) \\
&= p(s_{k+1} = j/s_k = i) \cdot p(\boldsymbol{y}_k/s_k = i, s_{k+1} = j) && (9.79) \\
&= p_k(i,j) \cdot \sum_{m'_k} p(\boldsymbol{y}_k/m'_k, s_k = i, s_{k+1} = j) \cdot q_k(i,j,m'_k) && (9.80)
\end{aligned}
$$

which for AWGN channel is also

$$
\gamma_k(i,j) = p_k(i,j) \cdot \sum_{m'_k} p_{\boldsymbol{n}_k}(\boldsymbol{y}_k - \boldsymbol{x}'_k(m'_k)) \cdot q_k(i,j,m'_k) \quad . \qquad (9.81)
$$

Equation (9.80) simplifies in the case of no parallel transitions to

$$
\gamma_k(i,j) = p_k(i,j) \cdot p_{\boldsymbol{y}_k/m_k} \quad . \qquad (9.82)
$$

α **computation** A forward recursion for $\alpha_k(j)$ is

$$
\begin{aligned}
\alpha_k(j) &= \sum_{i \in S_k} p(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}, \boldsymbol{y}_k) && (9.83) \\
&= \sum_{i \in S_k} p(s_{k+1} = j, \boldsymbol{y}_k/s_k = i, \boldsymbol{Y}_{0:k-1}) \cdot p(s_k = i, \boldsymbol{Y}_{0:k-1}) && (9.84) \\
&= \sum_{i \in S_k} p(s_{k+1} = j, \boldsymbol{y}_k/s_k = i) \cdot \alpha_{k-1}(i) && (9.85) \\
&= \sum_{i \in S_k} \gamma_k(i,j) \cdot \alpha_{k-1}(i) \quad . && (9.86)
\end{aligned}
$$

The initial condition is usually $\alpha_{-1}(0) = 1$ and $\alpha_{-1}(i \neq 0) = 0$ for trellises starting in state 0. Other initial distributions are possible including an "unknown" starting state with possibly uniform distribution of $\alpha_{-1}(i) = 1/|S_{-1}|$ $i = 0, ..., |S_{-1}| - 1$. The recursion in (9.86) essentially traces the trellis in a forward direction, very similar to the Viterbi algorithm computing a quantity for each state using the γ quantities on all the branches (which were computed first). Viterbi's add-compare-select is replaced by a sum-of-products operation.

β **computation** A backward recursion for $\beta_k(i)$ is

$$
\begin{aligned}
\beta_k(i) &= \sum_{j \in S_{k+2}} p(s_{k+2} = j, \boldsymbol{Y}_{k+1:K-1}/s_{k+1} = i) && (9.87) \\
&= \sum_{j \in S_{k+2}} p(s_{k+2} = j, \boldsymbol{y}_{k+1}, \boldsymbol{Y}_{k+2:K-1}/s_{k+1} = i) && (9.88) \\
&= \sum_{j \in S_{k+2}} p(\boldsymbol{y}_{k+1}/s_{k+1} = i, s_{k+2} = j, \boldsymbol{Y}_{k+2:K-1}) \cdot p(\boldsymbol{Y}_{k+2:K-1}, s_{k+2} = j/s_{k+1} = i) && (9.89)
\end{aligned}
$$

$$= \sum_{j \in S_{k+2}} \frac{p(s_{k+2} = j, \boldsymbol{y}_{k+1}/s_{k+1} = i, \boldsymbol{Y}_{k+2:K-1})}{p(s_{k+2} = j/s_{k+1} = i, \boldsymbol{Y}_{k+2:K-1})} \cdot p(\boldsymbol{Y}_{k+2:K-1}/s_{k+1} = i, s_{k+2} = j) \cdot p(s_{k+2} = j/s_{k+1} = i)$$

$$= \sum_{j \in S_{k+2}} p(s_{k+2} = j, \boldsymbol{y}_{k+1}/s_{k+1} = i) \cdot \beta_{k+1}(j) \tag{9.90}$$

$$= \sum_{j \in S_{k+2}} \gamma_{k+1}(i,j) \cdot \beta_{k+1}(j) \quad . \tag{9.91}$$

The boundary ("final") condition for β calculation follows from Equations (9.76) to (9.77) and determines a value of $\beta_{K-1}(j) = 1$ if the trellis is known to terminate in a final state, j, or $\beta_{K-1}(i) = 1/(|S_{K-1}| - 1),\ i = 0, ..., |S_{K-1}| - 1$ if the final state is not known and assumed to be equally likely. Other final values could also be used if some à priori distribution of the final states is known. The backward recursion is similar to backward Viterbi, again with sum-of-products operations replacing a add-compare-select operations.

EXAMPLE 9.3.1 (Wu's MAP decoder example) This example was provided by former Ph.D. student Zining "Nick" Wu, who also published a book through Kluwer (2000) on turbo codes and their use in storage media. Figure 9.14 illustrates Wu's example APP for the rate-1/2 convolutional code of Section 8.1 for a BSC with $p = .25$.

Again, the APP tracing of a trellis proceeds in much the same way as the Viterbi algorithm, except that costs along paths are multiplied and then summed at each node, rather than the Viterbi Algorithm's previous add, compare, and select operation, so multiply-and-add replaces add-compare-select. Figure 9.14 provides a listing of computations in both forward and backward directions (with α/β listed on each state). The channel inputs and outputs are identical to those for the Viterbi Algorithm Example in Section 9.1 (specifically see Figure 9.7) except that one additional bit error was introduced on the branch for $k = 3$, meaning 3 channel errors occurred. This code has d_{free} of 5 and so only corrects up to 2 errors with sequence detection and would have had a "tie" and resulting probability of sequence-detection error of 1/2, and an ambiguity in decoding.

At each state for the APP in Figure 9.14, the values of α_k/β_k appear while the value of γ_k appears on each branch. For instance, moving to the right from state $k = -1$ (where $\alpha_{-1}(0) = 1$), each of the upper emanating branches has the value

$$\gamma_0 = .0938 = \frac{1}{2}(.25)(.75) \quad , \tag{9.92}$$

where the .25 corresponds to the first code-output bit not matching the channel-output 0, while the .75 corresponds to second code-output bit matching the second channel-output bit of 1, and the factor of 1/2 represents p_{m_k} in (9.82). The product (.25).75 represents $p_{\boldsymbol{y}_k/m_k}$ in (9.82). The value of β_1 is available from the backward algorithm. The final distribution for the input bit is below the trellis and computed by summing the probabilities of the branches corresponding to a "1" and also a second sum for the "0" branches at each trellis stage, and then normalizing the two measures so that they add to 1. The largest then determines the decision.

With MAP detection, the extra channel bit error at $k = 3$ is corrected and the same input sequence of 000011 as in Section 7.1 is correctly detected even though the ML sequence detector would not have corrected this error. Generally speaking, there are simpler ways to augment the Viterbi algorithm than to use the MAP detector and still obtain soft information about each input bit. It is this soft information for each bit that allowed the extra channel bit error to be corrected. The next section discusses one such method.

The APP recursions could be executed for each bit of a message with $M > 2$ per sample. In this case, the probability of each bit being in error is minimized instead of the probability of message-symbol error being minimized. The essential difference in computation (hidden somewhat in Example 9.14 because

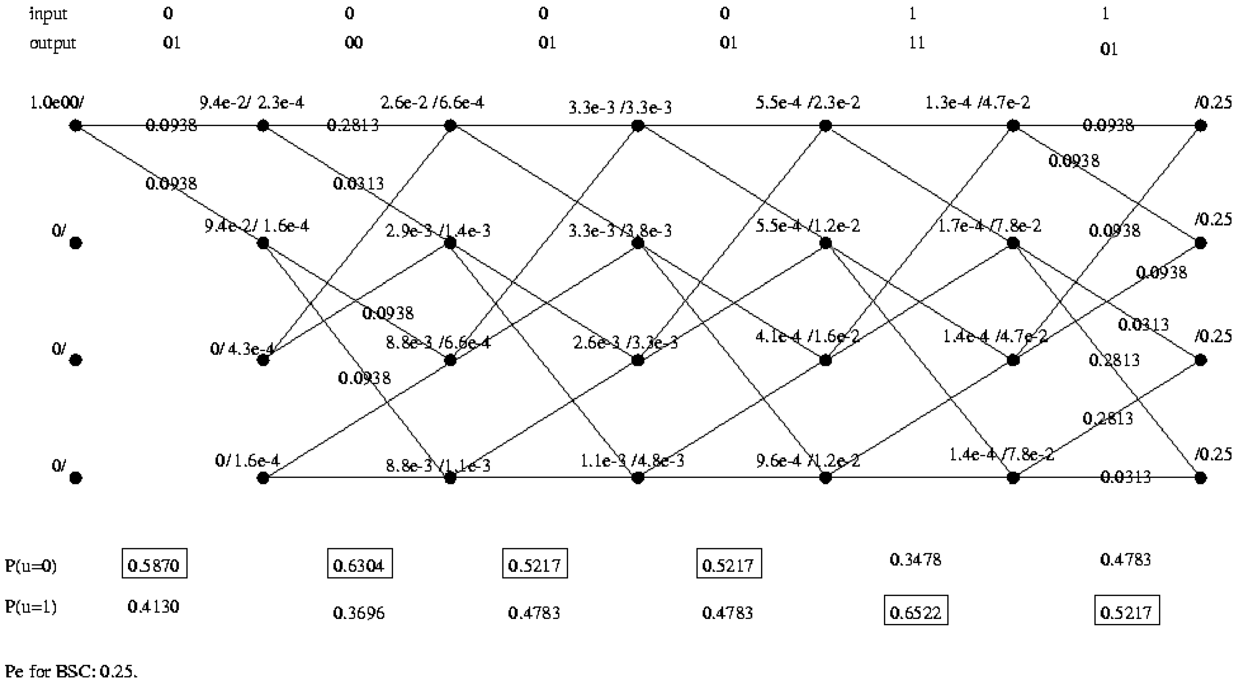


Figure 9.14: Example of MAP detector for rate 1/2 4-state Convolutional Code with BSC and $p = .25$

a message is a bit in that case) is that each of the γ terms is computed for a bit on a branch as if the others are either unknown (so each other bit on the branch equally likely to be 1 or 0) or as if some probability distribution for the other bits is already known. Examples 9.5.3 and 9.5.4 are illustrative examples of such calculations later in this Chapter in Section 9.5.

9.3.2 LOGMAP

The APP is often very well approximated by what is known as the **LOGMAP** algorithm, which removes multiplication expressions and replaces them with additions and the use of log tables. The LOGMAP algorithm begins with the logarithm of the equation

$$\alpha = \sum_i \alpha_i \gamma_i \quad , \tag{9.93}$$

to get

$$\ln(\alpha) = \ln\left(\sum_i \alpha_i \gamma_i\right) \quad . \tag{9.94}$$

The LOGMAP algorithms then attempt to efficiently compute the log of the sum of products. Each of the individual product's logarithm is the sum $\lambda_i \stackrel{\Delta}{=} \ln(\alpha_i) + \ln(\gamma_i)$. Then essentially, LOGMAP computes

$$\lambda = \sum_i e^{\lambda_i} \quad . \tag{9.95}$$

If there are only two terms, then

$$\lambda = \lambda_1 + \ln\left(1 + e^{\lambda_2 - \lambda_1}\right) = \lambda_1 + f(\lambda_2 - \lambda_1) \quad . \tag{9.96}$$

The function $f(\cdot)$ can be approximated and stored in a look-up table. Several successive uses allows computation of the entire sum. Note that terms for which $\lambda_2 << \lambda 1$ need not be computed because they contribute 0. Thus, by picking the largest of any two terms and letting it be λ_1 is a good way to simplify compoutation. If all terms are included in all sums, then LOGMAP is equivalent to the APP. Often inclusion of a few terms is sufficient. Various approximations to the function can make LOGMAP very simple to implement.

Some simplifications of LOGMAP use the approximation that the log of a sum of quantities is often dominated by its largest term, so

$$\log(\alpha) \approx \max_i \log(\alpha_i) + \log(\gamma_i) \quad . \tag{9.97}$$

One notes with some thought that the forward recursion essentially becomes the Viterbit algorithm in this case, leading to the next section.

9.4 Soft-Output Viterbi Algorithm (SOVA)

The **Soft-Output Viterbi Algorithm (SOVA)** was introduced by Hagenauer in 1989 as an augmentation to the usual Viterbi Detector to provide additional estimates of the $\boldsymbol{Y}_{0:K-1}$-conditioned probability of a given symbol value $\boldsymbol{x}_k$ at time k in the final surviving detected sequence. Such information suggests a reliability of each symbol $\boldsymbol{x}_k$ that was in the detected sequence and is generally useful for outer (or second) codes in concatenated systems.

The Viterbi Detection algorithm has been applied in this text to minimizing squared distances or Hamming distances for the AWGN or BSC respectively. Tacitly, direct use of squared distance or Hamming distance amounts to maximizing the natural logarithm of the probability distribution, often called a **Likelihood Function**,

$$L_{\boldsymbol{x}_k} \stackrel{\Delta}{=} \ln\left(p_{\boldsymbol{x}_k}\right) \quad . \tag{9.98}$$

The conditional likelihoods are correspondingly

$$\begin{aligned}
L_{\boldsymbol{x}_k/\boldsymbol{Y}_{0:K-1}} &\stackrel{\Delta}{=} \ln\left(p_{\boldsymbol{x}_k/\boldsymbol{Y}_{0:K-1}}\right) & (9.99)\\
L_{\boldsymbol{x}(D)/\boldsymbol{y}(D)} &\stackrel{\Delta}{=} \ln\left(p_{\boldsymbol{x}(D)/\boldsymbol{y}(D)}\right) & (9.100)\\
L_{\boldsymbol{y}(D)/\boldsymbol{x}(D)} &\stackrel{\Delta}{=} \ln\left(p_{\boldsymbol{y}(D)/\boldsymbol{x}(D)}\right) & (9.101)\\
L_{\boldsymbol{y}_k/\boldsymbol{x}_k} &\stackrel{\Delta}{=} \ln\left(p_{\boldsymbol{y}_k/\boldsymbol{x}_k}\right) \quad . & (9.102)
\end{aligned}$$

The sequence that maximizes the $L_{\boldsymbol{X}(D)/\boldsymbol{Y}(D)}$ function also minimizes the squared or Hamming distance from a received sequence on the AWGN or BSC respectively, and the latter distances are often easier to handle directly in computation than the exact likelihood function. The Viterbi detector inherently finds the best sequence and will not implement the MAP detector for an individual symbol value in a sequence. However, the two are often very close and Viterbi has noted that the sum of products operation in the APP algorithm is often dominated by the maximum term in the sum, and in this case the APP and Viterbi detector produce the same result[7]. However, the Viterbi detector does not normally provide "soft" information on the reliability, and so extra steps are added in SOVA to save and format such information before it is discarded in the internal calculations of the Viterbi detector.

For the general SOVA, it will be more convenient to consider the likelihood functions directly rather than squared or Hamming distance. The first advantage of considering the likelihood directly is that the Viterbi detector can be applied to the APP sequence criterion directly to maximize

$$\begin{aligned}
L_{\boldsymbol{x}(D)/\boldsymbol{y}(D)} &= L_{\boldsymbol{x}(D),\boldsymbol{y}(D)} - L_{\boldsymbol{y}(D)} & (9.103)\\
&= L_{\boldsymbol{y}(D)/\boldsymbol{x}(D)} + L_{\boldsymbol{x}(D)} - L_{\boldsymbol{y}(D)} \quad . & (9.104)
\end{aligned}$$

The last term in (9.104) is not a function of $\boldsymbol{x}(D)$ and thus cancels from all subsequent computations for a given received sequence $\boldsymbol{y}(D)$. Thus the likelihood $L_{\boldsymbol{x}(D),\boldsymbol{y}(D)}$ could be instead maximized. The term $L_{\boldsymbol{x}(D)}$ only cancels when all sequences are equally likely. If the receiver had a better estimate of the à priori input distribution, this term could also be propagated in the Viterbi algorithm through (9.104). When

$$L_{\boldsymbol{x}(D)} \approx \sum_k L_{\boldsymbol{x}_k} \quad , \tag{9.105}$$

the Viterbi algorithm follows directly with each branch metric including an additional $L_{\boldsymbol{x}_k}$ term for that branch's corresponding input, i.e.,

$$L_{\boldsymbol{x}_k,\boldsymbol{y}_k} = L_{\boldsymbol{y}_k/\boldsymbol{x}_k} + L_{\boldsymbol{x}_k} \quad . \tag{9.106}$$

The ability to propagate such à priori information is useful to some suboptimal, but reduced complexity, iterative decoding methods, as discussed in Sections 9.5 and 9.6.

[7] Example 9.3.1 illustrates a situation where the maximum is not equal to the sum and the Viterbi and APP are not the same algorithm.

For any surviving path in Viterbi detection, a selection of this path was made at each previous stage in the trellis. Sometimes the comparisons leading to this selection may have been very reliable if the likelihood functions of all the other paths were much smaller than that of the chosen path. At other stages, the comparisons leading to the selection may have been close in that another path was almost selected. The "closeness" of these selections are essentially the soft information. Let j_k^* denote the state of the selected path at time k. Then, a full SOVA implementation for the input message m_k corresponding to the symbol value $\boldsymbol{x}_k$ at time k retains the set of differences from the next-closest paths

$$\Delta_{m_k^*, m_k'} = \min_{j \in J_{k+\lambda}^*} \left\{ L_{\boldsymbol{X}_{0:k+\lambda}^*}(j_{k+\lambda}^*) - L_{\boldsymbol{X}_{0:k+\lambda}'}(j) \; m_k^* \neq m_k' \;, \lambda > 0 \right\} \tag{9.107}$$

over the survivor length for each state at each time. $J_{k+\lambda}^*$ is the set of other states (than the best survivor path at time $k+\lambda$ for which the soft metric is being computed). When Viterbi detection has completed at time $K = k + \lambda$ (or has progressed λ stages into the future with respect to the time k of interest, where λ is the maximum survivor length of the implementation), the best survivor path can be traced back to time k. Then for all other survivor paths, the one with the lowest cost $L_{\boldsymbol{X}_{0:k+\lambda}'}(j)$ that have different input message at time k (i.e., different input message λ stages earlier in the trellis) is found and subtracted from $L_{\boldsymbol{X}_{0:k+\lambda}^*}(j_{k+\lambda}^*)$ to get the soft information. The larger the value of $\Delta_{m_k^*, m_k'}$, the more likely the individual symbol decision at time k is to be correct. Such information does not improve nor change the selected sequence, but is useful other decoders, see Sections 9.5 and 9.6.

The difference in likelihoods can also be written as

$$\Delta_{m_k^*, m_k'} = L_{\boldsymbol{X}_{0:k+\lambda}^*}(j_{k+\lambda}^*) - \max_{j \in J_{k+\lambda}^*} \left\{ L_{\boldsymbol{X}_{0:k+\lambda}'}(j) \; m_k^* \neq m_k' \;, \lambda > 0 \right\} \quad . \tag{9.108}$$

For the BSC or the AWGN, the Viterbi Algorithm often directly uses Hamming distance or squared Euclidean distance directly (which is proportional to the negative of the log-likelihood function) and could be called $\mu_{\boldsymbol{X}_{0:k+\lambda}}$, so then (9.108) becomes

$$\Delta_{m_k^*, m_k'} = -\mu_{\boldsymbol{X}_{0:k+\lambda}^*} + \min_{j \in J_{k+\lambda}^*} L_{\boldsymbol{X}_{0:k+\lambda}'}(j) \tag{9.109}$$

$$= \min_{j \in J_{k+\lambda}^*} \mu_{\boldsymbol{X}_{0:k+\lambda}'}(j) - \mu_{\boldsymbol{X}_{0:k+\lambda}^*} \quad . \tag{9.110}$$

EXAMPLE 9.4.1 (SOVA for same Example as APP) For the example of Figure 9.15, the original Viterbi example for the 4-state rate-1/2 convolutional code is repeated. The Viterbi MLSD decoder correctly decoded the transmitted sequence because only two bit errors were made in the BSC, and the code had a free distance of 5. Just below the full Viterbi, the last 3 stages are repeated if the branch that previously had 00 (which was correct) changed to 01, which means the BSC now has introduced 3 bit errors, just as in the APP example of Figure 9.14.

First, the decoder now needs a rule for resolving ties at intermediate points. Rather than just randomly pick one of the paths tied, a slightly more intelligent strategy would be to pick that path which had the smallest cost on its previous state – for this example code, such a strategy leaves an unambiguous way of resolving ties. The MLSD proceeds for the last 3 stages using this rule for each tie that occurs, as shown in Figure 9.15. The code trellis is repeated in the lower left-hand corner for the reader's convenience.

The SOVA detector retains additional information for decisions at time k for the at each of the end states (there is actually such information for every stage, but this example focuses here only on time k). This additional information is the "soft" information about the best path decision made at each state according to (9.107). This soft information appears as superscripts in parenthesis on the costs of the two final states that had lowest metric, where $\lambda = 2$ in this example. The soft information of 0 in this case indicates that the next closest path corresponding to different input at time k was a tie, and is very unreliable. Thus, the soft information does not resolve the tie, as did the better soft information in the APP

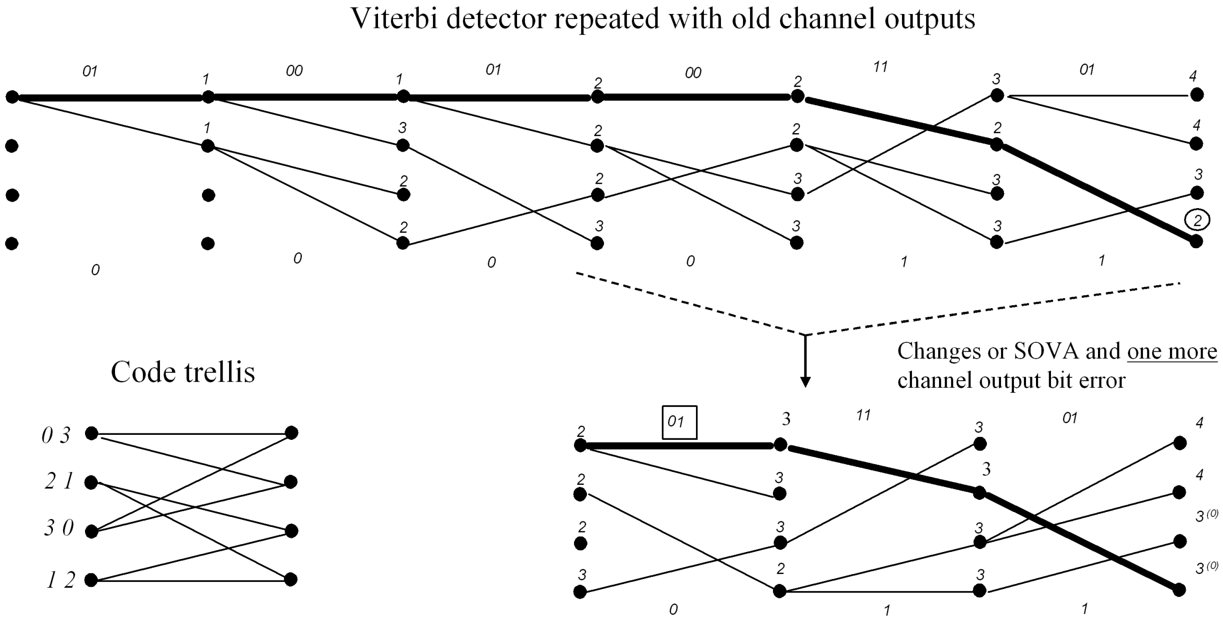

Viterbi detector repeated with old channel outputs
01
00
01
00
11
01
0
0
0
0
1
1
Code trellis
0 3
2 1
3 0
1 2
Changes or SOVA and one more channel output bit error
01
11
01
0
1
1
3^(0)
3^(0)

Figure 9.15: Example of rate 1/2 convolutional code with 3 output errors and SOVA decoding.

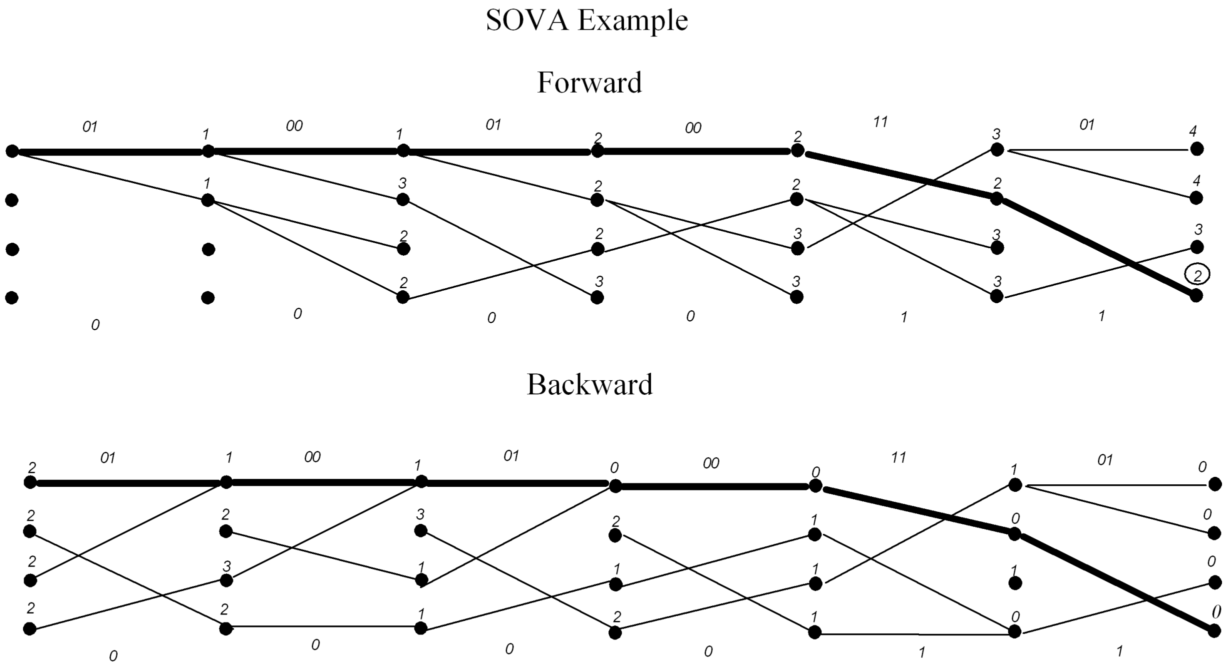
algorithm. Nonetheless, such soft information can be of value to other external decoders. This illustrates that the APP algorithm is actually always better than the SOVA algorithm, where the former could resolve the tie.

The SOVA algorithm is often implemented as a forward-backward Viterbi algorithm. The forward/backward formulation produces the same sequence decisions, but may in some cases produce slightly better soft information. The forward-backward approach may also have some regularities in structure than can be exploited in implementation. Equation (9.110) becomes

$$\Delta = \min_{\substack{j\in J_{k+1}\\ j\neq j^*}} \left[\mu_{\boldsymbol{X}'_{0:k}}(j) + \mu_{\boldsymbol{X}'_{k+1:k+\lambda}}(j)\right] - \mu_{\boldsymbol{X}^*_{0:k+\lambda}} \tag{9.111}$$

where the minimization has been broken into the minimum sum of a forward metric from time 0 to a given stage k in the trellis for non-best paths that correspond to different m_k and a backward metric from time $k+\lambda$ back to time $k+1$ into the same common state of the forward and backward paths j_{k+1} and a different value of m_k. Thus, the decoder runs Viterbi backward and forward in time. With some cleverness, one could envision a recursive "sliding window" version of this SOVA that always spans a fixed packet length that recursively moves one position to the right in time as each soft metric is computed, with both backward and forward path metrics updated on the basis of adjacent metrics and lost/new stages of trellis.

EXAMPLE 9.4.2 (Forward/Backward SOVA) The forward/backward SOVA is illustrated in Figure 9.16. The trellis for both forward and backward Viterbi are illustrated. The backward path is useful only for soft-information and does not exploit the known initial state, and so therefore produces different overall metrics. Nonetheless, at each state, a best forward path into it and best backward path into it can be added to get a potential value for soft-metric computation. For this example $\mu^*_{\boldsymbol{X}_{0:k+\lambda}} = 2$. At those states where such information is computed (they should not be on the optimum path), the path metric evaluated at time $k+1$ must correspond to a different value of the optimum-path message m_k. For instance at $k=0$ in Figure 9.16, the x_k on the best path is a 0 bit and $\mu^*_{\boldsymbol{X}_{0:6}} = 2$. Looking at all states in at time $k+1=1$ that have an input bit one into them from $k=0$



Δ	(1+2)-2=1	(1+2)-2=1	(2+2)-2=2	(2+1)-2=1	(3+1)-2=2	(3+0)-2=1
λ	6	5	4	3	2	1
k	0	1	2	3	4	5

Figure 9.16: Example of rate 1/2 convolutional code with 2 output errors and Forward/Backard SOVA decoding. (forward known state at $k = 0$ dominates branch selection.

(there is only 1), the calculation of the soft information is then 1 (forward) + 2 (backward) - 2 =1=Δ. In this example, there is only one Δ value at each step and this value corresponds to the other value of the binary input that is not on the best sequence path. In more general application, there would be $M - 1$ Δ's for all the other messages at each point in time (An infinite value for Δ is used when somehow a certain input simply cannot occur). The soft metrics are listed below the lower/backward trellis. The values are added for the next-best state that corresponds to different input on the trellis stage just before it. λ effectively varies in this fixed block version as shown in the table below. The decision at time $k = 2$ is the most reliable because $\Delta = 3$ is largest at $k = 2$. The situation where an extra bit error occurs (as in the earlier APP example) is investigated in Figure 9.17 for time $k = 3$. There are two possible optimum paths because of the tie at distance 3. Each of these has a $\Delta = 0$ for $k = 3$ where

$$\begin{aligned} \Delta(\text{upper path}) &= (2+1) - 3 = 0 \qquad (9.112) \\ \Delta(\text{lower path}) &= (3+0) - 3 = 0 \ , \qquad (9.113) \end{aligned}$$

so both decisions are highly unreliable. Unlike the APP, even forward/backward SOVA could not resolve this extra bit error – it is thus sometimes inferior to the APP in the soft information provided (and never better).

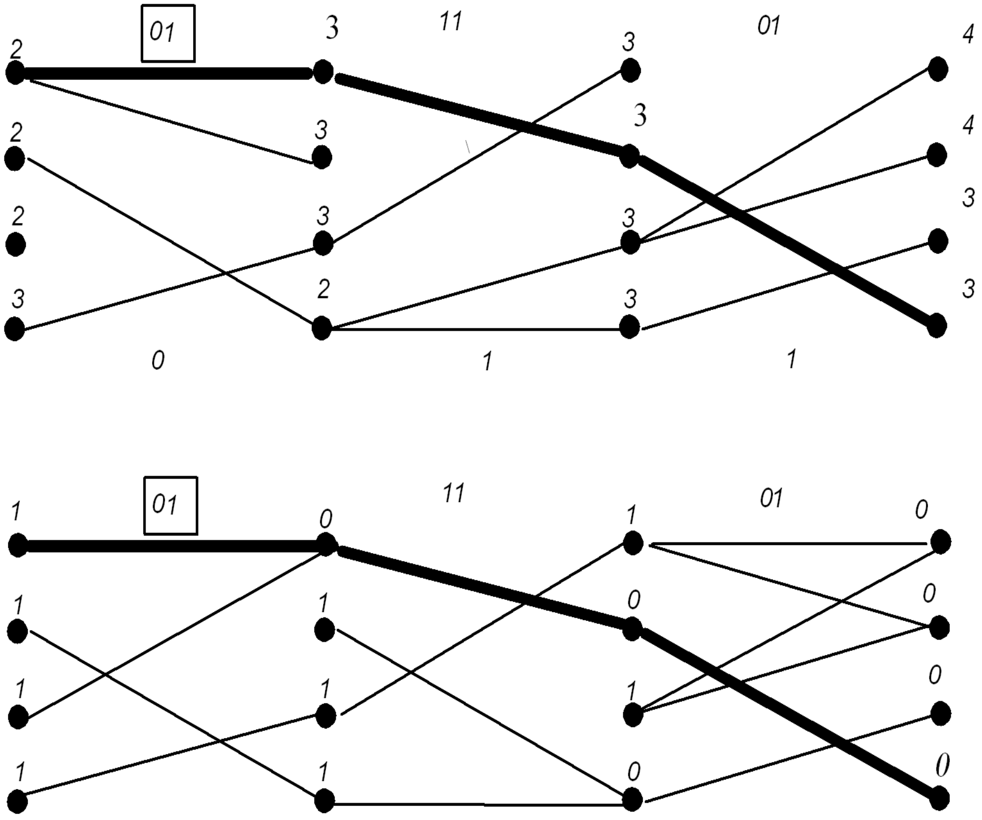


Figure 9.17: Example of rate 1/2 convolutional code with 3 output errors and SOVA decoding.

9.5 Decoding as a function of code constraints

A detailed reference on code constraints and soft information appears in the book by former 379B student and TA John Fan *Constrained Coding and Soft Iterative Decoding*, Kluwer, 2001.

An alternative means of viewing soft information and decoding for coded (or constrained) systems directly computes or approximates the à posteriori probabilities for each of the output symbols of an encoder; given channel outputs, any à priori distribution on these symbols, and the "constraints on the codewords" themselves. Code constraints are probabilistic events that are satisfied when the channel outputs qualify as codewords for whatever code is used. In addition to the trellis constraint implied in APP decoding of Section 9.3 and SOVA decoding in Section 9.4, there are several other types of constraints (or ways of viewing them)

- bit-level constraints (See Subsection 9.5.1)
- code-level constraints (See Subsection 9.5.2)
- constellation constraints (See Subsection 9.5.3)
- intersymbol interference constraint (see Subsection 9.5.4)

The probability density for each symbol (or bit) is computed rather than an overall sequence probability. Thus any particular symbol or bit at time k has a limited number of possibilities, thus averting a search over an exponentially growing number of possible sequences. The constraints supply extrinsic information on each bit or symbol in terms of known relationships to other bits or symbols – that is, the extrinsic information reflects the constraints.

Again, the information of code constraints becomes the extrinsic information while the à priori and channel probabilities essentially merge into the "intrinsic" information that is independent of the code constraints. The à posteriori probability can be construed as

$$p_{app} = p_{\boldsymbol{x}_k/\text{constraints}} \propto p_{\boldsymbol{x}_k,\text{constraints}} = p_{intrinsic} \cdot p_{extrinsic} \stackrel{\Delta}{=} p_{int} \cdot p_{ext} \quad . \tag{9.114}$$

The intrinsic probability will be a function of message or symbol to be estimated and the corresponding channel output at the same time (for a memoryless channel like BSC or AWGN). The extrinsic probability will be the probability contributed from all other channel outputs with regard to the most likely symbol/message value, knowing that only certain constrained sequences are possible.

A **constraint event** will be denoted by the letter E and will also provide extrinsic information to a decoding process. There is a probability for any channel that the constraint is satisfied $P(E)$, which is averaged over all inputs, but not a function of any specific input.

9.5.1 Bit-Level or Parity Constraints

A simple example of a constraint event for a binary code on a BSC could be the parity constraint on a binary encoder output bits v_k like

$$v_1 \oplus v_2 \oplus v_9 = 0 \quad . \tag{9.115}$$

The probability that the constraint is satisfied is trivially $P(E) = 1$ at the encoder output, but less at the BSC output. The bit probabilities at the BSC output (y_k) are easily determined as

$$p(y_k, v_k) = \begin{cases} (1-p)\cdot(1-p_0) & y_k = 1 \; v_k = 1 \\ p \cdot p_0 & y_k = 1 \; v_k = 0 \\ p \cdot (1-p_0) & y_k = 0 \; v_k = 1 \\ (1-p)\cdot p_0 & y_k = 0 \; v_k = 0 \end{cases} \quad . \tag{9.116}$$

For an AWGN with binary PAM inputs of $\pm\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}}$, channel-output probabilities are continuous and equal to

$$p(y, v_k) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{1}{2\sigma^2}(y-\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}})^2} \cdot (1-p_0) & v_k = 1 \\ \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{1}{2\sigma^2}(y+\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}})^2} \cdot p_0 & v_k = 0 \end{cases} \quad . \tag{9.117}$$

From the $p(y, v_k)$ probabilities for 2PAM AWGN or for the BSC, it is possible to compute the probability that the parity constraint is satisfied $P(E)$.

The bit-level constraint that these three bits always sum (modulo-2) to zero is useful in decoding and provides information from any of the individual bits to the other two for decoding. For any constraint event E, the maximum à posteriori decoder that observes or uses the constraint would then maximize (9.114)

$$\max_{v_k=0,1} p_{v_k/E} \quad . \tag{9.118}$$

Any encoder output, whether in bits v_k or in symbol vectors $\boldsymbol{x}_k$, can be uniquely mapped at any time into the corresponding input message (as long as the state of the encoder at that time is known – presumably from decoding of previous bits), so there is no loss of optimality in directly estimating the encoder output symbols. The APP can be rewritten as

$$p_{v_k/E} = \frac{1}{P(E)} \cdot \underbrace{p(v_k)}_{p_{int}} \cdot \underbrace{p_{E/v_k}}_{p_{ext}} \quad . \tag{9.119}$$

The scaling term $1/P(E)$ is a constant that is independent of any specific value of the encoder output v_k and does not affect the MAP decision for v_k given this constraint is satisfied. Generally, a constraint could be written as

$$E(v_1, v_2, ..., v_N) = E(\boldsymbol{v}) = 0 \quad . \tag{9.120}$$

The set of all $\boldsymbol{v}$ values that satisfy the constraint,

$$S_E \stackrel{\Delta}{=} \{\boldsymbol{v} \mid E(\boldsymbol{v}) = 0\} \quad , \tag{9.121}$$

has an extrinsic probability that is given by

$$p_{ext}(v_k) = p_{E/v_k} = c_k \cdot \sum_{\boldsymbol{v}\in S_E; v_k \text{ fixed}} \prod_{\substack{i=1 \\ i\neq k}}^{N} p(v_i) \quad , \tag{9.122}$$

and for a received y_k and AWGN

$$p_{ext}(v_k) = c_k \cdot \sum_{\boldsymbol{v}\in S_E; v_k \text{ fixed}} \prod_{\substack{i=1 \\ i\neq k}}^{N} p(v_i, y) \quad . \tag{9.123}$$

The sums in (9.122) and (9.123) as a function of v_k are executed over that subset of S_E that has a particular fixed value for v_k. The channel output y_k for the AWGN case is implied from here on in this section. The intrinsic probabilities are tacitly a function of the received channel output y_k, which is not shown but of course some fixed value for any given channel observation.

The constant c_k does not depend on v_k and could be computed according to

$$c_k = \left[\sum_{\boldsymbol{v}\in S_E; v_k \text{ fixed}} \prod_{\substack{i=1 \\ i\neq k}}^{N} p(v_i) \right]^{-1} \quad . \tag{9.124}$$

The constant is again inconsequential in subsequent maximization over v_k. The intrinsic and the à priori distribution are the same distribution when decoding is executed in terms of constraints on channel outputs. This intrinsic is thus given by

$$p_{int} = p(v_k) \quad . \tag{9.125}$$

Thus, the joint probability to be maximized by the decoder is then

$$p_{v_k,E} = c_k \cdot \sum_{\boldsymbol{v}\in S_E, v_k \text{ fixed}} \left[\prod_{i=1}^{N} p(v_i) \right] \quad , \tag{9.126}$$

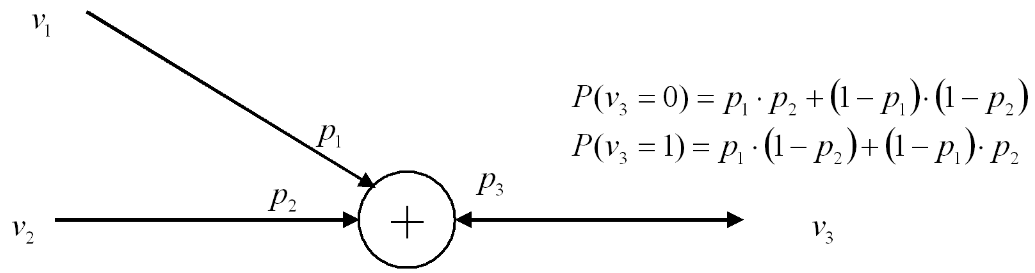

v_1
p_1
p_2
v_2
$+$
p_3
v_3
$P(v_3=0)=p_1\cdot p_2+(1-p_1)\cdot(1-p_2)$
$P(v_3=1)=p_1\cdot(1-p_2)+(1-p_1)\cdot p_2$

Figure 9.18: Graph of parity constraint.

and the à posteriori is

$$p_{v_k/E} = \underbrace{\frac{c_k}{P(E)}}_{c'_k} \cdot \sum_{\boldsymbol{v}\in S_E, v_k \text{ fixed}} \left[\prod_{i=1}^{N} p(v_i)\right] \quad . \tag{9.127}$$

The expressions in (9.126) and (9.127) are distributions that are computed for each fixed value of v_k, so this value is fixed in the sum over all sequences that satisfy the constraint E. The probability $P(E)$ is

$$P(E) = \sum_{\boldsymbol{v}\in S_E} \left[\prod_{i=1}^{N} p(v_i)\right] \quad , \tag{9.128}$$

where this time the sum over S_E does not hold the value v_k fixed and sums over all vectors $\boldsymbol{v}$ that satisfy the constraint. Each channel output sample initiates calculation of an intrinsic $p(v_k)$ in (9.119). The constraint manifests itself through the set S_E where certain bit combinations are not allowed.

EXAMPLE 9.5.1 (3-bit Parity constraint) A parity constraint involves t_r bits in a modulo-2 sum that must be zero. Basically, the non-zero entries in the row in a parity matrix H, such that $GH^* = 0$ (see Chapter 10 for definitions of G and H, which are not necessary to understand this example) determine those encoder output bits that are involved in any particular parity constraint. For instance, a block code might have the parity constraint with $t_r = 3$ so

$$v_1 \oplus v_2 \oplus v_3 = 0 \quad . \tag{9.129}$$

The set of values that satisfy this constraint are $(0,0,0)$, $(1,1,0)$, $(1,0,1)$, and $(0,1,1)$ and constitute the set S_E. Figure 9.18 illustrates this constraint. There will be two values for this probability, one for $v_k = 1$ and the other for $v_k = 0$. If the 3 input bits had intrinsic probabilities of being a 1 of p_1, p_2, and p_3, the extrinsic probability for bit 3 would be:

$$p_{ext}(v_3) = p_{E/v_3} = \begin{cases} p_1\cdot p_2 + (1-p_1)\cdot(1-p_2) & v_3 = 0 \\ p_1\cdot(1-p_2) + p_2\cdot(1-p_1) & v_3 = 1 \end{cases} \quad . \tag{9.130}$$

Similarly

$$p_{ext}(v_2) = p_{E/v_2} = \begin{cases} p_1\cdot p_3 + (1-p_1)\cdot(1-p_3) & v_2 = 0 \\ p_1\cdot(1-p_3) + p_3\cdot(1-p_1) & v_2 = 1 \end{cases} \quad , \tag{9.131}$$

and

$$p_{ext}(v_1) = p_{E/v_1} = \begin{cases} p_2\cdot p_3 + (1-p_2)\cdot(1-p_3) & v_1 = 0 \\ p_2\cdot(1-p_3) + p_3\cdot(1-p_2) & v_1 = 1 \end{cases} \quad . \tag{9.132}$$

The probability to be maximized for a decision for v_3 is

$$P(E, v_3) = c_3 \cdot \begin{cases} p_1\cdot p_2\cdot(1-p_3) + (1-p_1)\cdot(1-p_2)\cdot(1-p_3) & v_3 = 0 \\ p_1\cdot(1-p_2)\cdot p_3 + p_2\cdot(1-p_1)\cdot p_3 & v_3 = 1 \end{cases} \quad , \tag{9.133}$$

where $c_3 = 1 - (p_1 + p_2 + p_3) + 2 \cdot (p_1 \cdot p_3 + p_1 \cdot p_2 + p_2 \cdot p_3) - 4p_1 \cdot p_2 \cdot p_3$. The à posteriori probability, if ever desired directly, is the expression in (9.133) divided by $P(E) = (1 - p_1) \cdot (1 - p_2) \cdot (1 - p_3) + p_1 \cdot p_2 \cdot (1 - p_3) + p_1 \cdot (1 - p_2) \cdot p_3 + (1 - p_1) \cdot p_2 \cdot p_3$.

For the situation with information on bits 1 and 2 stating that $p_1 = p_2 = .99$ (that is high confidence they are 1's), then even if $p_3 = .75$ (pretty high confidence bit 3 is also a 1), the imposition of the constraint yields $P(v_3 = 0) = c_3' \cdot .49$ while $P(v_3 = 1) = c_3' \cdot 0.0099$ so the soft information here would change the unconstrained decision of a 1 for v_3 to favoring heavily a decision of a 0 for v_3.

If the channel were an AWGN, then for a received y, p_1, p_2, and p_3 would all be functions of a particular y value as in (9.117).

More generally, the i^{th} row of H, h_i, determines the parity constraint as

$$S_E = \{\boldsymbol{v} \mid \boldsymbol{v} h_i^* = 0\} \quad . \tag{9.134}$$

The extrinsic probability for both specific values of the bit in the parity constraint is (t_r is the number of bits in the parity equation)

$$p_{E/v_i}(v_i) = \sum_{\boldsymbol{v} \in S_E \text{ ; } v_i \text{ fixed}} \prod_{\substack{j=1 \\ j \neq i}}^{t_r} p_j(v_i) \quad . \tag{9.135}$$

Defining the soft bit $\chi_i = 2p_{ext}(v_i = 0) - 1 = 1 - 2p_{ext}(v_i = 1)$, induction shows (see problem 9.19) for a parity constraint[8] with $\chi_j = 2p_j(v_j = 0) - 1$ for $j \neq i$

$$\chi_i = \prod_{\substack{j=1 \\ j \neq i}}^{N} \chi_j \quad , \tag{9.136}$$

which is sometimes useful for simpler calculation of extrinsic information.

9.5.2 The Equality or Code-Level Constraint

Equality constraints basically observe that the same bit (or symbol) may be involved in more than one constraint. Then, the constraints provide information to one another through an equality constraint. A 3-bit equality constraint example initiates this section, where one could assume that the same bit named v_k appears in 3 other constraints (for instance 3 different parity constraints as in the example of Section 9.5.1).

EXAMPLE 9.5.2 (3-bit equality constraint) The basic equality constraint is sometimes drawn as shown in Figure 9.19. As an example, suppose 3 independent constraints return the probabilities that $v_k = 1$ of a_1, a_2, and a_3. The probability that the constraint is satisfied is

$$p_{v_k,E} = \begin{cases} c_k \cdot a_1 \cdot a_2 \cdot a_3 & v_k = 1 \\ c_k \cdot (1 - a_1)(1 - a_2)(1 - a_3) & v_k = 0 \end{cases} \quad , \tag{9.137}$$

where

$$c_k = \frac{1}{a_1 \cdot a_2 \cdot a_3 + (1 - a_1)(1 - a_2)(1 - a_3)} \quad . \tag{9.138}$$

The extrinsic probability returned to the second constraint as an intrinsic for further probability calculation is (E is the equality constraint while $p_{ext}(v_2)$ is the information provided to another constraint (say parity), which in that other constraint is viewed as an intrinsic probability for $v_k(2)$)

$$p_{ext_2} = p_{ext_2/v_k(2)} = \begin{cases} c_k' \cdot a_1 a_3 & v_k(2) = 1 \\ c_k' \cdot (1 - a_1)(1 - a_3) & v_k(2) = 0 \end{cases} \quad , \tag{9.139}$$

[8] May not hold for general constraints.

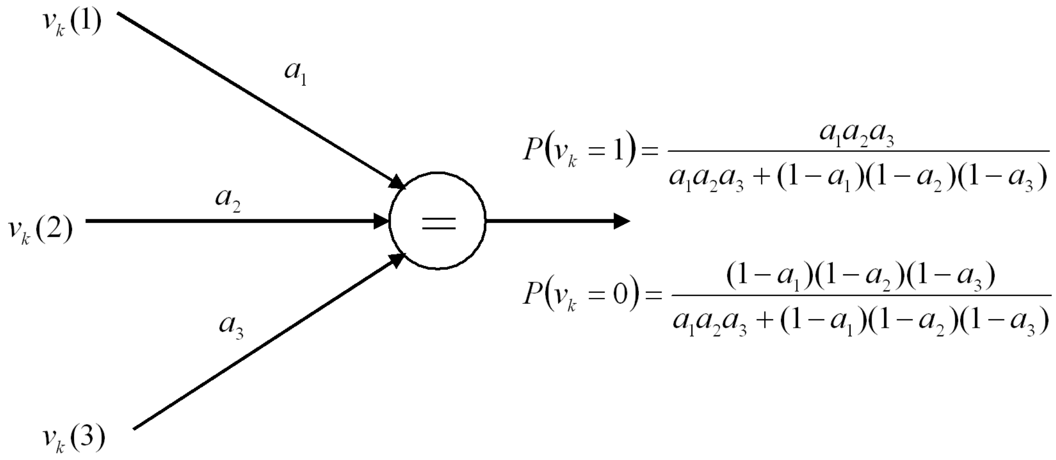

$v_k(1)$
a_1
$v_k(2)$
a_2
$=$
$P(v_k = 1) = \frac{a_1 a_2 a_3}{a_1 a_2 a_3 + (1-a_1)(1-a_2)(1-a_3)}$
$P(v_k = 0) = \frac{(1-a_1)(1-a_2)(1-a_3)}{a_1 a_2 a_3 + (1-a_1)(1-a_2)(1-a_3)}$
a_3
$v_k(3)$

Figure 9.19: Graph of equality constraint.

where

$$c'_k = \frac{1}{a_1 \cdot a_3 + (1-a_1)(1-a_3)} \quad . \tag{9.140}$$

Similar expressions hold for the extrinsic probabilities returned to the 1st constraint p_{E_1} and the third constraint p_{E_3}.

Diagrams such as in Figures 9.19 and 9.18 are useful later in depicting a flow of extrinsic information in decoding. In the diagram, several different extrinsic probabilities for a fixed encoder output bit v_k are available in the decoder. These extrinsic probabilities may come from different code constraints that control the same bit. These different inputs are denoted by $v_k(1)$, ..., $v_k(t_c)$ so that the particular bit v_k is used in t_c code constraints. This set could be alternatively described more compactly by the vector $\boldsymbol{v}_k$. The constraint set in this case is

$$S_E = \{\boldsymbol{v}_k | v_k(1) = v_k(2) = ... = v_k(t_c)\} \quad . \tag{9.141}$$

The set S_E is basically the two points $(0, 0, ..., 0)$ and $(1, 1, ...1)$. One might note that a simple repetition code of rate $1/t_c$ is another example of a situation where the equality constraint directly applies to each of the repeated bits. The joint probability of the event and the bit is thus

$$p_{v_k,E} = c_k \sum_{\boldsymbol{v}_k \in S_E} \prod_{i=1}^{t_c} p_i(v_k) \quad , \tag{9.142}$$

where

$$c_k = \left[\left\{ \prod_{i=1}^{t_c} p_i(v_k) \right\} + \prod_{i=1}^{t_c} (1 - p_i(v_k)) \right]^{-1} \quad . \tag{9.143}$$

The extrinsic probability for any given instance of this bit (returned to the constraint) is

$$p_{ext}(v_k(j)) = c'_k \sum_{\boldsymbol{v}_k \in S_E} \prod_{\substack{i=1 \\ i \neq j}}^{t_c} p_i(v_k) \quad , \tag{9.144}$$

where

$$c'_k = \left[\left\{ \prod_{\substack{i=1 \\ i \neq j}}^{t_c} p_i(v_k) \right\} + \prod_{\substack{i=1 \\ i \neq j}}^{t_c} (1 - p_i(v_k)) \right]^{-1} \quad . \tag{9.145}$$

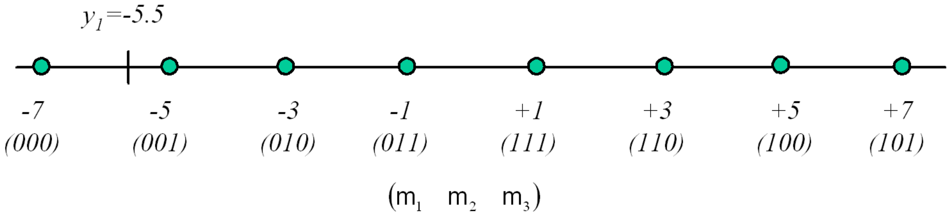


Figure 9.20: Example for 1-dimensional likelihood computation.

Often the other constraints for the bits are parity constraints. Each bit is presumed to be in t_c parity constraints, and the previous subsection discussed the processing of information for a parity constraint. One envisions a successive passing of information between constraints is possible – this is called iterative decoding and discussed further in Section 9.6.

9.5.3 Constellation bit constraints

A likelihood function can also be computed for each bit of the labeling for the points in a constellation or code:

EXAMPLE 9.5.3 (single-dimension bit-level likelihoods) An example for one dimension of a 64 QAM constellation and rate 2/3 code helps understanding. Figure 9.20 shows one dimension of the constellation, labeling, and a received value of $y = -5.5$. The received value of -5.5 in the first dimension corresponds to the 3 bits (m_1, m_2, m_3). There are 4 constellation points that correspond to each of the bits independently being 1 or 0. Thus, the likelihood for m_1 is

$$\begin{aligned} p(y_1 = -5.5, m_1 = 0) &= c_1 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(.5)^2} + e^{-\frac{1}{2\sigma^2}(1.5)^2} + e^{-\frac{1}{2\sigma^2}(2.5)^2} + e^{-\frac{1}{2\sigma^2}(4.5)^2} \right) \cdot (1 - p_1) \\ p(y_1 = -5.5, m_1 = 1) &= c_1 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(6.5)^2} + e^{-\frac{1}{2\sigma^2}(8.5)^2} + e^{-\frac{1}{2\sigma^2}(10.5)^2} + e^{-\frac{1}{2\sigma^2}(12.5)^2} \right) \cdot p_1 \quad , \end{aligned}$$

while the likelihood for m_2 is

$$\begin{aligned} p(y_1 = -5.5, m_2 = 0) &= c_2 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(.5)^2} + e^{-\frac{1}{2\sigma^2}(1.5)^2} + e^{-\frac{1}{2\sigma^2}(10.5)^2} + e^{-\frac{1}{2\sigma^2}(12.5)^2} \right) \cdot (1 - p_2) \\ p(y_1 = -5.5, m_2 = 1) &= c_2 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(2.5)^2} + e^{-\frac{1}{2\sigma^2}(4.5)^2} + e^{-\frac{1}{2\sigma^2}(6.5)^2} + e^{-\frac{1}{2\sigma^2}(8.5)^2} \right) \cdot p_2 \quad , \end{aligned}$$

and finally

$$\begin{aligned} p(y_1 = -5.5, m_3 = 0) &= c_3 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(1.5)^2} + e^{-\frac{1}{2\sigma^2}(2.5)^2} + e^{-\frac{1}{2\sigma^2}(8.5)^2} + e^{-\frac{1}{2\sigma^2}(10.5)^2} \right) \cdot (1 - p_3) \\ p(y_1 = -5.5, m_3 = 1) &= c_3 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(.5)^2} + e^{-\frac{1}{2\sigma^2}(4.5)^2} + e^{-\frac{1}{2\sigma^2}(6.5)^2} + e^{-\frac{1}{2\sigma^2}(12.5)^2} \right) \cdot p_3 \quad . \end{aligned}$$

Any of these could be used as à prior distributions for computation of likelihoods in other constraints or codes.

Of interest in implementation is that the likelihoods for all but the largest values of σ^2, or equivalently at SNR's above a few dB, will be very often dominated by one term on the AWGN channel. Furthermore, by computing the log likelihood with this single-term domination, the receiver truly only needs to compute the squared difference from the closest point in the constellation to compute the log-likelihood for each input bit value.

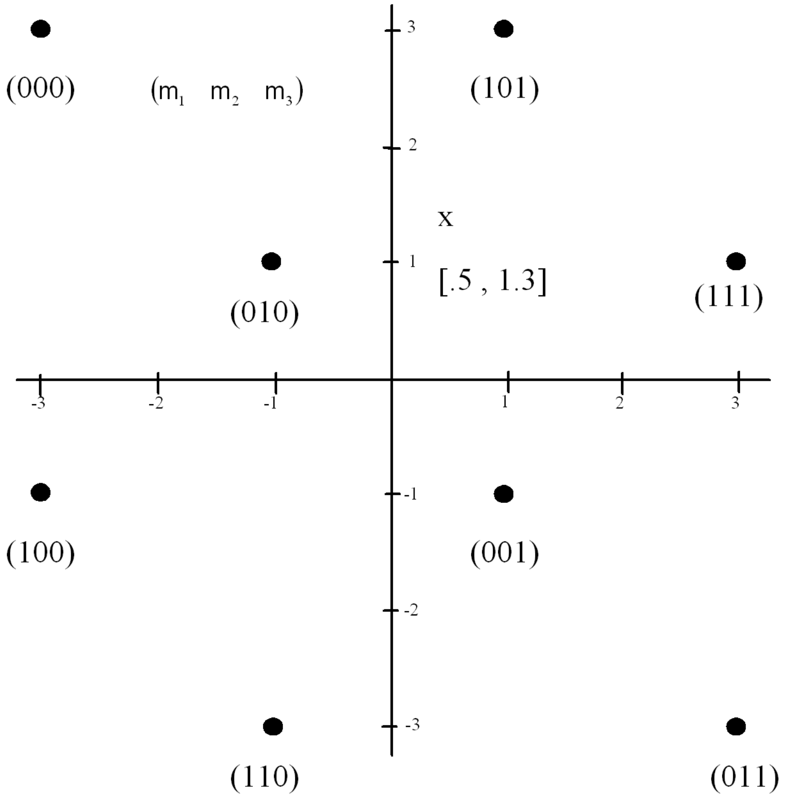


Figure 9.21: Example for 2-dimensional likelihood computation.

EXAMPLE 9.5.4 (2D example) Likelihoods for each bit can also be computed for more than one dimension as illustrated here for the 8SQ constellation in Figure 9.21. The two values of the likelihood for a $\boldsymbol{y} = [.5, 1.3]$ are:

$$\begin{aligned}
p(m_1 = 0) &= c_1 \cdot \frac{1}{2\pi\sigma^2} \left(e^{-\frac{(1.5)^2+(.3)^2}{2\sigma^2}} + e^{-\frac{(3.5)^2+(1.7)^2}{2\sigma^2}} + e^{-\frac{(.5)^2+(2.3)^2}{2\sigma^2}} + e^{-\frac{(2.5)^2+(4.3)^2}{2\sigma^2}} \right) \cdot (1-p_1) \\
p(m_1 = 1) &= c_1 \cdot \frac{1}{2\pi\sigma^2} \left(e^{-\frac{(.5)^2+(1.7)^2}{2\sigma^2}} + e^{-\frac{(2.5)^2+(.3)^2}{2\sigma^2}} + e^{-\frac{(3.5)^2+(2.3)^2}{2\sigma^2}} + e^{-\frac{(1.5)^2+(4.3)^2}{2\sigma^2}} \right) \cdot p_1 \quad .
\end{aligned}$$

The log likelihood ratio (LLR) that is defined in Subsection 9.5.5 later is the natural logarithm of the ratio of a the "1" probability to the "0" probability. If it is positive, then a "1" decision is favored; and if negative, a "0" decision is favored. For this example the computation of the LLRs yields:

$$\begin{aligned}
\text{LLR}(m_1) &= -3.4 && (9.146) \\
\text{LLR}(m_2) &= -.545 && (9.147) \\
\text{LLR}(m_3) &= -.146 \quad . && (9.148)
\end{aligned}$$

The decision would then be the point (m_1, m_2, m_3) = (0,0,0) not the point (0,1,0) that would result from simpler maximum likelihood symbol detection where the closest point to the received vector $\boldsymbol{y} = [.5, 1.3]$ is selected. The received vector, however, is very close to the decision-region boundary for a maximum likelihood symbol detector. The reason for the different decisions is that the points surrounding the received vector favor 0 in the middle position (m_2). That is, 3 of the 4 surrounding points have 0 while only 1 point has a 1 value for m_2.

At reasonably high SNRs so that points near the boundary rarely occur, the log likelihood would reduce to essentially a squared distance from the received value to the closest constellation point and the ML and $\bar{P}_b$-minimizing APP decisions will very often be the same.

9.5.4 Intersymbol-interference constraints

ISI constraints can directly produce soft information through use of the SOVA on the trellis as in Section 9.4 or the APP as in Section 9.3. This section provides a considerably simpler alternative known as the **soft canceler**.

ISI is handled with the so-called soft canceler that is used to approximate ML or MAP detection for each symbol or bit of a channel input from a sequence of channel outputs with ISI. The soft-canceler attempts to compute the à posteriori probabilities

$$p_{\boldsymbol{x}_i/\boldsymbol{y}_u} = \frac{p_{\boldsymbol{y}_u/\boldsymbol{x}_i} \cdot p_{\boldsymbol{x}_i}}{p_{\boldsymbol{y}_u}} \quad , \tag{9.149}$$

iteratively. If this function can be accurately estimated, then a MAP decision for symbol i can be implemented.

The non-$\boldsymbol{x}$-dependent nature of the probability distribution $p_{\boldsymbol{y}_u}$ in the denominator of (9.149) allows direct use of the likelihood function $p_{\boldsymbol{y}/x_i}$ instead of the à posteriori function when the input distribution for x_u is uniform, as is the presumed case here. Sometimes the log-likelihood function $L_{\boldsymbol{y}_u/\boldsymbol{x}_i} = \log(p_{\boldsymbol{y}_u/\boldsymbol{x}_i})$ is instead used of the probability density itself.

The probability distribution is a function of the discrete variable $\boldsymbol{a}_u$ that may take on any of the possible message values for the user $\boldsymbol{x}_u$. $\boldsymbol{a}_u$ is a vector when $\boldsymbol{x}_u$ is a vector.

The objective will be to maximize the probability over $\boldsymbol{a}_u$, or

$$\max_{\boldsymbol{a}_i} \; p_{\boldsymbol{y}_u/\boldsymbol{x}_i} \cdot p_{\boldsymbol{x}_i} \; \forall \; i = 1, ..., U \tag{9.150}$$

or with likelihoods

$$\max_{\boldsymbol{a}_i} \left(L_{\boldsymbol{y}_u/\boldsymbol{x}_i} + L_{\boldsymbol{x}_i} \right) = L_{ext}(\boldsymbol{a}_i) + L_{old}(\boldsymbol{a}_i) \ \ \forall \ \ i = 1, ..., U \ , \tag{9.151}$$

where the quantity $L_{ext}(\boldsymbol{a}_i)$ is the extrinsic likelihood that measures what all the other symbols relate regarding the possible values of symbol u. $L_{old}(\boldsymbol{a}_i)$ is an older likelihood based on previous information (or an initial condition).

The extrinsic probability distribution is assumed to be Gaussian and thus determined if an estimate of its mean and variance can be computed. It is assumed that previous estimates of the mean vector $\xi_u = E[\boldsymbol{x}_i/\boldsymbol{y}_u]$ and the autocorrelation $R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(u) = E[(\boldsymbol{x}_i - \xi_i)(\boldsymbol{x}_i - \xi_i)^*/\boldsymbol{y}_u]$ of all symbols is known from previous iterations or from initialization.

To compute the extrinsic probability distribution for user i, a noise estimate is computed as

$$w_{u,i}(\boldsymbol{a}_i) = \left(\boldsymbol{y}_u - \sum_{j \neq i} P_{uj} \cdot \xi_j \right) - P_{ui} \cdot \boldsymbol{a}_i \tag{9.152}$$

where $\boldsymbol{a}_i$ takes on all the possible discrete values in $\boldsymbol{x}_i$. Thus $w_{u,i}(\boldsymbol{a}_i)$ is a function that estimates the u^{th}-output dimension noise for each possible transmitted input. The autocorrelation of the noise estimate (or the estimated noise power) is

$$R_{\boldsymbol{ww}}(u,i) = R_{nn}(u) + \sum_{j \neq i} P_{uj} \cdot R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(j) \cdot P^*_{uj} \quad . \tag{9.153}$$

Hopefully, in a converged state, $R_{\boldsymbol{ww}}(u,i)$ would approach the noise autocorrelation, and $R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(u) \to 0$, leaving $\hat{\boldsymbol{x}}_i = \boldsymbol{x}_i$.

The new extrinsic probability distribution can thus be computed according to

$$p_{\boldsymbol{y}_u/\boldsymbol{x}_i}(\boldsymbol{a}_i) = \frac{1}{\pi |R_{\boldsymbol{ww}}(u,i)|} \cdot e^{-(w_{u,i}(\boldsymbol{a}_i))^* R^{-1}_{\boldsymbol{ww}}(u,i)(w_{u,i}(\boldsymbol{a}_i))} \quad . \tag{9.154}$$

The overall probability distribution is then the product of this new extrinsic probability distribution and the à priori distribution, from which new values of ξ_i and $R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(i)$ can be computed and used for extrinsic estimates of other symbols:

$$\xi_i = \frac{\sum_{\boldsymbol{a}_i} \boldsymbol{a}_i \cdot p_{\boldsymbol{y}_u/\boldsymbol{x}_i}(\boldsymbol{a}_i)}{\sum_{\boldsymbol{a}_i} p_{\boldsymbol{y}_u/\boldsymbol{x}_i}(\boldsymbol{a}_i)} \tag{9.155}$$

$$R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(i) = \frac{\sum_{\boldsymbol{a}_i} (\boldsymbol{a}_i - \xi_i)(\boldsymbol{a}_i - \xi_i)^* \cdot p_{\boldsymbol{y}_u/\boldsymbol{x}_i}(\boldsymbol{a}_i)}{\sum_{\boldsymbol{a}_i} p_{\boldsymbol{y}_u/\boldsymbol{x}_i}(\boldsymbol{a}_i)} \tag{9.156}$$

Figure 9.22 illustrates the flow chart for the algorithm.

If the soft value ξ_i exceeds the maximum value for the constellation significantly, it should be discarded and not used as clearly there is a "bad" soft quantity that would cause such a large value. Thus, for the pass of all the other symbols that follow, this "bad" value is not used in the soft cancelation, and a value of zero contribution to the variance is instead used. Such bad-value-ignoring has been found to speed convergence of the algorithm significantly.

The designer needs to remember that while the soft-canceler will approximate ML or MAP performance, such performance may still not be acceptable with severe intersymbol interference.

Initial Conditions

The initial soft values ξ_i, $i = 1, ..., U$ can be found by any pertinent detection method. Typically a pseudo-inverse (or inverse) for the channel P can process the output $\boldsymbol{y}_u$ to obtain initial estimates of the ξ_i

$$\boldsymbol{x_i} = P^+ \boldsymbol{y}_u \quad . \tag{9.157}$$

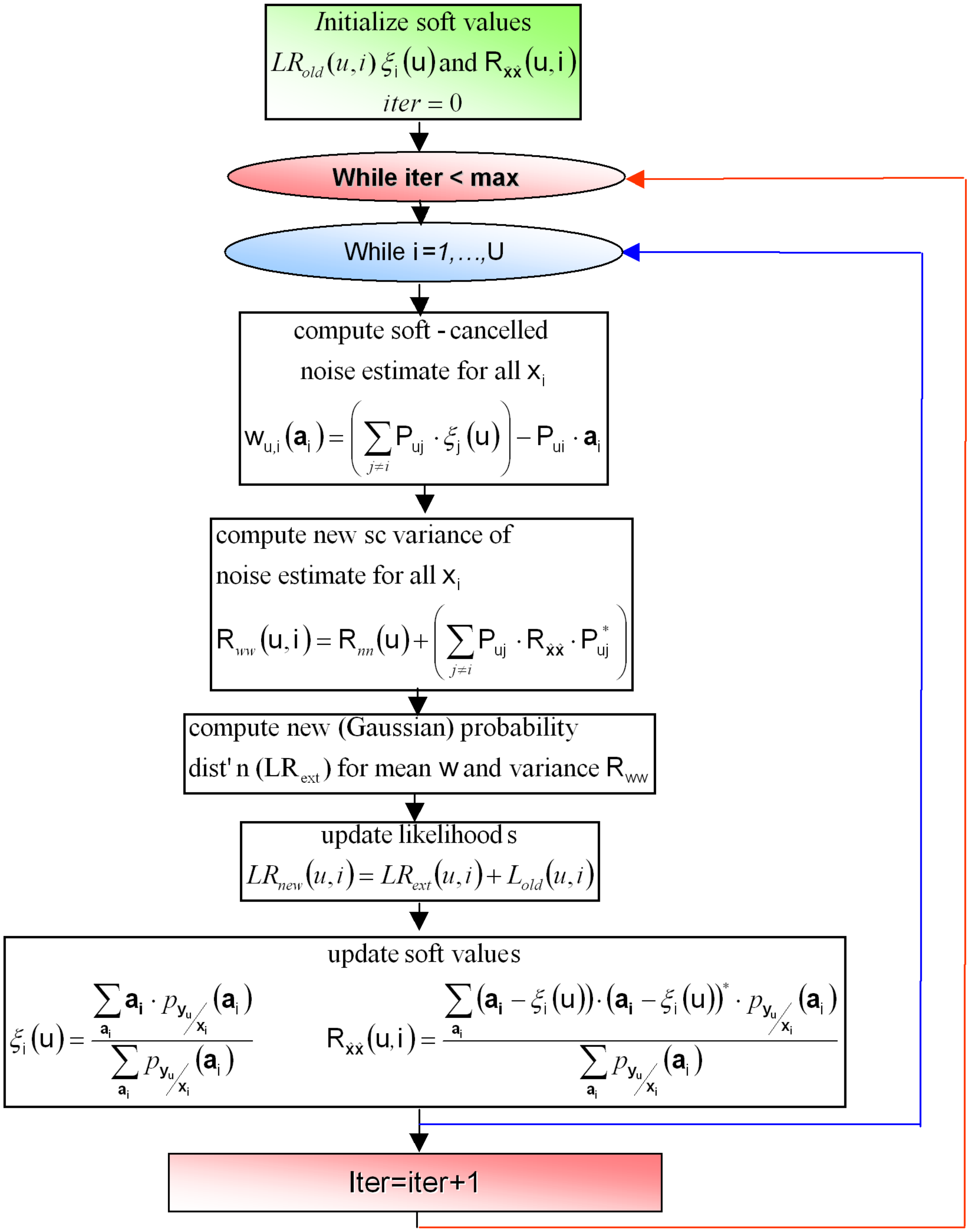

Initialize soft values
$LR_{old}(u,i)\ \xi_i(u)$ and $R_{\hat{x}\hat{x}}(u,i)$
$iter = 0$
While iter < max
While i=1,...,U
compute soft - cancelled
noise estimate for all x_i
$w_{u,i}(\mathbf{a}_i) = \left(\sum_{j\neq i} P_{uj}\cdot\xi_j(u)\right) - P_{ui}\cdot\mathbf{a}_i$
compute new sc variance of
noise estimate for all x_i
$R_{ww}(u,i) = R_{nn}(u) + \left(\sum_{j\neq i} P_{uj}\cdot R_{\hat{x}\hat{x}}\cdot P_{uj}^{*}\right)$
compute new (Gaussian) probability
dist' n (LR_{ext}) for mean w and variance R_{ww}
update likelihood s
$LR_{new}(u,i) = LR_{ext}(u,i) + L_{old}(u,i)$
update soft values
$\xi_i(u) = \frac{\sum_{\mathbf{a}_i}\mathbf{a}_i\cdot p_{y_u/x_i}(\mathbf{a}_i)}{\sum_{\mathbf{a}_i}p_{y_u/x_i}(\mathbf{a}_i)}$
$R_{\hat{x}\hat{x}}(u,i) = \frac{\sum_{\mathbf{a}_i}(\mathbf{a}_i-\xi_i(u))\cdot(\mathbf{a}_i-\xi_i(u))^{*}\cdot p_{y_u/x_i}(\mathbf{a}_i)}{\sum_{\mathbf{a}_i}p_{y_u/x_i}(\mathbf{a}_i)}$
Iter=iter+1

Figure 9.22: Soft Cancelation Flow Chart.

Zeroed initial values may be easier to compute, but can increase the convergence time of the algorithm in Figure 9.22. Large values outside the constellation boundaries produced by any such pseudo-inverse should be set to the closest constellation boundary point. An initial estimate of the variance can be set equal to the computed variance using the initial ξ_i values

$$R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(u) = P^{+} R_{nn}(u) \left[P^{+}\right]^{*} \quad . \tag{9.158}$$

9.5.5 Use of Likelihood Ratios

Transmission systems in practice are ultimately binary, even if $M > 2$ for symbols. Section 9.5.3 shows how to compute likelihoods for each bit value in a constellation constraint. The two bit values' likelihoods can be consolidated into a single quantity, a ratio: The likelihood ratio is

$$\Lambda \stackrel{\Delta}{=} \frac{p}{1-p} \tag{9.159}$$

where p is the probability that a 1 occured. Often the log likelihood ratio is used and is denoted by

$$\mathrm{LLR}(p) = \ln \Lambda \quad . \tag{9.160}$$

When the LLR (p) > 0, a hard decision is “1” and when negative, a hard decision is “0”.

Basically, all log-likelihood relations can be directly rewritten with the LLR for binary codes, simplifying the arithmetic and avoiding the calculation of two values for each distribution-like quantity, instead just computing one number. Since almost all communication systems ultimately reduce to bits transmitted, the likelihood ratio can be correspondingly computed and used no matter what the symbol constellations or codes are used. In practice, a MAP detector for the bits directly instead of for the symbols (with subsequent trivial translation into bits) can perform no worse that the symbol-based MAP detector when minimizing overall probability of bit error.[9] Note, the LLR is no longer explicityly a function of channel input value, so LLR(v) or LLR(y).

For channel information, one easily determines for the BSC input that[10]

$$\mathrm{LLR}_k = \begin{cases} \ln \frac{1-p}{p} & \text{if } v_k = 1 \\ \ln \frac{p}{1-p} & \text{if } v_k = 0 \end{cases} \tag{9.161}$$

and for the AWGN output with binary PAM input that

$$\mathrm{LLR}_k = \frac{2}{\sigma^2} \cdot y_k \quad , \tag{9.162}$$

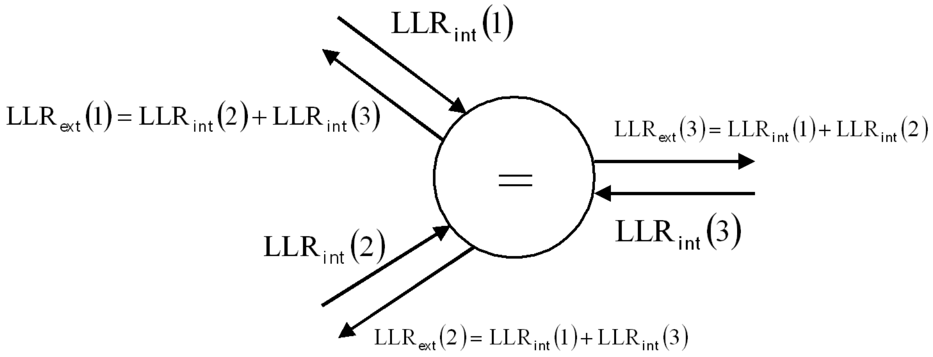
which provides for any given y the natural log of the ratio of the probability that the AWGN input is a 1 ($x = \sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}}$) to the probability that the input is a 0 ($x = -\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}}$) if 1 and 0 are equally likely. If 1 and 0 are not equally likely, add $\ln(\frac{p_1}{p_0})$.

EXAMPLE 9.5.5 (Continuation of Example 9.5.3) For the previous Example 9.5.3, the log likelihoods are approximated by using the dominant terms in each sum by

$$\begin{aligned} \mathrm{LLR}(m_1) &= \ln\left[e^{-\frac{1}{2\sigma^2}[(6.5)^2-(.5)^2]}\right] = -\frac{1}{2\sigma^2}[42] \ \text{ favors 0} && (9.163) \\ \mathrm{LLR}(m_2) &= \ln\left[e^{-\frac{1}{2\sigma^2}[(2.5)^2-(.5)^2]}\right] = -\frac{1}{2\sigma^2}[6] \ \text{ favors 0} && (9.164) \\ \mathrm{LLR}(m_3) &= \ln\left[e^{-\frac{1}{2\sigma^2}[(.5)^2-(1.5)^2]}\right] = \frac{1}{\sigma^2} \ \text{ favors 1.} && (9.165) \end{aligned}$$

[9] Optimum because the probability of bit error is minimized by definition.

[10] If the input is 0, the probability of an output 0 is $(1-p)p_0$ and of a 1 pp_0 so $\Lambda = \ln \frac{p}{1-p}$, or if the input is 1, the probability of an output 1 is $(1-p)p_1$ and of an output 0 is $(p \cdot p_1$, so $\Lambda = \ln \frac{1-p}{p}$.



LLR_int(1)
LLR_ext(1) = LLR_int(2) + LLR_int(3)
LLR_ext(3) = LLR_int(1) + LLR_int(2)
=
LLR_int(2)
LLR_int(3)
LLR_ext(2) = LLR_int(1) + LLR_int(3)

Figure 9.23: Equality constraint soft-information flow.

The log likelihood ratio when using the dominant terms is

$$\text{LLR} = \frac{y}{\sigma^2}(b-a) + \frac{a^2-b^2}{2\sigma^2} \tag{9.166}$$

where b is the closest 1-bit point in the constellation and a is the closest 0-bit point in the constellation.

the LLR generalizes to the form

$$\text{LLR} = \Re\frac{\{\boldsymbol{y}^*(\boldsymbol{b}-\boldsymbol{a})\}}{\sigma^2} + \frac{\|\boldsymbol{a}\|^2 - \|\boldsymbol{b}\|^2}{2\sigma^2} \tag{9.167}$$

for complex vectors.

Equality Constraints

Algebra on (9.144) shows that the extrinsic propagation of information from an equality node for the j^{th} instance of the same bit's use in t_r parity checks is given in terms of the LLR as

$$\text{LLR}(j) = \sum_{\substack{i=1 \\ i\neq j}}^{t_r} \text{LLR}(i) \tag{9.168}$$

and the APP simply sums all terms except the j^{th}. Thus in Figure 9.23, each extrinsic output log likelihood ratio is simply the sum of all the other input intrinsic log likelihood ratios. Specifically, the log likelihood ratio associated with each input is NOT included in the computation of its extrinsic output at the equality node. One can of course invert the LLR into a binary probability variable p, the probability of a 1. This is often done through the soft-bit formula[11]

$$\chi = 1 - 2p = -\tanh\left\{\frac{1}{2}\text{LLR}(p)\right\} = 2p(bit=0) - 1 \quad . \tag{9.170}$$

The computation of the LLR form of the soft-bit is

$$\text{LLR}(p) = \ln\frac{1-\chi}{1+\chi} \quad . \tag{9.171}$$

[11] Which follows from:

$$\frac{e^{\frac{1}{2}\ln\frac{p}{1-p}} - e^{-\frac{1}{2}\ln\frac{p}{1-p}}}{e^{\frac{1}{2}\ln\frac{p}{1-p}} + e^{-\frac{1}{2}\ln\frac{p}{1-p}}} = \frac{\sqrt{\frac{p}{1-p}} - \sqrt{\frac{1-p}{p}}}{\sqrt{\frac{p}{1-p}} + \sqrt{\frac{1-p}{p}}} = \frac{p-1+p}{p+1-p} \tag{9.169}$$

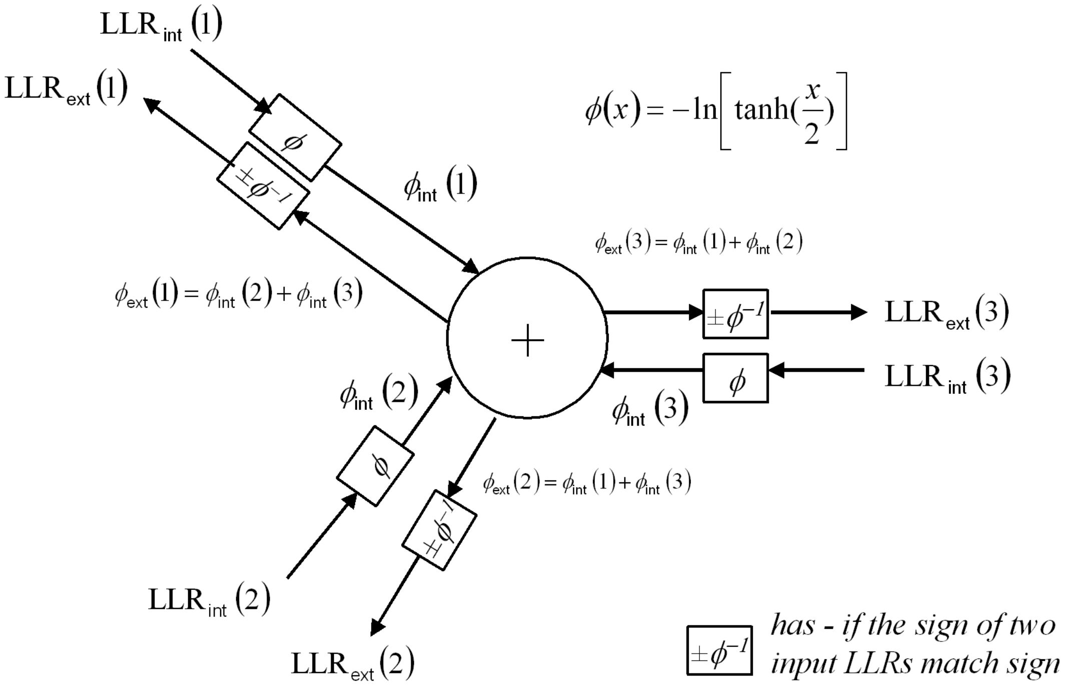

LLR$_{\text{int}}$(1)
LLR$_{\text{ext}}$(1)
$\phi(x) = -\ln\left[\tanh(\frac{x}{2})\right]$
ϕ
$\pm\phi^{-1}$
$\phi_{\text{int}}(1)$
$\phi_{\text{ext}}(3) = \phi_{\text{int}}(1) + \phi_{\text{int}}(2)$
$\phi_{\text{ext}}(1) = \phi_{\text{int}}(2) + \phi_{\text{int}}(3)$
+
$\pm\phi^{-1}$
LLR$_{\text{ext}}$(3)
ϕ
LLR$_{\text{int}}$(3)
$\phi_{\text{int}}(2)$
$\phi_{\text{int}}(3)$
ϕ
$\phi_{\text{ext}}(2) = \phi_{\text{int}}(1) + \phi_{\text{int}}(3)$
$\pm\phi^{-1}$
LLR$_{\text{int}}$(2)
LLR$_{\text{ext}}$(2)
$\pm\phi^{-1}$
has - if the sign of two input LLRs match sign

Figure 9.24: Parity constraint soft-information flow.

The hyperbolic tangent is

$$\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x}) \quad . \tag{9.172}$$

Table look-up is often used to convert the LLR to soft-bit, and vice-versa. The hyperbolic tangent's inverse satisfies (for $|y| < 1$) the relation

$$2 \cdot \tanh^{-1}(y) = \ln\left\{\frac{1+y}{1-y}\right\} \quad . \tag{9.173}$$

Parity Constraints

The parity constraint soft-information relationship is then from (9.136) with t_r terms in the parity constraint using (9.170)

$$\chi_i = \prod_{\substack{j=1 \\ j\neq i}}^{t_r} \chi_j = (-1)^{t_r-1} \cdot \prod_{\substack{j=1 \\ j\neq i}}^{t_r} \tanh\left(\frac{\text{LLR}(p_j)}{2}\right) \tag{9.174}$$

or with a natural logarithm for avoiding the product in implementation and using the look-up-table function $\phi(p) \triangleq -\ln\left(\frac{|\tanh(p)|}{2}\right)$. Then

$$LLR(p_i) \;=\; \ln\frac{1-\chi}{1+\chi} \tag{9.175}$$

$$\;=\; \ln\frac{1-\prod_{\substack{j=1 \\ j\neq i}}^{t_r}(1-2p_j)}{1+\prod_{\substack{j=1 \\ j\neq i}}^{t_r}(1-2p_j)} \tag{9.176}$$

$$= \quad \ln \frac{1-(-1)^{t_r-1}\prod_{\substack{j=1\\ j\neq i}}^{t_r}\tanh\left(\frac{1}{2}\mathrm{LLR}(p_j)\right)}{1+(-1)^{t_r-1}\prod_{\substack{j=1\\ j\neq i}}^{t_r}\tanh\left(\frac{1}{2}\mathrm{LLR}(p_j)\right)} \tag{9.177}$$

$$= \quad (-1)^{t_r}\cdot 2\cdot \tanh^{-1}\left[\prod_{\substack{j=1\\ j\neq i}}^{t_r}\tanh\left(\frac{1}{2}\mathrm{LLR}(p_j)\right)\right] \tag{9.178}$$

$$= \quad (-1)^{t_r}\cdot \prod_{\substack{j=1\\ j\neq i}}^{t_r}\left[\mathrm{sgn}\left(\mathrm{LLR}(p_j)\right)\right]\cdot \phi^{-1}\left(\sum_{\substack{j=1\\ j\neq i}}^{t_r}\phi(|\mathrm{LLR}(p_j)|\right) \tag{9.179}$$

where

$$\phi(x)=\phi^{-1}(x)=-\ln\left[\tanh\left(\frac{x}{2}\right)\right]=\ln\frac{e^x+1}{e^x-1} \quad . \tag{9.180}$$

See Figure 9.24 for the implementation diagram of the parity soft-information flows.

9.6 Iterative Decoding

Sections 9.3 - 9.5 illustrated several ways of generating soft information from a specific decoder or constraint. This soft information by itself allows MAP minimization of symbol or bit error, but has greater significance in its use to other decoders. Passing of soft information from one decoder to another, hopefully to improve the accuracy of an estimate of a likelihood for a bit or symbol value, is known as iterative decoding. Iterative decoding is the subject of this section.

Iterative decoding approximates ML or MAP detectors for a set of inputs and corresponding constraints/code with much lower complexity. Iterative decoding uses the extrinsic soft information generated as in Sections 9.3 - 9.5 along with intrinsic given or channel soft information in successive attempts to refine the estimate of the probability density (or LLR) of a transmitted bit or symbol. More sophisticated systems may use the fact that more than one code or constraint includes a given message bit, and soft information from constraint or code can become an á priori (intrinsic) probability for a second constraint. Soft information generated by one constraint is often called **extrinsic** information as in Section 9.5 because it can be used externally by other decoders. Such extrinsic information may help resolve "ties" or situations that are so close that the other decoders without assistance would not correctly decode with sufficiently high probability. With a combination of good fortune and skill, an iterative decoding process between two or more constraints, where each subsequently passes extrinsic information to the others in a cyclic order, can converge to a probability distribution for the input symbols that clearly indicates the same channel inputs as would be detected by a joint (for both codes viewed as a single giant code) optimum decoder, but with much less complexity. The concatenated and long-block-length codes of Chapter 11 are designed with anticipated use of iterative decoding. Implementation of iterative-decoding often enables a sufficiently low-cost approximation to an ML or MAP detector for very complex high-gain codes, allowing reliable transmission at data rates very close to capacity in transmission channels.

The key decomposition of the APP distribution for generating extrinsic information for iterative decoding is:

$$p_{\boldsymbol{x}_k/\boldsymbol{Y}_{0:K-1}} \propto p_{\boldsymbol{x}_k,\boldsymbol{Y}_{0:K-1}} = \underbrace{\underbrace{p_{\boldsymbol{x}_k}}_{\text{à priori}} \cdot \underbrace{p_{\boldsymbol{y}_k/\boldsymbol{x}_k}}_{\text{channel}}}_{\text{intrinsic}} \cdot \underbrace{p_{\boldsymbol{Y}_{0:k-1},\boldsymbol{Y}_{k+1:K-1}/\boldsymbol{x}_k,\boldsymbol{y}_k}}_{\text{extrinsic}} \quad . \tag{9.181}$$

The proportionality on the left in (9.181) simply notes that the normalization by the probability distribution $p_{\boldsymbol{Y}_{0:K-1}}$ changes nothing in a final decision for $\boldsymbol{x}_k$. The 3 terms correspond to the input or "à priori" information that either reflects knowledge about the input being nonuniform, or more likely represents soft information accepted from some other constraint/decoder. The channel information itself is simply that coming from symbol-by-symbol detection, which while sometimes insufficient by itself, is nonetheless often the strongest indication of the most likely symbol transmitted (see Section 9.5.3). The last term in (9.181) is the information captured from everything but the à priori and the current (time k) channel information. This last term can be used to export the soft information to all the other code/constraints involving $\boldsymbol{x}_k$ so that any coding (or intersymbol interference or in general "memory" in the transmission system) may give an indication of this message's probability of being different values. This is the "extrinsic" information of Section 9.5 and becomes the à priori information for another decoder in iterative decoding.

For a first decoding, the **à priori** term is often initially a uniform distribution (or some other known distribution) in iterative decoding. However, it can be replaced by refined estimates at each stage of iterative decoding. By contrast, the channel information remains the same. Often the bits/symbols are ordered differently for two decoders with respect to one another. This simply reflects that fact that the reordering redistributes the location of a burst of channel noise/errors, thus allowing better soft information from constraints less affected by bursts to be shared with those heavily affected by bursts. Interleaving, the engineer's name for redistribution, and its inverse de-interleaving are discussed in detail in Section 11.2.

It is often again convenient to use the log-likelihood function instead of the probability function itself,

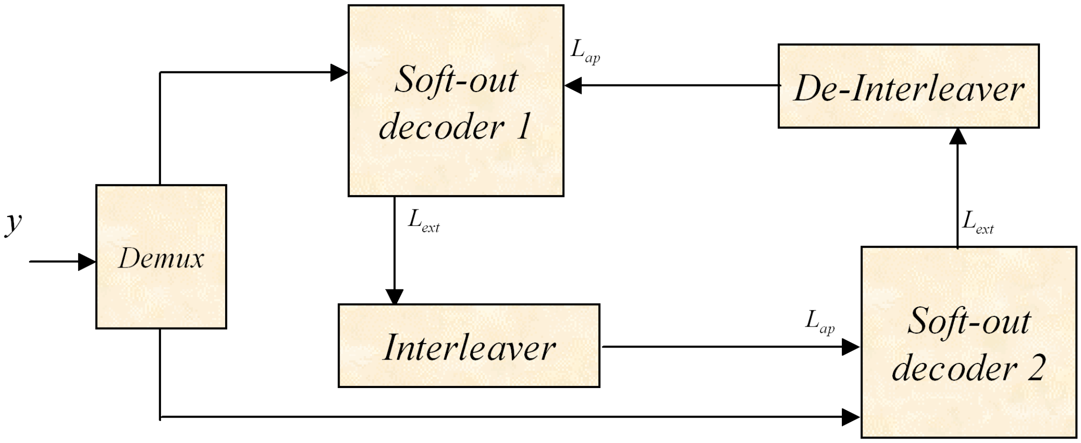


Figure 9.25: Iterative Decoding two codes.

so then (9.181) becomes

$$L_{\boldsymbol{x}_k,\boldsymbol{Y}_{0:K-1}} = L_{\boldsymbol{x}_k} + L_{\boldsymbol{y}_k/\boldsymbol{x}_k} + L_{\boldsymbol{Y}_{0:k-1},\boldsymbol{Y}_{k+1:K-1}/\boldsymbol{x}_k,\boldsymbol{y}_k} \tag{9.182}$$

$$= L_{\text{à priori}} + L_{\text{channel}} + L_{\text{extrinsic}} \quad . \tag{9.183}$$

If $L_{\boldsymbol{x}_k,\boldsymbol{Y}_{0:K-1}}$ is computed or approximated by a decoder, it is relatively simple to subtract L_{channel} and $L_{\text{à priori}}$ to get the extrinsic information for common symbols in the other decoder. The "L" can be replaced by "LLR" for bit decoding. Alternately for a given constraint as in Section 9.5, this extrinsic information may be directly computed.

Only information from different channel outputs is desirable as the à priori (or extrinsic accepted from another constraint) because the immediate channel output is already in use in (9.181). Such care to use only extrinisic information largely averts double counting the channel information in the likelihood calculations. A related potential problem with such "ad-hoc" iterative decoding with multiple constraints is that information could recycle in an unstable manner through several tightly related constraints allowing too much feedback in the successive refinement process. As with any feedback system, excessive feedback can destabilize the outputs. Code designs need to take this concern into account, or their gain will be limited when iterative decoding is used. Codes or constraints with such "short cycles" should be avoided when iterative decoding is used.

Figure 9.25 thus shows the basic iterative decoding concept where only the extrinsic likelihood from one decoder, L_{ext} as used as an à priori input distribution for the other decoder. The decoders can for instance correspond to two independent codes, in which case this is called "turbo coding" (really should be "turbo decoding") in an amusing analogy with the "turbo" engine in automobiles.[12] Effectively an equality constraint on all the commonly encoded bits is presumed between the codes used in Figure 9.25. One of the decoders could in fact be a MLSD decoder for intersymbol interference or some other mechanism for generating soft information from intersymbol interference. The two codes could be actually those of two different users sharing the same transmission system, and it happens that their signals "crosstalk" into one another (see Chapter 14).

Either decoder also produces an estimate of each and every symbol value $\boldsymbol{x}_i$, which is found as that symbol value at time i, $i = 0, ...K-1$ that maximizes the likelihood function values $L_{\boldsymbol{x}_i/\boldsymbol{Y}_{0:K-1}}$, or equivalently maximizes $L_{\boldsymbol{x}_i,\boldsymbol{Y}_{0:K-1}}$. A simple method of knowing when to stop is if and when the decisions made by the two decoders agree. Typically 5-20 iterations are necessary with reasonable code choices for the two decoders to agree.

[12] It seems more recent coding theorists pursue more exciting extracurricular activities than the gardening performed by their more tranquil predecessors.

Theorem 9.6.1 (Convergence of Iterative Decoding) *Iterative decoding will converge to the exact probability densities desired if the constraints are such that no information from one constraint can pass back to that constraint – that is the number of iterations is smaller than the length of a "cycle" among the constraints.*

Proof: *The proof follows directly from Equation 9.181, which is exact. If each of the terms in this expression are exactly known, then the entire equation is exactly computed. The first two terms on the right are trivially exactly computed. The extrinsic term is computed correctly if no approximations are used in its calculation. The only approximation that could be made in its calculation elsewhere is the use of other extrinsic terms that are approximated – the only way for them to be approximate is that somehow one decoder's processing used à priori information that came from another decoder's extrinsic information, which only happens if there are enough iterations for information to cycle through the constraints.* **QED.**

Theorem 9.6.1 is both profound in use and theoretically trivial. The very condition of "no cycles" prevents the nature of iterative decoding in some sense as avoiding cycles altogether would be very difficult (and would create some very difficult computations that are essentially equivalent to executing a full decoder). Thus, the condition for convergence usually won't occur exactly in most (or almost all) uses. Nonetheless, even if it is only approximately satisfied, the process often is very close to convergent and the decisions ultimately made will almost always be the same as the more complicated direct implementation of the decoder. Figure 9.25 immediately violates the short-cycles condition as clearly extrinsic information is cycling – yet, depending on the codes chosen and particularly the interleaver, the "long cycles" at a detailed bit level may be largely satisfied and the iterative decoding process can still then converge very quickly.

9.6.1 Iterative Decoding with the APP (BCJR) algorithm

In the APP algorithm of Section 9.3.1, the quantity

$$\gamma_k(i,j) = p(s_{k+1} = j, \boldsymbol{y}_k / s_k = i) = p_k(i,j) \cdot \sum_{\boldsymbol{x}'_k} p_{\boldsymbol{y}_k/\boldsymbol{x}'_k} \cdot q_k(i,j,\boldsymbol{x}'_k) \quad , \tag{9.184}$$

where $p_k(i,j) = p(s_{k+1} = j / s_k = i)$, allows the transfer of information between decoders. Recall this quantity is the one used to update the others (namely, α and β in BCJR) and to incorporate channel and à priori information. Ultimately decisions are based on the product of α, β, and γ in the BCJR algorithm, see Section 9.3. Since parallel transitions can only be resolved by symbol-by-symbol decoding, one presumes that step is accomplished outside of iterative decoding and so (9.184) simplifies to

$$\gamma_k(i,j) = p_k(i,j) \cdot p_{\boldsymbol{x}_k} \cdot p_{\boldsymbol{y}_k/\boldsymbol{x}_k} \tag{9.185}$$

where the first two terms on the right is the one used to incorporate the extrinsic information from the other decoder. Thus,

$$\ln\left(\gamma_k(i,j)\right) = L_{ext}(\text{old}) + L_{\boldsymbol{y}/\boldsymbol{x}} \quad . \tag{9.186}$$

or

$$\gamma_k(i,j) = e^{L_{ext}(\text{old})} \cdot e^{L_{\boldsymbol{y}/\boldsymbol{x}}} \quad . \tag{9.187}$$

Iterative Decoding with 2 or more decoders each individually using the APP passes information through the γ's. The α and β quantities are computed from the latest set of $\gamma_k(i,j)$ for each successive APP-decoding cycle with initial conditions as in Subsection 9.3.1. The extrinsic likelihood is then computed after completion of the algorithm as

$$L_{ext}(\text{new}) = L_{\boldsymbol{x}_k/\boldsymbol{Y}_{0:K-1}} - L_{ap} - L_{\boldsymbol{y}_k/\boldsymbol{x}_k} \tag{9.188}$$

where L_{ap} is the likelihood of the á priori (often the last value of the extrinsic likelihood), $L_{\boldsymbol{y}_k/\boldsymbol{x}_k}$ comes directly from the channel, and $L_{ext}(new)$ becoming $L_{int}(old)$ for the next decoder in the sequence.

9.6.2 Iterative Decoding with SOVA

The soft information of the SOVA method in Section 9.4 is somewhat ad-hoc and does not exactly correspond to computation of a probability distribution on the input $\boldsymbol{x}$. Nonetheless, larger values of $\Delta_{\boldsymbol{x}_k^*,\boldsymbol{x}_k'}$ imply more $\boldsymbol{x}_k^*$ is more likely than $\boldsymbol{x}_k'$. In fact the likelihood of any pair of $\boldsymbol{x}$ values, say $\boldsymbol{x}_k^{(1)}$ and $\boldsymbol{x}_k^{(2)}$ can be computed by subtracting corresponding values of Δ as

$$\Delta_{\boldsymbol{x}_k^{(1)},\boldsymbol{x}_k^{(2)}} = \Delta_{\boldsymbol{x}_k^*,\boldsymbol{x}_k^{(1)}} - \Delta_{\boldsymbol{x}_k^*,\boldsymbol{x}_k^{(2)}} \quad . \tag{9.189}$$

Values of Δ for all states must be maintained in the previous decoder's implementation of SOVA and not just for the surviving path. Then the SOVA internal update information in (9.106) for each comparison of branches becomes a function with respect to other values $\boldsymbol{x}_k^{(m)}$:

$$L_{\boldsymbol{x}_k^{(m)},\boldsymbol{y}_k}(\boldsymbol{x}_k^{(m)}) \leftarrow L_{\boldsymbol{y}_k/\boldsymbol{x}_k^{(m)}} + \Delta_{\boldsymbol{x}_k,\boldsymbol{x}_k^{(m)}} \;\; m = 0, ..., M_k - 1 \;\; , \;\; \boldsymbol{x}_k^{(m)} \neq \boldsymbol{x}_k \tag{9.190}$$

where the reference value of $\boldsymbol{x}_k$ is the value selected on this path in the previous use of SOVA in the other decoder. These new extrinsic-information-including branch likelihoods are added to previous state likelihoods and the SOVA proceeds as in Section 9.4. Use of the constraints rather than SOVA may be more direct.

9.6.3 Direct use of constraints in iterative decoding

Figure 9.26 illustrates the direct use of constraints, parity and equality, in an iterative-decoding flow diagram. Messages are passed from the initial channel outputs as LLR's from right to left into the equality nodes. Subsequently, the equality nodes pass this information to each of the parity nodes, which then return soft information to the equality nodes. The equality and parity then iteratively pass soft information and the output LLR's from the equality nodes are then monitored to see when they heavily favor 0 or 1 for each of the bits (and then the algorithm has converged). This diagram may be very useful in circuit implementation of iterative decoding of any (FIR) parity matrix H, which essentially specifies parity and equality constraints (for more, See LDPC codes of Chapter 11). Each of the equality nodes is essentially as in Figure 9.19 with the ultimate LLR returned to the right the sum of all inputs (while soft extrinsic information passed back to parity nodes excludes the particular parity-equations input to the equality node). Correspoindingly, each of the parity nodes is essentially as in Figure 9.18.

9.6.4 Turbo Equalization

Turbo equalization is not really equalization at all in the sense that this textbook defines equalization methods. Turbo equalization is iterative decoding where one of the two decoders in Figure 9.25 is an MLSD, SOVA, Soft-Canceler, or APP for the trellis or ISI constraint. The other decoder is a code. Of course, there can be two codes or more the decoding proceeds in order, processing the ISI constraint for information, and then the code constraints for information in a successive and recursive manner. Performance will essentially be the same as if a joint MAP detector were done for all.

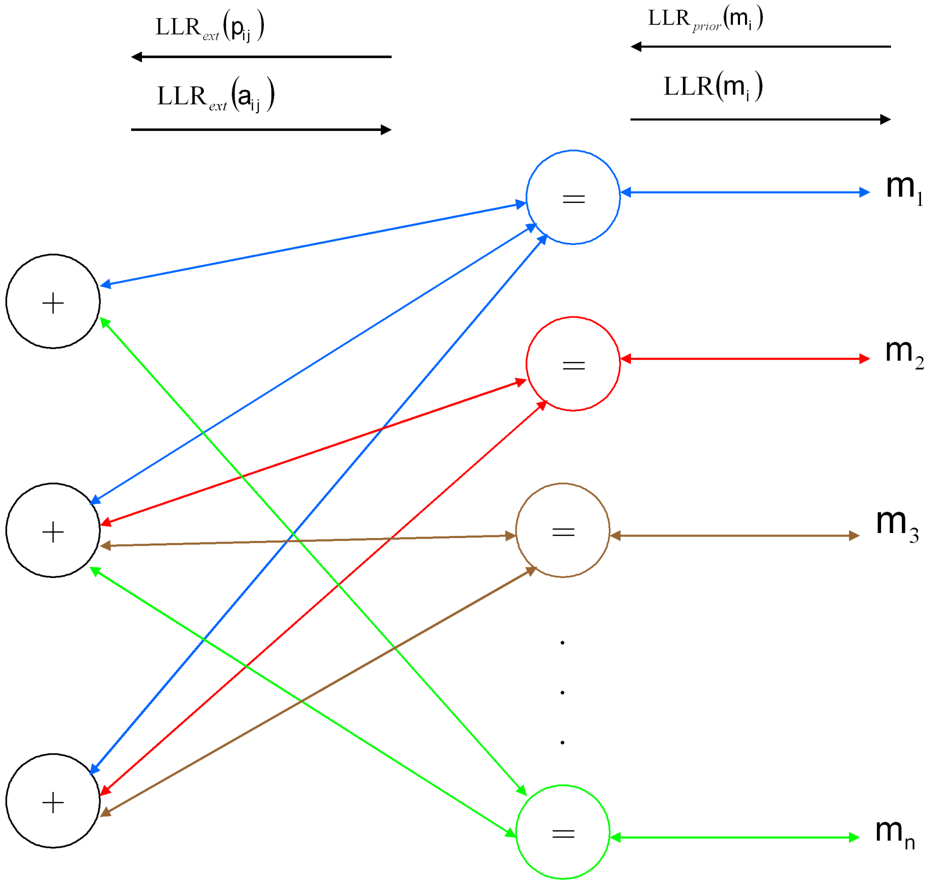


Figure 9.26: Direct use of constraints in Iterative Decoding.

Exercises - Chapter 9

9.1 *Error Events.*
For the trellis in Figure 9.2, let the input bits correspond to no precoding.

a. (3 pt) Find the error-event descriptions $\epsilon_m(D)$, $\epsilon_x(D)$ and $\epsilon_y(D)$ if the two inputs of concern are the sequences $m(D) = 0$ and $m(D)1 \oplus D \oplus D^2$.

b. (1 pt) What is the distance between these two events for the AWGN and consequent ML decoder?

c. (2 pts) What is the probability that one of the two input bit sequences is confused for the other?

9.2 *Partial Response Precoding and MLSD.*
A partial response channel with AWGN has $H(D) = (1 + D)^4(1 - D)$ and $M = 2$.

a. (1 pt) Find the channel response $h_k,\ k = 0, 1, ..., 5$.

b. (1 pt) Make a guess as to why this channel is called 'EPR6'.

c. (2 pts) Design a precoder for this channel and show the receiver decision regions.

d. (1 pt) In part (c), find the loss in performance with respect to the MFB.

e. (2 pts) Find the improvement of MLSD over precoding (ignoring N_e differences), given that $\epsilon_x(D) = 2(1 - D + D^2)$ is in E_{min}, the set of all error events that exhibit minimum distance.

f. (1 pt) In part (e), it was guessed that $\epsilon_x(D) = 2(1 - D + D^2)$ is in E_{min}. Some other sequences that are likely to be in E_{min} are $2(1 - D), 2(1 - D + D^2 - D^3)$, and $2(1 - D + D^2 - D^3 + D^4)$. Can you think of an intuitive reason why this should be so?

9.3 *Viterbi decoding of convolutional codes.*
A $\bar{b} = 2/3$ convolutional code is described by the 3 dimensional vector sequence $\boldsymbol{v}(D) = [v_3(D),\ v_2(D),\ v_1(D)]$ where $\boldsymbol{v}(D)$ is any sequence generated by the all the binary possibilities for an input bit sequence $\boldsymbol{u}(D) = [u_2(D),\ u_1(D)]$ according to

$$\boldsymbol{v}(D) = \boldsymbol{u}(D) \cdot \begin{bmatrix} 1 & D & 1+D \\ 0 & 1 & 1+D \end{bmatrix} \tag{9.191}$$

a. (2 pts) Write the equations for the output bits at any time k in terms of the input bits at that same time k and time $k - 1$.

b. (2 pts) Using the state label $[u_{2,k-1}\ u_{1,k-1}]$, draw the trellis for this code.

c. (4 pts) Use Viterbi decoding with a Hamming distance metric to estimate the encoder input if $\boldsymbol{y}(D =$

010 110 111 000 000 000 ...

is received at the output of a BSC with $p = .01$. Assume that you've started in state 00. Note that the order of the triplets in the received output is: $v_3v_2v_1$.

d. (2 pts) Approximate N_e and P_e for this code with ML detection.

e. (2 pts) Approximate N_b and P_b for this code with ML detection.

9.4 *Performance evaluation of MLSD, easy cases.*
Find d^2_{min}, the number of states for the Viterbi decoder, and the loss with respect to MFB for MLSD on the following channels with $M = 2$ and $d = 2$.

a. (2 pts) $H(D) = 1 + \pi D$

b. (2 pts) $H(D) = 1 + .9D$

c. (2 pts) $H(D) = 1 - D^2$

d. (3 pts) $H(D) = 1 + 2D - D^2$. Would converting this channel to minimum phase yield any improvement for MLSD? Discuss *briefly.*

9.5 *MLSD for the 1-D channel.*
Consider the discrete time channel $H(D) = 1 - D$.

a. (1 pt) Draw the trellis for $H(D)$ given binary inputs $x_k \in \{-1, +1\}$.

b. (2 pts) Show that the Viterbi algorithm branch metrics for $H(D)$ can be written as below.

$k-1$	k	branch metric
+1	+1	0
+1	-1	$z_k + 1$
-1	+1	$-z_k + 1$
-1	-1	0

c. (2 pts) Explain in words how you would use these branch metrics to update the trellis. Specifically, on each update two paths merge at each of the two states. Only one path can survive at each state. Give the rule for choosing the two surviving paths.

d. (2 pts) What can be said about the sign of the path metrics? If we are interested in using only positive numbers for the path metrics, how could we change the branch metrics? *Hint*: $min\{x_i\}$ is the same as $max\{-x_i\}$.

e. (2 pts) Use the MLSD you described in the previous part to decode the following sequence of outputs z_k:

0.5 1.1 -3 -1.9

It is acceptable to do this by hand. However, you should feel free to write a little program to do this as well. Such a program would be *very* helpful in the next problem without any changes if you apply it cleverly. Turn in any code that you write.

f. (1 pt) True MLSD does not decide on any bit until the whole sequence is decoded. In practice, to reduce decoding delay, at every instant k, we produce an output corresponding to symbol $k - \Delta$. For what value of Δ would we have achieved MLSD performance in this case?

9.6 *Implementing MLSD for the* $1 - D^2$ *channel.*
(4 pts) Use MLSD to decode the following output of a $1 - D^2$ channel whose inputs were 2-PAM with $d = 2$. *Hint*: You can make one not-so-easy problem into two easy problems.

0.5, 0, 1.1, 0.5, -3, 1.1, -1.9, -3, 0, -1.9

9.7 *Searching paths to find* d^2_{min}.
Consider the channel $H(D) = 0.8 + 1.64D + 0.8D^2$ with $\sigma^2 = 0.1$, $M = 2$, $d = 2$.

a. (3 pts) Find the minimum squared distance associated with an $L_x = 1$ (i.e., three branch) error event.

b. (4 pts) Find the minimum squared distance associated with an $L_x = 2$ (i.e., four branch) error event.

c. (4 pts) Find the minimum squared distance associated with an $L_x = 3$ (i.e., five branch) error event.

d. (3 pts) Assume that no longer error patterns will have smaller squared distances than those we have found above. What is ${d^2}_{min}$ for this MLSD trellis, and what is the loss with respect to the MFB? What is the loss with respect to the MFB if the last term of the channel changes to $-0.8D^2$?

9.8 *AMI Coding.*

Alternate Mark Inversion (AMI) Coding can be viewed as a discrete-time partial response channel, where the $H(D) = 1 - D$ shaping is entirely in the *transmit* filter, on a flat (ISI-free) AWGN channel. The transmit filter is preceeded by a binary differential encoder (ie. $M = 2$) and a modulator. The modulator output, which is binary PAM with levels $\{1, -1\}$, is the input to the transmit filter. The system diagram for AMI is shown in the figure.

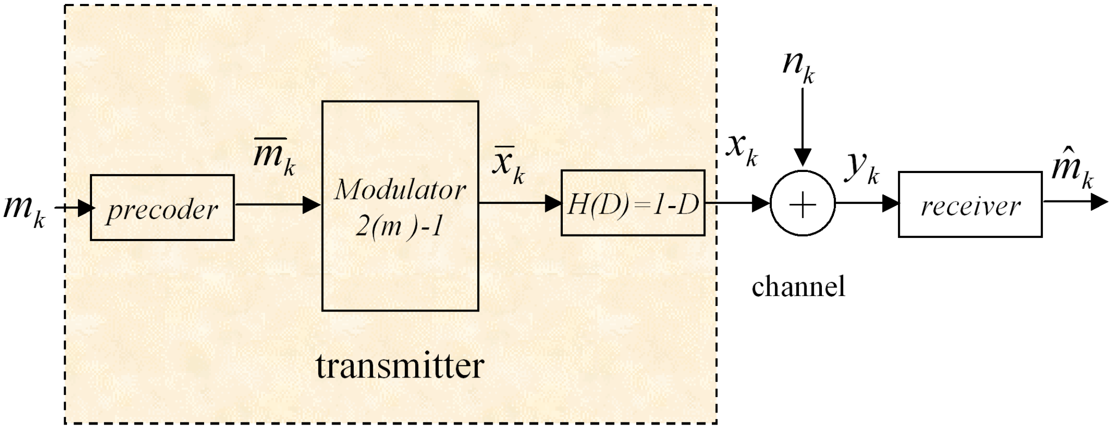


Figure 9.27: AMI transmission system.

a. (2 pts) Draw the transmitter in detail, showing the precoder with input bits m_k and output bits $\overline{m}_k$, a modulator with ouput $\overline{x}_k$, and the transmit filter with output x_k.

b. (1 pt) Find the channel input energy per dimension.

c. (2 pts) For a noise spectral density of $\sigma^2 = 0.5$, find the P_e for a symbol-by-symbol decoder.

d. (3 pts) Find the P_e for the MLSD detector for the system (with the same σ^2 as above). *Hint*: Its easy to find N_e in this case. Look at the analysis for the $1 + D$ channel in the text.

e. (2 pts) AMI coding is typically used on channels that are close to flat, but which have a notch at DC (such as a twisted-pair with transformer coupling). Argue as to why the channel input x_k, produced by AMI, may be better than the channel input x_k, produced by 2-PAM.

9.9 *Partial Response - Midterm 1997 11 pts* The partial-response channel shown in Figure 9.28 has the frequency magnitude characteristic, $|H(e^{-j\omega T})|$, shown in Figure 9.29. The power spectral density (or variance per sample) of the AWGN is $\sigma^2 = .01$.

a. What partial-response polynomial might be a good match to this channel? Can you name this choice of PR channel? (3 pts)

b. For binary PAM transmit symbols, find a simple precoder for this PR channel and determine the transmit energy per symbol that ensures the probability of bit error is less than 10^{-6} . (4 pts)

c. Using MLSD on this channel, and ignoring nearest neighbor effects, recompute the required transmit energy per symbol for a probability of symbol error less than 10^{-6}. (2 pts)

d. Would you use a precoder when the receiver uses MLSD? Why or why not? (2 pt)

9.10 *Trellis and Partial Response - Midterm 2003 16 pts*

A partial-response channel has response $P(D) = (1 + D)^q \cdot (1 - D)$ and additive white Gaussian noise, where $q \geq 1$ is an integer.

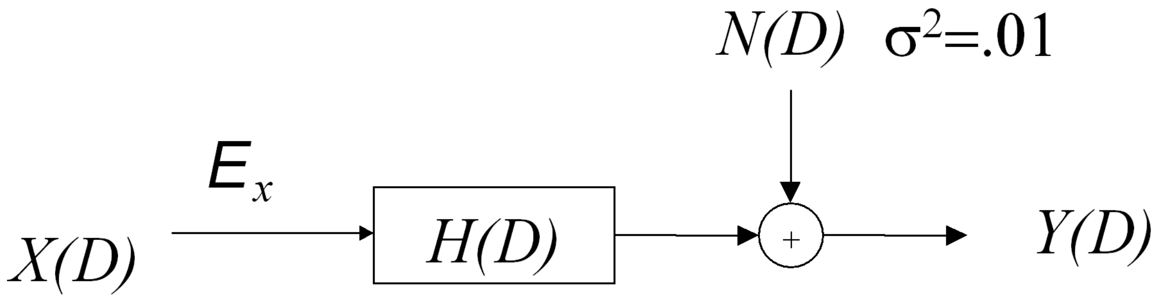


Figure 9.28: Partial-response channel.

a. For $q = 1$ and $M = 4$, draw two independent and identical trellises that describe this channel and associated times at which they function. (3 pts)

b. What is the d^2_{min} for part a and the corresponding gain over a ZF-DFE? (2 pts)

c. What error-event polynomials (input and output) produce the d_{min} in part b? (2 pts)

d. Design a precoder that avoids quasi-catastrophic error propagation for the use of MLSD on this channel. (2 pts).

e. If $q = 2$ and $M = 4$, how many states are used in trellis? What is the general formula for $q \geq 2$ and any M for the number of states? (2 pts)

f. What is d^2_{min} for $q = 2$ and $M = 4$? (2 pts)

g. Which MLSD works at lower P_e, the MLSD for part f or for part b? How do they compare with an AWGN with $P(D) = 1$? Why? (3 pts)

9.11 *Ginis dmin for other metrics 11 pts* Modify the bdistance.m file of Ginis program to compute squared distance instead of binary distance, and then

a. Use the program to verify the squared distance computations of the 4-state Trellis code used in class. (4 pts)

b. Use the program to verify the squared distance computations for the $1 + D$ channel with $M = 4, 8, 16$. (4 pts)

c. Use this program to determine the minimum distance for the channels $H(D) = (1 + D)^n(1 - D)$ for $n = 2, 3$ and $M = 2$. (4 pts)

d. Use this program to determine the performance of MLSD on the discrete channel with $P(D) = 1 + .5D + .5D^2 - .25D^3$ for $M = 2$ binary transmission in Figure 9.28. (4 pts)

9.12 *Al-Rawi's SBS detection and MLSD on EPR4 Channel:* In this problem you are asked to use matlab to compare SBS detection and MLSD using VA at the output of an EPR4 channel with AWGN. The matlab files that you need for solving this problem are available on the class web page.

Assume the input is binary. The input information sequence $m(D)$ that you need to use for this excercise is given in m.mat file.

a. Use matlab to implement a binary precoder, and generate the precoded sequence $\bar{m}(D)$ and the 2-level PAM $x(D)$ sequence Assume $d = 2$. (4 pts)

b. Pass $x(D)$ through the channel to get $y(D)$ Use the supplied matlab function channel.m, which takes $x(D)$ as an input and generates $y(D)$ as an output, for simulating the channel response The function channel.m takes the two-side noise power spectral density σ^2 as an argument. (2 pts)

c. Use SBS detection to detect the information bits $\hat{m}$. Find the number of bit errors in the detected sequence for the case of SNR $= \frac{\bar{\mathcal{E}}_{\boldsymbol{x}}}{\frac{N_0}{2}}$=4dB. (4 pts)

d. Use MLSD to detect the information bits $\hat{m}$. You may use the supplied matlab function MLSD.m, which implements Viterbi decoding for EPR4 channel. Find the number of bit errors in the detected sequence for the case of SNR $= \frac{\bar{\mathcal{E}}_{\boldsymbol{x}}}{\frac{N_0}{2}}$=2dB. (4 pts)

e. Plot the bit error rate $\bar{P}_b$=number of bits in error/length(m) versus SNR $= \frac{\bar{\mathcal{E}}_{\boldsymbol{x}}}{\sigma^2}$ for the SBSD and MLSD over the SNR range 0 to 6dB. (4 pts)

f. From your plots, what is the coding gain of MLSD over SBSD at $\bar{P}_b = 10^{-}3$? What is the expected theoretical coding gain? If the two are different, give a brief explanation. (4 pts)

9.13 *APP Algorithm*

Use of the 4-state rate-1/2 convolutional code with a systematic encoder given by $G(D) = [1 \ \frac{1+D^2}{1+D+D^2}]$ on the BSC results in the output bits shown in Figure 9.14. The transmitters input bits are equally likely.

a. (2 pt) Find the à posteriori probability distribution for each input bit in the systematic case.

b. (2 pts) Find the MAP estimate for each transmitted bit.

c. (2 pts) Why is it possible for the MAP estimate of the systematic encoder case to be different than the MAP estimate for the original nonsystematic realization in Figure 9.14. What about the MLSD – can the probability of bit error be different for the two encoders for this same code? What probability of error remains unchanged for systematic and nonsystematic realizations?

d. (3 pts) What is the probability of bit error for the case of MLSD with the systematic encoder? Is this better than the same probability of bit error with the nonsystematic encoder?

e. (2 pts) Comment on why the computation of probability of bit error may be difficult in the case of the MAP detector.

9.14 *Noise Estimation*

Consider the use of any code or partial response system with soft MLSD on the AWGN channel.

a. (1 pt) Does the execution Viterbi MLSD algorithm depend on the variance of the noise?

b. (1 pt) Does the execution APP algorithm depend on the variance of the noise?

c. (1 pt) How might your answer to part b present a practical difficulty?

d. (2 pts) Suppose you were given one training packet where all bits transmitted were known to both the transmitter and the receiver. How might the receiver estimate the noise variance?

e. (3 pts) For your answer in part d, what is the standard deviation for the noise variance estimate as a function of the training-packet length?

f. (2 pts) What is the equivalent practical difficulty to part c on the BSC?

g. (2 pts) How might you use the training packet in part e to solve the implementation problem in part f?

h. (1 pt) Suppose the parametrization of the AWGN or BSC was not exactly correct - what would you expect to see happen to the probability of bit error?

i. (2 pts) Does the SOVA algorithm have the same implementation concern? Comment on its use.

9.15 *SBS detection and MAP detection on EPR4 channel (Al-Rawi): (18 pts)*

In this problem you are asked to use matlab to compare SBS detection and MAP detection using the APP algorithm at the output of an EPR4 channel with AWGN. The matlab files that you need for this problem are available on the class web page.

Assume the input is binary and is equally likely. A single frame of the input sequence of length 1000 bits that you need to use for this exercise is given in the file m.mat. We would like to pass this frame through the channel EPR4channel.m 16 times and find the average BER. The function EPR4channel.m takes $\frac{\mathcal{N}_0}{2}$ and the frame number i ($i = 1, 2, .16$) as arguments, in addition to input frame xFrame$_i(D)$, and returns the output frame yFrame$_i(D)$. Let $\bar{\mathcal{E}}_{\boldsymbol{x}} = 1$, and assume SNR=4 dB, unless mentioned otherwise.

a. (1 pt) Use matlab to implement a binary precoder, and generate the precoded sequence $\bar{m}(D)$ and the 2-level PAM sequence $x(D)$.

b. (1 pt) After going through the channel, use SBS detection to detect the information bits. Find the number of bit errors in the detected sequence over the 16 transmitted frames.

c. (3 pts) For each received symbol y_k, $k = 1, ..., N$, where N is the frame length, calculate the channel probabilities $p_{y_k/\tilde{y}_k}$ for each possible value of $\tilde{y}_k$, where $\tilde{y}_k$ is the noiseless channel output. Normalize these conditional probabilities so that they add to 1. (3 pts)

d. (3 pts) Use the APP algorithm to detect the information bits. The matlab function app.m implements the APP algorithm for the EPR4 channel. Find the number of bit errors in the detected sequence over the 16 frames.

e. (3 pts) Repeat part d, but now instead of doing hard decisions after one detection iteration, do several soft detection iterations by feeding the extrinsic soft output of the APP algorithm as a priori soft input for the next iteration. Find the number of bit errors in the detected sequence over the 16 frames for 3 and 10 iterations.

f. (3 pts) Plot the BER versus SNR for SBS detection and MAP detection with 1, 3, and 10 iterations over the SNR range 0 to 12 dB. Use a step of 0.4 dB in your final plots. (3 pts)

g. (3pts) Does increasing the number of iterations have a significant effect on performance? Why, or why not?

h. (1 pt) What is the coding gain of MAP detection, with one iteration, over SBS detection at $\bar{P}_b = 10^{-3}$.

9.16 *APP (5 pts)*

Repeat the APP example in Section 9.3 for the output sequence 01, 11, 10, 10, 00, 10. and BSC with $p = 0.2$.

9.17 *SOVA (5 pts)*

Repeat the SOVA example in Section 9.3 for the output sequence 01, 11, 10, 10, 00, 10. and BSC with $p = 0.2$.

9.18 *Viterbi Detection - Midterm 2001 (12 pts)*

WGN is added to the output of a discrete-time channel with D-transform $1 + .9D$, so that $y_k = x_k + .9 \cdot x_{k-1} + n_k$. The input, x_k, is binary with values ± 1 starting at time $k = 0$ and a known value of $+1$ was transmitted at time $k = -1$ to initialize transmission.

a. Draw and label a trellis diagram for this channel.(2 pts)

b. Find the minimum distance between possible sequences. (2 pts)

c. What ISI-free channel would have essentially the same performance? (2 pts)

d. The receiver sees the sequence $y(D) = 1.91 - .9 \cdot D^2 + .1 \cdot D^5$ over the time period from $k = 0, ..., 5$ (and then nothing more). Determine the maximum likelihood estimate of the input sequence that occurred for this output. You may find it easiest to draw 6 stages of the trellis in pencil and then erase paths. Leave itermediate costs on each state. (6 pts).

9.19 *Soft-bit Induction Proof (6 pts)*
Prove Equation (9.136) via induction for a parity constraint with any number of terms.

9.20 *Equality Constraint Processing (7 pts)*
An equality constraint from 3 different constraints on a bit provides the probabilities $p(1) = .6$, $p(2) = .5$, and $p(3) = .8$.

a. Find the à posterior probability for the bit given the constraint and what value maximizes it. (1 pt)

b. Find the extrinsic probability for each bit for the constraint. (3 pts)

c. Show that the same value maximizing part a also maximizes the product of the intrinsic (à priori) and extrinsic information. (2 pts)

d. Compute the log likelihood ratio (LLR) for this bit. (1 pt)

9.21 *Parity Constraint Processing (12 pts)*
An parity constraint has $v_1 + v_2 + v_3 = 0$ with à priori probabilities $p_1 = .6$, $p_2 = .5$, and $p_3 = .8$.

a. Find the extrinsic probability for each bit for the constraint. (3 pts)

b. Find the value that maximizes the product of the intrinsic (à priori) and extrinsic information. (2 pts)

c. Find the à posterior probability for each bit given the constraint and what value maximizes it. (3 pts)

d. Repeat part b using Log Likelihood Ratios and the function ϕ of Section **??**. Show (4 pts)

9.22 *Constellation Constraint Processing (12 pts)*
For the constellations of Examples 9.5.3 and 9.5.4 using an SNR of 4 dB

a. Find the log likelihood ratios for each bit in each of the two constellations and the corresponding favored value of each bit (6 pts)

b. Find the LLR's if the received value is 2.9 for Example 9.5.3 for each of the 3 bits. (3 pts)

c. Find the LLR's if the received value is [-2.9,-1.1] for Example 9.5.4. for each of the 3 bits. (3 pts)

d. Suppose the 3 bits in part b and the 3 bits in part c actually correspond to the same transmitted bits in two independent transmission systems (i.e., diversity) - what is the APP decision for each bit. (3 pts)

9.23 *Simple Iterative Decoding Concept (9 pts)*
We begin using the constellation output for Problem 9.22 Part d's LLR's for each of the bits as à priori or intrinsic information to the parity constraint of Problem 9.21.

a. Find the result favored value for each bit and the corresponding LLR from the parity constraint. (3 pts)

b. Suppose a second equality constraint states that $v_1 = v_2$, then find the subsequent favored values for the 3 bits and the corresponding LLR's after also considering this equality constraint. (3 pts)

c. Proceed to execute the parity constraint one more time and indicate the resultant decisions and LLR's after this step. (3 pts)

9.24 *APP Decoding Program (6 pts)*

This problem investigates the use of APP Decoding software recently developed by former student K. B. Song. Please see the web page for the programs map_awgn.m , trellis.m , and siva_init.m .

a. For the sequence of channel outputs provided at the web page by the TA for the 4-state rate 1/2 convolutional code on the AWGN, run the program map_awgn.m to determine the APP estimates of the input bits. (3 pts)

b. Plot the LLR's that would constitute extrinsic information to any other decoder from your results in part a. You may need to modify the program somewhat to produce these results. (3 pts)

9.25 *APP and SOVA Comparison (11 pts)*

The objective here is to determine just how close APP and SOVA really are.

a. Compute N_D for any convolutional code of rate k/n bits per symbol with 2^ν states with the APP algorithm with log likelihood ratios. A table-look-up on a 1-input function (like e^x or $\ln(x)$) can be counted as one operation. Multiplication is presumed to not be allowed as an operation, so that for instance $\ln(e^x \cdot e^y) = \ln(e^{x+y}) = x + y$. (3 pts)

b. Note that the sum of a number of exponentiated quantities like $e^x + e^y$ is often dominated by the larger of the two exponents. Use this observation to simplify the APP algorithm with log likelihoods where appropriate. (3 pts)

c. Compute N_D for the SOVA algorithm. (3 pts)

d. Compare your answer in part b to part c. Does this lead to any conclusions about the relation of APP and SOVA? (2 pts)

9.26 *Constrained Constellation Decoding (12 pts)*

The figure below shows a 16QAM constellation that is used twice to create a 4D constellation with the first use corresponding to encoder output bits $[v_4\ v_3\ v_2\ v_1]$ and the second use corresponding to subsequent encoder output bits $[v_8\ v_7\ v_6\ v_5]$. The encoder output bits are also known to satisfy the constraints

$$\begin{aligned} v_1 + v_2 + v_5 &= 0 \qquad (9.192) \\ v_3 + v_4 + v_6 &= 0 \qquad (9.193) \\ v_1 + v_3 + v_7 &= 0 \qquad (9.194) \\ v_2 + v_4 + v_8 &= 0 \qquad (9.195) \end{aligned}$$

The distance between points in the QAM constellation is 2. The points are transmitted over an AWGN with SNR=13 dB, and the point [1.4, 2.6, 3.1, -.9] is received. Any point satisfying the constraints is equally likely to have been transmitted.

a. What is $\bar{b}$ for this transmission system? (.5 pt)

b. Draw a trellis for this code (use hexadecimal notation for the 4 bits on the 1^{st} QAM symbol followed by hexadecimal notation for the 4 bits on the 2^{nd} QAM symbol). (2 pts)

c. What 4D point is most likely to have been transmitted, and what is the probability of symbol error? (1 pt)

d. Find the parity matrix and a systematic generator matrix for this code ("I" matrix can appear for bits $[v_4\ v_3\ v_2\ v_1]$ in the systematic generator) . (2 pts)

e. Draw a constraint-based iterative decoding diagram for this code if probability of bit error is to be minimized for each of $[v_4\ v_3\ v_2\ v_1]$. What can you say about $[v_8\ v_7\ v_6\ v_5]$? How many cycles of the parity nodes are necessary to finish? (2.5 pts)

f. Find the values for each of the bits $[v_4\ v_3\ v_2\ v_1]$ by iterating your diagram in part e so that the probability of bit error for each of these 4 bits is individually minimized. (3 pts)

g. What is the probability of error for each of these bits? (1 pt)

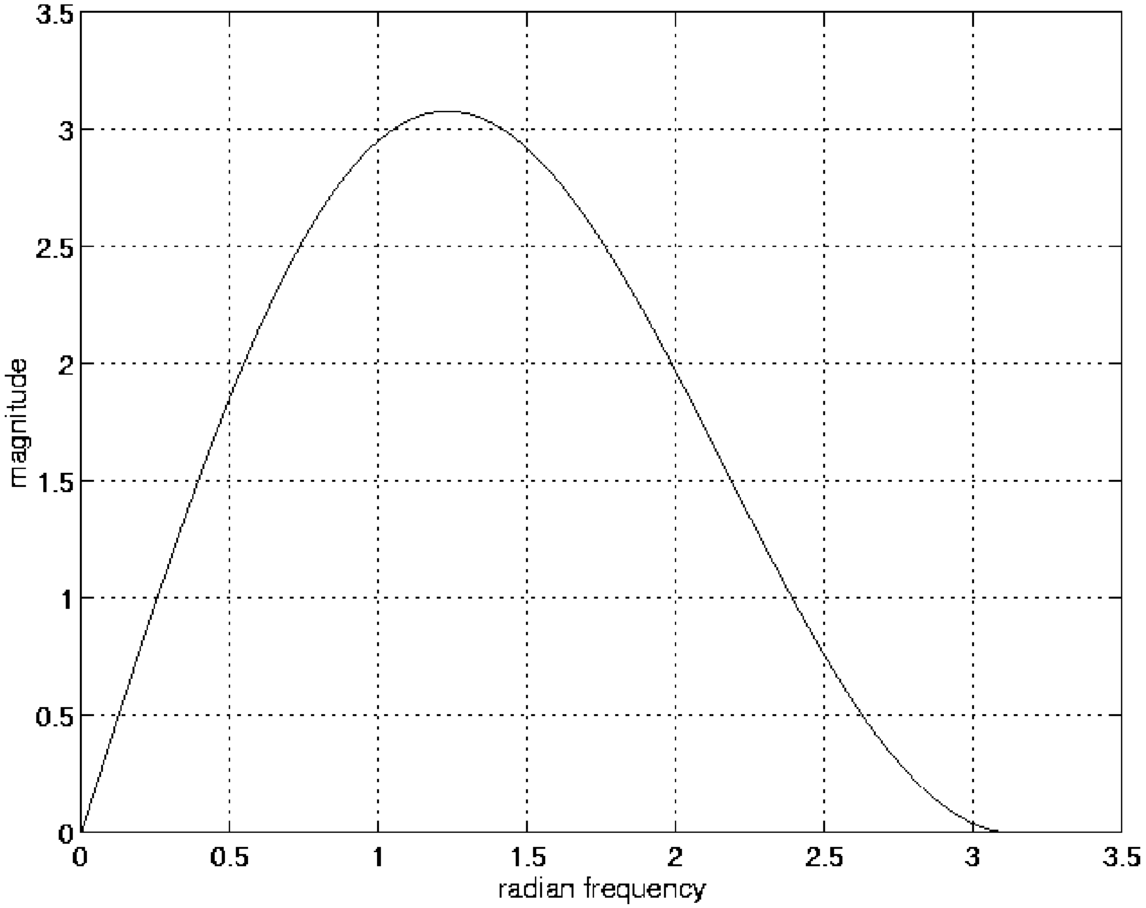


Figure 9.29: Plot of magnitude of $H(D)$ versus normalized frequency.

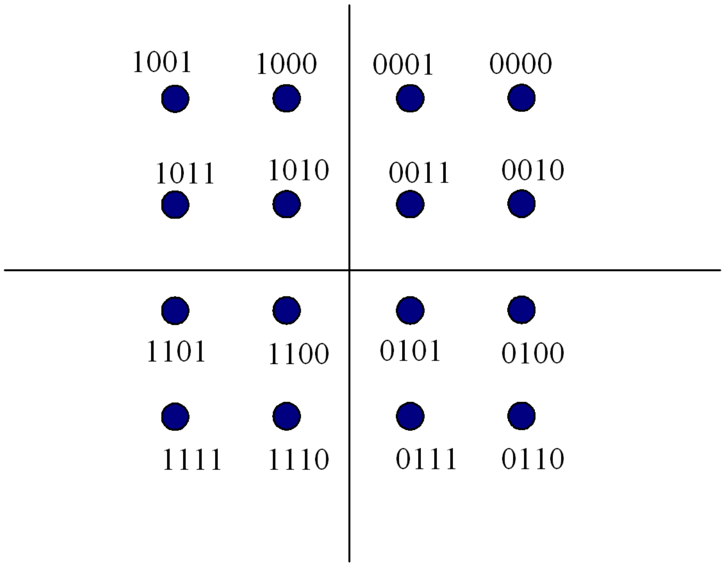


Figure 9.30: Constellation for Problem 9.26