Decoding Methods

Chapter 7

Decoding Methods

Chapter 2 introduced codes and sequences, tacitly assuming a maximum likelihood receiver that compares the channel-output sequences with all possible codewords and then selects the most likely sequence. With large-N, the decoder's implementation complexity can be enormous, approaching infinite complexity exponentially fast with N. Fortunately, many optimum and suboptimum decoding methods reduce this complexity. Even infinite-length-codeword codes can often be decoded with a complexity that is finite in real time. This chapter studies several decoding algorithms that are implementable.

Section 7.1 has examples and a general description of the **Viterbi Algorithm**, a method that allows exact implementation of **Maximum Likelihood Sequence Detection (MLSD)** for Chapter 2's convolutional codes and also for Chapter 3's partial-response channels. The Viterbi algorithm applies to decoding of any channel and/or code's treills with finite 2^{ν} states. if $\nu < \infty$ for any sequential encoder (including partial response). MLSD has finite complexity per subsymbol period and a reasonable decoding delay. The Viterbi Algorithm applies to both the BSC and the AWGN, as well as any channel that has a state-machine description, equivalent a Markov model. MLSD's analysis appears in Section 7.2, with particular examples of the $1 + D$ soft-decoded partial-response channel and a hard-decoded use of Chapter 2's 4-state convolutional code. Section 7.2 then also introduces former EE379 student George Ginis' code analysis "dmin" program. This program computes the essential parameters for any trellis so that its symbol-error and bit probabilities can be accurately estimated.

Section 7.3 proceeds to decoders that instead minimize bit, or possibly subsymbol, error probability, instead of sequences. These algorithms can improve performance and also provide soft information about the decoder decision's reliability for each bit or subsymbol. The fully optimum solution is the **"à posteriori probability " (APP) or Bahl-Cooke-Jelinek-Ravin (BCJR)** algorithm that minimizes individual bit (or also subsymbol) error probabilities. 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 8. An ad-hoc approximation of the MAP detector using the Viterbi Detector is known as the **Soft-Output Viterbi Algorithm (SOVA)** and also appears in Section 7.3. Section 7.4 looks at soft information from the perspective of channel or code constraints. Section 7.5 then proceeds with sub-optimum **iterative decoding** methods that can be applied to situations in which multiple codes have been concatenated as in Chapter 8, but is sufficiently general to apply to a number of different situations with just a single, or even no, code. Section 7.5 addresses the conversion of symbol-based system into binary likelihood-ratio-based coding, which can greatly simplify Sections 7.4 and 7.5's iterative decoder implementation without performance loss.

7.1 Trellis-based MLSD and the Viterbi Algorithm

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

$$\hat{\boldsymbol{x}}(D) = \arg\left\{\max_{\hat{\boldsymbol{x}}(D)} p_{\boldsymbol{y}(D)/\boldsymbol{x}(D)}\right\} \quad . \tag{7.1}$$

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

Figure 7.1 provides a simple example.

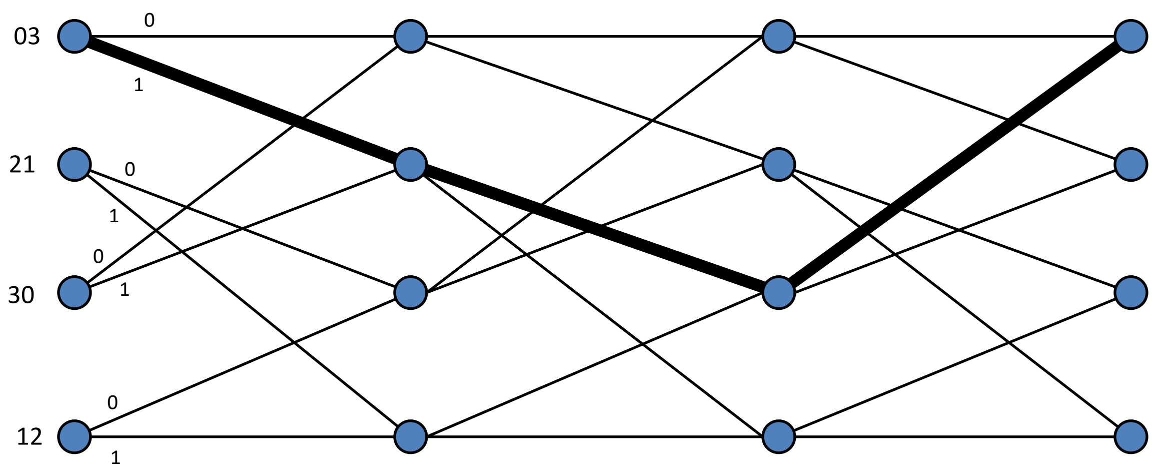


Figure 7.1: ML selects trellis path.

The thick 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 detector might wait until receiver acquires the entire sequence and then compare it against all possible encoder sequences. The Viterbi Algorithm recursively updates the 2^{ν} best trellis paths and eliminates other paths that no longer need consideration. These are the"surviving" paths into each state.

Thus, this optimum MLSD receiver's complexity need not be infinite (per unit time), as long as $\nu < \infty$, even for a semi-infinite input data sequence. Subsection 7.1.1 introduces the recursive Viterbi MLSD procedure using the example of a 4-state convolutional code. Subsection 7.1.2 then provides the general Viterbi Algorithm.

7.1.1 MLSD for a Convolutional Code

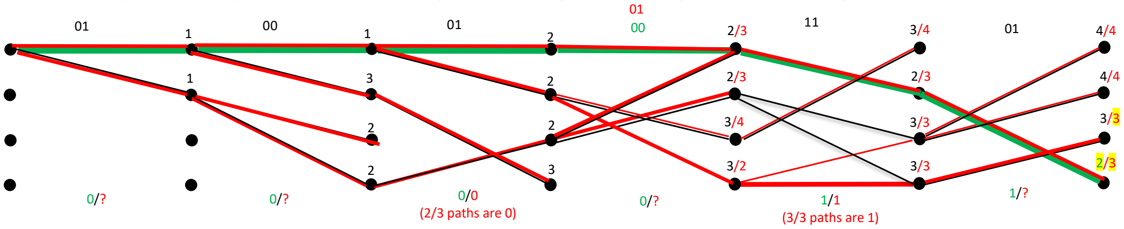
Chapter 2 introduced convolutional codes, while Chapter 8 details them further. This subsection's example applies to a BSC and MLSD compares sequences using the Hamming distance or number of channel-output bits that differ from a codeword. This example uses the Viterbi Algorithm.

Figure 7.2 illustrates the algorithmic calculations for Figure 7.1's trellis, with abbreviations for the code's subsymbol outputs on the left as: 0= 00; 1 =01; 2=10; and 3=11. The upper path out of each state corresponds to input bit 0, and the lower path to input bit 1. The black (or green / red for two

[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.

different) received sequence is above the trellis. Initial view should ignore all red-colored quantities. This test case has the known green input sequence below the trellis. Since this code has $r = 1/2$, there are two output bits for every input bit. The numbers above the states (black dots) are the survivor paths' Hamming distance from the received sequence. The VA at each state compares (up to) two paths into it. The VA adds Hamming distance for the branch's comparison to the received sequence to the accumulated distance for that path's previous state. The smallest then determines the survivor.

Green outputs – BSC-output 2 errors → correct sequence ; red output – 3 BSC output errors → two sequences tied (detect error)



All input bits correctly detected ; 2/6 input bits correct, 4 ambiguous (using majority vote)

Figure 7.2: Viterbi decoder example.

For instance, in the first stage, the trellis (code) starts in state 0 (at the top). Hamming distance is initially 0, but both branches differ by 1 from the received bits. Thus, both possible next states have cumulative distance 1. In the next stage, all 4 states have a survivor. with the distances shown. Stage 3 has two possibilities into each state that VA compares. The smallest survives and VA assigns the cumulative distance to that state. This ecample process terminates after 6 steps. The green path has accumulated distance 2, which is lowest and is the MLSD decision. This is also the known correct path for this example. Because this exampole code has $d_{free} = 5$, MLSD correct two output errors as it decides the green survivor sequence. The two output erred bits occurred in stage 1 and in stage 3, but the VA ML detection corrects them; more importantly the corresponding input sequence is correct (see green input bits below the trellis).

A second output sequence differs only in stage 4, adding 1 more BSC error in red color. Now with 3 errors, MLSD does not unambiguously find the correct path (2 final states both have the same cummulative distance 3 - MLSD could decide either. So in this second 3-output-bit-error case, VA MLSD detects the error, but cannot correct it. This would be an "error event" (see Subsection 7.2.1, Definition 7.2.1 for a formal definition). The input-bit decisions appear below Figure 7.2's trellis. All 6 inputs are correct for the first (green) case. However, only 3 of the 6 are correct for the red situation; for the correct bits, even though the detected sequence is ambiguous, the survivor's corresponding inputs are the same.

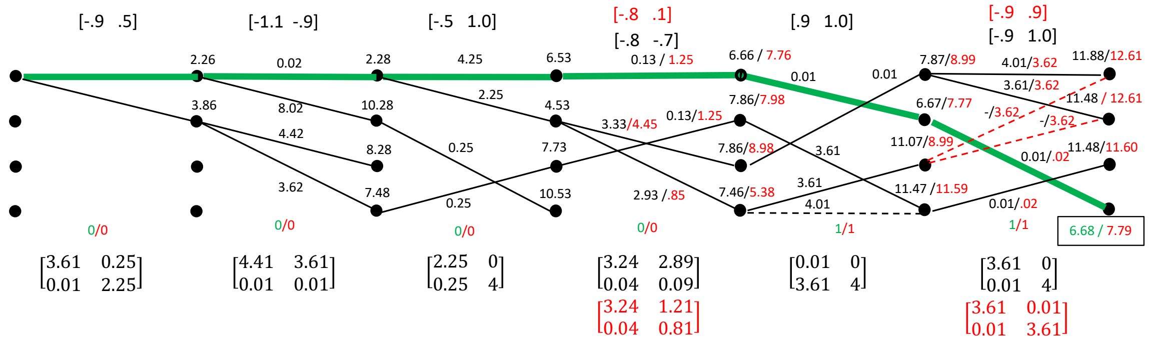

[-.9 .5]
[-1.1 -.9]
[-.5 1.0]
[-.8 .1]
[-.8 -.7]
[.9 1.0]
[-.9 .9]
[-.9 1.0]
2.26
0.02
2.28
4.25
6.53
0.13 / 1.25
6.66 / 7.76
0.01
0.01
7.87/8.99
4.01/3.62
11.88/12.61
3.86
8.02
10.28
2.25
4.53
3.33/4.45
0.13/1.25
7.86/7.98
6.67/7.77
3.61/3.62
11.48 / 12.61
-/3.62
-/3.62
4.42
8.28
0.25
7.73
7.86/8.98
3.61
11.07/8.99
0.01/.02
11.48/11.60
3.62
7.48
0.25
10.53
2.93 /.85
7.46/5.38
3.61
4.01
11.47 /11.59
0.01/.02
0/0
0/0
0/0
0/0
1/1
1/1
6.68 / 7.79
$\begin{bmatrix} 3.61 & 0.25 \\ 0.01 & 2.25 \end{bmatrix}$
$\begin{bmatrix} 4.41 & 3.61 \\ 0.01 & 0.01 \end{bmatrix}$
$\begin{bmatrix} 2.25 & 0 \\ 0.25 & 4 \end{bmatrix}$
$\begin{bmatrix} 3.24 & 2.89 \\ 0.04 & 0.09 \end{bmatrix}$
$\begin{bmatrix} 3.24 & 1.21 \\ 0.04 & 0.81 \end{bmatrix}$
$\begin{bmatrix} 0.01 & 0 \\ 3.61 & 4 \end{bmatrix}$
$\begin{bmatrix} 3.61 & 0 \\ 0.01 & 4 \end{bmatrix}$
$\begin{bmatrix} 3.61 & 0.01 \\ 0.01 & 3.61 \end{bmatrix}$

Figure 7.3: Viterbi decoder example, AWGN.

Figure 7.3 uses instead squared distance for the AWGN output samples shown. The procedure is the same, but the calculations are more tedious. The matrices below the trellis just appear to help the reader calculate the various brank metrics. The black channel outputs correspond to Figure 7.2's output sequence for both black and red/green. In this case, because soft decoding is 3 dB better (Chapter 2), the VA correctly decodes even the red paths. The reader may want to trace by finger with hand calculation the various survivor path selections and metrics. This helps understand the Viterbi Algorithm. This example assumes the code starts in state 00 (at the top). Indeed, as some software shows later in this section, a yet better path exists that starts in state 01 and produces the outputs 1 1 0 0 1 1 and a lower metric of 6.78.

7.1.2 Sequence Detection with the Viterbi Algorithm

Section 7.1.1's convolutional-code sequence-detection example specifically uses the Viterbi Algorithm. This section generalizes this algorithm's description. Viterbi introduced the algorthm for convolutional-code decoding in 1967 [1] - its application to partial-response channels was first published by Forney in 1972 [2]. 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 applies to any trellis with $|C|^{\nu} < \infty$ states. For convolutional codes, $M = 2$ even if BICM maps binary codewords to constellation points, while for partial-response uses, $|C|^{\nu}$.

Several quantities help describe the Viterbi Algorithm;

state index - i, $i = 0, 1, ..., M^{\nu} - 1$

state metric for state i at sampling time $k \stackrel{\Delta}{=} \mathscr{U}_{i,k}$ (sometimes called the "path metric")

previous-states set to state $i \stackrel{\Delta}{=} J_i$ (that is, states that have a transition into state i)

branch value $\tilde{\boldsymbol{y}}_k(j \to i)$ noiseless output corresponding to a transition 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, e.g. for BSC, $d_H(\boldsymbol{y}_k, \hat{\boldsymbol{v}}_k)$, or for AWGN $\Delta_{j,i,k} \stackrel{\Delta}{=} \|\boldsymbol{y}_k - \hat{\boldsymbol{x}}_k(j \to i)\|^2$

survivor path $\bar{j}_i$ - the path that has minimum metric coming into state i.

Definition 7.1.1 *[Viterbi Detection] Set $\mathscr{U}_{-1} = \mathscr{U}_{i,-1} = 0$ and recursively minimize for each state $i = 0, ..., 2^{\nu} - 1$ (or $|C|^{\nu} - 1$ for partial response) at each time $k \geq 0$*

$$\mathscr{U}_{i,k} = \min_{j \in J_i} \left[\mathscr{U}_{j,k-1} + \Delta_{j,i,k} \right] \quad . \tag{7.2}$$

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

The survivor paths save $\bar{j}_i$, which theoretically can have infinite length. Implemention truncates survivor paths to typically length $5 \cdot \nu$, a rule-of-thumb suggested by Viterbi. Such truncation has very little loss. Furthermore, to prevent the accumulated metrics' overflow , the same (negative) amount can be added, periodically, to bound them. This constant can equal the smallest metric's negative magnitude, thus always zeroing this smallest metric. Equivalently, circular overflow may be used as long as the maximum metric difference is less than one-half full dynamic range.

The basic Viterbi Decoder operations are 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 that state's metric with the new metric for subsequent iterations. The decoder then augments the survivor path by the corresponding branch's subsymbol. This operation set has the name **Add-Compare-Select (ACS)**.

7.1.3 MLSD for the Additive White Gaussian Noise Channel

The AWGN's MLSD finds the trellis' survivor path $\hat{\boldsymbol{x}}(D)$ that satisfies

$$\min_{\hat{x}(D)} \|\boldsymbol{y}(D) - H(D) \cdot \hat{\boldsymbol{x}}(D)\|^2 \quad . \tag{7.3}$$

When $H(D) = I$, this corresponds to the convolutional code with 2^{ν} states. For partial response channels, $H(D)$ represents the controlled intersymbol interference effect. This chapter generalizes the noiseless sequence's notation to $\tilde{\boldsymbol{y}}(D)$ where

$$\tilde{\boldsymbol{y}}(D) = H(D) \cdot \boldsymbol{x}(D) \quad . \tag{7.4}$$

The minimum distance d_{min} is then (as always) the distance between the two closest trellis sequences

$$d_{min} = \min_{\tilde{\boldsymbol{y}}'(D) \neq \tilde{\boldsymbol{y}}(D)} \|\tilde{\boldsymbol{y}}'(D) - \tilde{\boldsymbol{y}}(D)\| \quad . \tag{7.5}$$

Sometimes finding the two closest sequences is easy, and sometimes it is extremely difficult and requires Section 7.2's "Ginis dmin" search program. However, d_{min} always exists. Equivalently, an **error event** is

$$\boldsymbol{\epsilon}(D) \stackrel{\Delta}{=} \tilde{\boldsymbol{y}}'(D) - \tilde{\boldsymbol{y}}(D) \quad . \tag{7.6}$$

The zero sequence is not an error event. Thus, $d_{min} = \min_{\boldsymbol{\epsilon}(D) \neq 0} \|H \cdot \boldsymbol{\epsilon}(D)\|$.

MLSD's nearest-neighbor generalization: MLSD analysis' N_e counts ONLY those sequences at distance d_{min}, so

$$\begin{align} N_e \;\stackrel{\Delta}{=}\; & \text{the number of symbol errors \textit{at any single} subsymbol time index } k \text{ that could arise from any} \tag{7.7} \\ & \text{past ``first'' error with distance } d_{min} \ . \tag{7.8} \end{align}$$

A "first error" loosely means the earlier point in time where the error event begins; N_e counts any such first path separation that must end only at time k. This definition of N_e refines that of Chapter 1, which there includes any neighbor (even at larger distance) with a common decision boundary. MLSD usually complicates finding such an N_e by searching in an infinite-dimensional space for all neighbors who might have a common decision boundary, thus the refined definition includes only those at distance d_{min}. With this refined definition, the NNUB becomes an approximation of, and not necessarily an upper bound on, MLSD's error probability:

$$P_e \approx N_e \cdot Q\left(\frac{d_{min}}{2\sigma}\right) \quad . \tag{7.9}$$

Often (7.9) 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 (and partial-response) systems instead investigate the set of distances

$$\mathcal{D}_{min} = \{d_{min} = d_0 < d_1 < d_2 < .... < d_i\} \quad , \tag{7.10}$$

The set of distances includes d_{min} and also 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 ocurrences on average for all error-event sequences merging at the same point in time, $N_{e,1}$, $N_{e,2}$, ... $N_{e,i}$ (ordered in terms of the index i of the corresponding $d_{min}(i)$). The nearest neighbors can also simply be indexed to d as N_d.

Thus,

$$P_e \leq \sum_{i=0}^{\infty} N_{e,i} \cdot Q\left(\frac{d_i}{2\sigma}\right) = \sum_{d \in \mathcal{D}_{min}} N_d \cdot Q\left(\frac{d}{2\sigma}\right) \quad , \tag{7.11}$$

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.

Implied stationarity: In many cases, MLSD's probability of a specific error event's ocurrence tends to 1 if the sequence has infinite length (stating only that eventually an error is made and not necessarily poor performance.[2]). A oft-encountered performance is per-subsymbol error probability in that it applies to the common point in time that starts (or ends, but not both) all the evaluated sequences in computing the distance spectrum $\{d_{min}(i)\}$ and N_i. Such an error could commence (or end) at any time, so this is a subsymbol-error probability corresponding to choosing the best sequence $\hat{\boldsymbol{x}}(D)$. This subsymbol error probability tacitly presumes stationarity (a constant state diagram or equivalently a trellis description that is the same for each stage, as with all examples here). Most block codes do not exhibit such stationarity, so the designer must evaluate error events beginning at each and every symbol time instant within the block, and then average, to measure a meaningful subsymbol-error probability.

7.1.3.1 Viterbi MLSD for $H(D) = 1 + D$ Partial Response

Figure 7.4 displays the $1+D$ partial-response channel's trellis with $|C| = 2$. MLSD computes the squared distance $\|\boldsymbol{y}(D) - H(D) \cdot \hat{\boldsymbol{x}}(D)\|^2$ for the paths, selecting the path with smallest such squared distance.

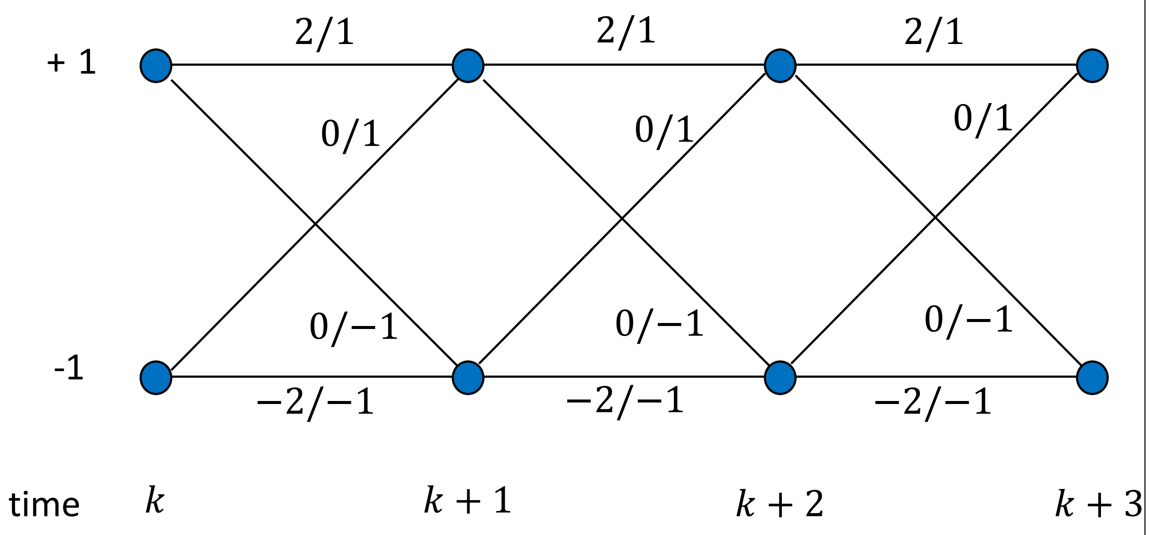


Figure 7.4: Trellis Diagram for $1 + D$ partial-response/sequential-encoder with $b = 1$ PAM.

More explicitly, this channel's the ML detector minimizes the "cost"

$$\mathscr{U}_k \triangleq \sum_{m=0}^{k} |y_m - (\hat{x}_m + \hat{x}_{m-1})|^2 = \|y - h * x\|^2 \quad . \tag{7.12}$$

[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.

When the MLSD-selected sequence is correct, the cost is the sum of squared channel noise sanokes over $k+1$ successive symbols. The MLSD-selected sequence of length $k+2$ minimizes

$$\sum_{m=0}^{k+1} |y_m - (\hat{x}_m + \hat{x}_{m-1})|^2 = \left[\sum_{m=0}^{k} |y_m - (\hat{x}_m + \hat{x}_{m-1})|^2 \right] + |y_{k+1} - (\hat{x}_{k+1} + \hat{x}_k)|^2 \tag{7.13}$$

$$\mathscr{U}_{k+1} = \mathscr{U}_k + |y_{k+1} - (\hat{x}_{k+1} + \hat{x}_k)|^2 \quad . \tag{7.14}$$

As with the convolutional code, and in general, the updated cost need not consider the history before the previous state because that is the lowest cost already to that state.

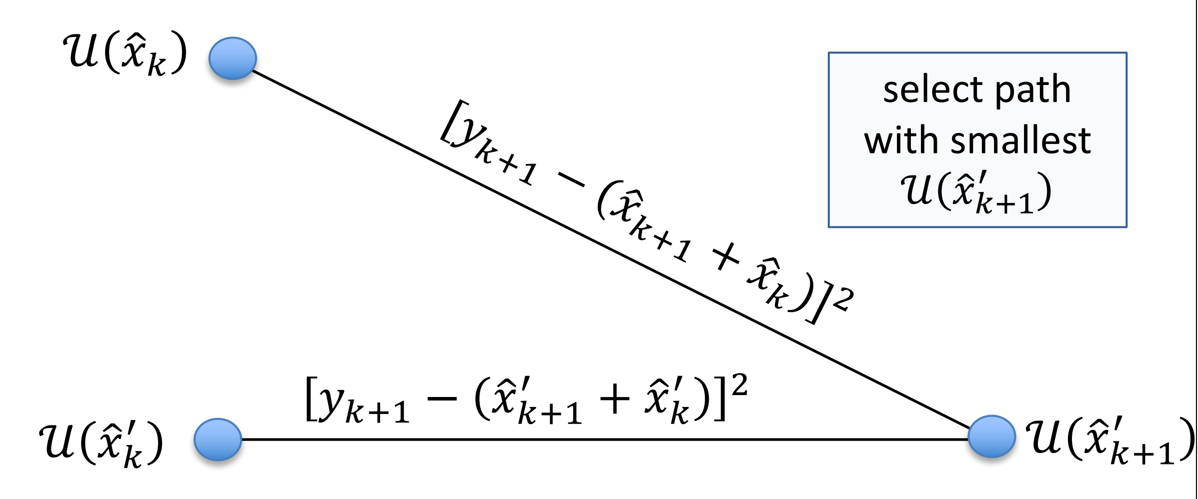

$\mathcal{U}(\hat{x}_k)$
$[y_{k+1} - (\hat{x}_{k+1} + \hat{x}_k)]^2$
select path with smallest $\mathcal{U}(\hat{x}'_{k+1})$
$[y_{k+1} - (\hat{x}'_{k+1} + \hat{x}'_k)]^2$
$\mathcal{U}(\hat{x}'_k)$
$\mathcal{U}(\hat{x}'_{k+1})$

Figure 7.5: Iterative cost minimization for Duobinary ($H(D) = 1 + D$)channel.

Figure 7.5 illustrates this concept. For each state at time k, there is a unique minimum cost $\mathscr{U}(\hat{x}_k)$ 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 $\mathscr{U}(\hat{x'}_k)$ has a different smallest cost for all paths up to time k that terminate in state x'_k. Both the paths in Figure 7.5 merge into a common state at time $k+1$. The smallest cost associated with that state at time $k+1$ adds the branch metric to the previous state's cost. The survivor path is the path with the smaller new cost. The process recursively repeats for future times.

For $H(D) = 1 + D$ with binary inputs, MLSD decides which 2 of the 4 possible extensions are survivors. The following example illustrates with some numbers.

EXAMPLE 7.1.1 *[Viterbi Decoding example for $H(D) = 1 + D$]* Suppose the transmitter uses differential encoding (or a precoder as in Section 3.13) with the binary $1 + D$ channel. The precoder input bits are [10101]. The encoder sets the initial channel state according to the last message being zero. The receiver does not know this in this example. The following table summarizes the channel signals for precoding and symbol-by-symbol detection:

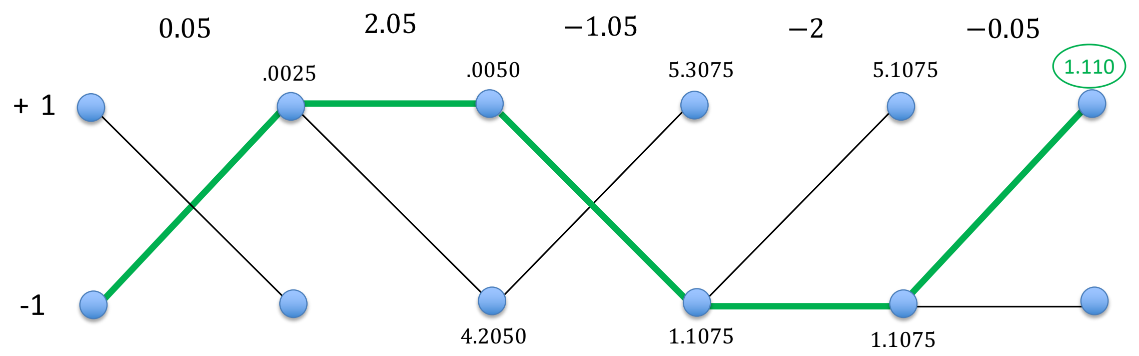


Figure 7.6: Sequence detection example.

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

Symbol-by-symbol selects $\hat{m} = [10001]$, which has one error in the middle position, $m_{k+3} \neq 0$.

Figure 7.6 illustrates the MLSD result and survivor. The precoder may appear unnecessary with the sequence detection system. However, without precoding, long trellis error-event strings (i.e., picking a path that deviates for a while from the correct path) will result in long strings of bit errors. However, for these same trellis paths, the precoding has only 2 input bit errors. 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-length trellis sequences – this is called **quasi-catastrophic error propagation** in partial-response channels.

7.1.3.2 Matlab's convolutional-code Viterbi Detector

Matlab provides the core vitdec.m program (no source code available[3] program. The vitdec.m program will handle Section 7.1.1's 3 convolutional-code examples:

```
vitdec Convolutionally decode binary data using the Viterbi algorithm.
    DECODED = vitdec(CODE,TRELLIS,TBLEN,OPMODE,DECTYPE) decodes the vector CODE
    using the Viterbi algorithm.  CODE is assumed to be the output of a
    convolutional encoder specified by the MATLAB structure TRELLIS.  See
    POLY2TRELLIS for a valid TRELLIS structure.  Each symbol in CODE consists
    of log2(TRELLIS.numOutputSymbols) bits, and CODE may contain one or more
    symbols.  DECODED is a vector in the same orientation as CODE, and each of
    its symbols consists of log2(TRELLIS.numInputSymbols) bits.  TBLEN is a
    positive integer scalar that specifies the traceback depth.
       OPMODE denotes the operation mode of the decoder. Choices are:
       'trunc' : The encoder is assumed to have started at the all-zeros state.
```

[3] The next subsection provides source code for a less-polished but helpful program that handles convolutional codes and partial response.

```
             The decoder traces back from the state with the best metric.
   'term'  : The encoder is assumed to have both started and ended at the
             all-zeros state.  The decoder traces back from the all-zeros
             state.
   'cont'  : The encoder is assumed to have started at the all-zeros state.
             The decoder traces back from the state with the best metric.  A
             delay equal to TBLEN symbols is incurred.
   DECTYPE denotes how the bits are represented in CODE.  Choices are:
   'unquant' : The decoder expects signed real input values.  +1 represents
               a logical zero and -1 represents a logical one.
   'hard'    : The decoder expects binary input values.
   'soft'    : See the syntax below.
DECODED = vitdec(CODE,TRELLIS,TBLEN,OPMODE,'soft',NSDEC) decodes the input
vector CODE consisting of integers between 0 and 2^NSDEC-1, where
0 represents the most confident 0 and 2^NSDEC-1 represents the most
confident 1.
Note that NSDEC is a required argument if and only if the decision type is
'soft'.
DECODED = vitdec(CODE, TRELLIS, TBLEN, OPMODE, DECTYPE, PUNCPAT)
decodes the input punctured CODE where PUNCPAT is the puncture pattern
vector of 1's and 0's with 0's indicating where the punctures occurred
in the data stream.
DECODED = vitdec(CODE, TRELLIS, TBLEN, OPMODE, DECTYPE, PUNCPAT, ERASPAT)
allows an erasure pattern (ERASPAT) vector to be specified for the input
CODE where the 1's indicate the corresponding erasures. ERASPAT and CODE
must be of the same length. If puncturing is not used, specify PUNCPAT
to be [].
DECODED = vitdec(..., 'cont', ..., INIT_METRIC,INIT_STATES,INIT_INPUTS)
provides the decoder with initial state metrics, initial traceback states
and initial traceback inputs.  Each real number in INIT_METRIC represents
the starting state metric of the corresponding state.  INIT_STATES and
INIT_INPUTS jointly specify the initial traceback memory of the decoder.
They are both TRELLIS.numStates-by-TBLEN matrices.  INIT_STATES consists of
integers between 0 and TRELLIS.numStates-1.  INIT_INPUTS consists of
integers between 0 and TRELLIS.numInputSymbols-1.  To use default values for
all of the last three arguments, specify them as [],[],[].
[DECODED FINAL_METRIC FINAL_STATES FINAL_INPUTS] = vitdec(..., 'cont', ...)
returns the state metrics, traceback states and traceback inputs at the end
of the decoding process.  FINAL_METRIC is a vector with TRELLIS.numStates
elements which correspond to the final state metrics.  FINAL_STATES and
FINAL_INPUTS are TRELLIS.numStates-by-TBLEN matrices.
Example:
    t = poly2trellis([3 3],[4 5 7;7 4 2]);  k = log2(t.numInputSymbols);
    msg = [1 1 0 1 1 1 1 0 1 0 1 1 0 1 1 1];
    code = convenc(msg,t);    tblen = 3;
    [d1 m1 p1 in1]=vitdec(code(1:end/2),t,tblen,'cont','hard')
    [d2 m2 p2 in2]=vitdec(code(end/2+1:end),t,tblen,'cont','hard',m1,p1,in1)
    [d m p in] = vitdec(code,t,tblen,'cont','hard')
    % The same decoded message is returned in d and [d1 d2].  The pairs m and
    % m2, p and p2, in and in2 are equal.  Note that d is a delayed version of
    % msg, so d(tblen*k+1:end) is the same as msg(1:end-tblen*k).
```

The program is both complex with many opotions and limited to convolutional codes. Several successive examples perhaps best illustrate its use, and in particular initially beginning with those in Subsection 7.1.1.

EXAMPLE 7.1.2 *[Revisit Subsection 7.1.1's examples with matlab]* Figure 7.2's MLSD corresponds to these matlab commands (Section 8.1 describes the poly2trellis command, which basically creates the encoder corresponding to $G(D) = [7\ \ 5]$ in octal (111 101 $=$ $D^2 + D + 1\ \ D^2 + 1$.

```
>> t=poly2trellis (3, [7 5]);
>> msg=[0 0 0 0 1 1];
code=convenc(msg,t) =
     0 0     0 0     0 0     0 0     1 1     0 1
% with no errors
>> vitdec(code,t,6,'trunc','hard')  =
     0     0     0     0     1     1
% with 2 errors
>> y=[0 1 0 0 0 1 0 0 1 1 0 1];
vitdec(y,t,6,'trunc','hard')  =
     0     0     0     0     1     1
% even with 3 errors
y3errors=[0 1 0 0 0 1 0 1 1 1 0 1];
>> vitdec(y3errors,t,6,'trunc','hard') =
     0     0     0     0     1     1
```

The decoder even corrects 3 errors, for which Section 7.4 suggests what vitdec.m may be executing internally when there are ties. The decoder also works for the systematic encoder, but this time only with 2 errors successfully

```
>> t3=poly2trellis(3,[7 5],7);
>> msg = [     0     0     0     0     1     1];
>> y3=convenc(msg,t3)
 0 0     0 0     0 0     0 0     1 1     1 0
 >> errors3=[ 0 1 0 0 1 0 0 0 1 0 0 0];
>> vitdec(xor(y3,errors3),t3,6,’trunc’,’hard’)
   0   1   1   0   0   1
>> errors2=[ 0 1 0 0 0 0 0 0 1 0 0 0];
>> vitdec(xor(y3,errors2),t3,6,’trunc’,’hard’) =
   0   0   0   0   1   1
```

Finally for the AWGN example:

```
>> yawgn=[-.9 .5 -1.1 -.9 -.5 1 -.8 -.7 .9 1 -.9 1];
>> vitdec(-yawgn,t,6,’trunc’,’unquant’) =
     0     0     0     0     1     1
>> % With revised 3-output-bit-errors
yawgn2=yawgn;
yawgn2(8)=.1;
yawgn2(12)=.9;
vitdec(-yawgn2,t,6,’trunc’,’unquant’) =
     0     0     0     0     1     1
```

Note the negation of the channel output to corresponding to matlab’s convention of a binary 1 corresponding to the level -1 in PAM2. Also note that the mode is “unquant,” not “soft,” for soft decoding. The mode “soft” corresponds to the channel output being soft information from another decoder.

Combined use of poly2trellis.m and vitdec.m unfortunately has a bug for most systematic (or feedback-using) encoder that has $k > 1$. Poly2trellis often increases unnecessarily the number of trellis states and often creates (see Chapter 8 a catastropic-encoder trellis. Section 7.3.3 provides an alternative deocoding solution that circumvents this matlab defficiency.

7.1.3.3 Matlab Partial-Response MLSD Programs:

Two MLSD matlab programs are available. The first is for partial-response channels or convolutional codes by former EE379 student Dr. Ghazi Al-Raw. It is self contained, but requires some effort to generate the input trellis descriptions:

```
help mlsd1D
  function mh = MLSD(y,v,b,Sk, Yk, Xk)
  MLSD using VA for input trellis
  Written by Ghazi Al-Rawi
    Updated significantly by J. Cioffi, 2023
  ********************************************************************
  INPUTS
  y  channel output sequence
        1 x K complex (usually real) number for partial response
           K is number of input bits in Xk
        n x K integer if BSC with convolutional code
    v  constraint length (log2 of the number of states)
    b  number of bits per subsymbol
    Sk 2^v x 2^b previous-state description matrix
        e.g., EPR4’s H=[1 1 -1 -1] has (binary) nextstate trellis
        nextState = [1 2; 3 4; 5 6; 7 8; 1 2; 3 4; 5 6; 7 8];
        so Sk = [1 5; 1 5; 2 6; 2 6; 3 7; 3 7; 4 8; 4 8];
    Yk 2^v x 2^b noiseless 1D trellis output corresponding to Sk, so EPR4
        Yk = [0 -2; 2 0; 2 0; 4 2; -2 -4; 0 -2; 0 -2; 2 0];
        for 4-state convolutional code
        Yk= [ 0 3 ; 2 1 ; 3 0 ; 1 2]
    Xk b x 2^v input vector, so EPR4 could be
        Xk = [0 1 0 1 0 1 0 1];
    e  if e=1, then euclidean distance, otherwise hamming distance (xor)
  OUTPUT
  mh is the detected message sequence
  ********************************************************************
```

The previous 1+D example works as

```
>> y=[.05 2.05 -1.05 -2 -.05];
v=1;
b=1;
>> Sk=[1 2 ; 1 2];
>> Yk1=[2 0 ; 0 -2];
>> Xk = [ 1 0];
>> mh = mlsd1D(y, v, b, Sk, Yk1, Xk,1)

mh =     1     1     0     0     1
```

The output matches Figure 7.6's 4th row or x_k with 1 mapping to 1 and 0 mapping to -1.

A more complex sxample uses the channel $H(D) = 1 + D - D^2 - D^3$ with binary inputs.

```
>> y = [     4     0    -4     2    -2     0     4     2 ];
>> v =     3;
>> b =     1;
>> Sk  = [
     1     5
     1     5
     2     6
     2     6
     3     7
     3     7
     4     8
     4     8 ];

>> Yk = [
     0    -2
     2     0
     2     0
     4     2
    -2    -4
     0    -2
     0    -2
     2     0 ];
>> Xk = [     0     1     0     1     0     1     0     1 ];
>> mh = mlsd1D(y, v, b, Sk, Yk, Xk,1)
mh =     1     0     1     0     0     1     1     1
```

This is the input that produces the output since there is no noise.

Running again with some significant noise produces different output:

```
mh = mlsd1D(y+[-.9 .3 .4 -.6 -1 -1.2 -.8 .1], v, b, Sk, Yk, Xk,1)

     0     1     0     1     1     1     0     1
```

The reader can try smaller noise offsets and see that the decoder produces the same as zero noise.

For the 4-state trellis in Figure 7.2, the use is

```
>> y = [     1     0     1     0     3     1 ];
>> v =     2 ;
>> Sk = [
     1     3
     1     3
     2     4
     2     4 ];

>> Yk = [
     0     3
     3     0
     2     1
     1     2 ];

>> Xk = [     0     1     0     1 ];

>> mh = mlsd1D(y, v, b, Sk, Yk, Xk,0)

mh =     0     0     0     0     1     1
```

This matches Figure 7.2. For the soft MLSD decoding of the same channel as in Figure 7.3.

```
>> ysoft = [   -0.9000   -1.1000   -0.5000   -0.8000    0.9000   -0.9000
                      0.5000   -0.9000    1.0000   -0.7000    1.0000    1.0000 ];
>> Yk
Yk(:,:,1) =
    -1     1
     1    -1
     1    -1
    -1     1
Yk(:,:,2) =
    -1     1
     1    -1
    -1     1
     1    -1
% Sk, v, b, Xk remain the same
 >> mh = mlsd1D(ysoft, v, b, Sk, Yk, Xk,1)

mh =     0     0     0     0     1     1
% or for the second output
>> ysoft(:,4)=[-.8 ; .1];
>> ysoft(:,6)=[-.9 ; .9];
>> ysoft =
  -0.9000   -1.1000   -0.5000   -0.8000    0.9000   -0.9000
    0.5000   -0.9000    1.0000    0.1000    1.0000    0.9000

>> mh = mlsd1D(ysoft, v, b, Sk, Yk, Xk,1)

mh =      0     0     0     0     1     1

>> Sk=[];
>> for i=1:8
[temp,I]=find(nextstate==i);
Sk=[Sk I];
end
```

The MLSD program matches the red sequence in Figure 7.3. Figure 7.3 assumes the code starts in state 00. However the program actually allows any starting state; indeed starting in other states can lead in some cases to a lower metric. The user may want to experiment with different outputs. A version of the program that forces a starting (or ending or both) state(s) awaits an extra-credit project by motivated student.

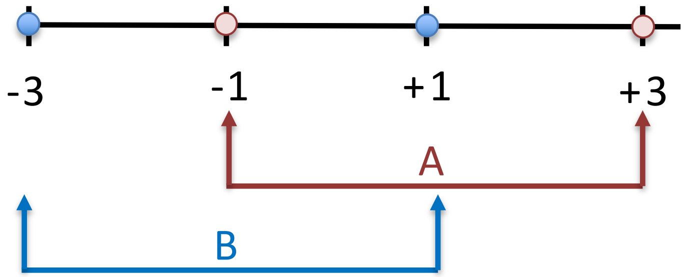


Figure 7.7: Partitioning for Reduced State Sequence Detection.

7.1.3.4 Reduced State Sequence Estimation Example

It is possible to reduce the number of states to 2 for the $1+D$ channel in implementing (nearly optimum) sequence detection for $|C| > 2$. Figure 7.7 illustrates a constellation labeling for the $|C| = 4$ case, introduced by Eyuboğlu and Qureshi [3]. The constellation partitions into two groups (A and B) with twice the minimum distance within each group.

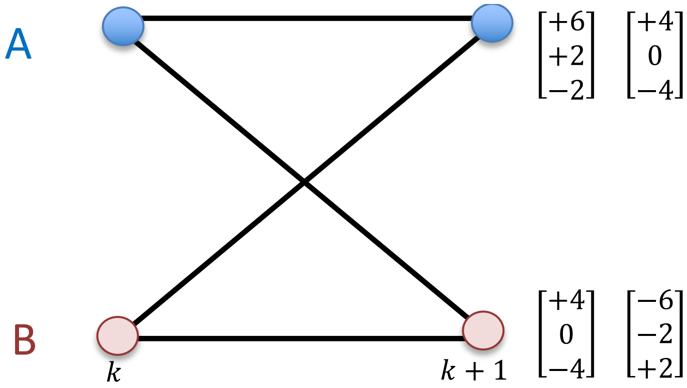

A
B
$\begin{bmatrix}+6\\+2\\-2\end{bmatrix}$ $\begin{bmatrix}+4\\0\\-4\end{bmatrix}$
$\begin{bmatrix}+4\\0\\-4\end{bmatrix}$ $\begin{bmatrix}-6\\-2\\+2\end{bmatrix}$
k
$k+1$

Figure 7.8: Reduced State Sequence Estimation trellis.

Figure 7.8 shows a new two-state trellis for the $|C| = 4$ $1+D$ channel. This new trellis states reflect only the set, A or B, that was last transmitted over the channel. The minimum distance between any two sequences through this trellis remains $d_{min} = 2\sqrt{2}$. 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 error probability of error will be approximately the same as the $|C|$-state detector.

In general for PAM or QAM constellations, it is possible to partition the signal constellation into subsets of increasing minimum distance (see the mapping-by-set-partitioning principles of Appendix B for a specific procedure for performing this partitioning). In reduced state sequence detection (RSSD, Eyuboglu, 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, $M' \leq |C|$ 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 RSSE cannot be used.

7.2 MLSD Analysis

MLSD Analysis computes the distance spectrum[4] $d_{min}(i),\ i = 0, 1, ...$ with $d_{min}(0) \triangleq d_{min}$ and their corresponding multiplicities N_i with $N_0 \triangleq N_e$. Determination of $d_{min}(i)$ and N_i can be complex. Several methods exist that find $\mathcal{D}_{min}$, the nearest-neighbor counts $N_{d\in\mathcal{D}_{min}}$, and even the corresponding bit-error probabilities.

Subsection 7.2.1 first discusses error events, Error-event analysis investigates directly the errors that can occur. Subsection 7.2.2 provides example analysis for the 4-state convolutional code, while Subsection 7.2.3 does so for the $1+D$ partial-response AWGN channel. Both subsections' results extend (with tedious calculation) to more complex convolutional codes and partial-response channels respectively. Section **??** provides Ginis's sequential encoder analysis program, which can input any sequential encoder's trellis and output the distance spectrum $\mathcal{D}_{min}$, $\{N_d\}$, and the average number of bit errors.

7.2.1 Error Events

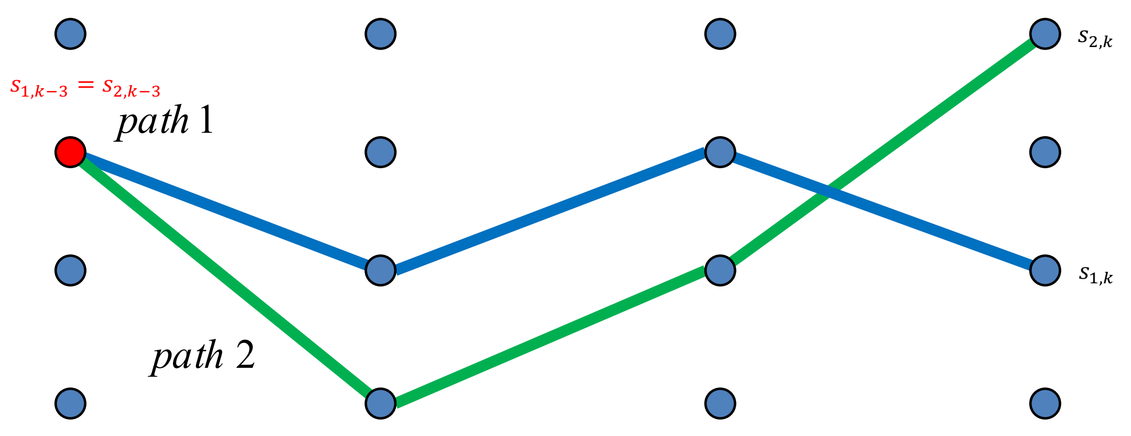


Figure 7.9: Example of trellis error event.

Error events enumerate the various ways in which a MLSD receiver can incorrectly decide upon a message sequence. Figure 7.9 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. This event may continue if the system continues operation. The event ends when the two paths merge again to perpetuity.

Definition 7.2.1 *[Error Event] An* **error event** *occurs when the decoder 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 this text's time-invariant trellis with $\nu \geq 1$, $k = 0$ without loss of generality.*

[4] For finite-field codes on a DMC, these distances have simpler indexing $d \in \mathcal{D}_{min} = \{d_{free} < d_{free}+1 < d_{free}+2...\}$, so d simplifies to an index for integers equal to or larger than d_{free}.

The two paths merge again at some later time $k+\ell_x = \ell_x$ *where* $0 < \ell_x < \infty$ *is the* **error event length** *and may tend to infinity with zero probability, but any interesting*[5] *non-zero probability error event always merges at some later finite time* ℓ_x*. The inputs correspondingly have* $\hat{\boldsymbol{x}}_m = \boldsymbol{x}_m \ \forall \ m < 0 \ , \ m \geq \ell_x$*.*

The **output error event sequence** *between the noiseless sequence,* $\tilde{\boldsymbol{y}}_k$ *and the decoder sequence,* $\hat{\boldsymbol{y}}_k$*, is*

$$\boldsymbol{\epsilon}_y(D) \stackrel{\Delta}{=} \tilde{\boldsymbol{Y}}(D) - \hat{\boldsymbol{Y}}(D) = H(D)\cdot \boldsymbol{X}(D) - H(D)\cdot \hat{\boldsymbol{X}}(D) \quad , \tag{7.15}$$

where the **input error event sequence** *is* $\boldsymbol{\epsilon}_x(D) \stackrel{\Delta}{=} \boldsymbol{X}(D) - \hat{\boldsymbol{X}}(D)$ *with length* ℓ_x*. Necessarily,* $\ell_x \leq \ell_y$ *for all error events. When* $H(D) = I$*, then* $\boldsymbol{\epsilon}_y(D) = \boldsymbol{\epsilon}_x(D)$ *and* $\ell_y = \ell_x$*. For partial response (strictly causal* $H(D) \neq I$ *and of degree* ν*)* $\ell_y = \ell_x + \nu$*. A corresponding* **encoder input or message error-event** *sequence is* $\epsilon_m(D) \stackrel{\Delta}{=} m(D) \ominus_M \hat{m}(D)$*, where the subtraction 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* $\boldsymbol{\epsilon}_v(D) = \boldsymbol{v}(D) \ominus \hat{\boldsymbol{v}}(D)$*, so vector modulo* $M = 2$ *in each element. For binary,* $\oplus$ *and* $\ominus$ *are the same operation. This text's analysis use lower-case epsilon notation ubiquitously for error events, whether in fields* $\mathbb{C}$ *or* $\mathbb{GF}$

The error event's application to partial-response channels, which like convolutional codes have a trellis and sequence descriptions, slightly complicates error-event definition above. This complication forces the distinction between channel-input and channel-output error events above and admits the possibility that $\ell_y > \ell_x$. For instance for the binary $1+D$ channel, an input error event of $\epsilon_x(D) = 2$ and $\ell_x = 1$ produces the channel output error event $\epsilon_y(D) = 2+2D$ with $\ell_y = 2$. These two descriptions, $\epsilon_x(D) = 2$ and $\epsilon_y(D) = 2+2D$, correspond to the same input error sequence $\hat{x}(D) = +1$ being the decoder output when the correct sequence was $x(D) = -1$.

7.2.1.1 Performance Analysis with Error Events

The sequence-error probability of sequence error is

$$P_e = Pr\{\hat{\boldsymbol{X}}(D) \neq \boldsymbol{X}(D)\} \quad . \tag{7.16}$$

MLSD minimizes P_e when all sequences are equally likely. P_e is the overall probability that any error event for a specific code can occur. Analysiis can upper bound P_e by enumerating all possible error events, then upper bounding the probability of each, and finally summing all. Different error events may have different probabilities. Some events have smaller $\|\boldsymbol{\epsilon}\|$ and thus occur with greater probability. Figure 7.10 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 7.10 are $\epsilon_x = 2$ and $\epsilon_x = 2-2D$. The first has probability of occuring $.5 \cdot Q(\sqrt{2}/\sigma)$ 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 that 2 is allowed is $1/2$ at any time k 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 Q-function multiplier that bounds error probability.

In general, Equation (7.11) bounds MLSD error probability on the AWGN channel. For the BSC, a

[5] 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.

similar expression from Chapter 1, Section 7, as well as Chapter 2, is

$$P_e \leq \sum_{i=0}^{\infty} N_{d_{free}+i} \cdot [4p(1-p)]^{(d_{free}+i)/2} = \sum_{d=d_{free}}^{\infty} N_d \cdot [4p(1-p)]^{d/2} \quad , \tag{7.17}$$

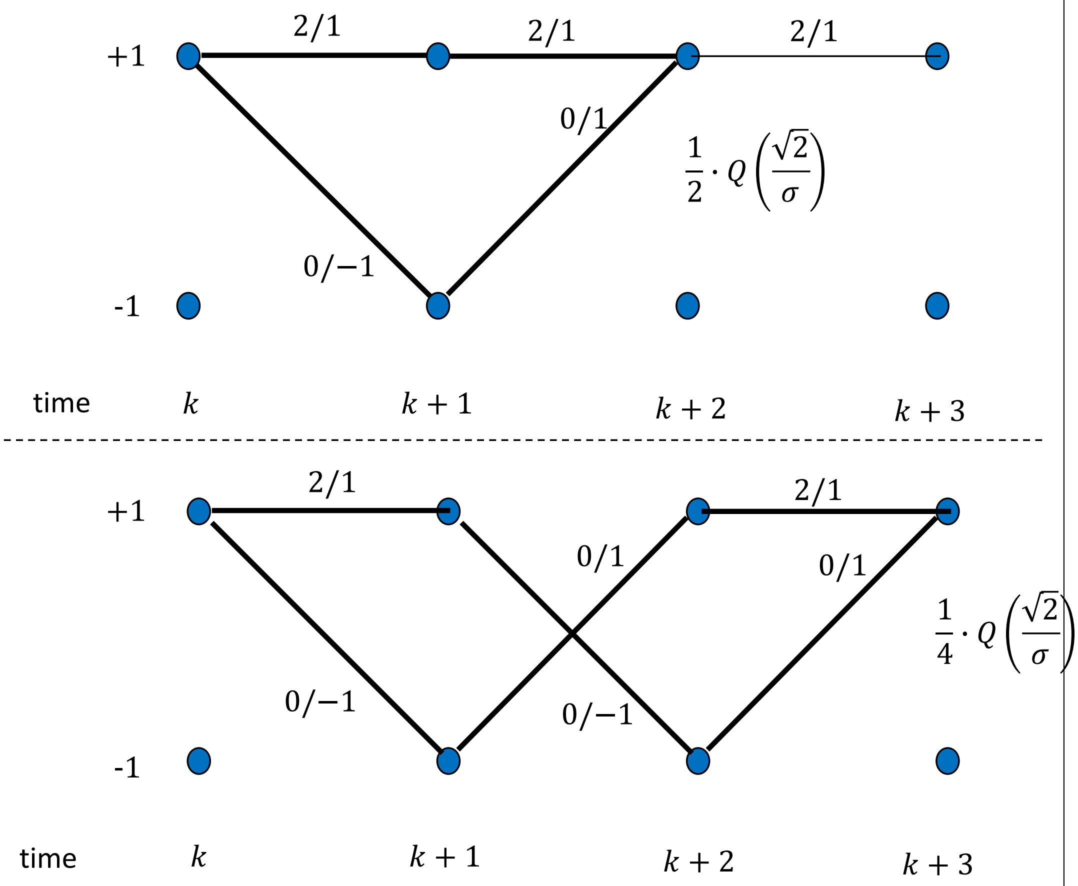

+1
-1
2/1
2/1
2/1
0/1
0/−1
$\frac{1}{2} \cdot Q\left(\frac{\sqrt{2}}{\sigma}\right)$
time
k
$k+1$
$k+2$
$k+3$
+1
-1
2/1
2/1
0/1
0/1
0/−1
0/−1
$\frac{1}{4} \cdot Q\left(\frac{\sqrt{2}}{\sigma}\right)$
time
k
$k+1$
$k+2$
$k+3$

Figure 7.10: Illustration of error events with same d_{min} and different probabilities.

The BSC's error probability then has bound:

$$P_e \leq \sum_{d=d_{free}}^{\infty} N_d \left[\sqrt{4p(1-p)}\right]^d \quad . \tag{7.18}$$

An aproximation to Equation (7.18) uses the first term

$$P_e \approx N_e \cdot \left(\begin{array}{c} d_{free} \\ \left\lceil \frac{d_{free}}{2} \right\rceil \end{array} \right) \cdot p^{\left\lceil \frac{d_{free}}{2} \right\rceil} \approx N_e \cdot [4p(1-p)]^{d_{free}/2} \quad , \tag{7.19}$$

effectively the BSC's nearest-neighbor union bound. This is the sequence-error probability. For convolutional codes, the bit-eerror probability $\bar{P}_b$ may be of more interest. $\bar{P}_b$ calculation for either the AWGN or the BSC requires the sequential encoder's input-message-bi- to-codeword mappings. Essentially, analysis requires the function

$$N(i,d) \stackrel{\Delta}{=} \text{number of error events of distance } d \text{ that correspond to } i \text{ input bit errors on average.} \tag{7.20}$$

The "on-average" part needs interpretation for partial-response channels. Otherwise $N(i,s)$ enumerates input-to-output mappings and associated input bit errors. $\bar{P}_b$ has then bound

$$\bar{P}_b < \frac{1}{b} \sum_{d=d_{free}}^{\infty} \sum_{i=1}^{\infty} i \cdot N(i,d) \cdot \left[\sqrt{4p(1-p)}\right]^d \quad . \tag{7.21}$$

The $1/b$ divider in front is the number of bits per symbol. Section 1.3.2.4's average total number of bit errors per (minimum-distance) error event (when all input messages are equally likely and independent from symbol to symbol) is

$$N_b = \sum_{i=1}^{\infty} i \cdot N(i, d_{free}) = \sum_{i=1}^{M-1} p_x(i) \cdot n_b(i) \quad . \tag{7.22}$$

For convolutional codes, the N_b follows a more code-directed expression that Chapter 1's simpler per-symbol definition intended for simpler symbols, and all same-length input sequences are assumed equally likely. Two other similar relationships are:

$$N_e = \sum_{i=1}^{\infty} N(i, d_{free}) \tag{7.23}$$

$$N_d = \sum_{i=1}^{\infty} N(i,d) \tag{7.24}$$

Further then the average bit-error rate has approximation

$$\bar{P}_b \approx \frac{N_b}{b} \cdot \binom{d_{free}}{\left\lceil \frac{d_{free}}{2} \right\rceil} \cdot p^{\left\lceil \frac{d_{free}}{2} \right\rceil} \approx \frac{N_b}{b} \left[4p(1-p)\right]^{d_{free}/2} \quad . \tag{7.25}$$

For the AWGN, again,

$$\bar{P}_b \approx \frac{N_b}{b} \cdot Q\left[\frac{d_{min}}{2\sigma}\right] \quad . \tag{7.26}$$

7.2.2 Example Analysis of the 4-state Convolutional code

Section 7.1's 4-state convolutional code example has the error events

$$\boldsymbol{\epsilon}_v(D) = \epsilon_u(D) \cdot G(D) \tag{7.27}$$

where all multiplication and addition is modulo-2 and $G(D) = [1 + D + D^2 \;\; 1 + D^2]$. For linear binary codes, the set of all error events is the same as the set of nonzero codewords. This greatly simplifies linear codes' analysis. The distance distribution $\mathcal{D}_{min}$ thus counts 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 $G(D)$, there is a second type of scalar "transfer function" that characterizes convolutional codes. 4-state convolutional code. Figure 7.11 redraws the underlying state-transition diagram 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 all-zeros codeword determines the other possible codewords, or equivalently the error events Figure 7.11 constructs this transfer function $T(W)$ for the

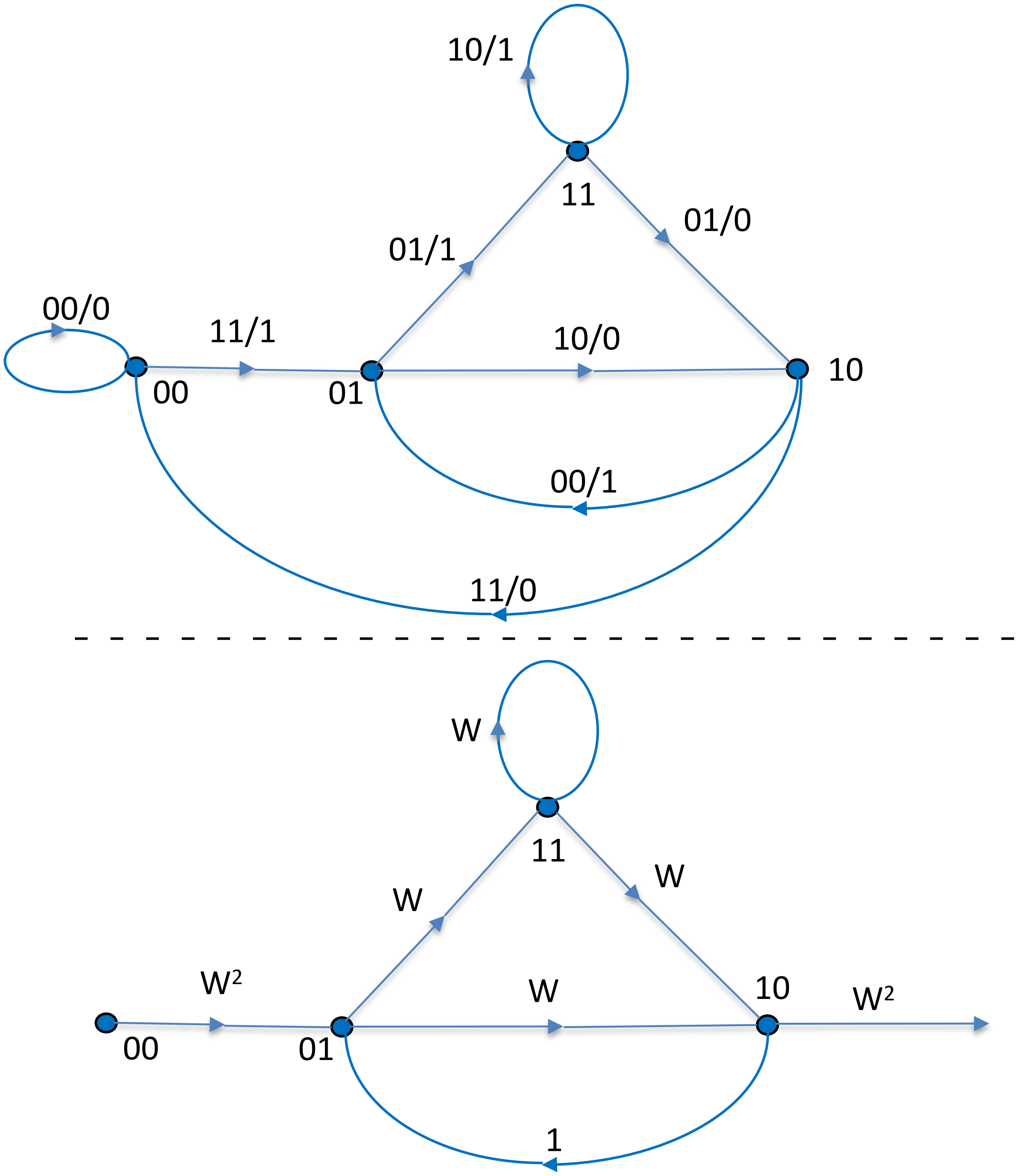


Figure 7.11: Transfer Function Construction for 4-State Example

Figure 7.11's lower half uses a place keeper for each branch/subsymbol's Hamming weight as exponent on the place-keeping variable W. The analysis construes this place keeping variable (with exponent) as $T(\mathrm{W})$. The Hamming weight is the exponent of W 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 s_{free}, comes from the W^5 path, so that $d_{free} = 5$.

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:

$$T(\mathrm{W}) = \frac{\sum T_k \cdot \Delta_k}{\Delta} \quad , \tag{7.28}$$

where

$$\Delta = 1 - (\text{sum loop gains}) + (\text{sum products of two non-touching loop gains}) - ... \quad , \tag{7.29}$$

$$\Delta_k = \Delta(\text{with the } k^{th} \text{ forward path removed}) \tag{7.30}$$

and

$$T_k = \text{gain of } k^{th} \text{ forward path} \quad . \tag{7.31}$$

For this example,

$$T(\mathrm{W}) = \frac{\mathrm{W}^5(1-\mathrm{W})+\mathrm{W}^6}{1-(\mathrm{W}+\mathrm{W}+\mathrm{W}^2)+\mathrm{W}\cdot\mathrm{W}} = \frac{\mathrm{W}^5}{1-2\mathrm{W}} = \mathrm{W}^5\sum_{k=0}^{\infty}(2\mathrm{W})^k \quad . \tag{7.32}$$

Thus,

$$T(\mathrm{W}) = \mathrm{W}^5\left[1+2\mathrm{W}+4\mathrm{W}^2+8\mathrm{W}^3+...\right] \quad . \tag{7.33}$$

There is one nearest neighbor at distance 5, 2 nearest neighbors at distance 6, 4 at distance 7, and so on. A similar approach applies to partial-response channels, with (squared) Euclidean distance replacing Hamming distance as the exponent of W. Any permutation of the order of the codeword's bit order will not affect $T(\mathrm{W})$. The nearest neighborl value $\overline{N}_e$ **for subsymbols** is the coefficient of $\mathrm{W}^{d_{min}}$ in $T(\mathrm{W})$ or

$$N_e^prime = \frac{1}{d_{free}!}\cdot\frac{\partial T(\mathrm{W})}{\partial \mathrm{W}^{d_{free}}}|_{\mathrm{W}=0} \quad . \tag{7.34}$$

The transfer function $T(W)$ can be more difficult to compute for codes with more states, perhaps more difficult than just searching the trellis. The next subsection produces the desired N_e-like coefficient for symbol-error probability expressions.

7.2.2.1 Expanded Use of Transfer Functions

Transfer-function's expanded analysis can use L as a placeholder variable for the output error-event length, I as a placeholder erred input bits, and J erred subsymbols. Each branch has gain multiplied by L, and by the power of I that corresponds to the number of input "1" bits, and by the power of J that corresponds to the number of symbol errors.

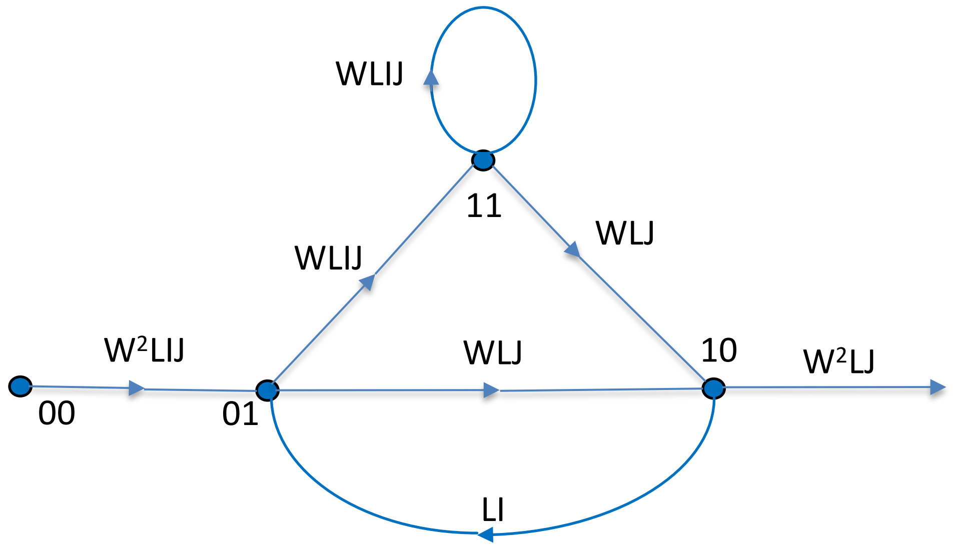


Figure 7.12: Expanded Transfer Function Computation

Figure 7.12 repeats Figure 7.11 with L, I, and J. Again, using Mason's Gain formula,

$$T(\mathrm{W}, L, I, J) = \frac{\mathrm{W}^5 L^3 I J^3 (1 - \mathrm{W}LIJ) + \mathrm{W}^6 L^4 I^2 J^4}{1 - (\mathrm{W}L^2 IJ + \mathrm{W}LIJ + \mathrm{W}^2 L^3 I^2 J^2) + \mathrm{W}^2 L^3 I^2 J^2} \tag{7.35}$$

$$= \frac{\mathrm{W}^5 L^3 I J^3}{1 - \mathrm{W}L^2 IJ - \mathrm{W}LIJ} = \frac{\mathrm{W}^5 L^3 I J^3}{1 - \mathrm{W}LIJ(1+L)} \tag{7.36}$$

$$= \mathrm{W}^5 L^3 I J^3 \left(1 + \mathrm{W}LIJ(1+L) + (\mathrm{W}LIJ)^2 (1+L)^2 + ...\right) \tag{7.37}$$

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

$$N_b = \frac{1}{d_{free}!} \cdot \frac{\partial T(\mathrm{W}, 1, I, 1)}{\partial \mathrm{W}^{d_{free}} \partial I} |_{\substack{\mathrm{W}=0 \\ I=1}} \tag{7.38}$$

and

$$\overline{N}_e \text{ symbol errors } = \frac{1}{d_{min}!} \frac{\partial T(\mathrm{W}, 1, 1, J)}{\partial \mathrm{W}^{d_{min}} \partial J} |_{\substack{\mathrm{W}=0 \\ J=1}} \tag{7.39}$$

This $\overline{N}_e$ is for subsymbols, not codewords.

7.2.3 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 $\left(\frac{d_{min}}{2\sigma_{pr}}\right)^2 = \left(\frac{\sqrt{2}d}{2\sigma_{pr}}\right)^2 = \mathrm{MFB}$. Thus MLSD attains the MFB, with finite real-time complexity and without equalization!. Contrast this detector with the ZFE, which suffers infinite noise enhancement on this channel, or the ZF-DFE which even with precoding remains 3 dB inferior to MLSD.[6]

7.2.3.1 Analysis by input error-event enumeration

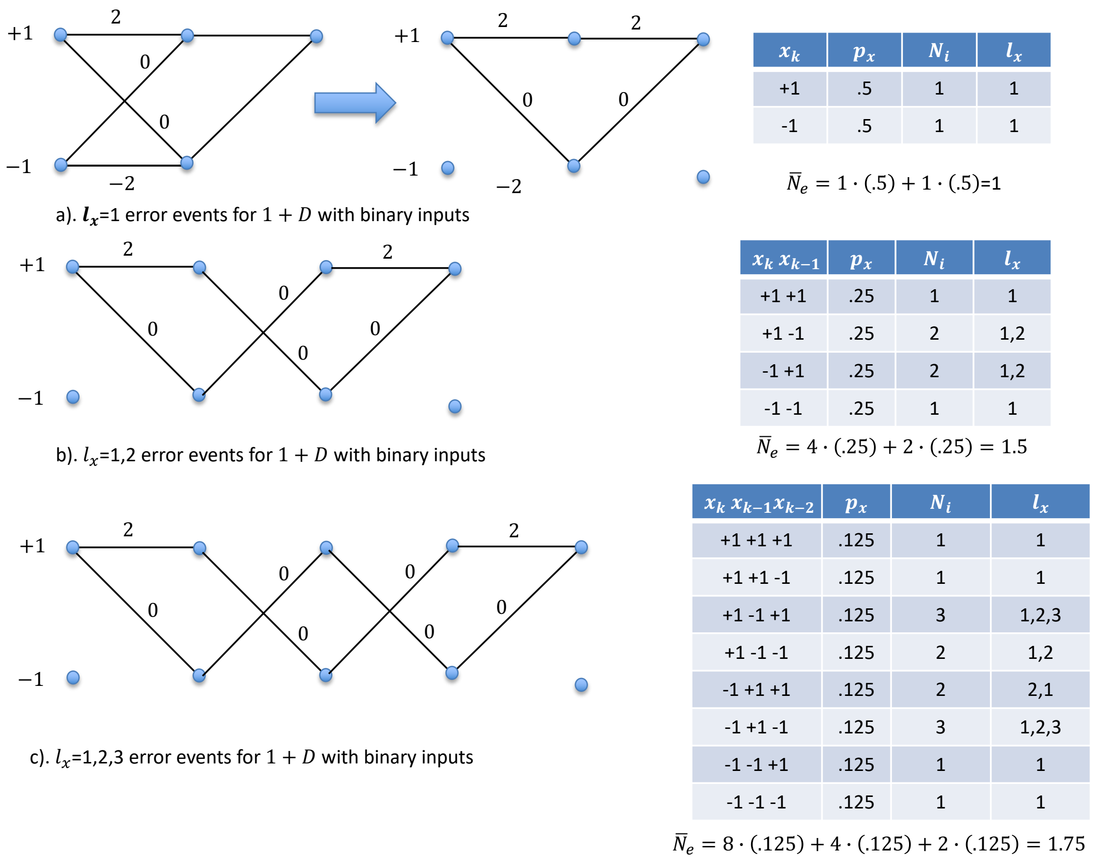
For the $1 + D$ channel with binary ($x_k = \pm 1$) inputs, the NNUB $\bar{P}_e$ expression is

$$\bar{P}_e \le \bar{N}_e \cdot Q\left[\frac{d_{min}}{2\sigma_{pr}}\right] = \bar{N}_e \cdot Q\left[\frac{\sqrt{2}}{\sigma_{pr}}\right] \quad , \tag{7.40}$$

since MLSD attains the matched-filter bound performance level on this channel. MLSD chooses from an infinite number of input sequences. Figure 7.13's trellises illustrate error-events of lengths $\ell_x = 1, 2, 3$. A length ℓ_x input error event must correspond to two paths that diverge (i.e., have a nonzero first entry) in the first trellis stage, and merge exactly $\ell_y = \ell_x + \nu = L + 1$ stages later. (If it merges sooner or diverges later, it is not of length ℓ_x.) [7] Symmetry permits consideration of only those error events beginning in one of the two states (+1 is used in Figure 7.13), because the other state's error events have identical length, number, and distribution. Also, the N_d are the same for either terminating state. This is because the input error-event sequence sample is 0 in the last ν stages (In Figure 7.13, $\nu = 1$).

[6] A higher error coefficient will occur for the Q-function with MLSD than in the strict MFB, which becomes evident later at higher values of M.

[7] 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.



a). l_x=1 error events for $1+D$ with binary inputs

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

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

b). l_x=1,2 error events for $1+D$ 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

$\bar{N}_e = 4 \cdot (.25) + 2 \cdot (.25) = 1.5$

c). l_x=1,2,3 error events for $1+D$ 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	2,1
-1 +1 -1	.125	3	1,2,3
-1 -1 +1	.125	1	1
-1 -1 -1	.125	1	1

$\bar{N}_e = 8 \cdot (.125) + 4 \cdot (.125) + 2 \cdot (.125) = 1.75$

Figure 7.13: Binary error events of length 1, 2, and 3 for the $1+D$ channel.

Analysis need include only merges into the top state (+1) in Figure 7.13: 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. Again, MLSD analysis sincludes only those error events corresponding to minimum distance in $\bar{N}_e$ because it is difficult to include those of greater distance (even if they have a common decision boundary).

For length $\ell_x = 1$, 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 $\ell_x = 1$, on the average. Thus, as $\bar{N}_e(1) = .5(1) + .5(1)$, where the argument of $\bar{N}_e(\ell_x)$ is the length of the input error event sequence. For lengths $\ell_x \leq 2$, 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 for which MLSD is in error.) From the table included in Figure 7.13(b), there are 2 error events of length 2 and 4. 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

$$\bar{N}_e(1,2) = \bar{N}_e(1) + \bar{N}_e(2) = 4(.25) + 2(.25) = 1.5 \quad . \tag{7.41}$$

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,

$$\bar{N}_e(1,2,3) = \bar{N}_e(1) + \bar{N}_e(2) + \bar{N}_e(3) = 8(.125) + 4(.125) + 2(.125) = 1.75 \quad . \tag{7.42}$$

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

$$\epsilon_x(D) = \begin{cases} \pm(2 - 2 \cdot D + 2 \cdot D^2 - .... + 2 \cdot D^{\ell_x - 1}) & \ell_x \text{ odd} \\ \pm(2 - 2 \cdot D + 2 \cdot D^2 - .... - 2 \cdot D^{\ell_x - 1}) & \ell_x \text{ even} \end{cases} \tag{7.43}$$

with the corresponding channel-output error events as

$$\epsilon_y(D) = (1 + D) \cdot \epsilon_x(D) = \begin{cases} \pm(2 + 2 \cdot D^{\ell_x}) & \ell_x \text{ odd} \\ \pm(2 - 2 \cdot D^{\ell_x}) & \ell_x \text{ even} \end{cases} \quad . \tag{7.44}$$

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

$$\bar{N}_e = 2 \sum_{\ell_x = 1}^{\infty} 2^{-\ell_x} = 2(\frac{1}{1 - .5} - 1) = 2 \quad , \tag{7.45}$$

so

$$\bar{P}_e \approx 2 \cdot Q\left(\frac{\sqrt{2}}{\sigma_{pr}}\right) \quad . \tag{7.46}$$

7.2.3.2 Analysis by error-event enumeration (d=2)

It can be tedious to enumerate all input sequences for a partial-response channel to compute $\bar{N}_e$. Instead, enumeration of the input error events, as in Equation (7.43)is simpler. 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

$$\bar{N}_e = 2 \cdot \sum_{\ell_x = 1}^{\infty} \left(\frac{M-1}{M}\right)^{\ell_x} = 2 \cdot (M - 1) \quad . \tag{7.47}$$

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 each error event's average number of bit errors occurring, let us first assume that MLSD uses no precoding, 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

$$N_b = \sum_b b \cdot N(b, d_{free}) \quad . \tag{7.48}$$

which is equivalent to the expression

$$N_b = \sum_i n_{b\epsilon}(i) \cdot p_\epsilon(i) \tag{7.49}$$

where $n_{b\epsilon}(i)$ is the number of bit errors corresponding to error event i and $p_\epsilon(i)$ 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

$$N_b = 2 \cdot \sum_{\ell_x = 1}^{\infty} (\ell_x) \left(\frac{M-1}{M}\right)^{\ell_x} = 2 \cdot M \cdot (M - 1) \quad , \tag{7.50}$$

Then, the bit error rate is accurately approximated by

$$\bar{P}_b(\text{no precode}) \approx \frac{N_b}{b} \cdot Q\left[\frac{\sqrt{2}}{\sigma_{pr}}\right] = \frac{2 \cdot M \cdot (M - 1)}{\log_2(M)} Q\left[\frac{\sqrt{2}}{\sigma_{pr}}\right] \quad . \tag{7.51}$$

The Q-function's coefficient in this expression is unacceptably high. However, precoding reduces it to a maximum of 2 input bit errors for any ℓ_x. Then $N_b = 2 \cdot \bar{N}_e$, and

$$\bar{P}_b(\text{precode}) \approx \frac{4 \cdot (M-1)}{\log_2(M)} \cdot Q\left[\frac{\sqrt{2}}{\sigma_{pr}}\right] \quad . \tag{7.52}$$

When $M = 2$, precoding does not reduce $\bar{P}_b$, but it does prevent a long string of bit errors from potentially occurring when a long error event occurs. Precoding is almost universally used with sequence detection on partial-response to avoid this "quasi-catastrophe." Also, as M increases above 2, precoding reduces $\bar{P}_b$ significantly. Definition 7.2.2 formalizes, along with Theorem 7.2.1:

Definition 7.2.2 *[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 the input encoding rule eliminates its possibility.

Theorem 7.2.1 *[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 Section 3.7's precoder), the controlled-ISI channel with MLSD cannot exhibit quasi-catastrophic error propagation.*

Proof: Since Section 3.7's partial-response precoder 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 error bursts.

7.2.4 Ginis' Code-Analysis Program

This section describes an algorithm that computes the d_{min}/d_{free} by search lists of a trellis' two diverging/merging paths (possible error events of interest). An early version of this algorithm was suggested in the dissertation of Dr. Sanjay Kasturia, a former Stanford Ph.D. student, but this particular matlab program with many enhancements is the product of former Stanford Ph.D. and EE379 student Dr. George Ginis.

7.2.4.1 Algorithm Description

The algorithm applies "Breadth-First-Search" to find merging paths. The search extends through the trellis one state at a time, until the minimum distance is found by separation of merged path pairs from candidate path pairs.

The algorithm maintains a state-pair list (so list of sets like $[s_1, s_2]$), each with a corresponding cost (distance between paths). This cost Δ is between two paths of equal length starting at some common state and ending at s_1 and s_2 (see Fig. 7.9). Equivalently, this list contains partially constructed (or completed) error events, for the common starting state. These two possible end states s_1 and s_2 may differ, but can also be the same state, in which latter case the list contains the corresponding error event. The algorithm consists of two stages: list initialization the and list update.

List initialization: Initially, the list (denoted as L) is the null set. For each trellis state, the list contains state-pairs that can be reached in one additional stage. A pair of states (s_1, s_2) is reachable from a state s_0 in one stage, 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 program's cost is "d," the Hamming distance for BSC or distance squared for AWGN, between the corresponding branches. The algorithm adds each and every state pair not already in L to 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 initialization's completion, L contains all state pairs that can be reached in trellis stage from any single state, together with their corresponding costs.

To complete list initialization, the algorithm sets an upper bound on minimum cost (e.g. some large value, which is certain to exceed the minimum distance). Each time an error event with lower cost is found (which initially is less than this upper bound), the new cost replaces upper-bound/lowest cost with this new smaller possible minimum distance. Also, a new list D is initialized to the null set. The list D will eventually contains all the searched state-pairs.

List update: The update step selects the state pair in L with the smallest associated cost, e.g. (s_1, s_2). If this cost is larger than the upper bound, the program exits and returns the upper bound. This is one of two possible algorithm exits. Otherwise, the algorithm extends the pair as follows: The set of all state pairs 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 an error event has been found. Up such finding, 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.

7.2.4.2 Using the MATLAB Program

Ginis' d_{min} program, dmin_main.m consists of 11 MATLAB m-files implementing the algorithm. The user only needs to be aware of few of these m-files, while the complete set of 11 listings appears in Appendix G. The program was initially written without inclusion/use of matlab's set commands. An enterprising student might try to update it with those commands, hopefully reducing run time for large trellises. 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 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 7.11. 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 8– see Problem 7.11.

The main program is really all the user needs to know.

```
>> help dmin_main
  [dmin,L] = dmin_main(nstates,b,nextstates,branchouts,E)

  Main program. Finds minimum distance in a general trellis.

  INPUTS
   nstates - the number of trellis states
   b - number of input bits
   nextstates -  (2^b x nstates) matrix
     each entry contains next state for current state(col) and input(row)
     example is =[1 2; 3 4; 1 2; 3 4] for 4-state conv code b=1, nstates=4
   branchouts - (2^b x nstates) matrix containing branch outputs
   E - E=1 for Euclidean sqared distance; otherwise 0 for Hamming

  OUTPUTS
   dmin - minimum distance
   The list L - columns are s1, s2, d , and list ID (0 or L, 1 for D)

  Calls to: dmin_init and dmin_iter

  George Ginis, April 2001 ; modified J. Cioffi 2023

  -----------------------------------------------

  >> help dmin_init
  L=dmin_init(nstates,b,nextstates,branchout, E)

  Subroutine:  Called by dmin_main
               calls:
                trellis_fn
                in_list
                extract_list
                add_list
                bdistance
                setdist_list
  Initializes list L.

  INPUTS
   nstates - the number of trellis states
   b - number of input bits
   nextstates -  (2^b x nstates) matrix
     each entry contains next state for current state(col) and input(row)
     example is =[1 2; 3 4; 1 2; 3 4] for 4-state conv code b=1, nstates=4
   branchouts - (2^b x nstates) matrix containing branch outputs
   E - E=1 for Euclidean sqared distance; otherwise 0 for Hamming

  OUTPUTS
   The list L - columns are s1, s2, d , and list ID (0 or L, 1 for D)
```

```
 George Ginis, April 2001 modified J. Cioffi 2023

 -----------------------------------------------------------------------------

>> help dmin_iter
 [dmin,L]= dmin_iter(L,nstates,b,nextstates,branchout,ub,E)

 Subroutine:  Called by dmin_main
              calls:
               trellis_fn
               in_list
               extract_list
               add_list
               bdistance
               setdist_list
               delete_list
               findmin_list
 updates list L, creates D

 INPUTS
  nstates - the number of trellis states
  b - number of input bits
  nextstates -  (2^b x nstates) matrix
    each entry contains next state for current state(col) and input(row)
    example is =[1 2; 3 4; 1 2; 3 4] for 4-state conv code b=1, nstates=4
  branchouts - (2^b x nstates) matrix containing branch outputs
  ub is an uppoer bound on minimum distance (chosen large)
  E - E=1 for Euclidean sqared distance; otherwise 0 for Hamming

 OUTPUTS
  dmin - minimum distance
  The list L - columns are s1, s2, d , and list ID (0 or L, 1 for D)

 George Ginis, April 2001 ; modified J.Cioffi 2023
```

7.2.5 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 sign$(h_0 = -$sign(h_2).

7.2.6 Decision Feedback Sequence Detection

In Decision Feedback Sequence Detection, the feedback section inputs are the trellis survivors. That is, the branch metric $\Delta_{i,j,k}$'s calculation 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 state.

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. This is intermediate to MLSD performance and a MMSE-DFE. It is sometimes also called **list decoding**.

7.3 MAP detection with the APP and SOVA Algorithms

As in Chapter 2, decoders need not initially make a hard decision about the transmitted symbol values. Instead, the decoder may measure input messages' relative likelihood (probability). This likelihood measure is "soft information." The decoder eventually makes a "hard" decision by selecting the input message value with the largest soft information. Such soft information may help decode other symbols, or more specifically other subsymbols or even bits. Section 7.5 examines successive soft-information exchange between decoders in"iterative decoding." This section describes the **à posteriori probability (APP) algorithm** that exactly computes the probabilities that a MAP detector uses with coded transmission. While simpler decoders might suffice, they may not consequently assist another decoder for the same information.

Subsection 7.3.1 investigates a recursive VA-like algorithm that maximizes individual bit/subsymbol-error's à posteriori probability (APP). This MAP detector name often applies to this individual probability while ML, or MLSD as per Sections 7.1 and 7.2's VA, applies to the maximum-likelihood sequence's probability. The MAP detector can be complex, and Subsection 7.2's soft-output VA (SOVA) largely obtains the same results by propagating log-likelihoods (LLs) or log-likehood ratios (LLRs) directly with some approximations that lead directly to a soft-output Viterbi Algorithm (SOVA). Both methods produce soft output information that can be useful to the detector itself as well as to other decoders as examples will illustrate.

7.3.1 MAP with the APP Algorithm

Maximum likelihood and MAP sequence detection respectively choose 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 correspond to the minimum error probability for each of its constituent symbols, nor does it correspondingly produce minimum probability of bit or message error for each individual bit/message. While Section 7.2 computed MLSD's stationary-channel (really sub-) symbol-error corresponding to minimum sequence-error probability, this subsymbol-error is not necessarily a minimum. A decoder could instead directly maximize $p_{m_k/\boldsymbol{y}(D)}$ for each sample time k. Such bit/sybsymbol decoding is more complex than MLSD for exact implementation. However, such maximization improves system performance if symbol error probability is a more important system measure than sequence-error probability.

The APP algorithm directly computes the AP probabilities when a code can be described by a trellis diagram over any finite block of K subsymbols. Each subsymbol/bit probability density has a maximum soft value to which a decoder can assign the hard MAP estimate of m_k. The APP algorithm also has the names "forward-backward" algorithm or the **Bahl-Cocke-Jelinek-Ravin (BCJR)**[4] algorithm.

The channel-output-subsymbol sample block is $\boldsymbol{Y}_{0:K-1}$. The APP decoder detects m_k , $k = 0, ..., K-1$ through the corresponding values of $p_{m_k/\boldsymbol{Y}_{0:K-1}}$ $k = 0, ..., K-1$. The latter are the APP's soft information.

A code's trellis description at any time $k \in [0 : K-1]$ 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 trellis branch between state $s_{k-1} = i$ and state $s_k = j$ has a conditional probability that is a function of the code:

$$p_k(i, j) = p(s_{k+1} = j/s_k = i) \quad . \tag{7.53}$$

Further an individual branch has a probability distribution for the input data message

$$q_k(i, j, m_k) = p(m_k/s_k = i, s_{k+1} = j) \quad , \tag{7.54}$$

which is usually 1 for the branch's assigned message and 0 for all other symbols. However, the branch probability could be $1/M'$ for each of M' equally likely parallel transitions, or generally some non-trivial parallel-transition distribution. A particularly useful APP measure 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 m_k value (that is, when there are no parallel transitions). The set of ordered pairs $B(m_k)$ contains all state pairs of beginning and ending trellis-branch state pairs (denoted by the branch endpoint states i and j in ordered pair (i, j)) on which

m_k can occur. The APP calculation (in terms of the beginning and ending states on which the specific input m_k occurs) is

$$p_k(m_k/\boldsymbol{Y}_{0:K-1}) = \sum_{(i,j)\in B(m_k)} p_k(s_k = i, s_{k+1} = j/\boldsymbol{Y}_{0:K-1}) \quad . \tag{7.55}$$

$p_k(s_k = i, s_{k+1} = j/\boldsymbol{Y}_{0:K-1})$ nominally divides $p_k(s_k = i, s_{k+1} = j,\ \boldsymbol{Y}_{0:K-1})$ by $p(\boldsymbol{Y}_{0:K-1})$. Since the values of $\boldsymbol{Y}_{0:K-1}$ are not a function of the *estimate* of $\boldsymbol{x}_k$, this division does not change the MAP estimate $\hat{m}_k$. Thus, the APP calculation ignores this normalization, The APP can compute the distribution $p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:K-1})$ and then proceed to the MAP estimate that maximizes (7.55). Individual decisions subsequently occur 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, (7.55) generalizes to

$$\begin{aligned}
p_k(m_k/\boldsymbol{Y}_{0:K-1}) &= \sum_{(i,j)\in B(m_k)} p_k(m_k/s_k = i, s_{k+1} = j, \boldsymbol{y}_k) \cdot p_k(s_k = i, s_{k+1} = j/\boldsymbol{Y}_{0:K-1}) && (7.56)\\
&= \sum_{(i,j)\in B(m_k)} \frac{p(\boldsymbol{y}_k, m_k/s_k = i, s_{k+1} = j) \cdot p(s_k = i, s_{k+1} = j)}{p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j) \cdot p(s_k = i, s_{k+1} = j)} \cdot p_k(s_{k-1} = i, s_k = j/\boldsymbol{Y}_{0:K-1}) \\
&= \sum_{(i,j)\in B(m_k)} \frac{p(\boldsymbol{y}_k, m_k/s_k = i, s_{k+1} = j)}{p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j)} \cdot p_k(s_{k-1} = i, s_k = j/\boldsymbol{Y}_{0:K-1}) \\
&= \sum_{(i,j)\in B(m_k)} \frac{p(\boldsymbol{y}_k/m_k, s_k = i, s_{k+1} = j) \cdot p(m_k/s_k = i, s_{k+1} = j)}{p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j)} \cdot p_k(s_{k-1} = i, s_k = j/\boldsymbol{Y}_{0:K-1}) \\
&= \sum_{(i,j)\in B(m_k)} \frac{p(\boldsymbol{y}_k/m_k, s_k = i, s_{k+1} = j) \cdot q_k(i,j,m_k)}{p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j)} \cdot p_k(s_{k-1} = i, s_k = j/\boldsymbol{Y}_{0:K-1}) \quad . && (7.57)
\end{aligned}$$

where $q_k(i,j,m_k) = p(m_k/s_k = i\ ,\ s_{k+1} = j)$ and

$$\begin{aligned}
p_k(\boldsymbol{y}_k/s_k = i, s_{k+1} = j) &= \sum_{m_k'} p(\boldsymbol{y}_k/m_k', s_k = i, s_{k+1} = j) \cdot p(m_k'/s_k = i, s_{k+1} = j) && (7.58)\\
&= \sum_{\boldsymbol{x}_k'} p(\boldsymbol{y}_k/\boldsymbol{x}_k', s_k = i, s_{k+1} = j) \cdot q_k(i,j,\boldsymbol{x}_k') \quad . && (7.59)
\end{aligned}$$

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 (7.59) simplifies on the right to $p(\boldsymbol{y}_k/m_k)$. Then this term is common to numerator and denominator in (7.57), and thus cancels, leaving (7.55).

7.3.1.1 3 APP Quantities

The APP algorithm computes $p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:K-1})$ for either (7.55) or (7.57) as a function of 2 recursively updated state-dependent quantities generated by progressing forward and backward through the code's trellis, and of a third non-recursive branch-dependent quantity that is a function that corresponds only to the time index k of interest.

Forward Trellis Quantity α: The first forward-recursively-computed quantity $\alpha_k(j)$ is the joint state and past output probability (tacitly a function of $\boldsymbol{Y}_{0:k}$)

$$\alpha_k(j) \stackrel{\Delta}{=} p(s_{k+1} = j, \boldsymbol{Y}_{0:k}) \quad j = 0, ..., |S_{k+1}| - 1 \quad . \tag{7.60}$$

Backward Trellis Quantity β: The second-backward recursively-computed quantity $\beta_k(j)$ is the state-conditional future output distribution (tacitly a function of $\boldsymbol{Y}_{k+1:K-1}$

$$\beta_k(j) \stackrel{\Delta}{=} p(\boldsymbol{Y}_{k:K-1}/s_{k+1} = j) \quad j = 0, ..., |S_{k+1}| - 1 \quad . \tag{7.61}$$

Branch Quantity γ: The third current-branch probability quantity $\gamma_k(i,j)$ is (tacitly a function of $\boldsymbol{y}_k$)

$$\gamma_k(i,j) = p(s_{k+1} = j, \boldsymbol{y}_k / s_k = i) \quad , i = 0, ..., |S_k| - 1 \ , \ j = 0, ..., |S_{k+1}| - 1 \quad . \tag{7.62}$$

Branch/Input Probability Calculation: Thus the quantity in (7.55) and (7.57) 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 $\gamma_k(i,j)$ for all trellis branches, then executes an α recursion and a β recursion. If the detector has these 3 quantities available at any time k, then

$$\begin{aligned} p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}) &= p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}, \boldsymbol{y}_k, \boldsymbol{Y}_{k+1:K-1}) & (7.63) \\ &= p_k(\boldsymbol{Y}_{k+1:K-1} / s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}, \boldsymbol{y}_k) \cdot p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}, \boldsymbol{y}_k) \\ &\quad \text{The state } k+1 \text{ captures all post, making } s_k \text{ and } \boldsymbol{Y}_{k+1:K} \text{ independent} \\ &= p_k(\boldsymbol{Y}_{k+1:K-1} / s_{k+1} = j) \cdot p_k(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}, \boldsymbol{y}_k) \\ &= \beta_k(j) \cdot p_k(s_{k+1} = j, \boldsymbol{y}_k / s_k = i, \boldsymbol{Y}_{0:k-1}) \cdot p_k(s_k = i, \boldsymbol{Y}_{0:k-1}) \\ &= \beta_k(j) \cdot \gamma_k(i,j) \cdot \alpha_{k-1}(i) \end{aligned}$$

Definition 7.3.1 *[APP Foundational Equation:] The important APP foundational equation depends on the 3-term branch product*

$$\beta_k(j) \cdot \gamma_k(i,j) \cdot \alpha_{k-1}(i) \tag{7.64}$$

and on the labeling $\mathcal{S}_{k,}$, which is the set of all allowed branch transitions from state any state s_k to any other state s_{k+1} for the given trellis description. The foundational equal is for caculation of the APP

$$Pr\{\boldsymbol{x}_k / \boldsymbol{Y}_{0:K-1}\} = \sum_{(i,j)|\boldsymbol{x}_k \in \mathcal{S}_k} \beta_k(j) \cdot \gamma_k(i,j) \cdot \alpha_{k-1}(i) \tag{7.65}$$

The MAP detector then selects the $\boldsymbol{x}_k$ subsymbol value at each time k that maximizes (7.65).

The foundational equation requires calculation of the 3 terms α, β, and γ throughout the trellis. The sum in (7.65) equivalently could be for each input corresponding to $\boldsymbol{x}_k$ to then produce $Pr\{u_{k,i} / \boldsymbol{Y}_{0:K-1}\}$, $i = 1, ...,$ and the sum modifies accordingly to be over those branches for which u_k, i. (Note the time index used was k so the equation avoided saying the maximum value of i is k, thus confusing an otherwise simple issue.) The calculation can also be over output bits $Pr\{v_{k,i} / \boldsymbol{Y}_{0:K-1}\}$, $i = 1, ..., n$. While the output bits APP may not seem of interest, such soft information may be of interest in an improved form of BICM, Section 7.5's BICM-ID (where ID is iterative decoding) where the Constellation's Gray-code "demapping" may be considered a code that exchanges soft information with a binary code.

γ computation: The APP computes $\gamma_k(i,j)$ first for all branches according to

$$\begin{aligned} \gamma_k(i,j) &= p(s_{k+1} = j, \boldsymbol{y}_k / s_k = i) & (7.66) \\ &= p(s_{k+1} = j / s_k = i) \cdot p(\boldsymbol{y}_k / s_k = i, s_{k+1} = j) & (7.67) \\ &\quad \text{See Equation (7.53)} \\ &= p_k(i,j) \cdot \sum_{m'_k} p(\boldsymbol{y}_k / m'_k, s_k = i, s_{k+1} = j) \cdot q_k(i, j, m'_k) & (7.68) \end{aligned}$$

which for AWGN channel is also

$$\gamma_k(i,j) = p_k(i,j) \cdot \sum_{m'_k} p_{\boldsymbol{n}_k}(\boldsymbol{y}_k - \boldsymbol{x}'_k(m'_k)) \cdot q_k(i, j, m'_k) \quad . \tag{7.69}$$

Equation (7.68) simplifies in the case of no parallel transitions to

$$\gamma_k(i,j) = p_k(i,j) \cdot p_{\boldsymbol{y}_k/m_k} \quad . \tag{7.70}$$

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

$$\begin{aligned}
\alpha_k(j) &= \sum_{i \in S_k} p(s_k = i, s_{k+1} = j, \boldsymbol{Y}_{0:k-1}, \boldsymbol{y}_k) && (7.71)\\
&= \sum_{i \in S_k} p(s_{k+1} = j, \boldsymbol{y}_k / s_k = i, \boldsymbol{Y}_{0:k-1}) \cdot p(s_k = i, \boldsymbol{Y}_{0:k-1}) && (7.72)\\
&= \sum_{i \in S_k} p(s_{k+1} = j, \boldsymbol{y}_k / s_k = i) \cdot \alpha_{k-1}(i) && (7.73)\\
&= \sum_{i \in S_k} \gamma_k(i,j) \cdot \alpha_{k-1}(i) \quad . && (7.74)
\end{aligned}$$

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 (7.74) 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). The APP replaces Viterbi's add-compare-select by a sum-of-products operation.

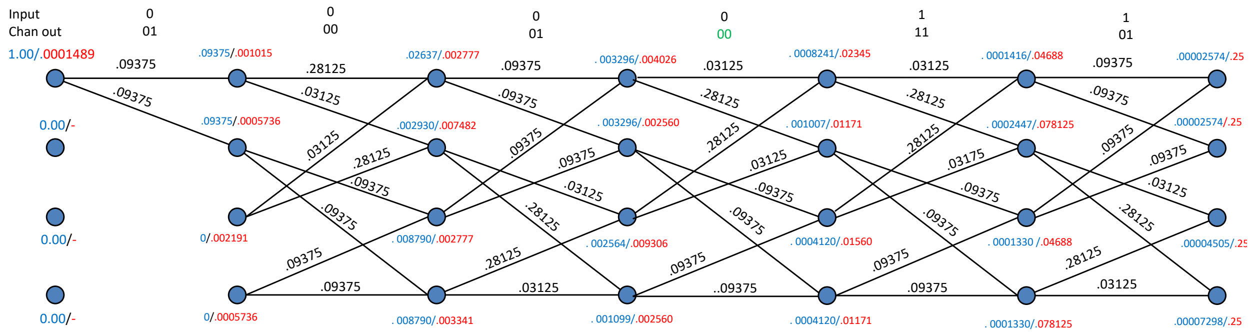
β computation: A backward recursion for $\beta_k(i)$ is

$$\begin{aligned}
\beta_k(i) &= \sum_{j \in S_{k+2}} p(s_{k+2} = j, \boldsymbol{Y}_{k+1:K-1} / s_{k+1} = i) && (7.75)\\
&= \sum_{j \in S_{k+2}} p(s_{k+2} = j, \boldsymbol{y}_{k+1}, \boldsymbol{Y}_{k+2:K-1} / s_{k+1} = i) && (7.76)\\
&= \sum_{j \in S_{k+2}} p(\boldsymbol{y}_{k+1} / s_{k+1} = i, s_{k+2} = j, \boldsymbol{Y}_{k+2:K-1}) \cdot p(\boldsymbol{Y}_{k+2:K-1}, s_{k+2} = j / s_{k+1} = i) && (7.77)\\
&= \sum_{j \in S_{k+2}} \frac{p(s_{k+2} = j, \boldsymbol{y}_{k+1} / s_{k+1} = i, \boldsymbol{Y}_{k+2:K-1})}{p(s_{k+2} = j / s_{k+1} = i, \boldsymbol{Y}_{k+2:K-1})} \cdot p(\boldsymbol{Y}_{k+2:K-1} / s_{k+1} = i, s_{k+2} = j) \cdot p(s_{k+2} = j / s_{k+1} = i) &&\\
&= \sum_{j \in S_{k+2}} p(s_{k+2} = j, \boldsymbol{y}_{k+1} / s_{k+1} = i) \cdot \beta_{k+1}(j) && (7.78)\\
&= \sum_{j \in S_{k+2}} \gamma_{k+1}(i,j) \cdot \beta_{k+1}(j) \quad . && (7.79)
\end{aligned}$$

The boundary ("final") condition for β calculation follows from Equations (7.63) to (7.64) 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 Section 7.4's backward Viterbi, again with sum-of-products operations replacing a add-compare-select operations.

EXAMPLE 7.3.1 *[Wu's MAP decoder example]* This example initially came from former Ph.D. student Zining "Nick" Wu[8], with several updates since. Figure 7.14 illustrates the APP for the 4-state rate-1/2 convolutional code for a BSC with $p = .25$. The inputs and outputs are the same as Subsection 7.1.1's examples.

[8] Dr. Zining Wu, a Chinese American electrical engineer; after graduation rose to Chief Technical Officer Marvell Semiconductor before present position of CEO, Innogrit.



$Pr\{u=0\}$	**.5843**	**.7048**	**.7470**	**.6747**	.2771	.3735
$Pr\{u=1\}$	.4157	.2952	.2530	.3253	**.7229**	**.6265**
Bit decision	0	0	0	0	**1**	**1**

Figure 7.14: Example of MAP detector for rate 1/2 4-state Convolutional Code with BSC and $p = .25$

The APP's trellis tracing 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. The forward quantities α_k appear in blue at each corresponding state. The backward quantities β_k also appear at each state. The γ_k quantities appear in black along each trellis 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

$$\gamma_0 = .0938 = \frac{1}{2}(.25)(.75) \quad , \tag{7.80}$$

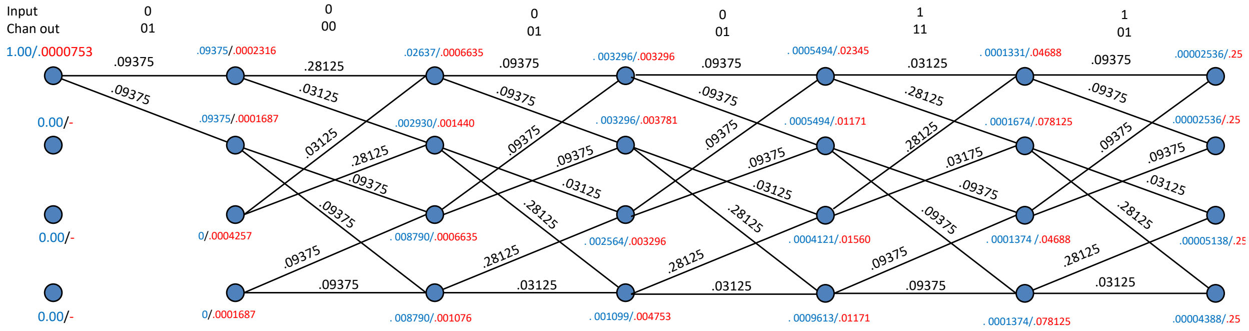
where the .25 corresponds to the first code-output bit not matching the channel-output 0, while the .75 corresponds to second code-output bit matching the second channel-output bit of 1, and the factor of 1/2 represents p_{m_k} in (7.70). Figure 7.14's table (below the trellis) provides the sum of that trellis stages probability distribution for the input bit given the entire channel output observation (over 6 subsymbol periods). The largest then determines the decision, which appears in green. This code has d_{free} of 5 and thus MLSD correctly finds the input sequence. Additionally, the APP in minimizing individual bit-error probability also obtains the correct inputs. In general, the detector does not know the correct sequence, but both criteria ML and APP both find the same set of input bits in this example. This need not always be true.

Indeed, by re-introducing the 3rd error that MLSD could only detect but not correct, Figure 7.24 illustrates the corresponding APP result:

Observations:

1. Each trellis stage should have the γ_k sum over all branches equal to one. The first stage does not exhibit this. The decoder designer could indeed change that stage to have $.09375 \rightarrow 0.5$, but this would only sale all future other α_k and β_k quantities by the same amount in each stage that includes calculations based on stage 0. This constant scaling will not change any MAP decision in maximizes (7.65).
2. The α and β quantities are indeed distributions, but are only computed for specific values in those distributions. Thus, they largely decrease in value as they progressing corresponding to the larger number of possible sequences growing exponentially with time (or backwards in time with the β quantities).
3. There is an initialization assumption on β_k that is somewhat arbitrary. This example chooses all final states as having equal $\beta = 1/n$. This is again constant scaling that does not affect decisions based on (7.65). The matlab program bcjr_conv.m at the Matlab open exchange has errors, for which this text provides the BCJR_AWGN and

BCJR_BSC that correct those known errors. If the final state is unknown, as in this example, it is perhaps best to assume all n-dimensional $\boldsymbol{v}_K$ values are equally likely. The programs bcjr_AWGN.m and bcjr_BSC.m that arrive later in this section assume the equally likely outputs. (The author welcomes critique from anyone who sees something I've missed.) These programs helped place the values on this example's trellis, and of course do the full decoding as well.



$Pr\{u = 0\}$	**.5870**	**.6304**	**.5217**	**.5217**	.3478	.4783
$Pr\{u = 1\}$	.4130	.3696	.4783	.4783	**.6522**	**.5217**
Bit decision	0	0	0	0	**1**	**1**

Figure 7.15: Example of MAP detector for rate 1/2 4-state Convolutional Code with BSC and $p = .25$, with additional erred bit.

The APP, equivalently per-bit MAP detection, also corrects the extra channel bit error at $k = 3$. In this case, this is the known correct input sequence of 000011, as in Section 7.1.1. In general because the detector does not know the correct sequence and/or bits, only estimates it/them, these two different detection results reflect the optimum decision for the two different criteria: VA/MLSD finding the best sequence and APP/MAP finding best individual bit decisions. The APP's success, when MLSD could not, simply follows from the MAP for each bit necessarily would include the MLSD as one option over its optimization. Viterbi (on the BSC) by itself cannot correctly find the sequence with 3 output-bit errors. However, if VA/MLSD additionally had such information, it can improve and identify the correct sequence, as is evident later in this section. Soft information can help correct the extra channel bit error. That soft information can also be helpful to other codes applied to the same input bits. The tables in Figures 7.14 and 7.24 also provide the bit-error probability, which is equivalent to the soft information. This is the minimum bit-error probability independently for each bit.

The APP recursions can apply for each bit of a message with $M > 2$ through Chapter 2's BICM, which Chapter 8 further studies.

7.3.1.2 LOGMAP

The **LOGMAP** algorithm approximates the APP by propagating log likelihoods and consequently replacing multiplication with logarithmic addition. The LOGMAP algorithm generally takes a log-likelihood (log of a probability distribution α) $\lambda = \ln(\alpha)$. The probability distribution itself may form as the sum of products as for instance in the APP's forward and backward recursions:

$$\alpha = \sum_i \alpha_i \cdot \gamma_i \quad . \tag{7.81}$$

As a log likelihood, then 7.81)'s individual products' logarithms similarly have two equivalent forms:

$$\begin{aligned} \lambda_i &\stackrel{\Delta}{=} \ln(\alpha_i) + \ln(\gamma_i) \quad (7.82) \\ e^{\lambda_i} &= \alpha_i \cdot \gamma_i \quad (7.83) \end{aligned}$$

Then essentially, LOGMAP computes

$$\lambda = \ln\left(\sum_i e^{\lambda_i}\right) \quad . \tag{7.84}$$

If there are only two terms, then

$$\lambda = \lambda_1 + \ln\left(1 + e^{\lambda_2 - \lambda_1}\right) = \lambda_1 + f(\lambda_2 - \lambda_1) \quad , \tag{7.85}$$

with tabular-implemented function $f(x) = \ln(1 + e^x)$. This requires no multiplication. A recursion applies - for instance with 3 terms and $\lambda_{12} \stackrel{\Delta}{=} \lambda_1 + f(\lambda_2 - \lambda_1)$, then $\lambda_3 = \lambda_1 + \lambda_{12} + f(\lambda_3 - \lambda_{12})$, with the extension requiring only more adds and f-table look ups.

Caution on log map sums: The log map sum does not apply directly for propagation of $\ln(\frac{1}{x} + \frac{1}{y})$, so designers should be careful in simply negating log-likelihoods for the AWGN, which leads to simple squared-distance metrics. These are usually positive and correspond to the negative of the log likelihood, correspondingly the reciprocal of the likelihood. Either keep the negative signs or revise (7.85) accordingly.

BCJR or APP Software: The author has revised the matlab file exchanges BCJR_conv.m program, which is nicely written but unfortunately has some errors as well as deficiencies. First, the programs are updated to allow codes with rates $r = k/n$ where $k \leq 4$ and n is arbitrary. The original matlab programs only work for $r = 1/n$ codes. Second, a version for the BSC channel is provided also; the original code was only for the AWGN. Further, there are some incorrect normalizations within the programs that this author has commented out. The programs seem to work more accurately without these normalizations. Finally, the original matlab program's recycling of forward end results to initial backward pass results has been removed, favoring a simple backward initialization to $1/2^n$ for each states initial β_K value.

```
>> help BCJR_AWGN
  >> help BCJR_BSC
  function BCJR_conv(y,trellis,p)
              BCJR_conv Decoder - HAMMING DISTANCE BSC
     This program derives from a nice matlab-file-xchange listing by K. Elhalil,
     of SUP'COM Tunisia.  It was modified by me (J. Cioffi) in 2023 to allow
     convolutional codes with k>1,r=k/n.

     It implements the Bahl, Cocke, Jelinek and Raviv (BCJR) APP algorithm.
     This function accepts the BSC output y, the trellis (from
     poly2trellis.  It uses a priori prob that is set to 1/2^k. Motivated
     users may want to add the ability to input a set of a priori inputs or
     extrinsic information.  It returns the APP LLR for each data bit input.
     The program replaces an alpha->beta turnaround at  last stage with just
     equal output probability 1/2^n for each initial  beta value.  I believe
     that avoids bias and is more accurate.  N=length(y) and N/n must be
     integers.  Also, I commented out the original matlab program's normalization
     line for alpha and beta that I believe incorrect.

    INPUTS:
       y - these are integers 1's or 0's in 1xn vector
```

```
      trellis - this is matlabs usual trellis description (see my text or
        class notes to avoid excessive computation for feedback systematic.
      p - this is 1-dimensional BSC error-probability for uncoded use.
   OUTPUTS:
      the decoded input bits' LLRs

 ******************************************************************************
```

To reproduce Figures 7.14 and 7.24, the following commands use BCJR_BSC to decode:

```
>> t=poly2trellis(3, [7 5]);
>> out=convenc([0 0 0 0 1 1],t)
>> out  =     0     0     0     0     0     0     0     0     1     1     0     1
>> BCJR_BSC(out,t,.25)  %=
          3.5981    3.1193     2.6526     2.2290    -1.9712   -1.4020   LLRs

         0                 0              0                 0                1               1
%checks.
outBSC2=[0 1 0 0 0 1 0 0 1 1 0 1];
>> BCJR_BSC(outBSC2,t,.25) =
    0.3406    0.8704    1.0826    0.7295   -0.9589   -0.5173  % less soft info/confidence
>> outBSC3=[ 0 1 0 0 0 1 0 1 1 1 0 1];
>> BCJR_BSC(outBSC3,t,.25) =
0.3514    0.5341    0.0870    0.0870   -0.6286   -0.0870 % yet soft info, all bits but first
>> BCJR_BSC(outBSC2,t,.49) =
    0.0008    0.0392    0.0016    0.0016   -0.0008   -0.0000 % same decisions, but
Less confident because p is large
>> BCJR_BSC(outBSC3,t,.49)= 0.0008    0.0392    0.0000    0.0000   -0.0008   -0.0000
```

The use shows the decrease in soft-information confidence as the channel worsens, and also as the number of output bit errors increase.

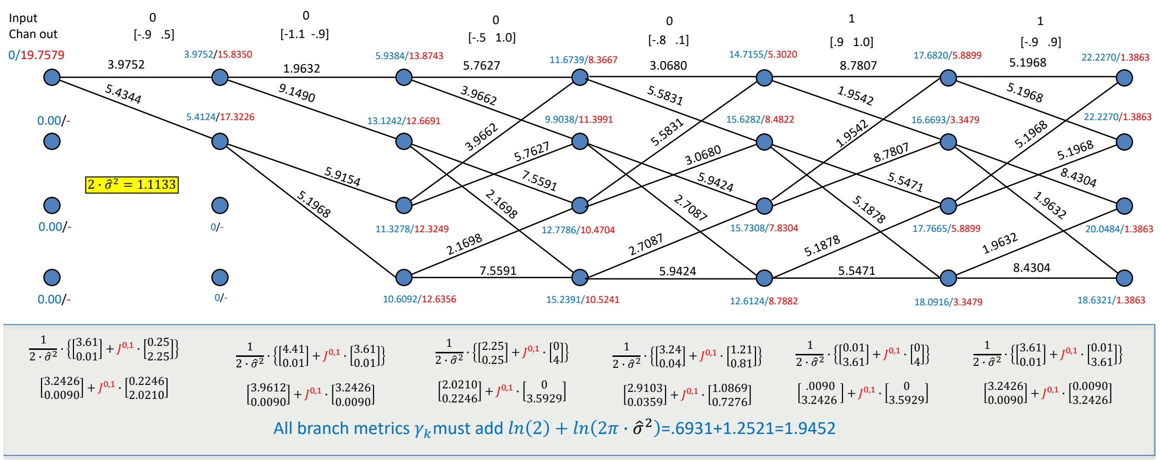
For the AWGN

```
>> help BCJR_AWGN
  function BCJR_AWGN(y,trellis,sigma)
                              BCJR_conv Decoder
     This program derives from a nice matlab-file-exchange listing by K. Elhalil,
     of SUP'COM Tunisia.  It was modified by me (J. Cioffi) in 2023 to allow
     convolutional codes with k>1,r=k/n.

     It implements the Bahl, Cocke, Jelinek and Raviv (BCJR) APP algorithm.
     This function accepts the channel output y, the trellis (from
     poly2trellis.  It uses a priori prob that is set to 1/2^k instead of
     the original matalb .  Motivated users may want to add the ability to
     input a set of a priori inputs (presumably extrinsic information from
     another code's use on same bits).   It returns the APP LLR for each
     data bit input.  The program replaces an alpha->beta turnaround at
     last stage with just equal output probability 1/2^n for each initial
     beta value.  I believe that avoids bias and is more accurate.
     N=length(y) and N/n must be integer.  Also, I commented out a
     normalization line for alpha and beta that I believe incorrect.

    INPUTS:
       y - these are real-valued N from some (AWGN likely) channel output
         multiply this by -1 to get the EE379 Class convention on 0->-1
       trellis - this is matlabs usual trellis description (see my text or
         379A class notes to avoid excessive computation for feedback
         systematic)
       sigma - this is 1-dimensional AWGN standard deviation
    OUTPUTS:
       the decoded input bits' LLRs
```

EXAMPLE 7.3.2 *[LOGMAP for the 4-state convolutional code]* Figure 7.16 illustrates the BCJR for the same 4-state convolutional code and AWGN output sequence. The BCJR_AWGN.m program (for which this text provides source code) has the values for the various states and branches within, so with one-time modification to print those quantities, Figure 7.16 avoids much complex hand calculation. Figure 7.16 illustrates a decoder's detailed calculation elements below each trellis stage in a way to illustrate the squared differences between branch values and received values. The $J^{0,1}\cdot$ simply means reorder the following vector or keep it's order for the various branch metric possibilities. Again blue quantities are the forward α_k recursion while red quantities are the backward β_k recursions. Chapter 4's multichannel normalizer addresses mechanisms that scale channel outputs adaptive when codes apply over frequency-indexed tones and the Gaussian variance varies, and in this case the noise-weighting becomes variable (not shown in this example).



Input
Chan out
0/19.7579
0.00/-
0.00/-
0.00/-
0
[-.9 .5]
0
[-1.1 -.9]
0
[-.5 1.0]
0
[-.8 .1]
1
[.9 1.0]
1
[-.9 .9]
$2\cdot\hat{\sigma}^2 = 1.1133$
3.9752/15.8350
5.4124/17.3226
0/-
0/-
5.9384/13.8743
13.1242/12.6691
11.3278/12.3249
10.6092/12.6356
11.6739/8.3667
9.9038/11.3991
12.7786/10.4704
15.2391/10.5241
14.7155/5.3020
15.6282/8.4822
15.7308/7.8304
12.6124/8.7882
17.6820/5.8899
16.6693/3.3479
17.7665/5.8899
18.0916/3.3479
22.2270/1.3863
22.2270/1.3863
20.0484/1.3863
18.6321/1.3863
3.9752
5.4344
1.9632
9.1490
5.9154
5.1968
5.7627
3.9662
3.9662
5.7627
7.5591
2.1698
2.1698
7.5591
3.0680
5.5831
5.5831
3.0680
5.9424
2.7087
2.7087
5.9424
8.7807
1.9542
1.9542
8.7807
5.5471
5.1878
5.1878
5.5471
5.1968
5.1968
5.1968
8.4304
1.9632
1.9632
8.4304
$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}3.61\\0.01\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0.25\\2.25\end{bmatrix}\right\}$
$\begin{bmatrix}3.2426\\0.0090\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0.2246\\2.0210\end{bmatrix}$
$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}4.41\\0.01\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}3.61\\0.01\end{bmatrix}\right\}$
$\begin{bmatrix}3.9612\\0.0090\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}3.2426\\0.0090\end{bmatrix}$
$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}2.25\\0.25\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0\\4\end{bmatrix}\right\}$
$\begin{bmatrix}2.0210\\0.2246\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0\\3.5929\end{bmatrix}$
$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}3.24\\0.04\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}1.21\\0.81\end{bmatrix}\right\}$
$\begin{bmatrix}2.9103\\0.0359\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}1.0869\\0.7276\end{bmatrix}$
$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}0.01\\3.61\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0\\4\end{bmatrix}\right\}$
$\begin{bmatrix}.0090\\3.2426\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0\\3.5929\end{bmatrix}$
$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}3.61\\0.01\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0.01\\3.61\end{bmatrix}\right\}$
$\begin{bmatrix}3.2426\\0.0090\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0.0090\\3.2426\end{bmatrix}$
All branch metrics γ_k must add $ln(2)+ln(2\pi\cdot\hat{\sigma}^2)$=.6931+1.2521=1.9452

$LL\{u=0\}$	19.8102	19.7914	20.0009	19.9890	22.9202	21.3274
$LL\{u=1\}$	22.7349	23.1900	21.2915	21.3361	19.8011	19.9913
Bit decision	0	0	0	0	1	1

Figure 7.16: Example of rate 1/2 convolutional code with 3 output errors and AWGN APP decoding.

The AWGN's LOGMAP APP also requires knowledge of the distribution parameter, specifically σ^2, as did the BSC LOGMAP require p. Previously, the VA did not require such knowledge. The LOGMAP implementation often estimates the noise variance. For this example, since the correct sequence is available but not σ^2 – a situation similar to when a training sequence might be known in advance and thus permit σ^2's estimation. The example estimates the noise variance from the 12 real-dimension noises according to

$$\hat{\sigma}^2 = \frac{1}{12}\cdot[.01+.025+.01+.01+.25+4+.09+.04+.01+0+.01+0] = .5567 \quad , \qquad (7.86)$$

leaving the subsymbol noise variance to divide each $|y-x|^2$ calculation as $2\hat{\sigma}^2 = 1.1133$ as in Figure 7.16. Figure 7.16's table shows the deviating bit in red. Again, the σ value does not change the decision, but does change the indicated confidence of the decision.

Instead, the BCJR_AWGN program finds the correct sequence (suggesting that this author has a mistake in Figure 7.16 although no one has found it yet):

```
> yawgn2
  Columns 1 through 8
   -0.9000    0.5000   -1.1000   -0.9000   -0.5000    1.0000   -0.8000    0.1000
  Columns 9 through 12
    0.9000    1.0000   -0.9000    0.9000
>> BCJR_AWGN(-yawgn2,t,1.1133/2) =
    5.7066    6.2779    2.5626    2.5684   -6.4242   -2.5681
```

Indeed the program finds the same input as other decoders have on this particular output.

7.3.2 Soft-Output Viterbi Algorithm (SOVA)

Hagenauer[9] first recognized the trellis contains additional "soft information" about the likelihood of a decision's correctness, coining the term "**Soft-Output Viterbi Algorithm (SOVA)**" [5] as an augmentation to the usual Viterbi Detector. SOVA additionally provides the $\boldsymbol{Y}_{0:K-1}$-conditioned probability of a specific subsymbol value $\boldsymbol{x}_k$ at time k in the final surviving detected sequence. This soft information measures each subsymbol $\boldsymbol{x}_k$'s reliability and finds use in outer (or second) codes in concatenated systems. This soft-information reliability indicator, usually in form of the LLR or ΔLLR also can resolve ties when two BSC path metrics are the same. This tie resolution improves upon simple random selection of one of the tired paths as the selected sequence.

LOGMAX: First, MLSD approximates MAP through the LOGMAX approximation, which replaces 7.81)'s log of the sum of products with $\lambda_{max} = \max_i \lambda_i$, presuming

$$e^{\lambda_{max}} \approx \sum_i e^{\lambda_i} \quad . \tag{7.87}$$

The maximum's selection leads to a VA-like survivor selection (and associated computation) for the particular LOGMAP in question. Equation (7.87)'s justification assumes $\lambda_i << \lambda_{max}$ because exponentiation amplifies the size difference. If instead all terms are included in all sums throughout APP, then LOGMAP is equivalent to the APP. Often inclusion of a few terms is sufficient. The LOGMAX retains the maximum and is indeed the VA when applied to the α_k recursion, as follows:

Relating MLSD to MAP through LOGMAX: Section 7.1's Viterbi Algorithm minimizes squared distances or Hamming distances for the AWGN and BSC respectively. Since the ln function is montonic, this does this maximization's optimum argument. Direct use of squared distance or Hamming distance therefore maximizes the probability distribution's natural logarithm, i.e., Chapter 1's likelihood function,

$$LL_{\boldsymbol{x}/\boldsymbol{y}} \stackrel{\Delta}{=} \ln\left(p_{\boldsymbol{x}_k/\boldsymbol{y}}\right) \quad . \tag{7.88}$$

The conditional likelihoods are correspondingly

$$\begin{aligned}
LL_{\boldsymbol{x}_k/\boldsymbol{Y}_{0:K-1}} &\stackrel{\Delta}{=} \ln\left(p_{\boldsymbol{x}_k/\boldsymbol{Y}_{0:K-1}}\right) && (7.89)\\
LL_{\boldsymbol{x}(D)/\boldsymbol{y}(D)} &\stackrel{\Delta}{=} \ln\left(p_{\boldsymbol{x}(D)/\boldsymbol{y}(D)}\right) && (7.90)\\
LL_{\boldsymbol{y}(D)/\boldsymbol{x}(D)} &\stackrel{\Delta}{=} \ln\left(p_{\boldsymbol{y}(D)/\boldsymbol{x}(D)}\right) && (7.91)\\
LL_{\boldsymbol{y}_k/\boldsymbol{x}_k} &\stackrel{\Delta}{=} \ln\left(p_{\boldsymbol{y}_k/\boldsymbol{x}_k}\right) \quad . && (7.92)
\end{aligned}$$

The sequence that maximizes the $LL_{\boldsymbol{X}(D)/\boldsymbol{Y}(D)}$ 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

[9] Joachim Hagenauer (1941 –), a German Electrical Engineering Professor at Technical University of Munich and specialized in information theory and satellite communication.

to handle directly in computation than the exact likelihood function. The Viterbi detector inherently finds the best sequence, unlike Subsection 7.3.1's APP detector that finds a sequence's best individual subsymbol (or bit) values.

SOVA processes the simpler, Euclidean or Hamming distance-based, likelihood functions directly. Often with binary codes, SOVA may propagate the LLRs directly. Section 7.3's APP process sums, for each sequence subsymbol value $\boldsymbol{x}_k$, probabilities associated with a product $\alpha_k \cdot \gamma_k \cdot \beta_k$. Indeed Figures 7.14 and 7.24 illustrate the calculation. The LOGMAP with maximum implementation implements the forward α's recursive calculation and also the backward β's recursive calculation. The branch metrics γ are LL's (or LLRs directly sometimes), so the forward MAP recursion (α recursion) becomes the normal VA when the maximum term replaces MAP's sum of products $\alpha_{k+1} = \sum_{\text{branches}} \alpha_k \cdot \gamma_k$ or

$$\ln(\alpha_{k+1}, s_{k+1}) \approx \max_{\substack{\text{branches} \\ \text{into } s_{k+1}}} \left\{ \ln(\alpha_k, s_k, \text{branch into}) + \ln(\gamma_k, \text{branch into}) \right\} \quad . \tag{7.93}$$

This is the VA in the forward direction. Similarly in the backward direction

$$\ln(\beta_k, s_k) \approx \max_{\substack{\text{branches} \\ \text{into } s_k}} \left\{ \ln(\beta_{k+1}, s_k, \text{branch into}) + \ln(\gamma_k, \text{branch into}) \right\} \quad . \tag{7.94}$$

The backward part is also the VA, just in the opposite direction. Thus, there are two VAs. The final step uses the sum of $\ln(\alpha) + \ln(\gamma) + \ln(\beta)$ for each stage to compute separately for each subsymbol (or bit) value (0 or 1) the individual LLRs. There are various recursions to develop for this last step, but they also are simple addition plus table-lookup based operations.

Again with the LOGMAX approximation taking the maximum of these 4 terms then the final ΔLL becomes (recalling that $LLR = \Delta LL$ for binary codes (only).

$$\begin{aligned} \Delta LL_{\boldsymbol{x}_k} &= \pm \Big[\max_{0 \text{ branches}} \left\{ \ln(\alpha_k, \text{branch}) + \ln(\gamma_k, \text{branch}) + \ln(\beta_k, \text{branch}) \right\} \\ &\quad - \max_{1 \text{ branches}} \ln(\alpha_k, \text{branch}) + \ln(\gamma_k, \text{branch}) + \ln(\beta_k, \text{branch}) \Big] \quad , \end{aligned} \tag{7.95}$$

where the ± simply is + when the maximum zero term is larger than the maximum one term, and − otherwise. When the two are equal, $\Delta LL = 0$. This is almost the same as selecting the survivor branch of the best surviving path. The difference is the bi-directional use of α for the past with respect to the decision and β for the future with respect to the decision. This is known as the forward-backward SOVA detector. It is the same if there is one surviving path with the same metric for the entire sequence/packet; otherwise, if there are "ties" with the same metric, the forward-backward SOVA provides additional soft information that exceeds that of the forward Viterbi alone.

SOVA's Soft Information: SOVA's soft information for subsymbol $\boldsymbol{x}_k$ comes from the full set of $2^\nu \cdot 2^{\tilde{b}}$ branches (for binary rate $1/n$ code, this is $2^{\nu+1}$). These branches subdivide into $2^{\tilde{b}}$ equal sets for each possible subsymbol value. For binary rate $1/n$ codes, this is just two sets. The LOGMAP selects the maximum from each set to obtain the log likelihood estimate $LL(\boldsymbol{x}_k)$. A final decision selects the largest. MAP bit estimates use (7.95), and repeat it if $\tilde{b} > 1$ (that is likely BICM in use) for each bit by averaging (integrating) over the other bits values in γ_k to determine the LLR_k value for each of the $\tilde{b}$ bits at time k. Before formalizing, the following example illustrates both the forward and backward SOVA calculations for the $r = 1/2$ 4-state convolutional code.

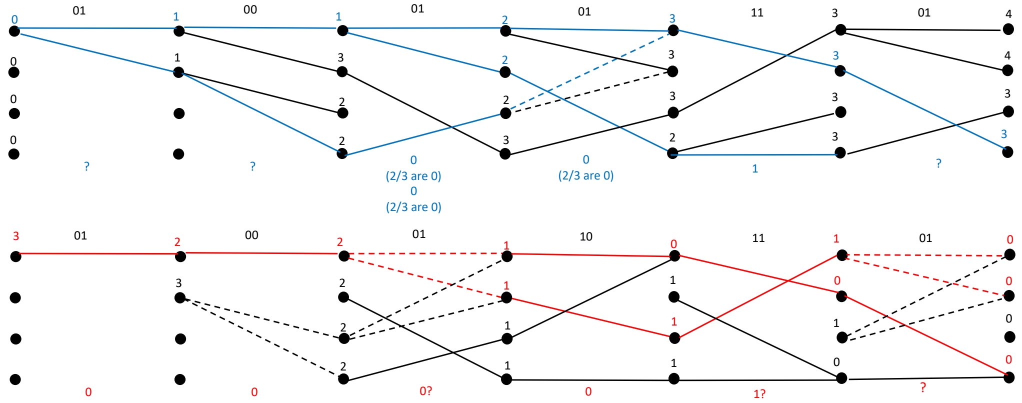
> **EXAMPLE 7.3.3** *[SOVA for Subsection 7.1.1's example and also Example 7.3.1]* Figure 7.17's upper blue-survivor trellis first lllustrates the forward SOVA with again the same 3 output-bit-error pattern for the BSC. This example immediately illustrates the SOVA assistance in the upper table because there is a tie: two survivor paths have $d_H = LL = 3$. MLSD nominally "flips a coin" and selects one. However, it is clear that the survivors have sets of values for decision of 0 or 1 at each stage, which Figure 7.17 illustrates in green color. For sample times $k = 2, 3, 4, 5$, the survivors either indicate a decision or a majority-vote decision, even though the ultimate path metric is the same. The forward-only SOVA cannot

resolve times $k = 0, 1$ without further noting that at time $k = 2$, one of the paths locally has a lower metric of 1 versus 2 for the other. The author believes this is how the earlier matlab VA produced the same result - thus it has undocumented tie resolution that is equivalent to soft-information use.

Forward SOVA Example with ties (3-error example revisited)

k	0	1	2	3	4	5
$\{LL(0)\}$	{3}	{3}	{3,3}	{3,3}	∅	{3}
$\{LL(1)\}$	{3}	{3}	{3}	{3}	{3,3}	{3}
ΔLL (**dec**)	0(**?**)	0(**?**)	$^2/_3$ (**0**)	$^2/_3$ (**0**)	-1 (**1**)	0 (**?**)

Green color indicates the minimum-metric path is a survivor in forward direction; all LL's in units of $ln(p)$.



Forward-Backward SOVA

k	0	1	2	3	4	5
$\{LL(0)\}$	$\left\{\underbrace{3}_{0+1+2}\right\}$	$\left\{\underbrace{3}_{1+0+2}\ \underbrace{4}_{1+1+2}\right\}$	$\left\{\underbrace{3}_{1+1+1}\ \underbrace{6}_{3+2+1}\ \underbrace{4}_{2+1+1}\ \underbrace{3}_{2+0+1}\right\}$	$\left\{\underbrace{3}_{2+1+0}\ \underbrace{5}_{2+2+1}\ \underbrace{3}_{2+1+0}\ \underbrace{4}_{3+0+1}\right\}$	$\underbrace{6}_{3+2+1}\ \underbrace{5}_{3+1+1}\ \underbrace{4}_{3+0+1}\ \underbrace{4}_{2+1+1}$	$\left\{\underbrace{4}_{3+1+0}\ \underbrace{5}_{3+2+0}\ \underbrace{4}_{3+0+1}\ \underbrace{3}_{3+0+0}\right\}$
$\{LL(1)\}$	$\left\{\underbrace{4}_{0+1+3}\right\}$	$\left\{\underbrace{6}_{1+2+3}\ \underbrace{4}_{1+1+2}\right\}$	$\left\{\underbrace{3}_{1+1+1}\ \underbrace{4}_{3\ +0+1}\ \underbrace{4}_{2+1+1}\ \underbrace{5}_{2+2+1}\right\}$	$\left\{\underbrace{4}_{2+1+1}\ \underbrace{3}_{2+0+0}\ \underbrace{4}_{2+1+1}\ \underbrace{6}_{3+2+1}\right\}$	$\underbrace{5}_{3+2+0}\ \underbrace{4}_{3+1+0}\ \underbrace{3}_{3+0+0}\ \underbrace{3}_{2+1+0}$	$\left\{\underbrace{4}_{3+1+0}\ \underbrace{3}_{3+0+0}\ \underbrace{4}_{3+1+0}\ \underbrace{5}_{3+2+0}\right\}$
ΔLL (**dec**)	1 (**0**)	1 (**0**)	$^2/_3$ (**0**)	$^2/_3$ (**0**)	-1 (**1**)	0 (**?**)

Green color indicates the minimum-metric path is a survivor in both forward and backward directions; all LL's in units of $ln(p)$

Figure 7.17: Example of rate 1/2 convolutional code with 3 output errors and SOVA decoding.

The backward VA also appear sin Figure 7.17. Neither the basic forward nor the backward basic VA can correctly detect the input sequence because of the tie. The question marks indicate bits for which the multiple equal-metric survivors do not favor either bit choice (0 or 1). However, Figure 7.17's lower forward-backward table lists the LL sets for each bit value. The last row indicates the decision that corresponds to taking the difference (and applying correct the sign). The table entries have underneath them 3-term sums that correspond to summing a branch's forward-VA metric (on the left), current metric, and backward-VA metric (on the right). These sums are in green color if both forward and backward paths are survivors. All other sums (which the detector need not compute in practice) are in black color When the sets have ties, only one branch decision in this example is in both blue forward and red backward survivors, thus the soft information resolves all the ties and decides the known correct sequence in this example. The lower forward-backward table also approximates the LLR by the ratio of occurrence of the lowest metric for each value (whether in a survivor or not).

As evident, forward-backward SOVA definitively resolves the correct input sequence, just as did the MAP. The LLR values may not be as accurate as MAP, but the SOVA process is far less complex. Further, the SOVA LLR has magnitude less than 1 when there are ties, but is clearly a rough approximation.

is still useful to another decoder that may have also used the corresponding bit. The use of a backward VA requires an endpoint. Good designers can find ways to update the backward path if the final time from which it originates advances. However, the soft-information generation most commonly finds use in concatenated systems with interleavers (see Chapter 8. Such systems necessarily block information into groups or packets, and ultimately the entire corresponding set of outputs is commonly available. Thus, the forward-backward SOVA directly applies.

SOVA Metric Formalization: In those cases, where only forward continuous update occurs, an alternative SOVA metric follows:

Definition 7.3.2 **[SOVA Forward-Only Soft-Information Metric]** *Let j_k^* denote the maximum-likelihood survivor's state at time k; so if SOVA looks ahead* **survivor-length** *λ time samples to trace-back a good survivor at time k, the state of interest is $j_{k+\lambda}$. $J_{k+\lambda}^*$ is the set of other states that do not include the best survivor path at time $k+\lambda$ for which SOVA computes the soft metric. SOVA retains the set of differences, or* **soft-information metric***, from the next-closest paths*

$$\Delta LL_{m_k^*,m_k'} \triangleq \Delta LL_k = \min_{j \in J_{k+\lambda}^*} \left\{ LL_{\boldsymbol{X}_{0:k+\lambda}^*}(j_{k+\lambda}^*) - LL_{\boldsymbol{X}_{0:k+\lambda}'}(j)\ m_k^* \neq m_k'\ , \lambda > 0 \right\} \tag{7.96}$$

over the survivor length for each state at each time. The gain factor $1/g$ appears below for each of BSC and AWGN.

When there are ties, forward SOVA tie resolution first checks the corresponding input to see its frequency of occurrence at time k among the survivors. Forward SOVA then selects the one with largest frequency of occurrence and sets the LLR magnitude equal to that fraction (which will be no larger than 1) and sign equal to ± corresponding to 1 and 0 respectively as the higher frequency of occurrence. When the frequencies of occurrence are equal, $LLR_k = 0$. A SOVA local decision may be made based on surround trellis stage's metrics (which may improve on random selection slight), the no SOVA soft information emanates in this case. Mathematically

$$BSC:\ \Delta LL_k = \frac{1}{g_{bsc}} \frac{\left|\left\{LL_{\boldsymbol{X}_{0:K-1}^*}(j_k^*) = \cdot L_{min}\right\}\right|}{\left|\left\{LL_{\boldsymbol{X}_{0:K-1}^*}(j_k^*) = L_{min}\right\}\right| + \left|\left\{LL_{\boldsymbol{X}_{0:K-1}'}(j_k^*) = L_{min}\right\}\right|} \tag{7.97}$$

$$AWGN:\ \Delta LL_k = \frac{1}{g_{awgn}} \cdot \ln \frac{\left|\left\{LL_{\boldsymbol{X}_{0:K-1}^*}(j_k^*) = L_{min}\right\}\right|}{\left|\left\{LL_{\boldsymbol{X}_{0:K-1}^*}(j_k^*) = L_{min}\right\}\right| + \left|\left\{LL_{\boldsymbol{X}_{0:K-1}'}(j_k^*) = L_{min}\right\}\right|} \tag{7.98}$$

with $g_{bsc} \triangleq \ln(p)$ and $g_{awgn} \triangleq 4 \cdot d_{free} \cdot SNR$.

For Example 7.3.3 with ties, this soft-information LLR will be zero for 1/2 the bits, illustrating that the forward-only indeed has less useful soft information. Thus, this text introduces a Forward-Backward SOVA metric that follows the example

Definition 7.3.3 **[Forward-Backward SOVA Metric]** *The forward backward SOVA metric for each stage computes first, with J_k^* corresponding to the best forward survivor path into stage k and J_k' corresponding to the best backward survivor path into stage k with the branch at time k contained in neither set,*

$$LL_k \triangleq \min_{\substack{j \in J_k^* \\ j' \in J_k'}} \left\{ LL_{\boldsymbol{X}_{0:k}^*}(j_k^*) + LL_{\boldsymbol{x}_k^*}(j_k^*) + LL_{\boldsymbol{X}_{k+1:K-1}^*}(j_k^*) \right. \tag{7.99}$$

$$\left. - \left[LL_{\boldsymbol{X}_{0:k}'}(j_k') + LL_{\boldsymbol{x}_k'}(j_k') + LL_{\boldsymbol{X}_{k+1:K-1}'}(j_k') \right]\ m_k^* \neq m_k'\ , \right\} \tag{7.100}$$

Ties are handled in the same way as the forward SOVA except that a survivors must occur jointly in both forward and backward paths.

7.3.2.1 Iterating the LLR Directly with Forward SOVA

SOVA and VA provide the same ML sequence estimate. There are 2^ν states and corresponding survivors, looking ahead to time $k+\lambda$, or simply at the end of a packet/codeword that the decoder processes. These have log likelihoods as LS. The sequence-error probability for the best survivor path for input bit value 0 at time k is $LL_k^*(0)$ is

$$Pr_{MLSD}\{x_k = -1\} = Pr_{MLSD}\{u_k = 0\} \propto e^{-LS_k^*(0)} \tag{7.101}$$

and similarly

$$Pr_{MLSD}\{x_k = +1\} = Pr_{MLSD}\{u_k = 1\} \propto e^{-LS_k^*(1)} \ , \tag{7.102}$$

which uses the abbreviation LS_k^* to indicate $\ln(p_{\boldsymbol{x}(D)/\boldsymbol{y}(D)}(u_k = 0)$, the largest survivor metric for which the MLSD decides 0, and similarly for 1. For binary codeswith MLSD. the quantities LLR_k and $\Delta LS_k = |LS_k^*(0) - LS_k^*(1)|$ relate through

$$LLR_k = LS_k(0) - LS_k(1) = x_k \cdot \Delta LS_k \quad . \tag{7.103}$$

The difference is that LLR_k measures the natural logarithm of the two probabilities that SOVA would decide 0 or 1 respectively for a specific bit at time k; however ΔLS_k measures the probability that the best sequence and the next best sequence differ at that same time k, in other words ΔLS_k measures sequence error (presuming all zeros is correct without generality loss for linear code)

$$P_e = \frac{e^{-LS_k^*(0)}}{e^{-LS_k^*(0)} + e^{-LS_k^*(1)}} = \frac{1}{1+e^{\Delta LS_k}} \quad . \tag{7.104}$$

They have the same magnitude, but differ in sign, effectively treating the all zeros sequence in a linear code as the correct sequence.

Another decoder's LLR estimate, $\widehat{LLR}_k$ estimates $\hat{\bar{P}}_b \approx \frac{1}{1+e^{\widehat{LLR}_k|}}$. As an à prior input to SOVA, the sequence log likelihood ratio is

$$\widehat{LLR}_k = \ln \frac{1-\hat{\bar{P}}_{b,k}}{\hat{\bar{P}}_{b,k}} \quad , \tag{7.105}$$

(7.105 relates the sequence-error probability to the likelihood of a specific input bit being erred. Intuitively a sequence error for a reasonably powerful code occurs when there are more than $d_{free}/2$ positions, or more noise than $d_{min}/2$ for the AWGN. The positions of input bit errors are likely to be anywhere along the sequence[10], then Hagenauer's SOVA provides the following update relation

$$\bar{P}_{b,k} \leftarrow \underbrace{\hat{\bar{P}}_{b,k}}_{\text{bit differs}} \cdot \underbrace{\frac{e^{\Delta LS_k}}{1+e^{\Delta LS_k}}}_{\text{survivor correct}} + \underbrace{(1-\hat{\bar{P}}_{b,k})}_{\text{bit same anyway}} \cdot \underbrace{\frac{1}{1+e^{\Delta LS_k}}}_{\text{survivor incorrect}} \quad . \tag{7.106}$$

[10] The one exception is going to be the first position in the sequence error event that necessarily must correspond to a bit error. Thus, the time k of interest should not be the first subsymbol instant of the sequence-error event.

Then, with algebra shown:

$$LLR_k \leftarrow \ln \frac{1+e^{\Delta LS_k} - \hat{\hat{P}}_{b,k}\cdot e^{\Delta LS_k} - (1-\hat{\hat{P}}_{b,k})}{\hat{\hat{P}}_{b,k}\cdot e^{\Delta LS_k} + (1-\hat{\hat{P}}_{b,k})} \tag{7.107}$$

$$\leftarrow \ln \frac{e^{\Delta LS_k}/\hat{\hat{P}}_{b,k} - e^{\Delta LS_k} + 1}{e^{\Delta LS_k} + (1-\hat{\hat{P}}_{b,k})/\hat{\hat{P}}_{b,k}} \tag{7.108}$$

$$\leftarrow \ln \frac{1+e^{\Delta LS_k}\cdot \frac{1-\hat{\hat{P}}_{b,k}}{\hat{\hat{P}}_{b,k}}}{e^{\Delta LS_k} + (1-\Delta LS_k)} \tag{7.109}$$

$$\leftarrow \ln \frac{1+e^{\Delta LS_k + \widehat{LLR}_k}}{e^{\Delta LS_k} + e^{\widehat{LLR}_k}} \ . \tag{7.110}$$

If either $\widehat{LLR}_k = 0$ or $\Delta LS_k = 0$, then the Hagenauer update provides zero soft information, which makes sense. However when both are non-zero, there is nonzero soft information that updates. The multiplication of This can be used to return information the other (or other) decoder(s). The addition $\Delta LS_k + \widehat{LLR}_k$ ignores that the ΔLS_k had a common scale factor in front of the Gaussian distribution that depends on the number of terms, and so when added to a single-bit term, $\widehat{LLR}_k$, there is a scaling difference in Equation (7.110) that should be adjusted accordingly. Thus, for the AWGN, typically the relationship

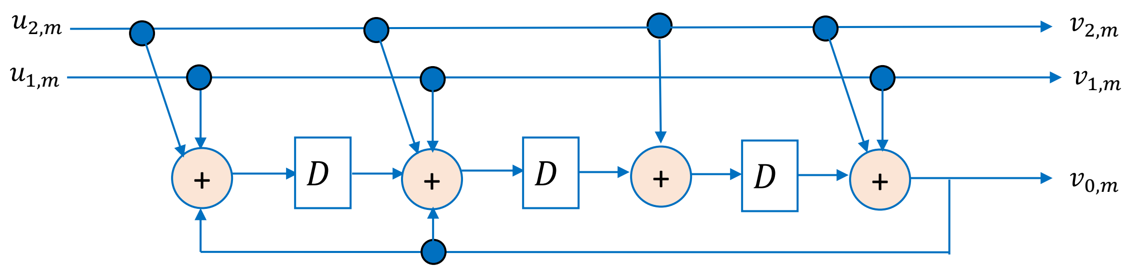
$$\Delta LS_k = \frac{\Delta_k}{4\cdot d_{free}\cdot SNR} \tag{7.111}$$

is used to bound this term's size, where Δ_k is directly the time k squared erros between channel output and postulated channel input. For the BSC, the author suggests similarly

$$\Delta LS_k = \frac{\Delta_{Hamming,k}}{d_{free}} \ . \tag{7.112}$$

7.3.3 Using a feedback-free generator's decoder to decode systematic with feedback

Appendix shows that two different generators for the same code have a feedback-free transformation between them. This is useful with systematic encoders with feedback, when the receiver may use the decoder for a different encoder. This arises for instance in matlab poly2trellis' unnecessary increase in number of states for systematic codes. An example helps illustrate the issue and solution. Figure 7.18 illustrates a systematic encoder for the best known 8-state $r = 2/3$ convolutional code.



$$H(D) = [17 \quad 15 \quad 13] = [D^3+D^2+D+1 \quad D^3+D^2+1 \quad D^3+D+1]$$

$$H_{sys}(D) = \begin{bmatrix} \frac{D^3+D^2+D^1+1}{D^3+D+1} & \frac{D^3+D^2+1}{D^3+D+1} & 1 \end{bmatrix}$$

$$G_{sys}(D) = \begin{bmatrix} 1 & 0 & \frac{D^3+D^2+D^1+1}{D^3+D+1} \\ 0 & 1 & \frac{D^3+D^2+1}{D^3+D+1} \end{bmatrix}$$

Figure 7.18: Example of rate 1/2 convolutional code with 3 output errors and SOVA decoding.

The next commands illustrate the poly2trellis issue

```
tfeed=poly2trellis([4 4],[13 0 17 ; 0 13 15], [13 13]);
     tfeed.numInputSymbols: 4
    tfeed.numOutputSymbols: 8
           tfeed.numStates =  64
          tfeed.nextStates: [64x4 double]
             tfeed.outputs: [64x4 double]
```

64 states is too many. Indeed matlab's own "istrellis" command says its own trellis is invalid.

The solution uses Appendix B's invariant factors decomposition to find

$$G_{sys}(D) = \left[\begin{array}{cc} 1+D+D^2+D^3 & 1+D+D^2 \\ 1+D^2+D^3 & D+D^2 \end{array} \right] \cdot \left[\begin{array}{cc} \frac{1}{1+D+D^3} & 01+D+D^2 \\ 0 & 1 \end{array} \right] \cdot \left[\begin{array}{ccc} D & 1+D^2 & 1+D^2 \\ 1+D & D & 1 \end{array} \right] . \qquad (7.113)$$

Because the first two matrices are invertible, an equivalent 8-state encoder is given by the last matrix above. The following matlab commands then illustrate use of vitdec to decode a sequence encoded with the systematic encoder, but decoded in terms of the equivalent 8-state feedback-free decoder to find an input. That input is then processed through the 8-state decoder to get the estimated channel output that corresponds to the systematic encoder. This final sequence's first two bits (because it is systematic) of each subsymbol are the MLSD estimates for the original systematic encoder.

```
tmin=poly2trellis([3 2], [2 5 5; 3 2 1])
     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 8
>> tfeed=poly2trellis([4 4],[13 0 17 ; 0 13 15], [13 13])
     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 64
 outfeed=convenc([ 0 0       0 0      0 0       1 0       1 1      0 1     0 0      0 1],tfeed)
>> error2 = [ 0 0 1  0 0 0   0 0 0   0 0 0  0 0 0   0 0 0   0 1 0  0 0 0 ];  % 2 errors introduced
>> informin2=vitdec(+xor(outfeed,error2),tmin,6,'trunc','hard')
0 0     0 0     0 0     1 1     0 1     1 1     0 1     1 1  % not same as systematic enc's input
>> vmin2 = convenc(informin2,tmin)
     000   000  000  101   111  011   001  011
```

The first two bits of vmin2 match the input to the systematic encoder. As Appendix B shows, the invariant factors decomposition can be tedious to execute, but is straightforward. Unfortunately, this method, also sometimes better known as Smith Normal form when not in a finite field, does not appear to have canned software anywhere that executes it for a finite field. (This is probably a great project for motivated student.)

7.4 Soft Information Generation

Section 7.3's soft-information generation derives from a trellis model for a code (and/or ISI). Soft-information also arises in block codes as well as from multi-level signal constellations. Chapter 2's binary block codes have parity-check matrices[11]. Each parity-check equation can provide soft-information, as in Subsections 7.4.1 and 7.4.2. The assignment of code bits to a multilevel constellation also provides soft information as in Subsection 7.4.3. The use of a full SOVA to provide ISI-based soft-information from a trellis may simplify as in Subsection 7.4.4's soft-canceller approximation.

Overall, a constraint can supply extrinsic soft information on each bit or subsymbol in terms of known relationships to other bits or symbols – that is, the extrinsic information reflects the constraints. Additionally, the à priori and channel probabilities provide"intrinsic" soft information. Notationally, the à posteriori probability ($p_{app} = p_{subsymbol/channel-output}$) factors as

$$p_{app} = p_{\boldsymbol{x}_k/\text{constraints}} \propto p_{\boldsymbol{x}_k,\text{constraints}} = p_{intrinsic} \cdot p_{extrinsic} \stackrel{\Delta}{=} p_{int} \cdot p_{ext} \quad . \tag{7.114}$$

The intrinsic probability measures the channel-related input bit or subsymbol at time k, essentially the BCJR's quantity γ_k. The extrinsic probability represents information contributed from all other channel subsymbols or bits at other times $k' \neq k$.

The letter E represents a **constraint event**, which provides extrinsic information to a decoding process. $Pr\{E\}$ is probability that the event occurs. This event satisfaction probability is not a function of any specific input, and instead represents an average over all the possible inputs.

7.4.1 Bit-Level or Parity Constraints

A parity-check equation sums BSC output bits that correspond to the 1 entries in the code's parity matrix H. With encoder outputs as v_k, a 3-bit parity constraint example is

$$v_1 \oplus v_2 \oplus v_9 = 0 \quad , \tag{7.115}$$

and corresponding BSC outputs as y_1, y_2, and y_9, $P(E) = 1$ at the encoder output, but is less at the BSC output. The BSC has parameter p, while the encoder output bit has some probability $Pr\{0\} = p_0$ (which may represent for instance soft information from another code or constraint, and thus may not be simply 1/2). That prior distribution for the encoder-output bit is independent of the (to-be-applied) constraint before it is applied. The independence allows BSC joint channel-input-output-bit probabilities then to follow easily as ($i = 1, 2, 9$ in (7.115))

$$p(v_i, y_i) = p(y_i/v_i) \cdot p(v_i) = \begin{cases} (1-p) \cdot p_i & y_i = 1 \; v_i = 1 \\ \quad p \cdot (1-p_i) & y_i = 1 \; v_i = 0 \\ \quad p \cdot p_i & y_i = 0 \; v_i = 1 \\ \underbrace{(1-p)}_{p_{int}} \cdot \underbrace{(1-p_i)}_{p_{ext}} & y_i = 0 \; v_i = 0 \end{cases} \quad . \tag{7.116}$$

(7.116)'s p-dependent factors are the channel's intrinsic information while the p_i factors are the extrinsic information, which may arise as soft information from other constraints' decoders. With a memoryless-channel assumption, an AWGN with binary PAM inputs $\pm\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}}$ has

$$p(v_i, y) = \begin{cases} \underbrace{\frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{1}{2\sigma^2}(y-\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}})^2}}_{p_{int}} \cdot \underbrace{p_i}_{p_{ext}} & v_i = 1 \\ \underbrace{\frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{1}{2\sigma^2}(y+\sqrt{\bar{\mathcal{E}}_{\boldsymbol{x}}})^2}}_{p_{int}} \cdot \underbrace{(1-p_i)}_{p_{ext}} & v_i = 0 \end{cases} \quad . \tag{7.117}$$

As in the following, the distribution $p(y, v_i)$ enables computation of $P(E)$.

[11] See also Chapter 8 for a more complete development of binary block codes.

The event and probability calculation: (7.115)'s 3-bit constraint views the "other 2" bits as extrinsic information, equivalently each having a p_i value in (7.116) or (7.117). For any constraint event E, the maximum à posteriori decoder that observes or uses the constraint would then maximize (7.114)

$$\max_{v_i=0,1} p_{v_i/E} \quad . \tag{7.118}$$

This MAP decision is constraint-specific and produces soft information along with any hard decision based thereupon. Any encoder output, whether in bits $\{v_i\}$ or as the subsymbol value $\boldsymbol{x}_i$, maps uniquely at subsymbol sampling[12] time i into the corresponding encoder-output/channel-input bit v_i (as long as encoder state at that time is known), so there is no loss of optimality in directly estimating the encoder output symbols. The event-satisfaction APP is

$$p_{v_i/E} = \frac{p(v_i, E)}{P(E)} = \frac{1}{P(E)} \cdot \underbrace{p(v_i)}_{p_{int}} \cdot \underbrace{p_{E/v_i}}_{p_{ext}} \quad . \tag{7.119}$$

Dependency upon the channel output $\boldsymbol{y}$ is tacitly implied through event $E/$ The scaling term $1/P(E)$ is a constant that is not a function of any specific encoder-output bit v_k; therefore this scaling term does not influence the MAP decision for v_k. Generally, a constraint is[13]

$$E([v_n, v_2, ..., v_1]) = E\left(\boldsymbol{v}\right) = 0 \quad . \tag{7.120}$$

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

$$S_E \stackrel{\Delta}{=} \{\boldsymbol{v} \mid E(\boldsymbol{v}) = 0\} \quad . \tag{7.121}$$

The notation $S_{E\setminus i}(y_i)$ denotes all set members may take any value except that each member's element i must satisfy $v_i = y_i$. Any specific fixed channel (BSC or AWGN) output y_i therefore has extrinsic probability factor

$$p_{E/v_i} = p_{ext}(E, y_i) = c_i \cdot \sum_{\boldsymbol{v} \in S_{E\setminus i}(y_i)} \prod_{\substack{j=1 \\ j \neq i}}^{n} p_j(E, y_i) \quad . \tag{7.122}$$

Equation (7.122)'s product values $p_j(E, y_i)$ are from the contraints other bits' probabilities. Again, (7.122)'s sum, as a function of y_i, executes over that subset of S_E that has a specific fixed value for $y_i = v_i$. There will be 2 values for the BSC and a continuous distribution for the AWGN. The constant c_i's inverse sums (integrates for AWGN) over these values, but is not necessary for the constraint's MAP decoder.

The constant c_i's calculation: c_i is

$$c_i = \left[\sum_{\boldsymbol{v} \in S_E} \prod_{j=1}^{n} p_j(E, y_i) \right]^{-1} \quad . \tag{7.123}$$

The constant is again inconsequential in subsequent maximization over v_i. The intrinsic or à priori distribution is

$$p_{int} = p(v_i = y_i) \quad , \tag{7.124}$$

essentially incorporating the current subsymbol index i's value into the overall APP and being soft information that may find use in another constraint's decoder. The decoder thus maximizes the joint probability

$$p_{ext}(v_i, E) = \frac{1}{P(E)} \cdot \sum_{\boldsymbol{v} \in S_{E\setminus i}(y_i)} \left[\prod_{j=1}^{n} p(v_j) \right] \quad , \tag{7.125}$$

[12] The index $i = 1, ..., n$ for encoder output bits and $i = 1, ..., k$ for encoder input bits.

[13] The symbol E should not be confused with the notation $\mathbb{E}$ for expectation.

and the à posteriori is

$$p_{v_i/E} = \underbrace{\frac{c_i}{P(E)}}_{c'_i} \cdot \sum_{\boldsymbol{v} \in S_{E \setminus i}(y_i)} \left[\prod_{j=1}^{n} p(v_j) \right] \quad . \tag{7.126}$$

The expressions in (7.125) and (7.126) are distributions. A decoder computes them for each v_k value; the calculation holds this value fixed in the sum over all sequences that satisfy the constraint E. The probability $P(E)$ is

$$P(E) = \sum_{\boldsymbol{v} \in S_E} \left[\prod_{i=1}^{n} p(v_i) \right] \quad , \tag{7.127}$$

where this sum over S_E does not fix the value v_k and so is over all vectors $\boldsymbol{v}$ that satisfy the constraint. Each channel output sample initiates calculation of an intrinsic $p(v_k)$ in (7.119). The constraint manifests itself through the set S_E that excludes all-non-event-satisfying bit combinations.

EXAMPLE 7.4.1 *[3-bit Parity constraint]* A 3-bit parity-check constraint event specifies a modulo-2 sum that must be zero

$$v_1 \oplus v_2 \oplus v_3 = 0 \quad . \tag{7.128}$$

Basically, these bits correspond to an H-matrix row's 1 entries. To simplify here, the 3 indices will be successive, but this of course not necessary in general. Further notational simplification just denotes the 3 à priori (which may be extrinsic from other constraints) probabilities as p_i for $i = 1, 2, 3$.

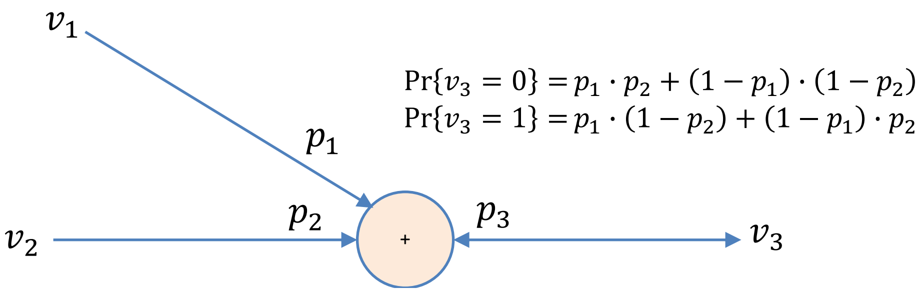

v_1
$\Pr\{v_3 = 0\} = p_1 \cdot p_2 + (1-p_1) \cdot (1-p_2)$
$\Pr\{v_3 = 1\} = p_1 \cdot (1-p_2) + (1-p_1) \cdot p_2$
p_1
p_2
p_3
v_2
+
v_3

Figure 7.19: Graph of parity constraint.

The set of values that satisfy this constraint is $S_E = \{(0,0,0),\ (1,1,0),\ (1,0,1),\ (0,1,1)\}$. Figure 7.19 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 k =3, arising from bits 1 and 2, would be:

$$p_{ext}(v_3) = p_{E/v_3} = \begin{cases} p_1 \cdot p_2 + (1-p_1) \cdot (1-p_2) & v_3 = 0 \\ p_1 \cdot (1-p_2) + p_2 \cdot (1-p_1) & v_3 = 1 \end{cases} \quad . \tag{7.129}$$

This is a specific instance of Equation (7.122). Similarly for the other 2 bits in the parity check:

$$p_{ext}(v_2) = p_{E/v_2} = \begin{cases} p_1 \cdot p_3 + (1-p_1) \cdot (1-p_3) & v_2 = 0 \\ p_1 \cdot (1-p_3) + p_3 \cdot (1-p_1) & v_2 = 1 \end{cases} \quad , \tag{7.130}$$

and

$$p_{ext}(v_1) = p_{E/v_1} = \begin{cases} p_2 \cdot p_3 + (1-p_2) \cdot (1-p_3) & v_1 = 0 \\ p_2 \cdot (1-p_3) + p_3 \cdot (1-p_2) & v_1 = 1 \end{cases} \quad . \tag{7.131}$$

The decoder decision for v_3 maximizes (letting $p_3 = p_{BSC} = p$)

$$(c_3')^{-1} \cdot \begin{cases} p_1 \cdot p_2 \cdot (1-p_3) + (1-p_1) \cdot (1-p_2) \cdot (1-p_3) & v_3 = 0 \\ p_1 \cdot (1-p_2) \cdot p_3 + p_2 \cdot (1-p_1) \cdot p_3 & v_3 = 1 \end{cases} \quad , \tag{7.132}$$

where the sum is $c_3^{-1} = 1 - (p_1 + p_2 + p_3) + (2 \cdot p_1 \cdot p_3 + p_1 \cdot p_2 + 2 \cdot p_2 \cdot p_3) - 4p_1 \cdot p_2 \cdot p_3$, although not necessary for MAP decoding. Similar MAP decisions follow for bits 1 and 2.

For a decoding situation with soft 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, using (7.126), $P(v_3 = 0) = c_3' \cdot .49$ 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 Example 7.4.1 and received value y, p_1, p_2, and p_3 would all be functions of that particular y value as in (7.117).

More generally for memoryless channels, the i^{th} row of H, h_i, determines the i^{th} parity constraint's event set as

$$S_E = \{\boldsymbol{v} \mid \boldsymbol{v} \cdot h_i^* = 0\} \quad . \tag{7.133}$$

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

$$p_{ext}(E, (v_i) = e^{LLR_{ext,i}} = \sum_{\boldsymbol{v} \in S_{E \setminus k}(y_k)} \prod_{\substack{j=1 \\ j \neq k}}^{t_r} p_j(v_k) \quad . \tag{7.134}$$

Soft Bits: Defining the soft bit $\chi_i = 2 \cdot p_{ext}(v_i = 0) - 1 = 1 - 2 \cdot p_{ext}(v_i = 1)$, induction shows (see problem 7.19) for a parity constraint[14] with $\chi_j = 2 \cdot p_j(v_j = 0) - 1$ for $j \neq i$

$$\chi_i = \prod_{\substack{j=1 \\ j \neq i}}^{t_r} \chi_j \quad , \tag{7.136}$$

which is sometimes useful for simpler extrinsic-information calculation. The soft bit relates directly to the LLR as

$$LLR_{ext,i} = \ln\left(\frac{\chi_i + 1}{\chi_i - 1}\right) \tag{7.137}$$

or

$$\chi_i = -\tanh\left(\frac{LLR_{ext,i}}{2}\right) \quad . \tag{7.138}$$

and further introducing a involutory function that allows adding of soft-bit related information

$$\phi(LLR_{ext,i}) \stackrel{\Delta}{=} + \ln\left(\frac{e^{LLR_{ext,i}} + 1}{e^{LLR_{ext,i}} - 1}\right) = -\ln\left(\tanh\left[\frac{LLR_{ext,i}}{2}\right]\right) \quad , \tag{7.139}$$

or more generally

$$\phi(x) = \phi^{-1}(x) = -\ln\left[\tanh\left(\frac{x}{2}\right)\right] = \ln\left(\frac{e^x + 1}{e^x - 1}\right) \quad . \tag{7.140}$$

When $x = LLR_{ext,i}$, then $\chi_i \cdot \chi_j$ corresponds to $\phi(LLR_i) + \phi(LLR_j)$, avoiding multiplication and allowing only addition and table-look-up operations.

[14] Which follows from:

$$\frac{e^{\frac{1}{2}\ln\frac{p}{1-p}} - e^{-\frac{1}{2}\ln\frac{p}{1-p}}}{e^{\frac{1}{2}\ln\frac{p}{1-p}} + e^{-\frac{1}{2}\ln\frac{p}{1-p}}} = \frac{\sqrt{\frac{p}{1-p}} - \sqrt{\frac{1-p}{p}}}{\sqrt{\frac{p}{1-p}} + \sqrt{\frac{1-p}{p}}} = \frac{p - 1 + p}{p + 1 - p} \tag{7.135}$$

7.4.1.1 Parity-Constraint Implementation

The parity constraint soft-information relationship is then from (7.136) with t_r terms, using also (7.138),

$$\chi_i = \prod_{\substack{j=1 \\ j \neq i}}^{t_r} \chi_j = (-1)^{t_r - 1} \cdot \prod_{\substack{j=1 \\ j \neq i}}^{t_r} \tanh\left(\frac{\text{LLR}(p_j)}{2}\right) \tag{7.141}$$

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

$$\begin{aligned}
LLR(p_i) \quad &= \quad \ln \frac{1-\chi}{1+\chi} && (7.142) \\
&= \quad \ln \frac{1 - \prod_{\substack{j=1 \\ j \neq i}}^{t_r} (1 - 2p_j)}{1 + \prod_{\substack{j=1 \\ j \neq i}}^{t_r} (1 - 2p_j)} && (7.143) \\
&= \quad \ln \frac{1 - (-1)^{t_r - 1} \prod_{\substack{j=1 \\ j \neq i}}^{t_r} \tanh\left(\frac{1}{2}\text{LLR}(p_j)\right)}{1 + (-1)^{t_r - 1} \prod_{\substack{j=1 \\ j \neq i}}^{t_r} \tanh\left(\frac{1}{2}\text{LLR}(p_j)\right)} && (7.144) \\
&= \quad (-1)^{t_r} \cdot 2 \cdot \tanh^{-1}\left[\prod_{\substack{j=1 \\ j \neq i}}^{t_r} \tanh\left(\frac{1}{2}\text{LLR}(p_j)\right)\right] && (7.145) \\
&= \quad (-1)^{t_r} \cdot \prod_{\substack{j=1 \\ j \neq i}}^{t_r} \left[\text{sgn}\left(\text{LLR}(p_j)\right)\right] \cdot \phi^{-1}\left(\sum_{\substack{j=1 \\ j \neq i}}^{t_r} \phi(|\text{LLR}(p_j)|\right) \quad . && (7.146)
\end{aligned}$$

[Example 7.4.1 continued] See Figure 7.20 for the implementation diagram of the parity soft-information flows.

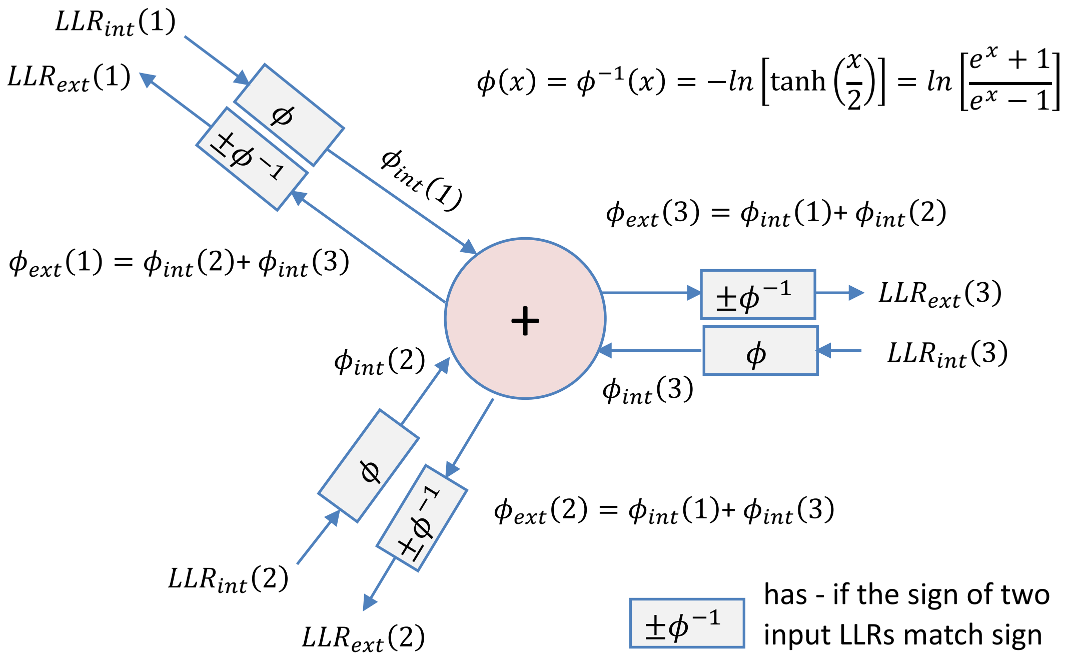


Figure 7.20: Parity constraint soft-information flow.

7.4.2 The Equality or Code-Level Constraint

Equality constraints basically observe that multiple parity checks may contain the same bit (or symbol). Then, the constraints provide information to one another through an equality constraint. A 3-bit equality-constraint example initiates this section, where the same bit named v_k appears in 3 different parity-check constraints.

EXAMPLE 7.4.2 *[3-bit equality constraint]*

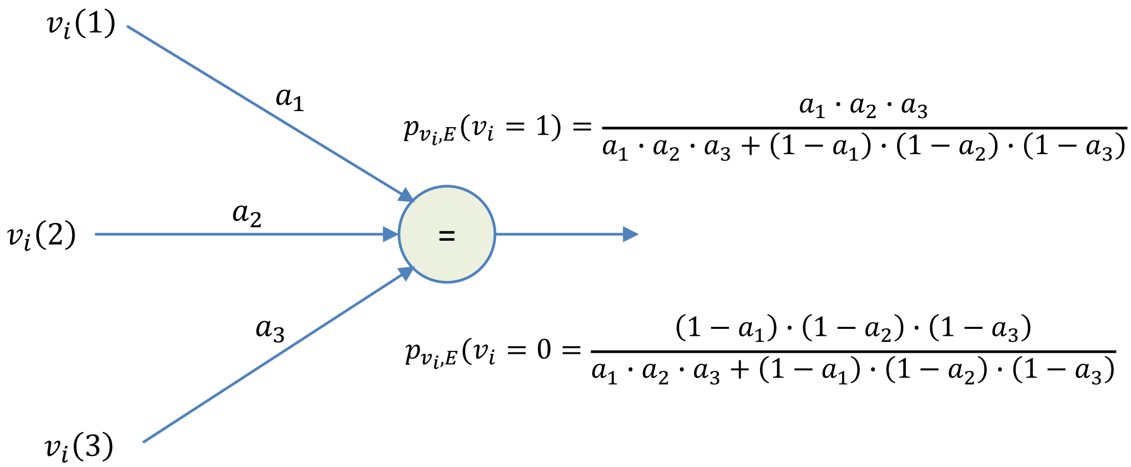

$v_i(1)$
a_1
$p_{v_i,E}(v_i = 1) = \frac{a_1 \cdot a_2 \cdot a_3}{a_1 \cdot a_2 \cdot a_3 + (1-a_1)\cdot(1-a_2)\cdot(1-a_3)}$
$v_i(2)$
a_2
$=$
a_3
$p_{v_i,E}(v_i = 0 = \frac{(1-a_1)\cdot(1-a_2)\cdot(1-a_3)}{a_1 \cdot a_2 \cdot a_3 + (1-a_1)\cdot(1-a_2)\cdot(1-a_3)}$
$v_i(3)$

Figure 7.21: Graph of equality constraint.

Figure 7.21 illustrates these 3 instances of the same bit. Constraint satisfaction means all 3 are the same. The probability that all 3 instances are the same has two possibilities, $S_E = \{(000), (111)\}$:

$$p_{v_i,E} = \begin{cases} c_i' \cdot a_1 \cdot a_2 \cdot a_3 & v_i = 1 \\ c_i' \cdot (1-a_1)(1-a_2)(1-a_3) & v_i = 0 \end{cases} , \tag{7.147}$$

where

$$c_i' = \frac{1}{a_1 \cdot a_2 \cdot a_3 + (1-a_1)(1-a_2)(1-a_3)} . \tag{7.148}$$

The equality constraint accepts extrinsic probabilities from the "other" 2 bits (coming from different parity check calculations) and then returns an intrinsic probability to its own parity-check constraint. For bit instance 2, the extrinsic probability calculated from the other two parity check constraints' results is

$$p_{ext}(v_2/y_2) = \begin{cases} c_i \cdot a_1 a_3 & v_i(2) = 1 \\ c_i \cdot (1-a_1)(1-a_3) & v_i(2) = 0 \end{cases} , \tag{7.149}$$

where

$$c_i = \frac{1}{a_1 \cdot a_3 + (1-a_1)(1-a_3)} . \tag{7.150}$$

This then determines the intrinsic information returned to parity-check 2. Similar expressions hold for the extrinsic probabilities returned to parity checks 1 and 3.

Figure 7.22's flow diagram illustrates the calculations and flows for the simpler equality constraint in that LLR values simply are added for the "other" bits.

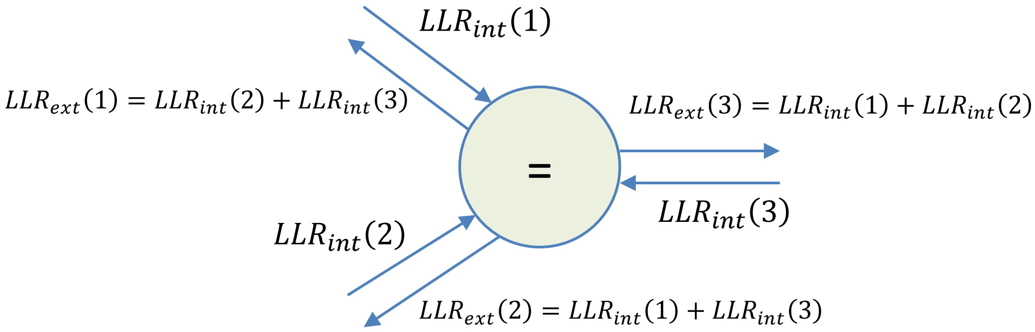

$LLR_{int}(1)$
$LLR_{ext}(1) = LLR_{int}(2) + LLR_{int}(3)$
$LLR_{ext}(3) = LLR_{int}(1) + LLR_{int}(2)$
=
$LLR_{int}(3)$
$LLR_{int}(2)$
$LLR_{ext}(2) = LLR_{int}(1) + LLR_{int}(3)$

Figure 7.22: Equality constraint soft-information flow.

7.4.2.1 Equality-Constraint Implementation

Equation (7.154) essentially states 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 LLRs as

$$\text{LLR}(j) = \sum_{\substack{i=1 \\ i \neq j}}^{t_c} \text{LLR}(i) \tag{7.151}$$

and the extrinsic calculation simply sums all terms except the j^{th}. Thus, as in Example 7.4.2's Figure 7.22, 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.

repetition code: A simple repetition code of rate $1/t_c$ is another equality constraint directly that applies to each of the repeated bits. $S_E = \{000000....0,\ 111111....1\}$. The joint probability of the event and the bit is thus

$$p_{v_i,E} = c_k \sum_{\boldsymbol{v}_i \in S_E} \prod_{j=1}^{t_c} p_j(v_i) \quad , \tag{7.152}$$

where

$$c_i = \left[\left\{ \prod_{j=1}^{t_c} p_j(v_i) \right\} + \prod_{j=1}^{t_c} \left(1 - p_j(v_i)\right) \right]^{-1} \quad . \tag{7.153}$$

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

$$p_{ext}(v_i(j)) = c_i' \cdot \sum_{\boldsymbol{v}_i \in S_E} \prod_{\substack{j=1 \\ j \neq i}}^{t_c} p_j(v_i) \quad , \tag{7.154}$$

where

$$c_i' = \left[\left\{ \prod_{\substack{j=1 \\ j \neq i}}^{t_c} p_j(v_i) \right\} + \prod_{\substack{j=1 \\ j \neq i}}^{t_c} \left(1 - p_j(v_i)\right) \right]^{-1} \quad . \tag{7.155}$$

7.4.3 Constellation bit constraints

A decoder can compute a likelihood function for each bit of a constellation's labels.

EXAMPLE 7.4.3 *[single-dimension bit-level likelihoods]* An example for one dimension of a 64 QAM constellation and rate 2/3 code helps understanding. Figure 7.23 shows one Gray-code labelled constellation dimension and a received value of $y = -5.5$.

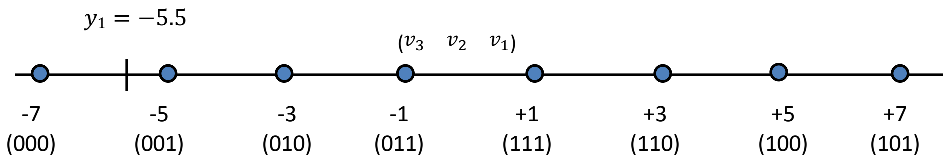


Figure 7.23: Example for 1-dimensional likelihood computation.

The received value of -5.5 in the first dimension corresponds to the 3 bits (v_3, v_2, v_1). There are 4 constellation points that correspond to each of the bits independently being 1 or 0. Thus, the likelihood for v_3 is

$$\begin{aligned}
p(y_1 = -5.5, v_3 = 0) &= c_1 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(.5)^2} + e^{-\frac{1}{2\sigma^2}(1.5)^2} + e^{-\frac{1}{2\sigma^2}(2.5)^2} + e^{-\frac{1}{2\sigma^2}(4.5)^2} \right) \cdot (1 - p_3) \\
p(y_1 = -5.5, v_3 = 1) &= c_1 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(6.5)^2} + e^{-\frac{1}{2\sigma^2}(8.5)^2} + e^{-\frac{1}{2\sigma^2}(10.5)^2} + e^{-\frac{1}{2\sigma^2}(12.5)^2} \right) \cdot p_3 \quad ,
\end{aligned}$$

while the likelihood for v_2 is

$$\begin{aligned}
p(y_1 = -5.5, v_2 = 0) &= c_2 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(.5)^2} + e^{-\frac{1}{2\sigma^2}(1.5)^2} + e^{-\frac{1}{2\sigma^2}(10.5)^2} + e^{-\frac{1}{2\sigma^2}(12.5)^2} \right) \cdot (1 - p_2) \\
p(y_1 = -5.5, v_2 = 1) &= c_2 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(2.5)^2} + e^{-\frac{1}{2\sigma^2}(4.5)^2} + e^{-\frac{1}{2\sigma^2}(6.5)^2} + e^{-\frac{1}{2\sigma^2}(8.5)^2} \right) \cdot p_2 \quad ,
\end{aligned}$$

and finally

$$\begin{aligned}
p(y_1 = -5.5, v_1 = 0) &= c_3 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(1.5)^2} + e^{-\frac{1}{2\sigma^2}(2.5)^2} + e^{-\frac{1}{2\sigma^2}(8.5)^2} + e^{-\frac{1}{2\sigma^2}(10.5)^2} \right) \cdot (1-p_1) \\
p(y_1 = -5.5, v_1 = 1) &= c_3 \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \left(e^{-\frac{1}{2\sigma^2}(.5)^2} + e^{-\frac{1}{2\sigma^2}(4.5)^2} + e^{-\frac{1}{2\sigma^2}(6.5)^2} + e^{-\frac{1}{2\sigma^2}(12.5)^2} \right) \cdot p_1 \quad .
\end{aligned}$$

Any of these could serve as à prior distributions for other codes' subseqent LL computations.

AWGN likelihood values 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. Furthermore, this dominant term only computes the squared difference from the closest constellation point for each input bit value, with normalization by $2\sigma^2$.

EXAMPLE 7.4.4 (2D example) Likelihoods' calculation for each bit in non-square constellations can also be computed as in 8SQ constellation in Figure 7.24, although such constellation mappings do not compliment well codes in BICM.

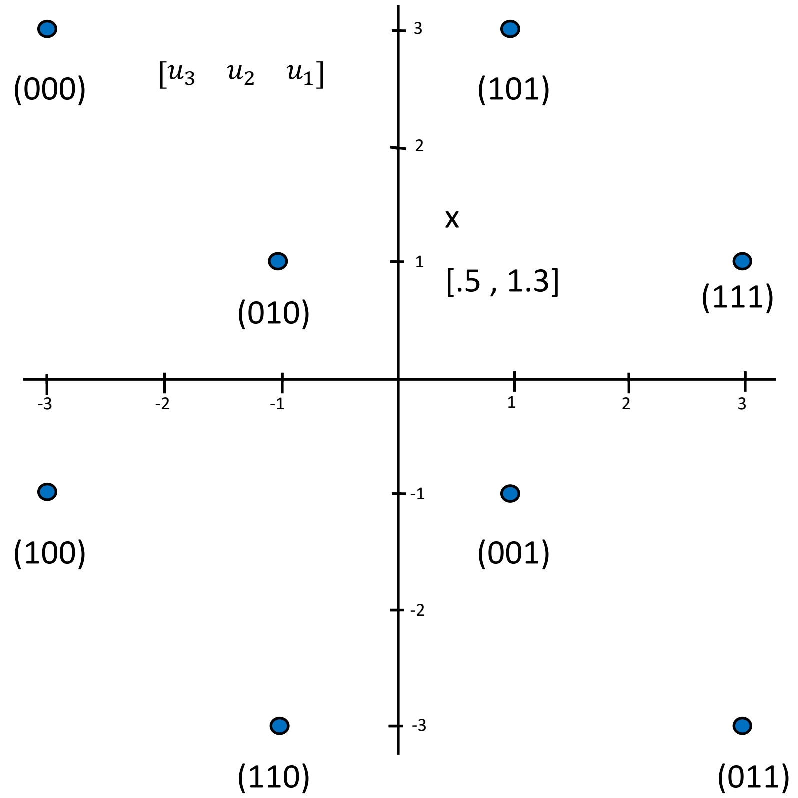


Figure 7.24: Example for 2-dimensional likelihood computation.

The two values of the likelihood for a $\boldsymbol{y} = [.5, 1.3]$ are:

$$\begin{aligned}
p(u_1 = 0) &= c_1 \cdot \frac{1}{2\pi\sigma^2} \left(e^{-\frac{(1.5)^2+(.3)^2}{2\sigma^2}} + e^{-\frac{(3.5)^2+(1.7)^2}{2\sigma^2}} + e^{-\frac{(.5)^2+(2.3)^2}{2\sigma^2}} + e^{-\frac{(2.5)^2+(4.3)^2}{2\sigma^2}} \right) \cdot (1-p_1) \\
p(u_1 = 1) &= c_1 \cdot \frac{1}{2\pi\sigma^2} \left(e^{-\frac{(.5)^2+(1.7)^2}{2\sigma^2}} + e^{-\frac{(2.5)^2+(.3)^2}{2\sigma^2}} + e^{-\frac{(3.5)^2+(2.3)^2}{2\sigma^2}} + e^{-\frac{(1.5)^2+(4.3)^2}{2\sigma^2}} \right) \cdot p_1 \quad .
\end{aligned}$$

For this example the computation of the LLRs yields:

$$\begin{aligned} \text{LLR}(u_3) &= -3.4 & (7.156)\\ \text{LLR}(u_2) &= -.545 & (7.157)\\ \text{LLR}(u_1) &= -.146 \quad . & (7.158) \end{aligned}$$

The MAP decision is $(u_3, u_2, u_1) =$ (0,0,0) not the point (0,1,0) that would result from simpler maximum likelihood symbol detection, which instead selects the closest point or (0 1 0). 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. Usually SQ QAM constellations with even power of 2 number of points are used with BICM and Gray coding, so to preserve the binary code's gain.

7.4.4 Soft intersymbol-interference cancellation

ISI constraints can directly produce soft information through SOVA's processing of ISI (like with partial response) on the trellis or APP as in Section 7.3. This subsection provides a considerably simpler alternative known as the **soft canceler**.

Soft cancellers handle ISI and approximate ML or MAP detection for each input subsymbol or bit. The soft-canceller specifically approximates the à posteriori probabilities

$$p_{\boldsymbol{x}_i/\boldsymbol{y}_u} = \frac{p_{\boldsymbol{y}_u/\boldsymbol{x}_i} \cdot p_{\boldsymbol{x}_i}}{p_{\boldsymbol{y}_u}} \quad , \tag{7.159}$$

iteratively, where u is a time index or possible a user index when there is crosstalk.

(7.159)'s independence of $\boldsymbol{x}$ allows direct use of the likelihood function $p_{\boldsymbol{y}/x_i}$ instead of the à posteriori function when the initial input distribution for x_u is uniform; when soft extrinsic information from a code is provided, then the intrinsic non-uniform distribution incorporates also the $p_{\boldsymbol{x}_i}$ distribution. Often the log-likelihood function $L_{\boldsymbol{y}_u/\boldsymbol{x}_i} = \log(p_{\boldsymbol{y}_u/\boldsymbol{x}_i})$ is directly propagated.

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 subsymbol $\boldsymbol{x}_u$. $\boldsymbol{a}_u$ is a vector when $\boldsymbol{x}_u$ is a vector.

The objective maximizes the probability over $\boldsymbol{a}_u$, or

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

or with likelihoods

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

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).

An AWGN's extrinsic probability distribution depends on an estimate of its mean and variance: Previous estimates of the mean vector $\xi_u = \mathbb{E}[\boldsymbol{x}_i/\boldsymbol{y}_u]$ and the autocorrelation $R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(u) = \mathbb{E}[(\boldsymbol{x}_i - \xi_i)(\boldsymbol{x}_i - \xi_i)^*/\boldsymbol{y}_u]$ are known from training or previous soft-canceller iterations.

To compute the extrinsic probability distribution for subsymbol index i, a noise estimate is

$$w_{u,i}(\boldsymbol{a}_i) = \left(\boldsymbol{y}_u - \sum_{j \neq i} H_{uj} \cdot \xi_j \right) - H_{ui} \cdot \boldsymbol{a}_i \tag{7.160}$$

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 estimated noise autocorrelation matrix is

$$R_{\boldsymbol{ww}}(u,i) = R_{nn}(u) + \sum_{j \neq i} H_{uj} \cdot R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(j) \cdot H_{uj}^* \quad . \tag{7.161}$$

The converged soft canceller's goal is that $R_{\boldsymbol{ww}}(u,i)$ approaches the noise autocorrelation and that $R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(u) \to 0$, leaving $\hat{\boldsymbol{x}}_i = \boldsymbol{x}_i$.

A new extrinsic probability distribution is

$$p_{\boldsymbol{y}_u/\boldsymbol{x}_i}(\boldsymbol{a}_i) = \frac{1}{\pi |R_{\boldsymbol{ww}}(u,i)|} \cdot e^{-(w_{u,i}(\boldsymbol{a}_i))^* R^{-1}_{\boldsymbol{ww}}(u,i)(w_{u,i}(\boldsymbol{a}_i))} \quad . \tag{7.162}$$

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:

$$\xi_i = \frac{\sum_{\boldsymbol{a}_i} \boldsymbol{a}_i \cdot p_{\boldsymbol{y}_u/\boldsymbol{x}_i}(\boldsymbol{a}_i)}{\sum_{\boldsymbol{a}_i} p_{\boldsymbol{y}_u/\boldsymbol{x}_i}(\boldsymbol{a}_i)} \tag{7.163}$$

$$R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(i) = \frac{\sum_{\boldsymbol{a}_i} (\boldsymbol{a}_i - \xi_i)(\boldsymbol{a}_i - \xi_i)^* \cdot p_{\boldsymbol{y}_u/\boldsymbol{x}_i}(\boldsymbol{a}_i)}{\sum_{\boldsymbol{a}_i} p_{\boldsymbol{y}_u/\boldsymbol{x}_i}(\boldsymbol{a}_i)} \tag{7.164}$$

Figure 7.25 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.

7.4.4.1 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

$$\boldsymbol{x_i} = H^+ \cdot \boldsymbol{y}_u \quad . \tag{7.165}$$

Zeroed initial values may be easier to compute, but can increase the convergence time of the algorithm in Figure 7.25. 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

$$R_{\hat{\boldsymbol{x}}\hat{\boldsymbol{x}}}(u) = H^+ \cdot R_{nn}(u) \cdot \left[H^+\right]^* \quad . \tag{7.166}$$

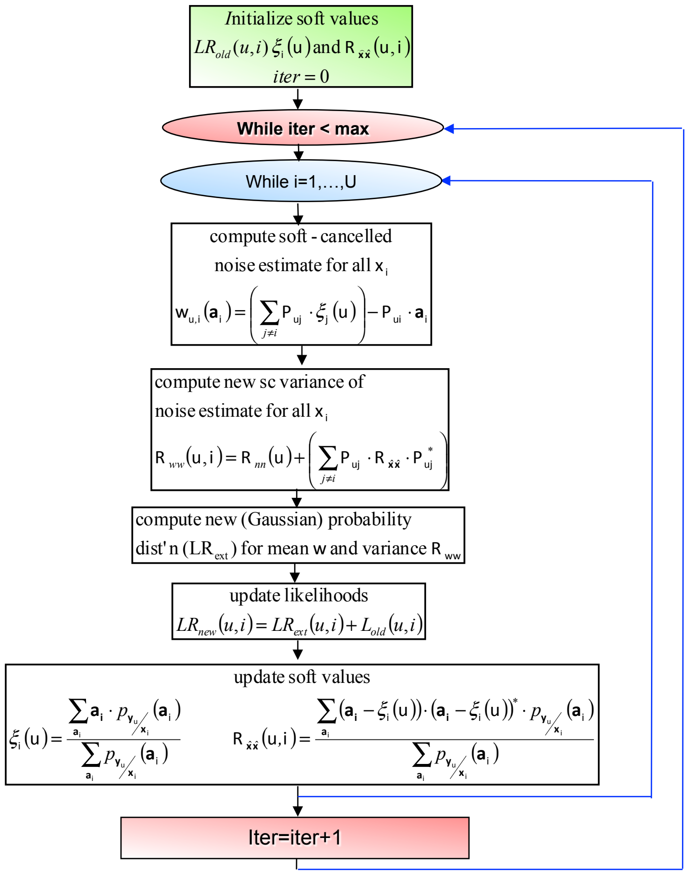
For channel information, one easily determines for the BSC input that[15]

$$\text{LLR}_k = \begin{cases} \ln \frac{1-p}{p} & \text{if } v_k = 1 \\ \ln \frac{p}{1-p} & \text{if } v_k = 0 \end{cases} \tag{7.167}$$

and for the AWGN output with binary PAM input that

$$\text{LLR}_k = \frac{2}{\sigma^2} \cdot y_k \quad , \tag{7.168}$$

[15] 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 $\Lambda = \ln \frac{1-p}{p}$.



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 likelihoods
$LR_{new}(u,i) = LR_{ext}(u,i) + L_{old}(u,i)$
update soft values
$\xi_i(u) = \frac{\sum_{\mathbf{a}_i} \mathbf{a}_i \cdot p_{\mathbf{y}_u/\mathbf{x}_i}(\mathbf{a}_i)}{\sum_{\mathbf{a}_i} p_{\mathbf{y}_u/\mathbf{x}_i}(\mathbf{a}_i)}$
$R_{\hat{x}\hat{x}}(u,i) = \frac{\sum_{\mathbf{a}_i} (\mathbf{a}_i - \xi_i(u)) \cdot (\mathbf{a}_i - \xi_i(u))^* \cdot p_{\mathbf{y}_u/\mathbf{x}_i}(\mathbf{a}_i)}{\sum_{\mathbf{a}_i} p_{\mathbf{y}_u/\mathbf{x}_i}(\mathbf{a}_i)}$
Iter=iter+1

Figure 7.25: Soft Cancelation Flow Chart.

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 7.4.5 *[Continuation of Example 7.4.3]* For the previous Example 7.4.3, the lLOGMAX has dominant terms:

$$\begin{aligned}
\text{LLR}(u_3) &= \ln\left[e^{-\frac{1}{2\sigma^2}[(6.5)^2-(.5)^2]}\right] = -\frac{1}{2\sigma^2}[42] \;\; \text{favors } 0 && (7.169)\\
\text{LLR}(u_2) &= \ln\left[e^{-\frac{1}{2\sigma^2}[(2.5)^2-(.5)^2]}\right] = -\frac{1}{2\sigma^2}[6] \;\; \text{favors } 0 && (7.170)\\
\text{LLR}(u_1) &= \ln\left[e^{-\frac{1}{2\sigma^2}[(.5)^2-(1.5)^2]}\right] = \frac{1}{\sigma^2} \;\; \text{favors } 1. && (7.171)
\end{aligned}$$

The LOGMAX *LLR* esimate simplifes to

$$\text{LLR} = \frac{y}{\sigma^2}\,(b-a) + \frac{a^2-b^2}{2\sigma^2} \qquad (7.172)$$

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

$$\text{LLR} = \Re\frac{\{\boldsymbol{y}^*(\boldsymbol{b}-\boldsymbol{a})\}}{\sigma^2} + \frac{\|\boldsymbol{a}\|^2 - \|\boldsymbol{b}\|^2}{2\sigma^2} \qquad (7.173)$$

for complex vectors.

7.5 Iterative Decoding

Sections 7.3 and 7.4 describe soft-information generation from a specific decoder or constraint. This soft information by itself allows MAP symbol- or bit-error minimization, but has greater significance for other decoders. Iterative decoding passes soft information from one decoder to another, hopefully improveing LLR accuracy.

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 7.3 - 7.5 along with intrinsic given or channel soft information in successive attempts to refine the estimate of the transmitted bit-or-symbol probability distribution (or LLR). Concatenated coding systems use more than one code or constraint and exploit soft information from each constraint or code to improve the others' decoding. As in Sections 7.3 and 7.4, soft information from a first decoder/constraint that will be sent to another decoder/constaint is extrinsic information. 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. An iterative-decoding process between two or more constraints passes extrinsic information to the other decoders, Each decoder retains and uses its own intrinsic information, which typically arises from a specific subsymbol channel-output sampling time-k. The decoders cyclically exchange extrinsic information, combining the incoming extrinsic information with the retained intrinsic information.. The resultant cycle approximates an optimum decoder, but with much less complexity. Chapter 8's concatenated and long-block-length codes anticipate iterative decoding's use. The approximation usually allows reliable transmission at data rates very close to capacity, essentially canonical performance.

The APP decomposition that enables the iterative-decoding cycle is:

$$p_{\boldsymbol{x}_k/\boldsymbol{Y}_{0:K-1}} \propto p_{\boldsymbol{x}_k,\boldsymbol{Y}_{0:K-1}} = \underbrace{\underbrace{p_{\boldsymbol{x}_k}}_{\text{à priori}} \cdot \underbrace{p_{\boldsymbol{y}_k/\boldsymbol{x}_k}}_{\text{channel}}}_{\text{intrinsic}} \cdot \underbrace{p_{\boldsymbol{Y}_{0:k-1},\boldsymbol{Y}_{k+1:K-1}/\boldsymbol{x}_k,\boldsymbol{y}_k}}_{\text{extrinsic}} \quad . \tag{7.174}$$

The proportionality on the left in (7.174) simply notes that the normalization by the probability distribution $p_{\boldsymbol{Y}_{0:K-1}}$ has no influence upon a final decision for $\boldsymbol{x}_k$. A first decoder retains the first 2 terms locally for the intrinsic information. Iterative decoding's initial à prior distribution is usually uniform. The 3rd term is the extrinsic information that finds both local use and also exported use at (an)other (decoder)s. Those other decoders may return extrinsic information that then replaces the à priori information at the first decoder. That first decoder then may consequently update its extrinsic information for another export to other decoders. When there are more than 2 codes/decoders, the design follows an extrinsic information-passing schedule. The "channel" information itself arises from a memoryless processing of the time-k channel output data. This "symbol-by-symbol" detection, while sometimes insufficient by itself, nonetheless often provides strong indication of the most likely transmitted subsymbol, as in Subsection 7.4.3). This channel information remains the same on each iterative decoding cycle. Extrinsic information does not include the channel information to avoid an consequent bias that would accumulate with it on each cycle.

Often the two decoders order differently the bits/symbols through Chapter 8's interleaving. This interleaving redistributes the location of large channel noise/errors or "bursts," thus allowing better soft information from constraints less affected by bursts to be shared with those heavily affected by bursts. This process also helps make independent BICM's mapping of bits to/from constellations that have nonzero redundancy $\rho > 0$ in $|C| = 2^{b+\rho}$..

Log Likelihood Implementation: It is again convenient to use the log-likelihood function instead of the probability function, so then (7.174) becomes

$$\begin{align} LL_{\boldsymbol{x}_k,\boldsymbol{Y}_{0:K-1}} &= LL_{\boldsymbol{x}_k} + LL_{\boldsymbol{y}_k/\boldsymbol{x}_k} + LL_{\boldsymbol{Y}_{0:k-1},\boldsymbol{Y}_{k+1:K-1}/\boldsymbol{x}_k,\boldsymbol{y}_k} \tag{7.175} \\ &= \underbrace{LL_{\text{à priori}} + LL_{\text{channel}}}_{\text{bias accumulation risk}} + LL_{\text{extrinsic}} \quad . \tag{7.176} \end{align}$$

If a decoder computes $LL_k \stackrel{\Delta}{=} LL_{\boldsymbol{x}_k,\boldsymbol{Y}_{0:K-1}}$ further calculations subtract $LL_{\text{channel},k}$ and $L_{\text{à priori},k}$ to produce the extrinsic information for another decoder. Decoder implementation often replaces the LL_k by LLR_k for bit decoding. Section 7.4's constraint events directly compute this extrinsic information from their associated constraint. Equation (7.176) illustrates potential bias accumulation. This potentially unstable feedback bias mechanism may not be so clear in constraint-based code designs, which nevertheless need to account for it to profit best from iterative decoding. Chapter 8's LDPC codes, whose preferred decoders pass soft-information between constraints, avoid such short soft-information-message-passing cycles that would allow soft-information's bias accumulation.

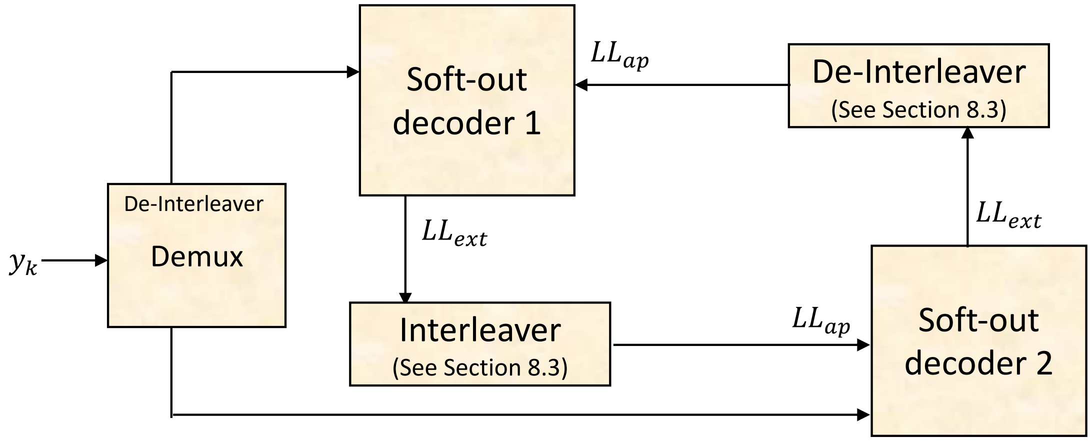


Figure 7.26: Iterative Decoding two codes.

Iterative decoding flow: Figure 7.26 illustrates iterative decoding for two interleaved codes. Chapter 8, Section 3 further details interleaving. The extrinsic likelihood from one decoder, LL_{ext} becomes the à priori input distribution for the other decoder. The channel information remains local to each decoder for the current subsymbol time instant. This time-instant almost always differs with interleaving. The decoders usually correspond to two independent codes, in which case the system uses "turbo coding" (really should be "turbo decoding," see Chapter 8, Section 3.2) in an amusing analogy with automobile's "turbo" engine[16]. Figure 7.26 could also add a 3rd decoder to the cycle, or even more; this then tends towards codes based on event constraints. Thus, an alternate view has an equality constraint on all the commonly encoded bits for Figure 7.26's two codes. One decoder could also be an MLSD decoder for intersymbol interference or a soft canceller. A decoder can be as simple as the demapping of BICM's Gray Coded constellation points (which indeed is a code that can use/approximate MAP decoders, albeit those decoders perhaps conceptually simple). The two codes could be actually those of two different users sharing the same channel, and it happens that their signals "crosstalk" into one another, as in Chapters 2 and 5.

Either decoder also produces an estimate of each and every symbol value $\boldsymbol{x}_i$, which is found as that symbol value at time k, $k = 0, ...K-1$ that maximizes the likelihood function values $L_{\boldsymbol{x}_k/\boldsymbol{Y}_{0:K-1}}$, or equivalently maximizes $L_{\boldsymbol{x}_k,\boldsymbol{Y}_{0:K-1}}$. 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 to converge to a common final decision. The following Theorem helps understand convergence:

Theorem 7.5.1 [Iterative-Decoding Convergence] *Iterative decoding converges to the exact probability densities desired if the constraints/decoders are such that no information from one constraint/decoder can return to that constraint – that is the number of iterations is smaller than the length of a message-passing "cycle" among the constraints/decoders.*

[16] It seems these codes' inventors pursue more exciting extracurricular activities than the gardening performed by their more tranquil predecessors in Chapter 3's RAKE receivers.

Proof: The proof follows directly from Equation 7.174, which is exact. If each term is exact, then the entire equation is also exact. The first two right-hand terms on the right are trivially exact. The extrinsic term will be exact if no approximations are used in its calculation. The only approximation that could be made in its calculation elsewhere is the use of other approximated extrinsic terms. The only way for these 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 7.5.1 is both profound and theoretically trivial. The very condition of "no cycles" prevents the nature of iterative decoding, because averting cycles altogether is difficult in practice. Such aversion creates some very difficult computations that are essentially equivalent to executing a full ML decoder. Thus, the convergence condition usually won't occur exactly in most (or almost all) uses. Nonetheless, its approximate satisfaction often enables convergence. Such decisions ultimately are almost always the same as the more complicated direct ML decoder. Figure 7.26 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 quickly and nearly exactly.

7.5.1 Iterative Decoding with the APP (BCJR) algorithm

Section 7.3.1's APP algorithm computes the quantity

$$\gamma_k(i,j) = p(s_{k+1}=j, \boldsymbol{y}_k / s_k = i) = p_k(i,j) \cdot \sum_{\boldsymbol{x}'_k} p_{\boldsymbol{y}_k/\boldsymbol{x}'_k} \cdot q_k(i,j,\boldsymbol{x}'_k) \quad , \tag{7.177}$$

where $p_k(i,j) = p(s_{k+1} = j/s_k = i)$, allows decoders to transfer information. 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, BCJR/APP decisions are upon the product of α, β, and γ, see Section 7.3. Since parallel transitions can only be resolved by symbol-by-symbol decoding, one presumes that step is accomplished outside of iterative decoding and so (7.177) simplifies to

$$\gamma_k(i,j) = p_k(i,j) \cdot p_{\boldsymbol{x}_k} \cdot p_{\boldsymbol{y}_k/\boldsymbol{x}_k} \tag{7.178}$$

where iterative decoding incorporates another decoders extrinsic information through the $p_k(i,j) \cdot p_{\boldsymbol{x}_k}$ component terms. Thus,

$$\ln\left(\gamma_k(i,j)\right) = LL_{ext}(\text{old}) + LL_{\boldsymbol{y}_k/\boldsymbol{x}_k} \quad . \tag{7.179}$$

or

$$\gamma_k(i,j) = e^{LL_{ext}(\text{old})} \cdot e^{LL_{\boldsymbol{y}/\boldsymbol{x}}} \quad . \tag{7.180}$$

The local decoder updates its α and β quantities rom the latest set of $\gamma_k(i,j)$ for each successive APP-decoding cycle with initial condition usually a uniform distribution on $p_{\boldsymbol{x}_k}$. Similarly, the new extrinsic information for another decoder subtracts $LL_{\boldsymbol{y}_k/\boldsymbol{x}_k}$ and the earlier provided extrinsic information before new export as:

$$LL_{ext}(\text{new}) = L_{\boldsymbol{x}_k/\boldsymbol{Y}_{0:K-1}} - L_{ap} - L_{\boldsymbol{y}_k/\boldsymbol{x}_k} \tag{7.181}$$

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 $L_{ext}(old)$ for the next decoder.

7.5.2 Iterative Decoding with SOVA

SOVA's soft information (Section 7.4) is somewhat ad-hoc and does not exactly correspond to $p_{\boldsymbol{x}_k}$. Nonetheless, larger values of $LL_{\boldsymbol{x}^*_k,\boldsymbol{x}'_k}$ imply more $\boldsymbol{x}^*_k$ is more likely than $\boldsymbol{x}'_k$. In fact the likelihood of any pair of $\boldsymbol{x}_k$ values, say $\boldsymbol{x}^{(1)}_k$ and $\boldsymbol{x}^{(2)}_k$ can be computed by subtracting corresponding values of ΔLL_k as

$$\Delta LL_{\boldsymbol{x}^{(1)}_k,\boldsymbol{x}^{(2)}_k} = \Delta LL_{\boldsymbol{x}^*_k,\boldsymbol{x}^{(1)}_k} - \Delta LL_{\boldsymbol{x}^*_k,\boldsymbol{x}^{(2)}_k} \quad . \tag{7.182}$$

SOVA uses values of ΔLL_k for all time k paths, particularly in the forward-backward recursion. The new log likelihoods allow an update of the VA and consequent new LL values, from which the current branch LL subtracts before export as extrinsic to another decoder. Then (**??**)'s internal information update for each branch comparison becomes a function with respect to other values $\boldsymbol{x}_k^{(m)}$:

$$LL_{\boldsymbol{x}_k^{(m)},\boldsymbol{y}_k}(\boldsymbol{x}_k^{(m)}) \leftarrow LL_{\boldsymbol{y}_k/\boldsymbol{x}_k^{(m)}} + \Delta LL_{\boldsymbol{x}_k,\boldsymbol{x}_k^{(m)}} \;\; m = 0, ..., M_k - 1 \;\; , \;\; \boldsymbol{x}_k^{(m)} \neq \boldsymbol{x}_k \qquad (7.183)$$

where the reference is this path's $\boldsymbol{x}_k$ value selected in the previous decoder's SOVA use. These new extrinsic-information-including branch likelihoods are added to previous state likelihoods and the SOVA proceeds as in Section 7.4.

7.5.3 Direct use of constraints in iterative decoding

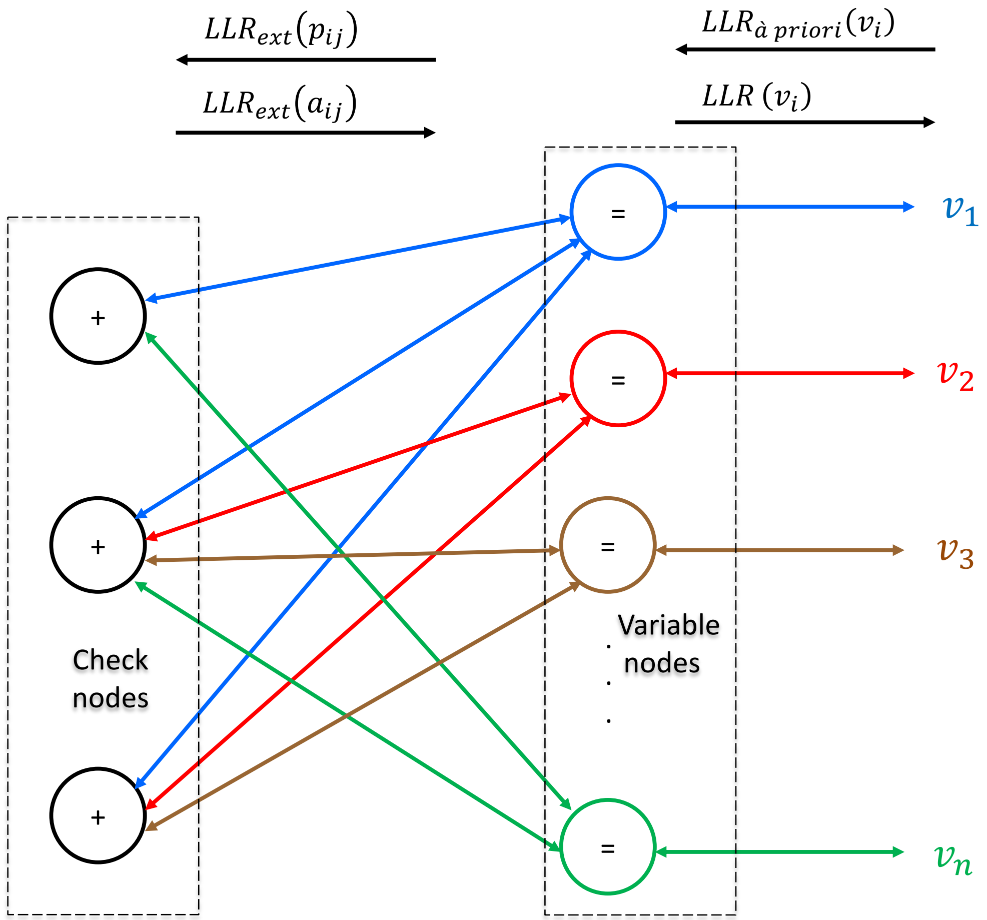


Figure 7.27: Tanner Graph: Direct use of constraints in Iterative Decoding.

Figure 7.27 illustrates the direct use of constraints, parity and equality, in an iterative-decoding flow diagram. This is often called a Tanner Graph [6].[17] Messages are passed from the initial channel outputs as LLR_{ext}'s from right to left into the equality nodes. Subsequently, the equality nodes pass updated extrinsic information to each of the parity nodes (right to left), which then return soft information to

[17] After Dr. Michael Tanner (1956 -), an American information theorist, with PhD from Stanford University in 1971, and a Professor at UC Santa Cruz.

the equality nodes in a subsequent cycle. 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 8). Each of the nodes use Figures 7.19 and 7.21.

7.5.4 Turbo Equalization

Turbo equalization uses the soft-information from decoders in the MLSD or APP priors for an ISI decoder. Similarly, this information may export extrinsic information in the same way as if it were a code. When soft cancellers are used, the current branch metric LL must be subtracted as well as any previous extrinsic information.

Exercises - Chapter 7

7.1 *Error Events.*

For the trellis in Figure 7.4, 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?

7.2 *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?

7.3 *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

$$\boldsymbol{v}(D) = \boldsymbol{u}(D) \cdot \begin{bmatrix} 1 & D & 1+D \\ 0 & 1 & 1+D \end{bmatrix} \tag{7.184}$$

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.

7.4 *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.*

7.5 *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?

7.6 *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

7.7 *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 $-0.8D^2$?

7.8 *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.

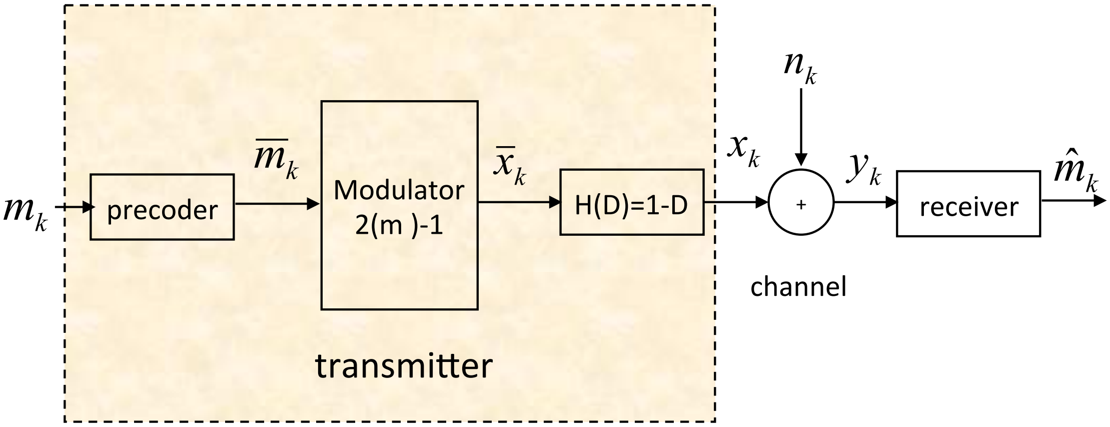


Figure 7.28: AMI transmission system.

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.

7.9 *Partial Response - Midterm 1997 11 pts* The partial-response channel shown in Figure 7.29 has the

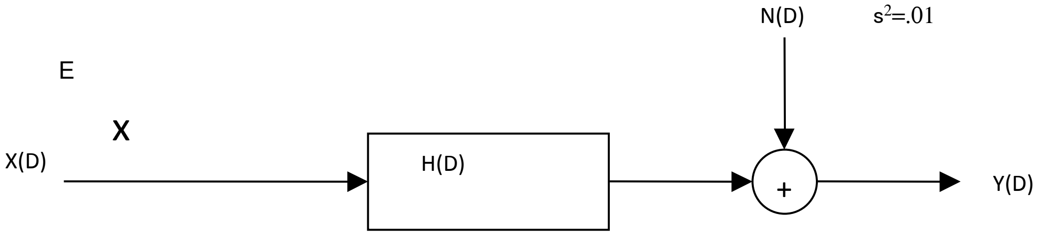


Figure 7.29: Partial-response channel.

frequency magnitude characteristic, $|H(e^{-j\omega T})|$, shown in Figure 7.30. 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)

7.10 *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.

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)

7.11 *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 7.29. (4 pts)

7.12 *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)

7.13 *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 7.3.3. 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 7.3.3. 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.

7.14 *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.

7.15 *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 $\bar{P}_b = 10^{-3}$.

7.16 *APP (5 pts)*

Repeat the APP example in Section 7.3 for the output sequence 01, 11, 10, 10, 00, 10. and BSC with $p = 0.2$.

7.17 *SOVA (5 pts)*

Repeat the SOVA example in Section 7.3 for the output sequence 01, 11, 10, 10, 00, 10. and BSC with $p = 0.2$.

7.18 *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).

7.19 *Soft-bit Induction Proof (6 pts)*
Prove Equation (7.136) via induction for a parity constraint with any number of terms.

7.20 *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)

7.21 *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)

7.22 *Constellation Constraint Processing (12 pts)*
For the constellations of Examples 7.4.3 and 7.4.4 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 7.4.3 for each of the 3 bits. (3 pts)

c. Find the LLR's if the received value is [-2.9,-1.1] for Example 7.4.4. 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)

7.23 *Simple Iterative Decoding Concept (9 pts)*
We begin using the constellation output for Problem 7.22 Part d's LLR's for each of the bits as à priori or intrinsic information to the parity constraint of Problem 7.21.

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)

7.24 *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)

7.25 *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)

7.26 *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

$$
\begin{aligned}
v_1 + v_2 + v_5 &= 0 && (7.185)\\
v_3 + v_4 + v_6 &= 0 && (7.186)\\
v_1 + v_3 + v_7 &= 0 && (7.187)\\
v_2 + v_4 + v_8 &= 0 && (7.188)
\end{aligned}
$$

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)

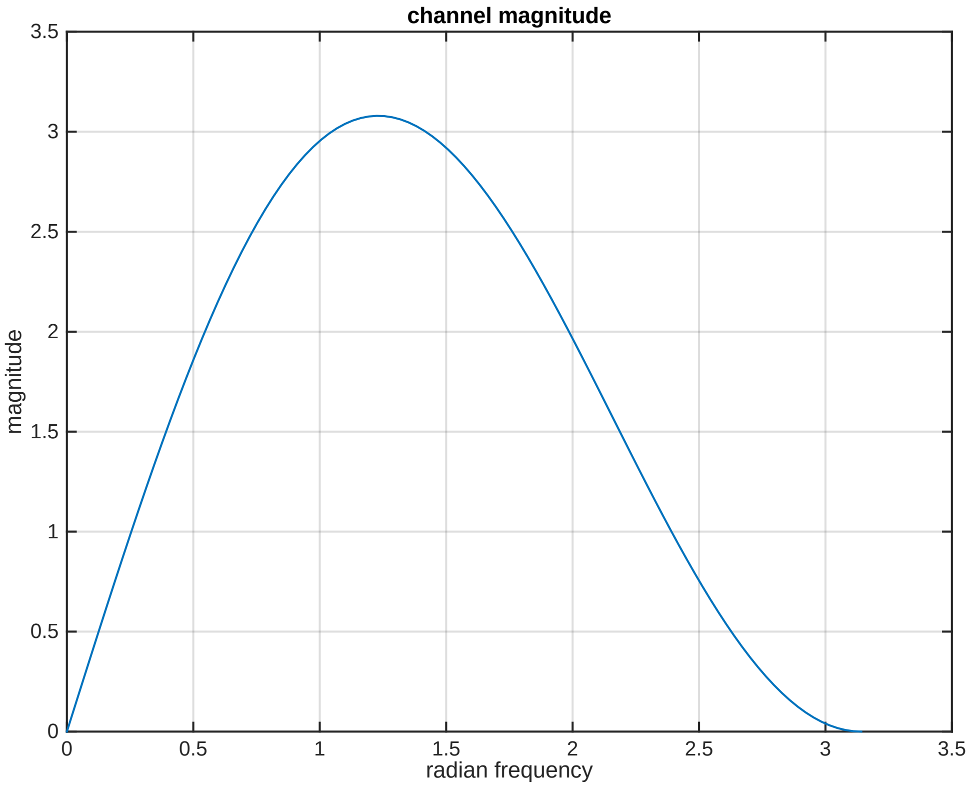


Figure 7.30: Plot of magnitude of $H(D)$ versus normalized frequency.

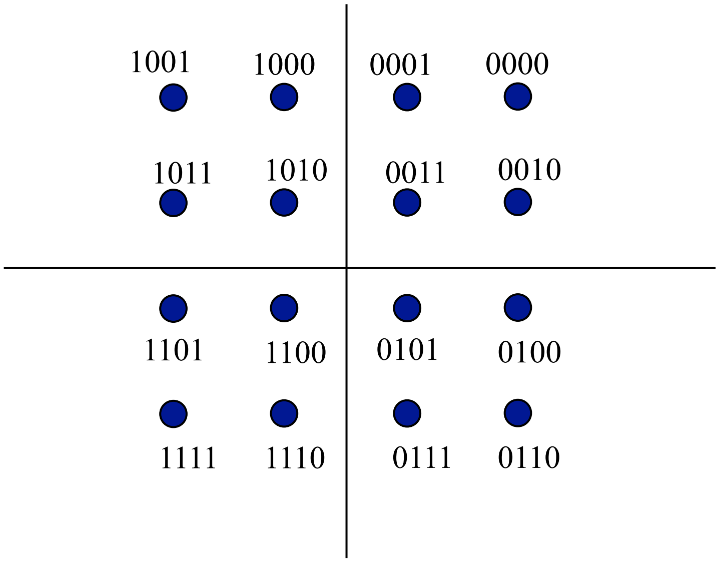


Figure 7.31: Constellation for Problem 7.26

Bibliography

[1] Andrew J. Viterbi *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm* IEEE Transactions on Information Theory. Vol. 13, No. 2, April 1967, pp. 260-269.

[2] G. David Forney, Jr. *Maximum-Likelihood SeauencecEstimation of Digital Sequences in the Presence of Intersymbol Interference* IEEE Transactions on Information Theory. Vol. 18, No. 5, May 1972, pp. 363-378.

[3] Vedat Eyuboğlu and Shahid U.H. Qureshi *Reduced-State Sequence Estimation with Set Partitioning and Decision Feedback* IEEE Transactions on Communications. Vol. 36, No. 1, January 1988, pp. 13-20.

[4] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv *Optimal Decoding of Linear Codes for Minimizing Symbol-Error Rate* IEEE Transactions on Information Theory. Vol. 20, No. 2, March 1974, pp. 284-287.

[5] Joachim Hagenauer and Peter Hocher *A Viterbi Algorithm with Soft-Decision Outputs and its Applications* Proceedings Globecom 1989 Dallas, TX. November 27-30, pp. 1680-1686.

[6] R. Michael Tanner *A Recursive Approach to Low Complexity Codes* IEEE Transactions on Information Theory. Vol 27, No. 5, September 1981, 533-547.

Index