STANFORD

Lecture 8

Decoding

February 1, 2024

John M. Cioffi

Hitachi Professor Emeritus (recalled) of Engineering

Instructor EE379A – Winter 2024

Announcements & Agenda

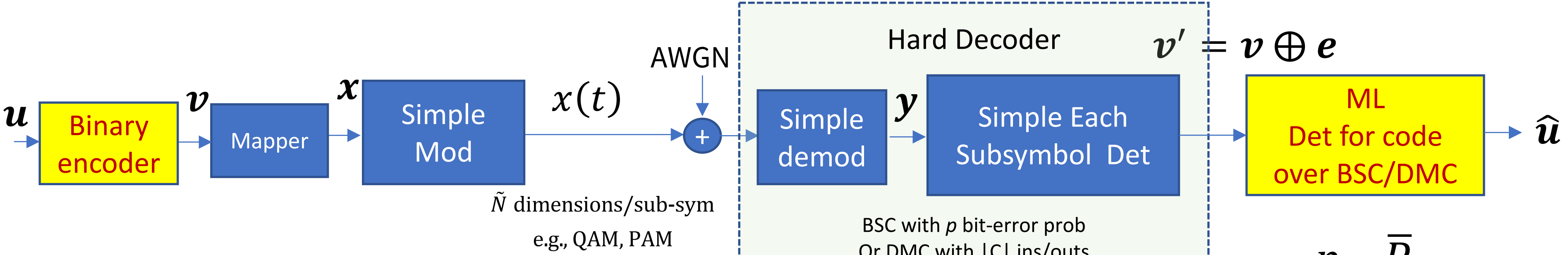
- Announcements
  - PS3 extended to Friday – see updated HWH3 at website.
  - PS4 is due TUESDAY, no late. (HWH4 is already at website.)
    - PS4 solutions will post immediately, and thus be available for your midterm study.
    - You should expect less time than PS2 or PS3.
  - There is no homework assigned next week.

- Today
  - Continue from L7
  - Viterbi Sequence Decoding (MLSD)
  - A Posteriori Probability (APP) bit decoding
  - Soft-Output Viterbi Algorithm (SOVA)
  - Backup – not covered: Invariant Factors Decomposition
    - Minimal Generators (and thus minimal decoder complexity)
    - Matlab code-structure error warning

Continue L7

Section 7.2

Hard decoder – first decodes the “v” bit sequence



$$p = \bar{P}_b$$

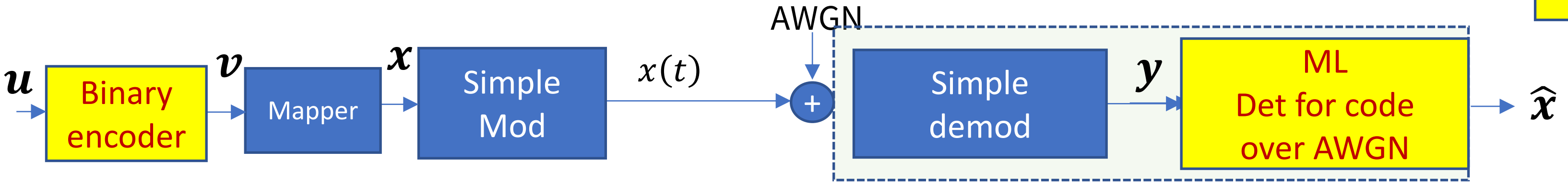
$$p_{v'/v} = \begin{cases} 1 - p & v' = v \\ p & v' \neq v \end{cases}$$

- Encoder-output subsymbols are decoded independently – e.g., a “hard” decision.
- The remaining channel is a BSC model, to which the binary code applies.
- The BEC with the “erasure” output is a first step from hard to soft….

Soft Decoder – decode the symbol

Finite Real-Time Complexity

$$\underbrace{p_{y/x} = (2\pi\sigma^2)^{-N/2} \cdot e^{-\|\boldsymbol{y}-\boldsymbol{x}\|^2/2\sigma^2}}$$



$\tilde{N}$ dimensions/sub-sym

e.g., QAM, PAM

$$\boxed{\hat{\boldsymbol{x}} = \arg\left\{\min_{\hat{\boldsymbol{x}}} \sum_{n=1}^{\bar{N}} \|\tilde{\boldsymbol{y}}_n - \hat{\tilde{\boldsymbol{x}}}_n\|^2\right\}}$$

- The demodulator samples ($\in \mathbb{C}$) pass to the detector for comparison of codewords (subsymbol sequences).
- The $\boldsymbol{y}$ information is "soft" in that it is not pre-quantized into a decision (or at least not to $|C|$ **subsymbol values).**
- Deployed systems often have ADC on y_n ; quantize $\frac{d_{min}(|C|)}{\sigma_q} = 4^3$; i.e., 3 bits cover intra-point distance.
  - This 3-bit quantization **of dmin** limits decoder loss (w.r.t. infinite precision) to .25 dB distortion (one more bit reduces to .06 dB distortion).
  - Same rule applies per dimension for both ADCs in quadrature receivers.
  - Total ADC bits will then be these 3, plus $\bar{b}$, plus 1-2 bits for peak-to-average (analog coverage), so $b_{ADC} = \bar{b}$ +4, or possibly $\bar{b}$ +5 .

AWGN Error Probability for Conv Codes

- AWGN $\bar{P}_e = \bar{N}_e \cdot Q\left(\frac{d_{min}}{2\sigma}\right) = \bar{N}_e \cdot Q\left(\sqrt{d_{free} \cdot \frac{\mathcal{E}_x}{\sigma^2}}\right) = \bar{N}_e \cdot Q\left(\sqrt{d_{free} \cdot \frac{k}{n} \cdot SNR}\right)$
  - Because d_{min}= $\sqrt{d_{free} \cdot 4 \cdot \mathcal{E}_x}$

energy-spread reduces energy/subsym
(assumes $\frac{1}{T'}$ can increase, so no filter on AWGN)

- AWGN $\bar{P}_b = \frac{N_b}{b} \cdot Q\left(\sqrt{d_{free} \cdot r \cdot SNR}\right)$
  - Where N_b=$\sum_{i=1}^{\infty} i \cdot N(i, d_{free})$ and $N(i,d)$ for conv code is the number of i-input-bit error events with distance d .
  - Finding N_b can require exhaustive search in general, but Section 7.2 (L9) shows how to compute $N(i,d)$ for CC, also distspec.m.
  - Yes, it is equal to Chapter 1's $\sum_{i=1}^{\infty} p_x(i) \cdot n_b(i)$, which is actually harder to compute.
- BC **coding gain** $\gamma = 10 \cdot \log_{10}(r \cdot d_{free})$ (for AWGN with binary subsymbols ..) and **energy/bit** $\bar{\mathcal{E}}_b$.

HAZARD WARNING☠ – BINARY CODING THEORIST'S FALLACY – assumes "free bandwidth"

Binary-code fair comparison: hold 2 of 3 $\{\bar{b} \quad \bar{\mathcal{E}}_x \quad \bar{P}_e\}$ fixed and compare 3rd ;

But $N_{coded} = \frac{1}{r} \cdot N_{uncoded}$ so then BOTH $\bar{\mathcal{E}}_x$ & $\bar{b}$ decrease for coded w.r.t uncoded (~ holding power & rate constant), not fair.

$\bar{b}_{coded} = r \cdot \bar{b}_{uncoded}$ $\quad \bar{\mathcal{E}}_{x,coded} = r \cdot \bar{\mathcal{E}}_{x,coded}$; So $\mathcal{E}_b = \frac{\bar{\mathcal{E}}_x}{\bar{b}}$ is the same, **BUT** $W \cdot T \rightarrow {}^{W \cdot T}/_{r}$

So, either the coded design increased bandwidth (may not be possible) or otherwise reduced rate;
adding a code to reduce rate is somewhat antithetical to Shannon if $R < C$. Increasing W is "cheating."

BSC Error Probability

- BSC $\bar{P}_e = \bar{N}_e \cdot [4p(1-p)]^{\left\lceil \frac{d_{free}}{2} \right\rceil}$
- BSC $\bar{P}_b = \frac{N_b}{b} \cdot [4p(1-p)]^{\left\lceil \frac{d_{free}}{2} \right\rceil}$

- Chapter 1's B-Bound can be used to show that this is roughly 3dB inferior to soft decoding (AWGN).

- Fair-comparison discussion is for AWGN.
  - Strictly speaking with BSC, data rate must reduce to improve with codes.
  - From BSC capacity, $r \le \underbrace{1 + p \cdot \log_2 p + (1-p) \cdot \log_2 (1-p)}_{\mathcal{C}} \le 1$ for reliable transmission with a code $0 < p < \frac{1}{2}$.

Coding Tables –best known rate ½ conv codes

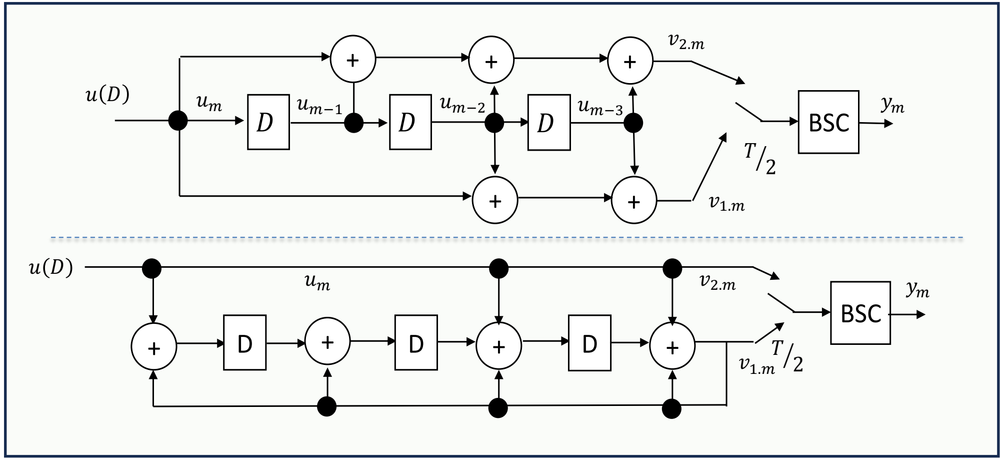
- Section 8.2 – Conv Code Tables see the octal entries, chap 8 [6])

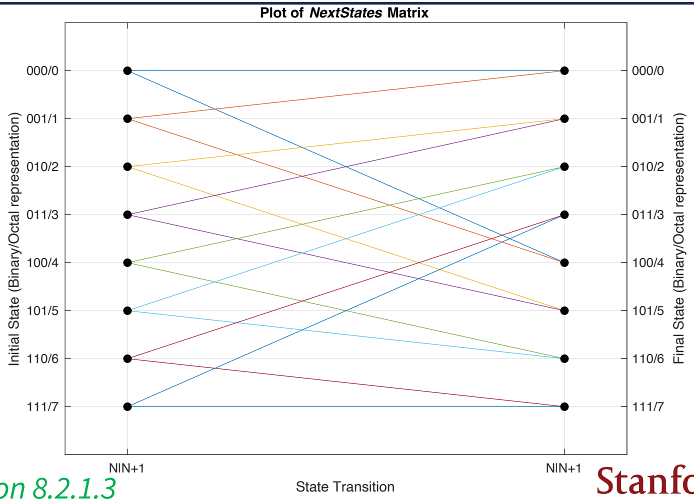
2^ν	$g_{11}(D)$	$g_{12}(D)$	d_{free}	γ	(dB)	N_e	N_1	N_2	N_b	L_D
4	7	5	5	2.5	3.98	1	2	4	1	3
8	17	13	6	3	4.77	1	3	5	2	5
16	23	35	7	3.5	5.44	2	3	4	4	8
(2G) 16	31	33	7	3.5	5.44	2	4	6	4	7
32	77	51	8	4	6.02	2	3	8	4	8
64	163	135	10	5	6.99	12	0	53	46	16
(802.11a) 64	155	117	10	5	6.99	11	0	38	36	16
(802.11b) 64	133	175	9	4.5	6.53	1	6	11	3	9
128	323	275	10	5	6.99	1	6	13	6	14
256	457	755	12	6	7.78	10	9	30	40	18
(3G) 256	657	435	12	6	7.78	11	0	50	33	16
512	1337	1475	12	6	7.78	1	8	8	2	11
1024	2457	2355	14	7	8.45	19	0	80	82	22
2048	6133	5745	14	7	8.45	1	10	25	4	19
4096	17663	11271	15	7.5	8.75	2	10	29	6	18
8192	26651	36477	16	8	9.0	5	15	21	26	28
16384	46253	77361	17	8.5	9.29	3	16	44	17	27
32768	114727	176121	18	9	9.54	5	15	45	26	37
65536	330747	207225	19	9.5	9.78	9	16	48	55	33
131072	507517	654315	20	10	10	6	31	58	30	27

Table 8.1: Rate 1/2 Maximum Free Distance Codes

L_D = length of Min-dist event

```
>> t8=poly2trellis(4,[17  13]) =
   numInputSymbols: 2
  numOutputSymbols: 4
         numStates: 8
        nextStates: [8 × 2 double]
           outputs: [8 × 2 double]
>> plotnextstates(t8.nextStates)
```





Best rate-1/3 convolutional codes

- Codes listed for other rates, example 1/3 here, see Sec 8.2 for ¼, 2/3, ¾,

2^ν	$g_{11}(D)$	$g_{12}(D)$	$g_{13}(D)$	$g_{14}(D)$	d_{free}	γ	(dB)	N_e	N_1	N_2	N_b	L_D
4	7	7	7	5	10	2.5	3.98	1	1	1	2	4
8	17	15	13	13	13	3.25	5.12	2	1	0	4	6
16	37	35	33	25	16	4	6.02	4	0	2	8	7
32	73	65	57	47	18	4.5	6.53	3	0	5	6	8
64	163	147	135	135	20	5	6.99	10	0	0	37	16
128	367	323	275	271	22	5.5	7.40	1	4	3	2	9
256	751	575	633	627	24	6.0	7.78	1	3	4	2	10
512	0671	1755	1353	1047	26	6.5	8.13	3	0	4	6	12
1024	3321	2365	3643	2277	28	7.0	8.45	4	0	5	9	16
2048	7221	7745	5223	6277	30	7.5	8.75	4	0	4	9	15
4096	15531	17435	05133	17627	32	8	9.03	4	3	6	13	17
8192	23551	25075	26713	37467	34	8.5	9.29	1	0	11	3	18
16384	66371	50575	56533	51447	37	9.25	9.66	3	5	6	7	19
32768	176151	123175	135233	156627	39	9.75	9.89	5	7	10	17	21
65536	247631	264335	235433	311727	41	10.25	10.1	3	7	7	7	20

- Code complexity measure $N_D = \underbrace{2^\nu}_{states} \cdot \left(\underbrace{2^k}_{adds} + \underbrace{2^k - 1}_{compares} \right)$

Design Example

- An AWGN has SNR = 5 dB.

- The uncoded ($M = 2$) error rate is $P_e = Q\left(10^{5/20}\right) = .0377 \ \ (not\ very\ good)$.

- A better design uses best 64-state rate $r =$ ½ code, so bandwidth expands by 2x.
  - The gain is 7 dB.
  - New $P_e = Q\left(10^{(5+7)/20}\right)$= 3.4303e-05 (better, see Slide L8:8's table for this code).

- To get $P_e \approx 10^{-6}$?
  - Need 8.5 dB of coding gain with rate ½ , so use this table's 1024-state code
  - $P_e = Q\left(10^{(5+8.5)/20}\right) \approx 10^{-6}$

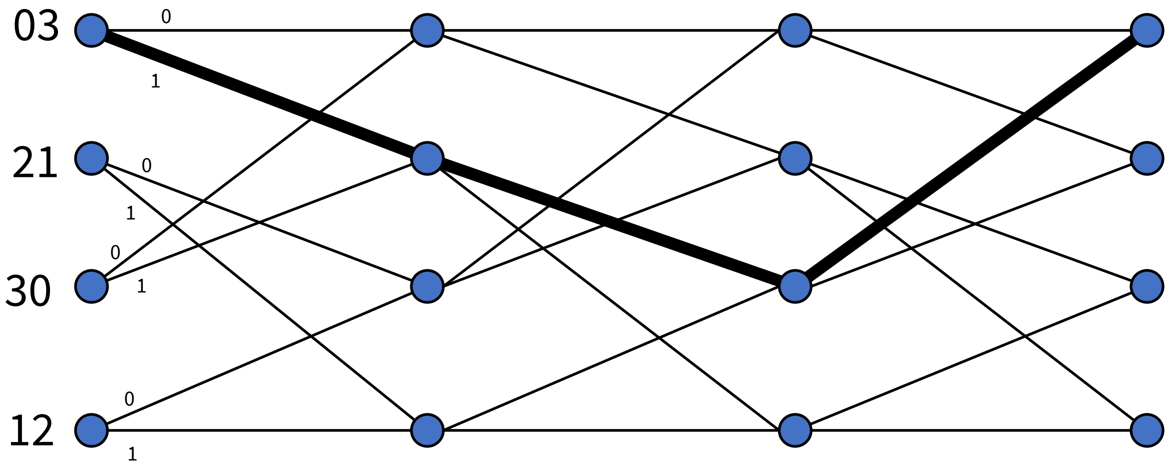
- Encoder is $G(D)$ = $\left[\underbrace{1 + D + D^2 + D^3 + D^5 + D^8 + D^{10}}_{2457} \quad \underbrace{1 + D^2 + D^3 + D^5 + D^6 + D^7 + D^{10}}_{2355}\right]$

1024 is a lot of states: larger distances may have large N_i that increase P_e.
Design instead should use better (not CC) code (see Lectures 9-11).
The 7 dB and 8.5 dB here often reduce in practice to about 5.5-6.0 dB, because of large N_i.

Viterbi Sequence Decoding

Section 7.1

Example, rate $r = ½$ CC surviving path



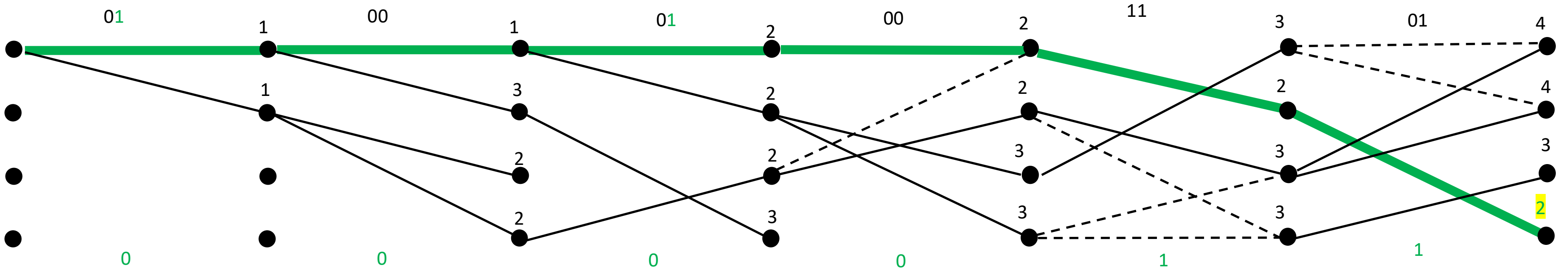
One path/sequence maximizes $p_{Y(D)/X(D)}$

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

- $\hat{\boldsymbol{x}}(D)$ is the best **survivor path.** There are 2^ν possible survivors at each time.
- Each state thus at time k has one best survivor, so 2^ν possible survivors at stage k,
  - from which any stage $k+1$ survivor must follow.

Example BSC: 6 input bits & 12 output bits

Green outputs – BSC-output 2 errors from correct sequence

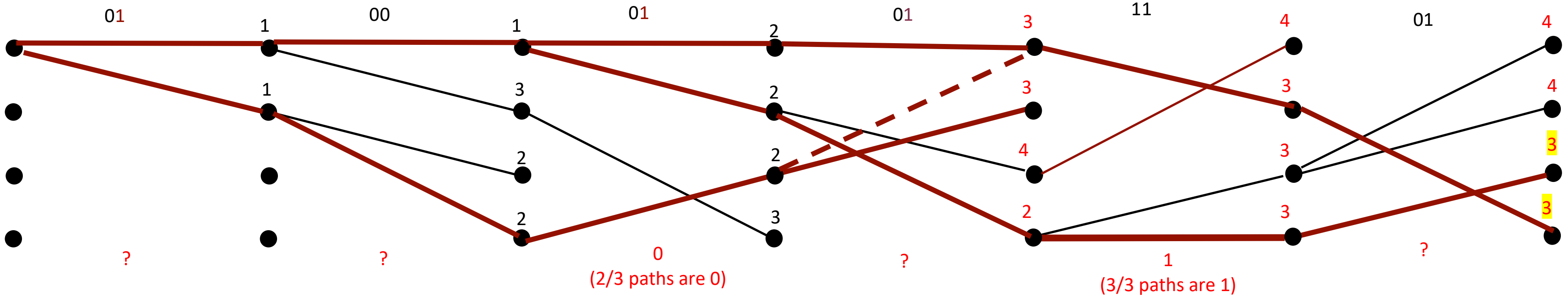


All input bits correctly detected

- This example's input bits are known and shown in green (in actual transmission, message is not known).
- Process can continue ad-infinitum, but after reasonable time ($\sim 5\nu$) – trace back path with lowest distance.
  - If a tie, pick one of them (probably an uncorrectable error has occurred).
- This is exactly ML if extended to infinity, and usually close with **finite survivor-path length.**

Example with 6 bits of input (12 output)

Red output – 3 BSC output errors → two sequences tied (detect error)



2/6 input bits are correct; 4 are ambiguous (using majority vote).

- Tie means sequence error is likely – must pick one of two equally likely.
- Detector needs more information to decode correctly.
  - This can include more future channel outputs that extend the 4 states (if available).

VA in General

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

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

previous-states set to state $i \triangleq J_i$ (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)

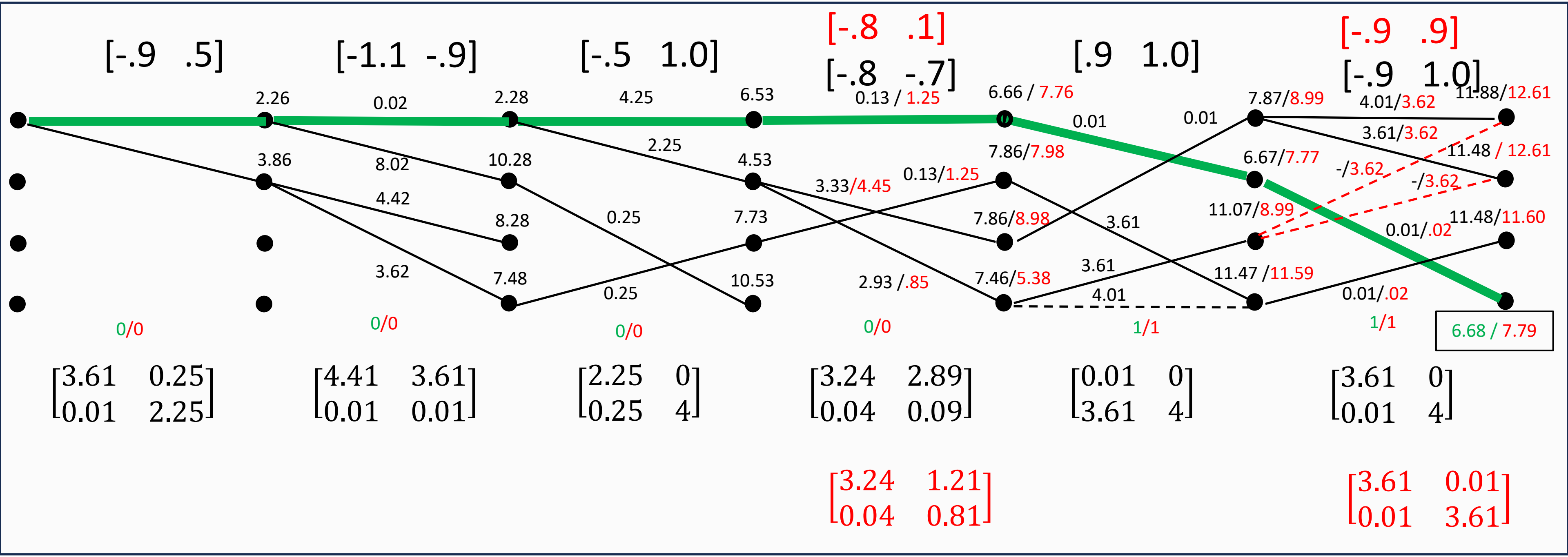
branch metric in going from state j to state i at time k, e.g. for BSC, $d_H(\boldsymbol{y}_k, \hat{\boldsymbol{v}}_k)$, or for AWGN $\Delta_{j,i,k} \triangleq \|\boldsymbol{y}_k - \hat{\boldsymbol{x}}_k(j \to i)\|^2$

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

- Formal – Many students just study trellis examples (L8:4-6) first, and then the above follows easily.

For AWGN?

- Same process with squared-distance replacing Hamming distance on branches



- The green path corresponds to the 2 “BSC errors” in hard-decoder example’s (L8:13) positions.
- The red numbers correspond to the 3 “BSC hard errors” in L8:14 positions, and they are corrected!

Soft decoding performs better than hard.

Matlab's vitdec program

```
DECODED = vitdec(CODE,TRELLIS,TBLEN,OPMODE,DECTYPE)

  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.
             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.
```

- **INPUTS**
  - Needs trellis, $y(D)$, survivor length
  - Indicate "opmode"
  - Hard/soft
- **OUTPUTS**
  - Detected bits (sometimes "delayed")
  - Last-state metrics
  - Survivor paths from last state
  - Survivor's bits on survivor path

Program-use examples are next.

Use of matlab vitdec for 4-state example

```
>> t=poly2trellis (3, [7 5]);
```

```
>> convenc([0 0 0 0 1 1],t)  =  00  00  00  00  11  01
```

```
>> t.nextStates =
   0  2
   0  2
   1  3
   1  3
>> t.outputs =
   0  3
   3  0
   2  1
   1  2
```

```
>> msg=[0 0 0 0 1 1];
>> code=convenc(msg,t)
code =   0   0   0   0   0   0   0   0   1   1   0   1
>> [d m p in] = vitdec(code,t,10,'cont','hard')
  % 'cont' mode's d output does not include the 00001101
d =   0   0   0   0   0   0 %(all before msg – this is survivor delay)
m=    3   2   3   0         % final pathmetrics
p =                         % previous states
   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   2   2   2   2   2
   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   2   2   2   2   2
in =                        % last 6 stages of inputs on paths p
   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   1   1   1   1   1   1
   0   0   0   0   0   1   1   1   1   1
```

0 0 0 0 1 1

OR with no survivor delay:

```
>> vitdec(code,t,6,'trunc','hard') =
   0   0   0   0   1   1
      % 'trunc' mode includes the 000011
```

Also for t3 in L7:27 Example (same code, with feedback)

```
>> code=convenc(msg,t3) =
   0   0   0   0   0   0   0   0   1   1   1   0

>> vitdec(code,t3,6,'trunc','hard') =
   0   0   0   0   1   1
```

- This uses matlab's ugly trellis

Now with Errors

- Repeat the earlier 2-output-bit error example decoding with matlab vitdec:

```
>> 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
```

- The program vitdec actually decodes bits with ties too (3-output-bit 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
```

- Surprisingly, this is correct.   Later we see a soft-output Viterbi (SOVA) that calculates additional local information for sequences with ties; it will also decode correctly.

- It is not clear what matlab vitdec is doing internally, but result is same. The full program is available by typing "edit vitdec" in matlab, but it is 414 lines with a lot of subroutine calls (the comments do not seem to help on this) and these subroutines are not visible with edit.
    - Motivated student encourage to take a look and tell me and rest of class how vitdec.m resolves the ties.

Viterbi Example with AWGN

- The vitdec.m function accepts the AWGN output as dectype = "**unquant**" ("soft" is for iterative decoders and is best used with biased all-positive log likelihood ratio soft information.)
  - vitdec.m 's detected input-of-the-channel uses opposite sign on channel output to this class/text's convention.

```
% original 2-output-bit errors
>> 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
```

Soft decoding performs better than hard

8-state rate 2/3 code

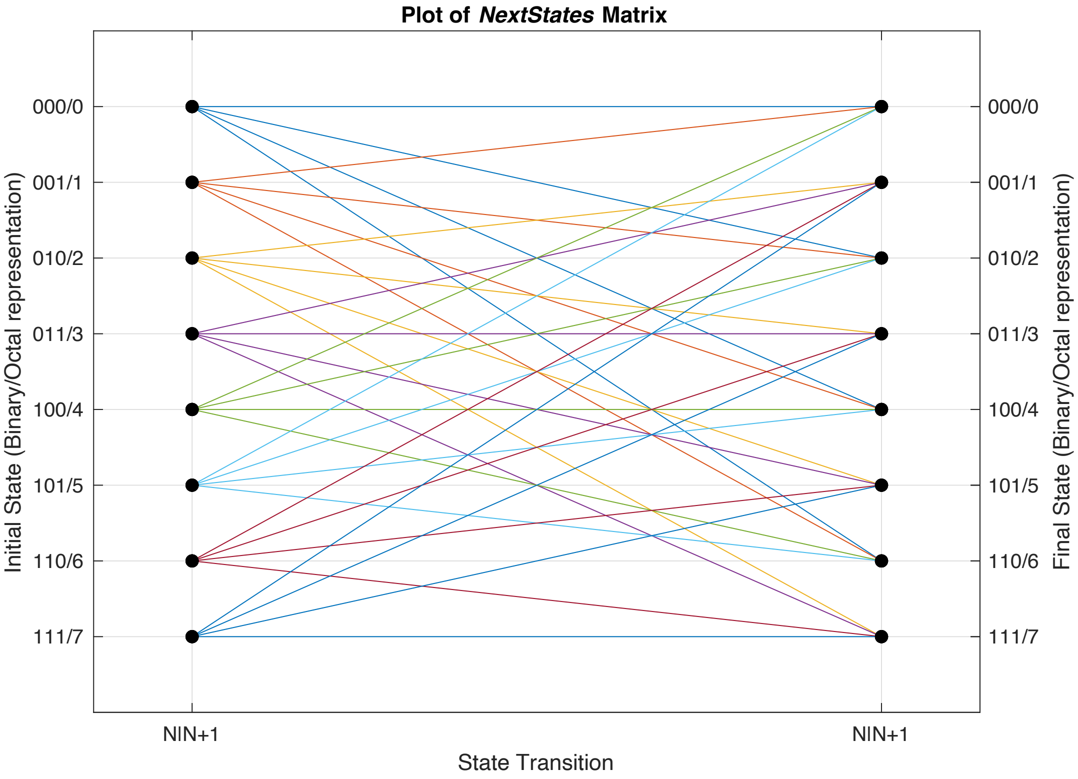
- Form gen & output

$$G_{best\,\frac{2}{3},\,8-state}(D) = \begin{bmatrix} D & 1+D^2 & 1+D^2 \\ 1+D & D & 1 \end{bmatrix}$$

```
tmin=poly2trellis([3 2], [2 5 5; 3 2 1])
    numInputSymbols: 4
   numOutputSymbols: 8
          numStates: 8
         nextStates: [8 × 4 double]
            outputs: [8 × 4 double]

>> tmin.nextStates
    0    4    2    6
    0    4    2    6
    1    5    3    7
    1    5    3    7
    0    4    2    6
    0    4    2    6
    1    5    3    7
    1    5    3    7

>> tmin.outputs
    0    6    3    5
    3    5    0    6
    4    2    7    1
    7    1    4    2
    5    3    6    0
    6    0    5    3
    1    7    2    4
    2    4    1    7
>> inmin= [0 0   0 0   0 0  1 0   1 1   0 1  0 0    0 1];
>> outmin=convenc(inmin, tmin)
            000  000  000  011  001  100  110  110
>> plotnextstates(tmin.nextStates)
```



8-state decode

- Decoding with dfree = 4:

```
>> vitdec(outmin,tmin,6,'trunc','hard’) % no errors
  0 0   0 0   0 0   1 0   1 1  0 1  0 0  0 1
>> inmin=
  0 0   0 0   0 0   1 0   1 1  0 1  0 0  0 1];
---------------
error2=[0 1 0, zeros(1,9), 1 0 0 , zeros(1,9)]; % 2 errors
>> vitdec(+xor(outmin,error2),tmin,6,'trunc','hard’)

  0 0   0 0   0 0   1 0   1 1  01  0 0  0 1
--------------------------------------
3 errors – can’t correct (smaller free distance)
>> error = 0 0 1   0 0 0   0 0 0   1 0 0   0 0 0   0 0 0   0 1 0   0 0 0

>> vitdec(+xor(outmin,error),tmin,6,'trunc','hard’)
  0 0   0 0   0 0   1 1   0 0   1 0   0 0   1 1
```

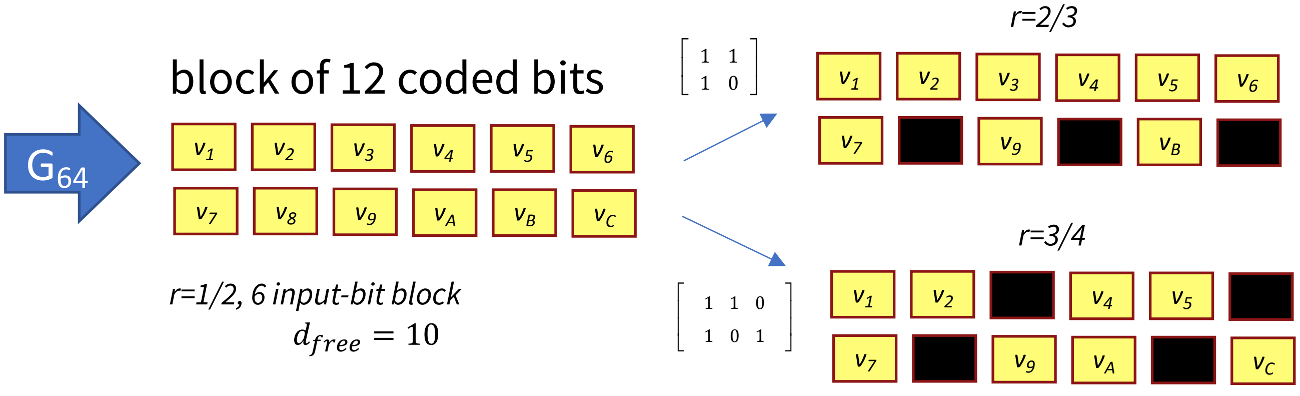
Higher rate, more complex (8 states), but lower distance of dfree = 4

- We encourage you to play a bit.
  - Start with examples here and then vary inputs / channel outputs and see.

It’s possible with only $\left\lceil \frac{d_{free}-1}{2} \right\rceil$ output bit errors to select wrong sequence – why?

- The error sequence relative to correct may not yet have merged.
  - This is why decoders typically report decoded output by tracing backward $5 \cdot \nu$ subsymbol periods

Decoders with puncturing?



BSC Decoder inserts "0" for punctured bit into branch value (branch Hamming distance calculation uses only transmitted bits)

AWGN Decoder similarly includes 0 into branch Euclidean metric

- Receiver knows where punctured bits would have been.
- The decoder enters **distance** 0 in the punctured channel-output position,
  - and otherwise proceeds.
  - vitdec.m has an optional input to say where this occurs.
- This reduces d_{free} by (usally, and at most) 1 for every punctured bit (10 for ½, 7 for 2/3, and 6 for ¾).

☠️ Caution on Matlab's awgn.m

- With $r < 1$ and binary AWGN, matlab's "SNR" is not defined the same as this course.

- For example, uncoded binary channel with SNR_{379}= 7 dB

```
>> y=awgn(x,7);
```

The 7 dB is for $\bar{\mathcal{E}}_x = 1$,
IN THE SAME BANDWIDTH

- For $r = 1/2$ and binary AWGN, matlab's "SNR"

```
>> y=awgn(x,10);
% or
>> y=awgn(x,7,-3)
```

The 7 dB IN THE SAME BANDWIDTH as uncoded causes matlab's 10 dB in 2x BANDWIDTH (because the noise is relative to $\bar{\mathcal{E}}_x = 1$ for the larger 2x bandwidth)

Or 2x-rate encoder output has $\bar{\mathcal{E}}_x = 1/2$,
So 7- (-3) = 10 dB

- This highly counter-intuitive, but matlab people were not considering fixed power over variable bandwidths.
- **Recommendation:** Use randn (Gaussian) noise directly with (1/sqrt(SNR/Exbar))*randn(# of points).
  - Avoid awgn command.

Maximum a Posteriori & the APP Algorithm

Section 7.3

Minimize instead each subsymbol error prob

- The **MAP detector** has criteria $\hat{u}_k = arg \min_{u_k} p_{u_k / Y_{0:K-1}}$.
  - Let $m \rightarrow k$ to emphasize time here (there are also different k input bits)
  - Also, let $\bar{N} \rightarrow K$.
- Reminder: this is the **APP** (à posteriori probability).
- Usually, the messages u_k are bits, so MAP minimizes each bit's error probability "separately."
- MAP decoding results are often very close to MLSD results, but not always:
  - If bit-error is the criterion, the MAP is better .. by definition.
  - If sequence (packet) error is the criterion, then MLSD is better.
  - Both MAP and MLSD initially assume the input values are equally likely.
- There is a "Viterbi-like" procedure **"Bahl Jelinek Cocke and Raviv (BCJR)"** that also uses the trellis for MAP.
- MAP or APP is more complex, but also produces "soft information" (LLR) that might be used by another code's decoder, if both share different encoders that act on same bit.
  - This product or concatenated code is a way to increase block length (better code possible) but retain simple decoding.

APP Method (largely for a packet of K subsymbols)

- Depends on 3 quantities from state i at time k to state j at time $k+1$
  - Forward trellis quantity

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

  - Backward trellis quantity

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

  - Branch quantity

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

- Tedious algebra and bookkeeping (See Section 7.3)
  - Branch calculation (do them all first)

    $$\gamma_k(j) = p_{m_k / \boldsymbol{y}_k}$$

  - Forward recursion

    $$\alpha_k(j) = \sum_{i \in S_k} \gamma_k(i,j) \cdot \alpha_{k-1}(i)$$

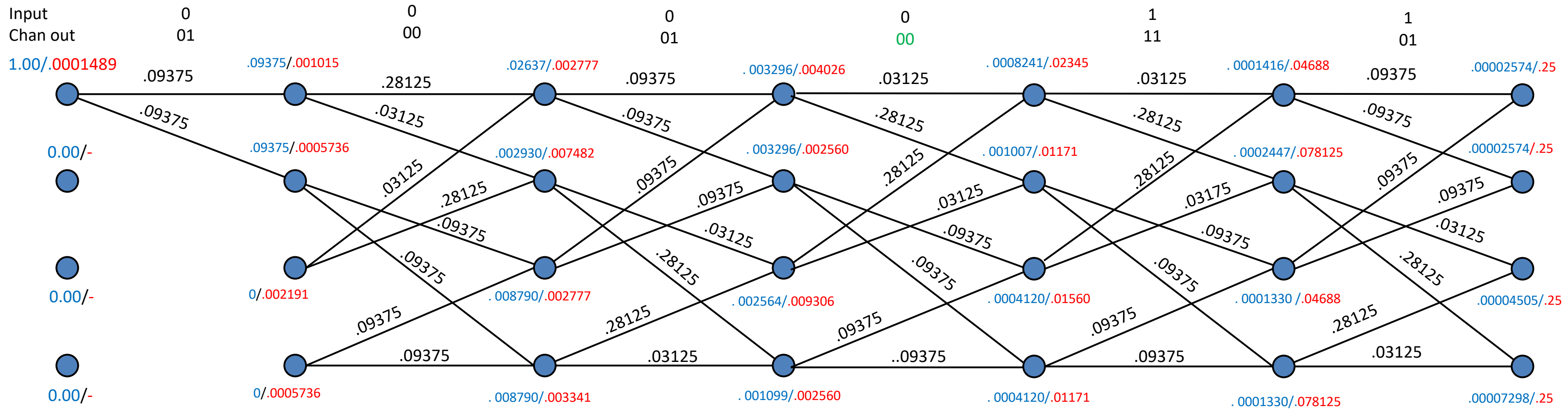
  - Backward recursion

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

Example with same rate ½ code - BSC

- Branch γ calculations are all of the form $\frac{1}{2} \cdot p^i \cdot (1-p)^{2-i} \quad i = 0,1,2$ for BSC with $p = {}^1/_4$.
- Forward pass sums two products at each state to get new α .
- Backward pass sums two products at each state to get new β .

$\frac{3}{32}$=.09375



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

Compute Likelihoods: Foundation Equations

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

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

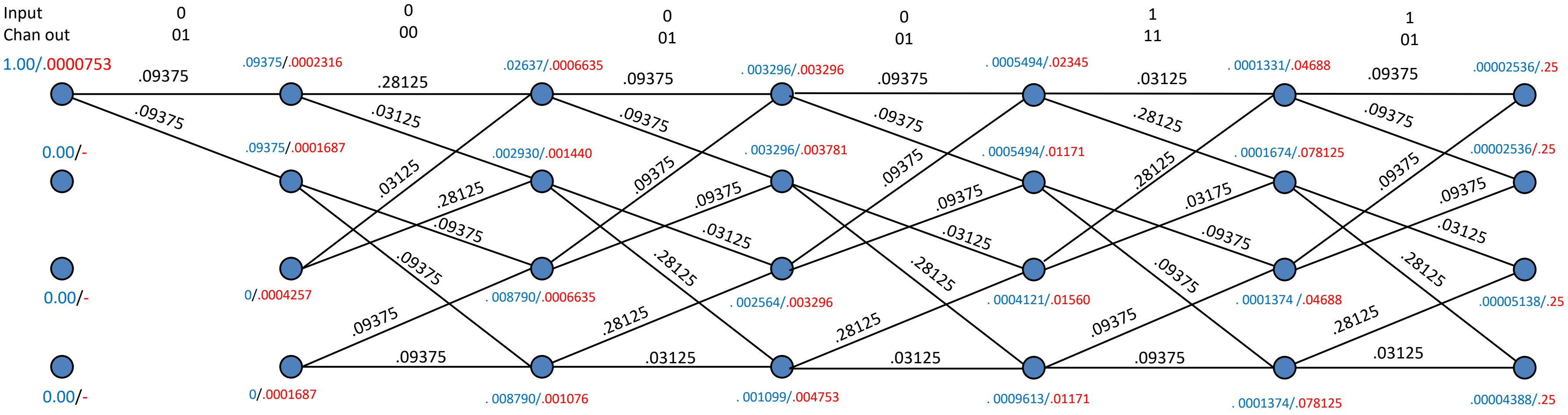
and on the labeling $\mathcal{S}_{k,}$, *which is the set of all allowed branch transitions from state any state* s_k *to any other state* s_{k+1} *for the given trellis description. The foundational equal is for caculation of the APP*

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

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

- Compute the 3 quantities for each branch (γ) and for each state $(\alpha\ ,\beta)$.
  - The decoder can also sum over the bit values corresponding to $\boldsymbol{x}_k$,
  - which usually includes the input bit values $u_{k,i}$.
  - In some iterative-decoding situations is better the output values $v_{k,i}$.
- Then compute sum to get à posteriori probabilities → decision for each bit.

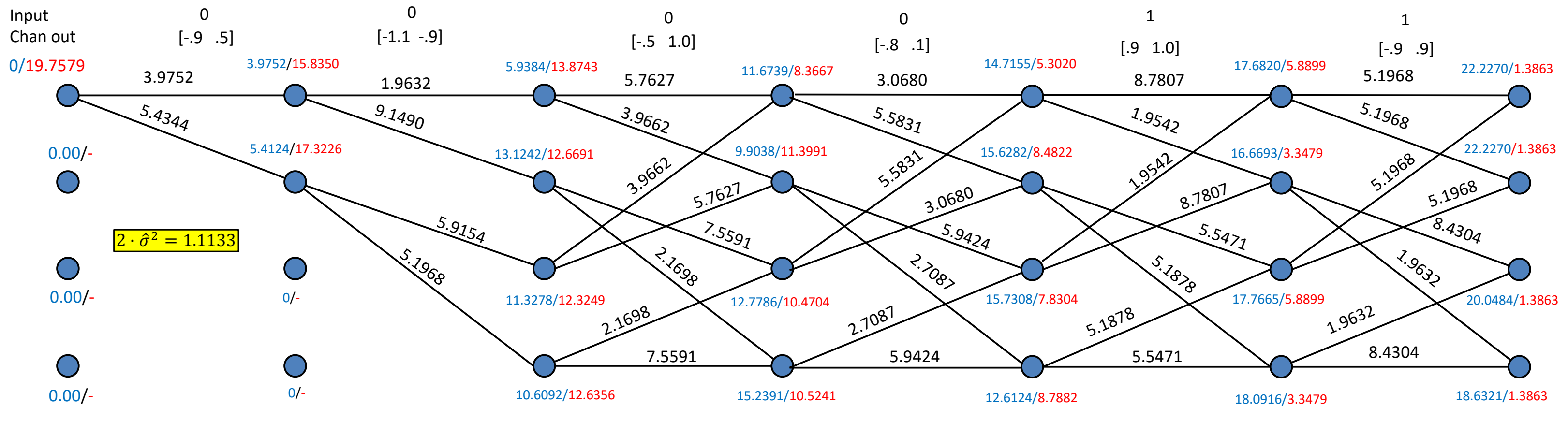
How about 3 errors on BSC?



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

- It corrects all 3 – even with hard decisions; however, the decoder uses a p. This is additional soft info.
- Decoder already knew $p = .25$, but MLSD did not use it (with Viterbi Algorithm) – just Hamming distance.
- This heads toward soft decoding, slightly. Decisions won't change for another $p < 0.5$, but the soft info does.

AWGN Case with BCJR (uses log likelihoods)



$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}3.61\\0.01\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0.25\\2.25\end{bmatrix}\right\}$ $\quad$ $\begin{bmatrix}3.2426\\0.0090\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0.2246\\2.0210\end{bmatrix}$

$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}4.41\\0.01\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}3.61\\0.01\end{bmatrix}\right\}$ $\quad$ $\begin{bmatrix}3.9612\\0.0090\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}3.2426\\0.0090\end{bmatrix}$

$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}2.25\\0.25\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0\\4\end{bmatrix}\right\}$ $\quad$ $\begin{bmatrix}2.0210\\0.2246\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0\\3.5929\end{bmatrix}$

$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}3.24\\0.04\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}1.21\\0.81\end{bmatrix}\right\}$ $\quad$ $\begin{bmatrix}2.9103\\0.0359\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}1.0869\\0.7276\end{bmatrix}$

$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}0.01\\3.61\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0\\4\end{bmatrix}\right\}$ $\quad$ $\begin{bmatrix}.0090\\3.2426\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0\\3.5929\end{bmatrix}$

$\frac{1}{2\cdot\hat{\sigma}^2}\cdot\left\{\begin{bmatrix}3.61\\0.01\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0.01\\3.61\end{bmatrix}\right\}$ $\quad$ $\begin{bmatrix}3.2426\\0.0090\end{bmatrix}+J^{0,1}\cdot\begin{bmatrix}0.0090\\3.2426\end{bmatrix}$

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**

- Calculation of branch metrics by hand can use the items inside the box.
- Final decisions are in the table.

LOGMAP APP

- This LOGMAP APP algorithm computes LL to avoid multiplication:

$$\alpha = \sum_i \alpha_i \cdot \gamma_i$$

- LOGMAP defines

$$\boxed{\begin{aligned} \lambda_i &\triangleq \ln(\alpha_i) + \ln(\gamma_i) \\ e^{\lambda_i} &= \alpha_i \cdot \gamma_i \end{aligned}}$$

- This simplifies multiplication (per term) to addition (and reduces arithmetic range requirement).

- Addition of original terms (different i) requires the calculation:

$$\lambda = \ln\left(\sum_i e^{\lambda_i}\right)$$

- LOGMAP recursively recruits the sum with table look ups and additions/differences:

$$\boxed{\lambda = \lambda_1 + \ln\left(1 + e^{\lambda_2 - \lambda_1}\right) = \lambda_1 + f(\lambda_2 - \lambda_1)}$$

Matlab’s BCJR (with some edits,@ website)

- Section 8.2 – Conv Code Tables (see the octal entries)

function **BCJR_AWGN**(y,trellis,sigma)

BCJR_conv Decoder

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 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 vectors from some (AWGN likely) channel output multiply this by -1 to get the EE379 Class convention on 0->-1

trellis - this is matlab’s usual trellis description (see text or class notes to avoid excessive computation for feedback systematic).

sigma - this is 1-dimensional AWGN standard deviation

OUTPUTS:

The decoded bits’ LLRs

```
> yawgn2  %(same 4-state code, same outputs – 3 error case)
-0.9000   0.5000  -1.1000  -0.9000  -0.5000   1.0000  -0.8000   0.1000
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   LLRs

     0         0        0         0         1         1      Bits
```

The entries on the earlier trellis were obtained by going into source code and printing gamma, alpha, beta, and LLs.

BCJR_BSC @ website

- Section 8.2 – Conv Code Tables (see the octal entries)

function **BCJR_BSC**(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 has been tested on easy (r=2/3) codes but not for k>2. Maximum k value is 4, so up to rate 4/5 codes.

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

The program returns the a posteriori probability's LLR for each data bit input. The program sets the stage beta initial probabilities to 1/2^n each. JC believes that avoids bias and is more accurate. N=length(y) and N/n must be integer.

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

WITH 0 OUTPUT BIT ERRORS:

```
>> 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
```

Bits

WITH 2 OUTPUT BIT ERRORS:

```
>> 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
```
% less 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
```

***p<1/2* just scales confidence**

Rate 2/3 examples

- The original program (BCJR_conv.m) from Matlab File exchange only handed AWGN.
  - It also only handled $k = 1$, so then only rate $r = 1/n$ codes.
  - BCJR_BSC.m and BCJR_AWGN.m handle respectively Hamming and Euclidean distance and r=k/n for k=1,2,3,4

```
tmin=poly2trellis([3 2], [2 5 5; 3 2 1]);
>> inmin =
  [ 00  00  00  10  11  01  00  01];
>> outmin=convenc(inmin,tmin)  =
   000   000  000 011  001 100 110  110

>> (-sign(BCJR_BSC(outmin,tmin,0.125))+1)/2
    00   00   00  10  11   01 00  01

>> (-sign(BCJR_BSC(xor(outmin,error2),tmin,0.125))+1)/2=
    00   00   00   10 11   01 00  01

------ 3 errors breaks --------------------
>> error = 0 0 1   0 0 0   0 0 0   1 0 0   0 0 0   0 0 0   0 1 0   0 0 0

>> (-sign(BCJR_BSC(xor(outmin,error),tmin,0.125))+1)/2
    00   00   00  10 11   11 00  11
```

$$G_{best\,\frac{2}{3},\,8-state}(D) = \begin{bmatrix} D & 1+D^2 & 1+D^2 \\ 1+D & D & 1 \end{bmatrix}$$

Note fewer bit errors with BCJR than with Viterbi (vitdec had 6 errors on L8:14).

- Soft Information?

```
>> BCJR_BSC(outmin,tmin,0.125) =
   6.3363   6.0916   5.5102   5.2724   4.8885   4.7739  -4.3927   4.3377
  -3.9469  -3.8931   3.5362  -3.3284   3.1898   2.8062   2.2747  -2.3525
```

Errors → less confidence

```
>> BCJR_BSC(xor(outmin,error2),tmin,0.125)
   1.4981   1.1619   1.0144   0.5441   1.7154   1.5290  -0.3864   1.0702
  -1.2267  -0.5590   0.6631  -0.7614   1.1868   0.6030   0.4964  -0.6719
```

Worse channel → less confidence

```
>> BCJR_BSC(outmin,tmin,0.25) =
   2.1147   2.0436   1.4665   1.3846   1.1293   1.1448  -0.9088   0.8865
  -0.7604  -0.6870   0.6312  -0.5003   0.5364   0.3806   0.2365  -0.2868
```

Soft-Output Viterbi Algorithm SOVA

Section 7.3.2

SOVA

- LOGMAX – approximates the sum in sum of products by maximum term.
  - Often very true in decoding.

$$\ln(\alpha_{k+1}, s_{k+1}) \approx \max_{\text{branches into } s_{k+1}} \ln(\alpha_k, s_k, \text{branch into}) + \ln(\gamma_k, \text{branch into}) \ .$$

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

$$\ln(\beta_k, s_k) \approx \max_{\text{branches into } s_{k+1}} \ln(\beta_{k+1}, s_k, \text{branch into}) + \ln(\gamma_k, \text{branch into}) \ .$$

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

- Look familiar?
  - Yes, back to Viterbi.
  - But now we have 2, one forward and one backward.

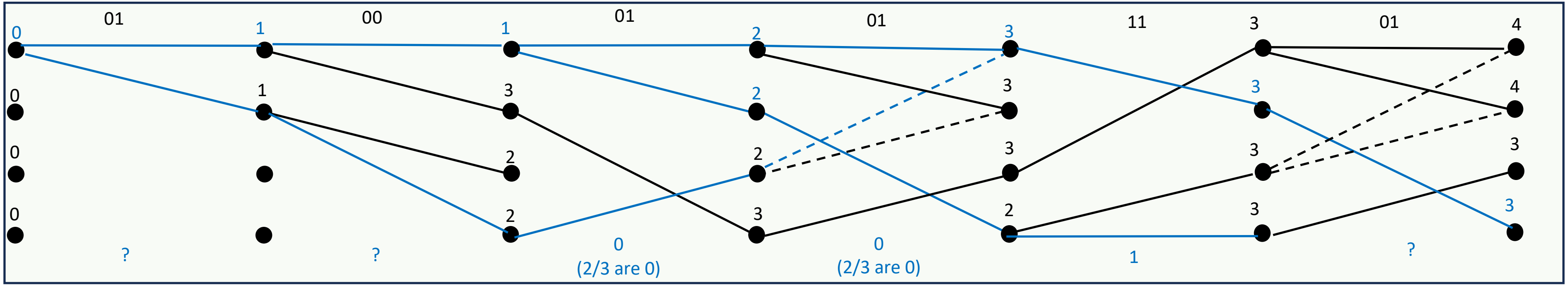
Forward SOVA Example with Ties

- It’s pretty easy without ties – just find other path with other input with next lowest survivor metric
  - And take the difference, which magnitude (an integer for BSC) is indication of confidence (+ sign for 0 and – sign for 1)

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 $ln(p)$.

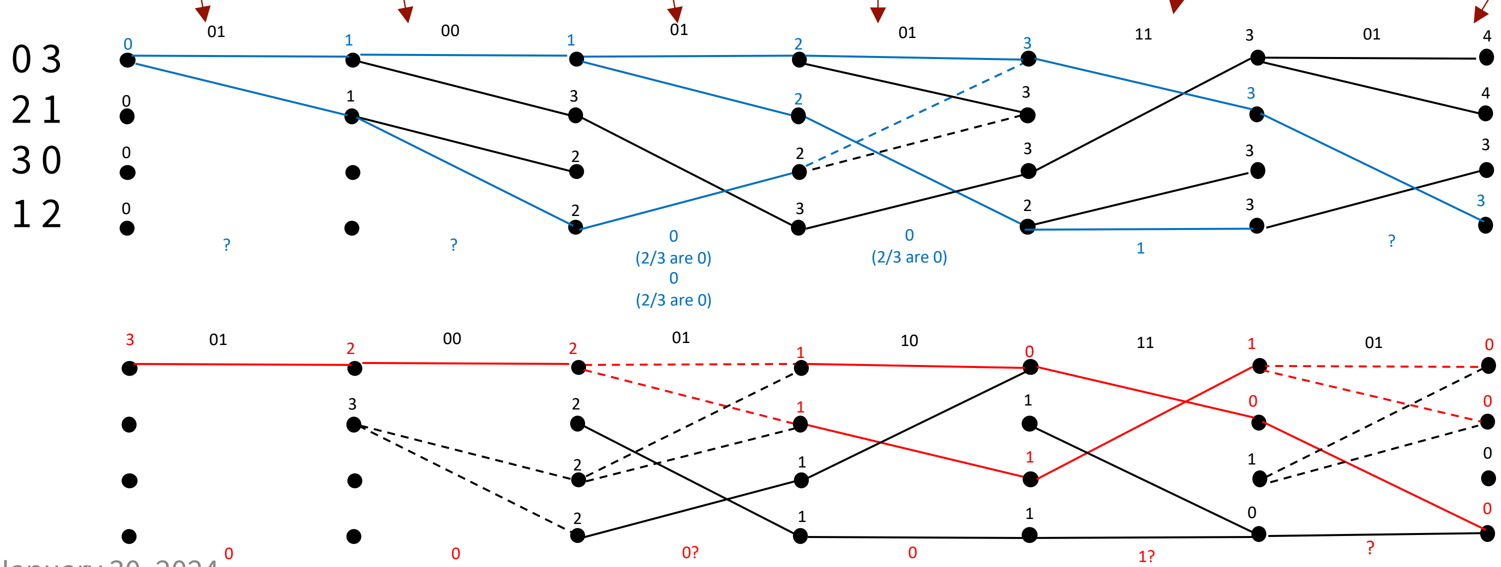


- The local resolution and majority voting appear to be what matlab is doing (requires examination/test of source code).
  - Probably could be confirmed by someone testing various situations
  - Nonetheless, the above is viable Forward-SOVA tie resolution

Forward-Backward SOVA Example

Forward-Backward SOVA						
k	0	1	2	3	4	5
$\{LL(0)\}$	$\left\{\underbrace{3}_{0+1+2}\right\}$	$\left\{\underbrace{3}_{1+0+2}\ \underbrace{4}_{1+1+2}\right\}$	$\left\{\underbrace{3}_{1+1+1}\ \underbrace{6}_{3+2+1}\ \underbrace{4}_{2+1+1}\ \underbrace{3}_{2+0+1}\right\}$	$\left\{\underbrace{3}_{2+1+0}\ \underbrace{5}_{2+2+1}\ \underbrace{3}_{2+1+0}\ \underbrace{4}_{3+0+1}\right\}$	$\left\{\underbrace{6}_{3+2+1}\ \underbrace{5}_{3+1+1}\ \underbrace{4}_{3+0+1}\ \underbrace{4}_{2+1+1}\right\}$	$\left\{\underbrace{4}_{3+1+0}\ \underbrace{5}_{3+2+0}\ \underbrace{4}_{3+0+1}\ \underbrace{3}_{3+0+0}\right\}$
$\{LL(1)\}$	$\left\{\underbrace{4}_{0+1+3}\right\}$	$\left\{\underbrace{5}_{1+2+2}\ \underbrace{4}_{1+1+2}\right\}$	$\left\{\underbrace{3}_{1+1+1}\ \underbrace{4}_{3+0+1}\ \underbrace{4}_{2+1+1}\ \underbrace{5}_{2+2+1}\right\}$	$\left\{\underbrace{4}_{2+1+1}\ \underbrace{3}_{2+0+0}\ \underbrace{4}_{2+1+1}\ \underbrace{6}_{3+2+1}\right\}$	$\left\{\underbrace{5}_{3+2+0}\ \underbrace{4}_{3+1+0}\ \underbrace{3}_{3+0+0}\ \underbrace{3}_{2+1+0}\right\}$	$\left\{\underbrace{4}_{3+1+0}\ \underbrace{3}_{3+0+0}\ \underbrace{4}_{3+1+0}\ \underbrace{5}_{3+2+0}\right\}$
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 $ln(p)$



Did better than Forward

Hagenauer's LLR SOVA update

- Prob of VA sequence error

$$Pr_{ML}\{x_k = -1\} = Pr\{u_k = 0\} \propto e^{-LS_k^*(0)}$$
$$Pr_{ML}\{x_k = +1\} = Pr\{u_k = 1\} \propto e^{-LS_k^*(1)}$$

- Magnitude difference of two bit choices is
  - $\Delta LS_k = LS_k^*(0)$- $LS_k^*(1)$
  - $LLR_k = x_k \cdot \Delta LS_k$
- Linear-code analysis: all 0's is correct, so

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

- Another decoder provides

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

- It includes soft info through:

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

- Algebra provides

$$LLR_k \leftarrow \ln\left[\frac{1+e^{\Delta LS_k+\widehat{LLR}_k}}{e^{\Delta LS_k}+e^{\widehat{LLR}_k}}\right]$$

- Ignores scaling difference between sequence and bit, so

$$\Delta LS_k \rightarrow \frac{(y_k-x_k)^2}{4 \cdot d_{free} \cdot SNR}$$

or $\Delta LS_k \rightarrow \frac{d_H(y_k,v_k)}{d_{free}}$ for BSC

End Lecture 8

IFD is backup

interesting, not enough time

Invariant Factors Decomposition

Appendix B.7

Parity Code Tables, Feedback, and poly2trellis

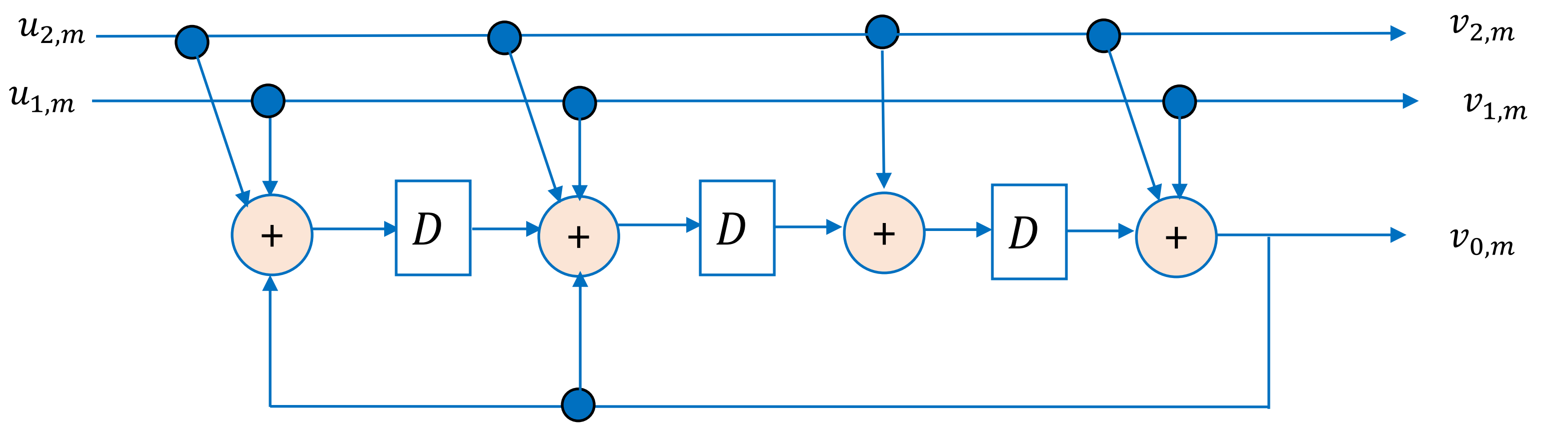
- Poly2trellis has a third input that is feedback – example best 8-state $r = 2/3$ conv code from tables

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

$$H_{sys}(D) = \left[\frac{D^3 + D^2 + D^1 + 1}{D^3 + D + 1} \quad \frac{D^3 + D^2 + 1}{D^3 + D + 1} \quad 1\right]$$

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

- Circuit has 8 states (3 flip flops)



So what does Matlab do?

```
>> tfeed=poly2trellis([4 4],[13 0 17 ; 0 13 15], [13 13])

tfeed =

     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 64  OUCH!
          nextStates: [64 × 4 double]
             outputs: [64 × 4 double]
```

- I could find no way to use this command other than the above valid (but nonminimal trellis).
- The matlab page examples do the same thing – increase number of states excessively.
- This is NOT a problem if code is $r = {}^{1}/{}_{n}$, then number of states is preserved.
- Here it was square of number of states (64), for rate ¾, it would cube number of states.

But there is a fix!

Work-Around

- This is tedious and so matlab probably wanted to avoid it (See Appendix B on **Invariant Factors Decomp**).
  - It is Smith-Normal Form, but in binary polynomials:

$$G_{sys}(D) = \underbrace{\begin{bmatrix} 1+D+D^2+D^3 & 1+D+D^2 \\ 1+D^2+D^3 & D+D^2 \end{bmatrix}}_{\substack{A \\ |A|=1}} \cdot \underbrace{\begin{bmatrix} \frac{1}{D^3+D+1} & 0 \\ 0 & 1 \end{bmatrix}}_{\substack{\Gamma \\ |\Gamma|\neq 0}} \cdot \begin{bmatrix} D & 1+D^2 & 1+D^2 \\ 1+D & D & 1 \end{bmatrix}$$

- The first two matrices are 1-to-1, so only remap all possible binary inputs to the SAME codewords.
  - They do not affect the set of codewords (or the code).
- Minimal 8-state feedback-free encoder is $G_{min}(D) = \begin{bmatrix} D & 1+D^2 & 1+D^2 \\ 1+D & D & 1 \end{bmatrix}$.

- Encode with $G_{sys}(D)$ convenc.m has no issues (even though it uses 64 states) or just encode with 8 state circuit on slide 34; the codewords are the same (so MLSD will find closest codeword).
- Decoder assumes $G_{min}(D)$ and finds $\hat{u}_{min}(D)$; then $\hat{v}_{min}(D)$= $\hat{u}_{min}(D) \cdot G_{min}(D)$ - recode the decoded.
- $\hat{u}_{sys}(D) = [\hat{v}_{2,min}(D) \quad \hat{v}_{1,min}(D)]$ because the original encoder was systematic.
  - Further any finite number of output errors only cause a finite (possibly less, but not more) number of input bit errors.

Example: 8-state rate 2/3 code

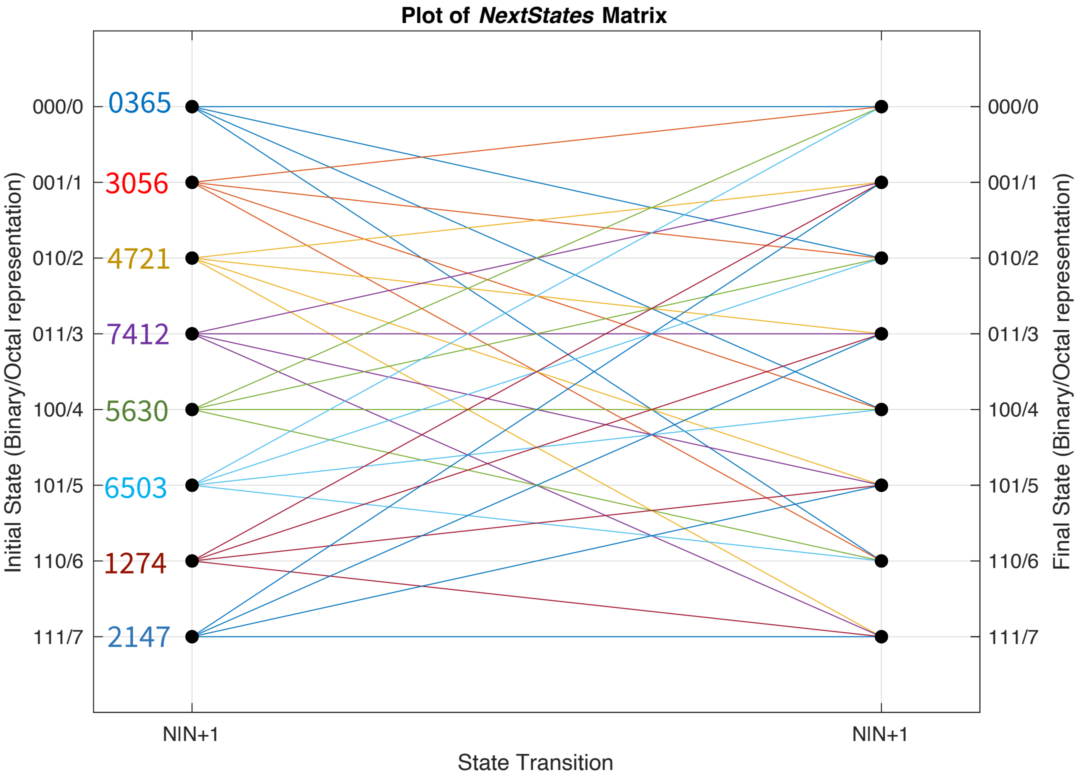
- Saving commands

```
tmin=poly2trellis([3 2], [2 5 5; 3 2 1])

    numInputSymbols: 4
   numOutputSymbols: 8
          numStates: 8
         nextStates: [8 × 4 double]
            outputs: [8 × 4 double]

>> tmin.nextStates
   0   4   2   6
   0   4   2   6
   1   5   3   7
   1   5   3   7
   0   4   2   6
   0   4   2   6
   1   5   3   7
   1   5   3   7

>> tmin.outputs
   0   6   3   5
   3   5   0   6
   4   2   7   1
   7   1   4   2
   5   3   6   0
   6   0   5   3
   1   7   2   4
   2   4   1   7
>> outmin=convenc([ 0 0  0 0  0 0  1 0  1 1  0 1  0 0  0 1],tmin)
   000   000   000   011 001   100   110   110
>> plotnextstates(tmin.nextStates)
```



8-state decode

- Minimal Direct Works – dfree = 6

```
>> vitdec(outmin,tmin,6,'trunc','hard')
  00   00   00   10   11   01  00   01
>> inmin=
  00   00   00   10   11   01   00   01];
---------------
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
>> vitdec(+xor(outmin,error2),tmin,6,'trunc','hard')

  00   00   00   10   11   01   00   01
```

- Systematic feedback encoder – different output

```
>> tfeed=poly2trellis([4 4],[13 0 17 ; 0 13 15], [13 13])
    numInputSymbols: 4
   numOutputSymbols: 8
          numStates: 64
         nextStates: [64 × 4 double]
            outputs: [64 × 4 double]
>> outfeed=convenc([ 0 0    0 0    0 0    1 0    1 1    0 1    0 0    0 1],tfeed)
                  0 0 0   0 0 0   0 0 0   1 0 1   1 1 1   0 1 1   0 0 1   0 1 1 %systematic
>> informin=vitdec(outfeed,tmin,6,'trunc','hard')
                   00     00     00     11     01     11     00     00 % map differs
>> vmin = convenc(informin,tmin)  =
                   000    000    000   101    111    011   001    011
```

Have to leave spaces in matlab, but it looks better without them here

```
>> 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
>> vmin2 = convenc(informin2,tmin)
   000  000  000  101  111  011  001  011

>> outfeed  % check
   000  000  000  101  111  011  001  011

% So, this fixes matlab's high-complexity-trellis problem with 8-state decoder
```

This works for any decoder,
But of course most helpful
With matlab poly2trellis issues