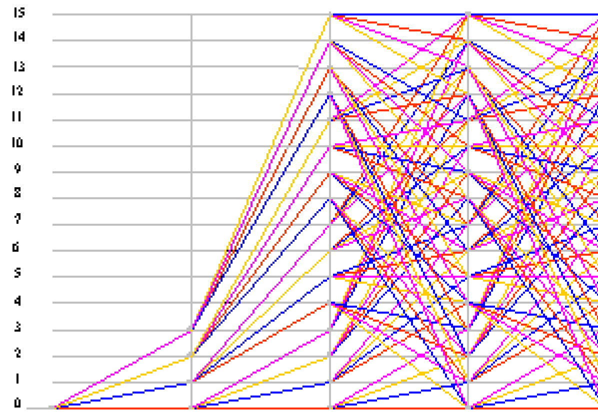
Convolution - Viterbi Code/Decode

Brief Help edited by Juan Chamero, jach_spain@yahoo.es

Updated as of April 2006

References:

1. Viterbi School HomePage: http://viterbi.usc.edu/about/viterbi/viterbi_digital_age.htm, and http://home.netcom.com/%7Echip.f/viterbi/algrthms.html.
2. Steve Gardner and Tom Hardin, "Systems design and implementation of modems and FEC codecs for wireless and wireline systems". They can be reached at shg@istari-design.com and cth@istari-design.com.
3. Library of Comm issues in C++ **SPUC** Org: http://spuc.sourceforge.net/ , http://spuc.sourceforge.net/iterbi Class (Full): http://spuc.sourceforge.net/



INDEX

Parameters
Interpretation
Some Convolution/Viterbi Experimentation References
Shortcut for a NO ERROR condition
Examples
- Running for K = 5
- Other Polynomials set:

Convolution seen as from a States Machine

Appendices
- i) Going Backward in case of No errors condition
- II) Another Sources of Viterbi related soft:
- III) Convolution - Viterbi Source Codes
- IV) Something about efficiency

Parameters

Ratio: 1:2, enter one bit, two bits outcome (a two bits symbol in the Jargon)
K=3, constraint, "reach" of the convolution algorithm
Polynomial pair: (a, b): binary numbers that point to Polynomial Tab Bits to XOR in order to generate output. For instance (3, 5) means (0011), (0101)

Sample: [010111001010001] plus three 0 flushing bits, remains:

[010111001010001000]

We are going then to generate two streams: Upper and Lower

Polynomial for Upper Stream 1: $1 + X + X^2$
Polynomial for Lower Stream 2: $1 + X$

K=3 stands for a "shift register" of size K-1=2, or in the GSM jargon the size of the "Convolution States Machine" because the convolution coder could be seen as a "black box" that changes states when triggered by input bits, to generate an output of two bits in this case, one for the Upper stream and another for the Lower stream.

Let's see a states transition scheme, step by step, a transition at a time:

State	Input	B0	B1	output	Us	Ls
0	-	0	0	-	-	-
1	0	0	0	0	0	0
2	1	1	0	0	1	1
3	0	0	1	0	1	0
4	1	1	0	1	0	0
5	1	1	1	0	0	1
6	1	1	1	1	1	0
7	0	0	1	1	0	1
8	0	0	0	1	1	1
9	1	1	0	0	1	1
10	0	0	1	0	1	0
11	1	1	0	1	0	0
12	0	0	1	0	1	0
13	0	0	0	1	1	1
14	0	0	0	0	0	0
15	1	1	0	0	1	1
16	0	0	1	0	1	0
17	0	0	0	1	1	1
18	0	0	0	0	0	0

Generating the following Us and Ls pairs:

00 11 10 00 01 10 01 11 11 10 00 10 11 00 11 10 11 00

Return

Interpretation

Al possible states of this box are then:

B0	B1
0	0
0	1
1	0
1	1

Each state "matches" with a bit input to change its state

Next State Table

States bits		Input bit	
B0	B1	0	1
0	0	00	10
0	1	00	10
1	0	01	11
1	1	01	11

And for each triad (state, input bit, output bit) we have precisely determined the stream output, both upper and lower stream in this case, throughout a Table Lookup as follows:

Output Symbol Table

Current State	Input = 0:	Input = 1:
00	00	11
01	11	00
10	10	01
11	01	10

Note: observe that the "Hamming distance" between possible output streams as a function of input bit at any cycle is 2.

Now we may build the output as a function of states change with a third Input Inference Table as a function of states, before and after being triggered by the input bit:

Input Inference table

	00 = 0	01 = 1	10 = 2	11 = 3
00 = 0	0	X	1	X
01 = 1	0	X	1	X
10 = 2	X	0	X	1
11 = 3	X	0	X	1

Warning: With this table we may decode convolution straightforwardly (see below the procedure to follow for a NO ERROR condition).

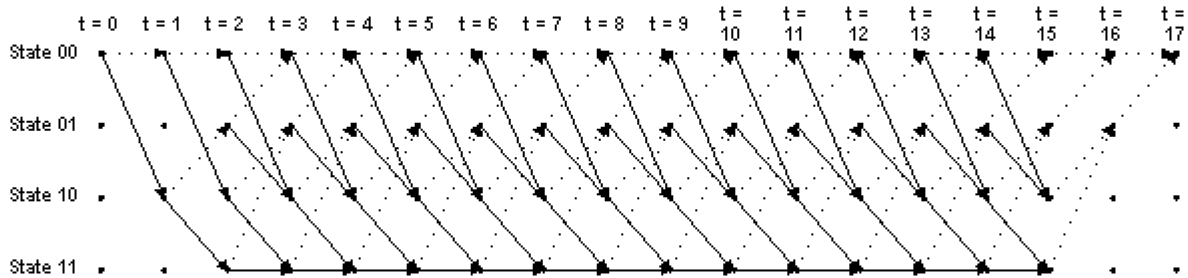
Return

Some Convolution/Viterbi Experimentation References

Source: http://home.netcom.com/%7Echip.f/viterbi/tutorial.html , www.netcom.com

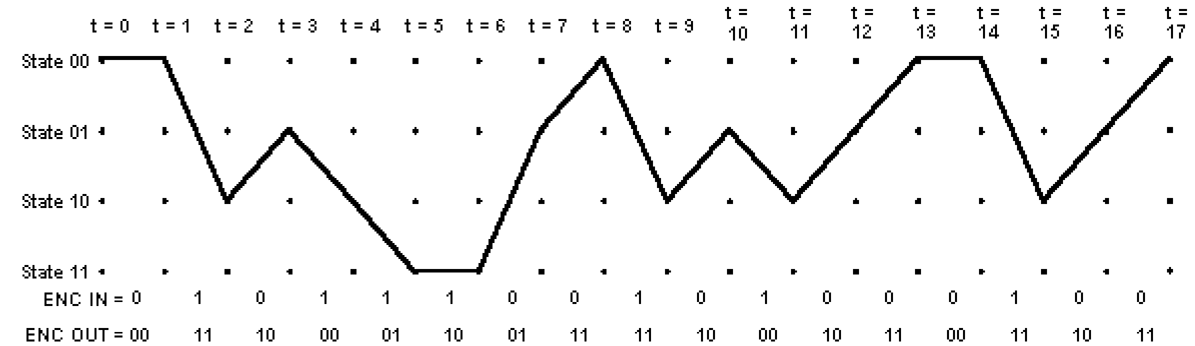
Description of the Algorithm

The Viterbi decoder itself is the primary focus of this tutorial. Perhaps the single most important concept to aid in understanding the Viterbi algorithm is the Trellis diagram. The figure below shows the trellis diagram for our example rate ½, K = 3 convolution encoder, for a 15-bit message:

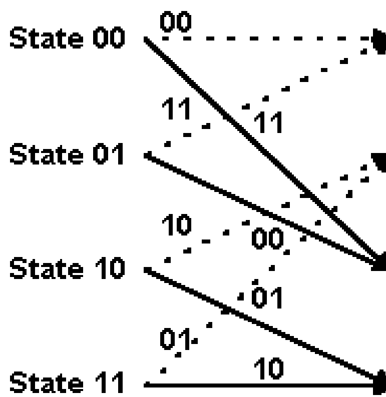


The four possible states of the encoder are depicted as four rows of horizontal dots. There is one column of four dots for the initial state of the encoder and one for each time instant during the message. For a 15-bit message with two encoder memory flushing bits, there are 17 time instants in addition to t = 0, which represents the initial condition of the encoder. The solid lines connecting dots in the diagram represent state transitions when the input bit is a one. The dotted lines represent state transitions when the input bit is a zero. Notice the correspondence between the arrows in the trellis diagram and the state transition table discussed above. Also notice that since the initial condition of the encoder is State 00_2, and the two memory flushing bits are zeroes, the arrows start out at State 00_2 and end up at the same state.

The following diagram shows the states of the trellis that are actually reached during the encoding of our example 15-bit message:

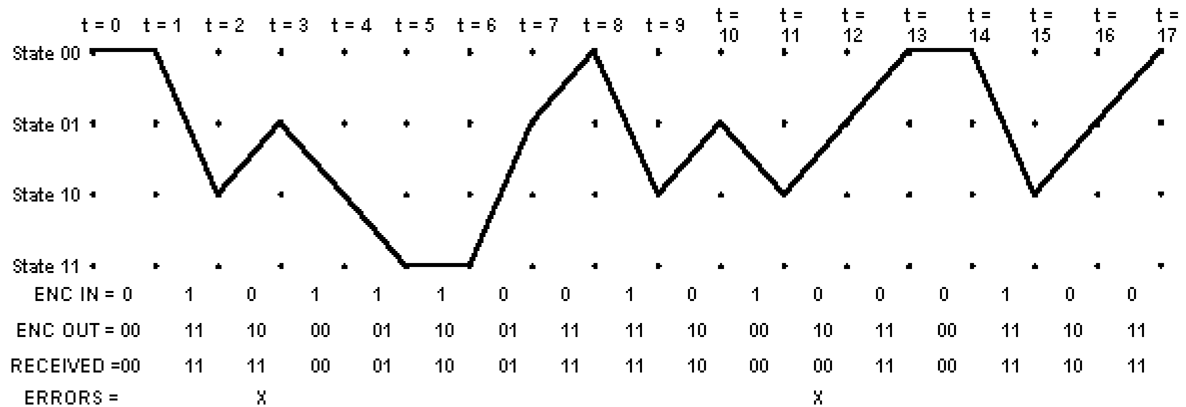


The encoder input bits and output symbols are shown at the bottom of the diagram. Notice the correspondence between the encoder output symbols and the output table discussed above. Let's look at that in more detail, using the expanded version of the transition between one time instant to the next shown below:



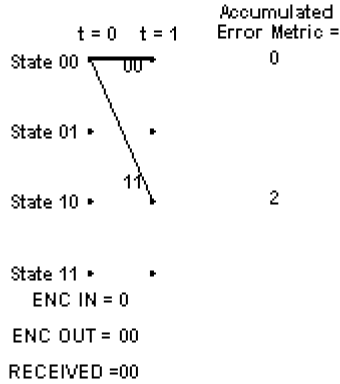
The two-bit numbers labeling the lines are the corresponding convolution encoder channel symbol outputs. Remember that dotted lines represent cases where the encoder input is a zero, and solid lines represent cases where the encoder input is a one. (In the figure above, the two-bit binary numbers labeling dotted lines are on the left, and the two-bit binary numbers labeling solid lines are on the right.)

OK, now let's start looking at how the Viterbi decoding algorithm actually works. For our example, we're going to use hard-decision symbol inputs to keep things simple. (The example source code uses soft-decision inputs to achieve better performance.) Suppose we receive the above encoded message with a couple of bit errors:



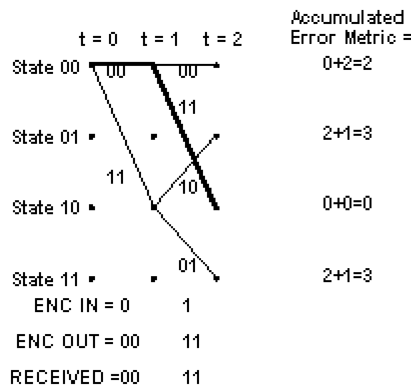
Each time we receive a pair of channel symbols, we're going to compute a metric to measure the "distance" between what we received and all of the possible channel symbol pairs we could have received. Going from t = 0 to t = 1, there are only two possible channel symbol pairs we could have received: 00_2, and 11_2. That's because we know the convolution encoder was initialized to the all-zeroes state, and given one input bit = one or zero, there are only two states we could transition to and two possible outputs of the encoder. These possible outputs of the encoder are 00_2 and 11_2. The metric we're going to use for now is the Hamming distance between the received channel symbol pair and the possible channel symbol pairs. The Hamming distance is computed by simply counting how many bits are different between the received channel symbol pair and the possible channel symbol pairs. The results can only be zero, one, or two. The Hamming distance (or other metric) values we compute at each time instant for the paths between the states at the previous time instant and the states at the current time instant are called branch metrics. For the first time instant, we're going to save these results as "accumulated error metric" values, associated with states. For the second time instant on, the accumulated error metrics will be computed by adding the previous accumulated error metrics to the current branch metrics.

At t = 1, we received 00_2. The only possible channel symbol pairs we could have received are 00_2 and 11_2. The Hamming distance between 00_2 and 00_2 is zero. The Hamming distance between 00_2 and 11_2 is two. Therefore, the branch metric value for the branch from State 00_2 to State 00_2 is zero, and for the branch from State 00_2 to State 10_2 it's two. Since the previous accumulated error metric values are equal to zero, the accumulated metric values for State 00_2 and for State 10_2 are equal to the branch metric values. The accumulated error metric values for the other two states are undefined. The figure below illustrates the results at t = 1:



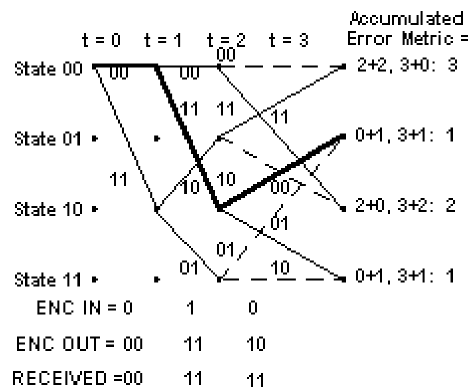
Note that the solid lines between states at t = 1 and the state at t = 0 illustrate the predecessor-successor relationship between the states at t = 1 and the state at t = 0 respectively. This information is shown graphically in the figure, but is stored numerically in the actual implementation. To be more specific, or maybe clear is a better word, at each time instant t, we will store the number of the predecessor state that led to each of the current states at t.

Now let's look what happens at t = 2. We received a 11_2 channel symbol pair. The possible channel symbol pairs we could have received in going from t = 1 to t = 2 are 00_2 going from State 00_2 to State 00_2, 11_2 going from State 00_2 to State 10_2, 10_2 going from State 10_2 to State 01_2, and 01_2 going from State 10_2 to State 11_2. The Hamming distance between 00_2 and 11_2 is two, between 11_2 and 11_2 is zero, and between 10_2 or 01_2 and 11_2 is one. We add these branch metric values to the previous accumulated error metric values associated with each state that we came from to get to the current states. At t = 1, we could only be at State 00_2 or State 10_2. The accumulated error metric values associated with those states were 0 and 2 respectively. The figure below shows the calculation of the accumulated error metric associated with each state, at t = 2.



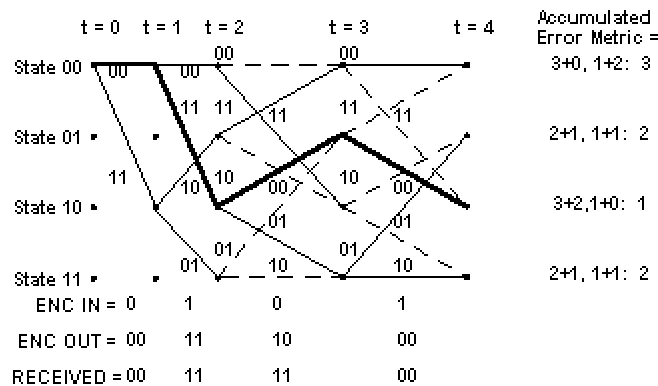
That's all the computation for t = 2. What we carry forward to t = 3 will be the accumulated error metrics for each state, and the predecessor states for each of the four states at t = 2, corresponding to the state relationships shown by the solid lines in the illustration of the trellis.

Now look at the figure for t = 3. Things get a bit more complicated here, since there are now two different ways that we could get from each of the four states that were valid at t = 2 to the four states that are valid at t = 3. So how do we handle that? The answer is, we compare the accumulated error metrics associated with each branch, and discard the larger one of each pair of branches leading into a given state. If the members of a pair of accumulated error metrics going into a particular state are equal, we just save that value. The other thing that's affected is the predecessor-successor history we're keeping. For each state, the predecessor that survives is the one with the lower branch metric. If the two accumulated error metrics are equal, some people use a fair coin toss to choose the surviving predecessor state. Others simply pick one of them consistently, i.e. the upper branch or the lower branch. It probably doesn't matter which method you use. The operation of adding the previous accumulated error metrics to the new branch metrics, comparing the results, and selecting the smaller (smallest) accumulated error metric to be retained for the next time instant is called the add-compare-select operation. The figure below shows the results of processing t = 3:

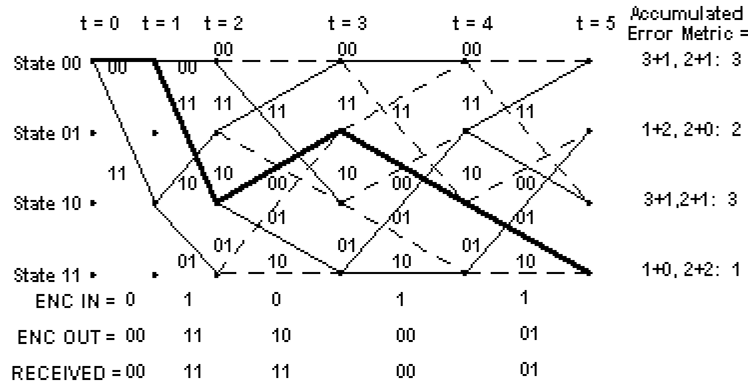


Note that the third channel symbol pair we received had a one-symbol error. The smallest accumulated error metric is a one, and there are two of these.

Let's see what happens now at t = 4. The processing is the same as it was for t = 3. The results are shown in the figure:

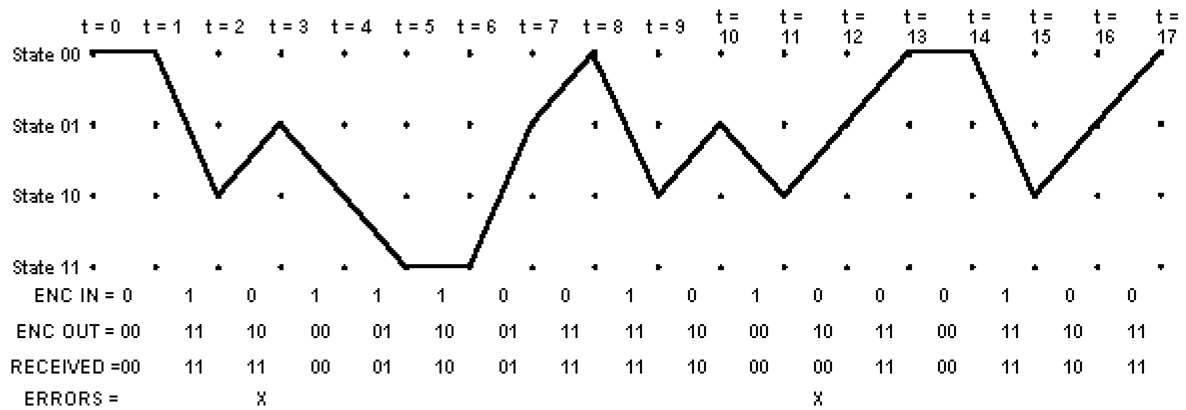


Notice that at t = 4, the path through the trellis of the actual transmitted message, shown in bold, is again associated with the smallest accumulated error metric. Let's look at t = 5:



At t = 5, the path through the trellis corresponding to the actual message, shown in bold, is still associated with the smallest accumulated error metric. This is the thing that the Viterbi decoder exploits to recover the original message.

Perhaps you're getting tired of stepping through the trellis. I know I am. Let's skip to the end. At t = 17, the trellis looks like this, with the clutter of the intermediate state history removed:



The decoding process begins with building the accumulated error metric for some number of received channel symbol pairs, and the history of what states preceded the states at each time instant t with the smallest accumulated error metric. Once this information is built up, the Viterbi decoder is ready to recreate the sequence of bits that were input to the convolution encoder when the message was encoded for transmission. This is accomplished by the following steps:

- First, select the state having the smallest accumulated error metric and save the state number of that state.
- Iteratively perform the following step until the beginning of the trellis is reached: Working backward through the state history table, for the selected state, select a new state which is listed in the state history table as being the predecessor to that state. Save the state number of each selected state. This step is called traceback.
- Now work forward through the list of selected states saved in the previous steps. Look up what input bit corresponds to a transition from each predecessor state to its successor state. That is the bit that must have been encoded by the convolution encoder.

The following table shows the accumulated metric for the full 15-bit (plus two flushing bits) example message at each time t:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
State 00_2		0	2	3	3	3	3	4	1	3	4	3	3	2	2	4	5	2
State 01_2			3	1	2	2	3	1	4	4	1	4	2	3	4	4	2	
State 10_2		2	0	2	1	3	3	4	3	1	4	1	4	3	3	2		
State 11_2			3	1	2	1	1	3	4	4	3	4	2	3	4	4		

It is interesting to note that for this hard-decision-input Viterbi decoder example, the smallest accumulated error metric in the final state indicates how many channel symbol errors occurred. The following state history table shows the surviving predecessor states for each state at each time t:

t =	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
State 00_2	0	0	0	1	0	1	1	0	1	0	0	1	0	1	0	0	0	1
State 01_2	0	0	2	2	3	3	2	3	3	2	2	3	2	3	2	2	2	0
State 10_2	0	0	0	0	1	1	1	0	1	0	0	1	1	0	1	0	0	0
State 11_2	0	0	2	2	3	2	3	2	3	2	2	3	2	3	2	2	0	0

The following table shows the states selected when tracing the path back through the survivor state table shown above:

t =	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
	0	0	2	1	2	3	3	1	0	2	1	2	1	0	0	2	1	0

Using a table that maps state transitions to the inputs that caused them, we can now recreate the original message. Here is what this table looks like for our example rate ½ K = 3 convolution code:

	Input was, Given Next State =			
Current State	**00_2 = 0**	**01_2 = 1**	**10_2 = 2**	**11_2 = 3**
00_2 = 0	0	x	1	X
01_2 = 1	0	x	1	X
10_2 = 2	X	0	x	1
11_2 = 3	X	0	x	1

Note: In the above table, x denotes an impossible transition from one state to another state.

So now we have all the tools required to recreate the original message from the message we received:

t =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	1	0	1	1	1	0	0	1	0	1	0	0	0	1

The two flushing bits are discarded.

Shortcut for a NO ERROR condition

We have to take into account that in summary the convolution is a injective transformation F, meaning that given an initial state S(0) and an input stream I of length h bits a convolution with (in this case) a couple of functions f and g (corresponding to two polynomials) generates via a sort of Shift Register F applied to (S(0), I) gives the stream II formed by h pairs ob bits that are named “symbols”, namely:

F(S(0), I) = II

Existing then the inverse function F´ such that

F´(II) = I

Let´s see how this inverse behaves. We may now use the Input Inference Table defined above. Knowing the output symbols stream, for example:

00 11 10 00 01 10 01 11 11 10 00 10 11 00 11 10 11 00

Input Inference table

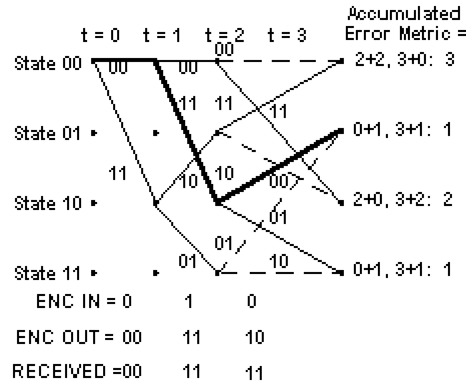
	00 = 0	01 = 1	10 = 2	11 = 3
00 = 0	0	X	1	X
01 = 1	0	X	1	X
10 = 2	X	0	X	1
11 = 3	X	0	X	1

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S	00	00	10	01	10	11	11	01	00	10	01	10	01	00	00	10	01	00
II	00	11	10	00	01	10	01	11	11	10	00	10	11	00	11	10	11	00
I	0	1	0	1	1	1	0	0	1	0	1	0	0	0	1	0	0	0

First step: given S(t), current state at time t, and II(t), output symbol at time t, from Output Symbol Table we “infer” Input (I(t) at time t.

Second step: Once inferred Input we compute the next state S(t+1) from Next State Table

Here's an insight into how the traceback algorithm eventually finds its way onto the right path even if it started out choosing the wrong initial state. This could happen if more than one state had the smallest accumulated error metric, for example. I'll use the figure for the trellis at t = 3 again to illustrate this point:



See how at t = 3, both States 01_2 and 11_2 had an accumulated error metric of 1. The correct path goes to State 01_2 -notice that the bold line showing the actual message path goes into this state. But suppose we choose State 11_2 to start our traceback. The predecessor state for State 11_2 , which is State 10_2 , is the same as the predecessor state for State 01_2! This is because at t = 2, State 10_2 had the smallest accumulated error metric. So after a false start, we are almost immediately back on the correct path.

For the example 15-bit message, we built the trellis up for the entire message before starting traceback. For longer messages, or continuous data, this is neither practical nor desirable, due to memory constraints and decoder delay. Research has shown that a traceback depth of K x 5 is sufficient for Viterbi decoding with the type of codes we have been discussing. Any deeper traceback increases decoding delay and decoder memory requirements, while not significantly improving the performance of the decoder. The exception is punctured codes, which I'll describe later. They require deeper traceback to reach their final performance limits.

To implement a Viterbi decoder in software, the first step is to build some data structures around which the decoder algorithm will be implemented. These data structures are best implemented as arrays. The primary six arrays that we need for the Viterbi decoder are as follows:

- A copy of the convolution encoder next state table, the state transition table of the encoder. The dimensions of this table (rows x columns) are $2^{(K - 1)}$ x 2^k. This array needs to be initialized before starting the decoding process.
- A copy of the convolution encoder output table. The dimensions of this table are $2^{(K - 1)}$ x 2^k. This array needs to be initialized before starting the decoding process.
- An array (table) showing for each convolution encoder current state and next state, what input value (0 or 1) would produce the next state, given the current state. We'll call this array the input table. Its dimensions are $2^{(K - 1)}$ x $2^{(K - 1)}$. This array needs to be initialized before starting the decoding process.
- An array to store state predecessor history for each encoder state for up to K x 5 + 1 received channel symbol pairs. We'll call this table the state history table. The dimensions of this array are $2^{(K - 1)}$ x (K x 5 + 1). This array does not need to be initialized before starting the decoding process.
- An array to store the accumulated error metrics for each state computed using the add-compare-select operation. This array will be called the accumulated error metric array. The dimensions of this array are $2^{(K - 1)}$ x 2. This array does not need to be initialized before starting the decoding process.
- An array to store a list of states determined during traceback (term to be explained below). It is called the state sequence array. The dimensions of this array are (K x 5) + 1. This array does not need to be initialized before starting the decoding process.

Before getting into the example source code, for purposes of completeness, I want to talk briefly about other rates of convolution codes that can be decoded with Viterbi decoders. Earlier, I mentioned punctured codes, which are a common way of achieving higher code rates, i.e. larger ratios of k to n. Punctured codes are created by first encoding data using a rate 1/n encoder such as the example encoder described in this tutorial, and then deleting some of the channel symbols at the output of the encoder. The process of deleting some of the channel output symbols is called puncturing. For example, to create a rate 3/4 code from the rate 1/2 code described in this tutorial, one would simply delete channel symbols in accordance with the following puncturing pattern:

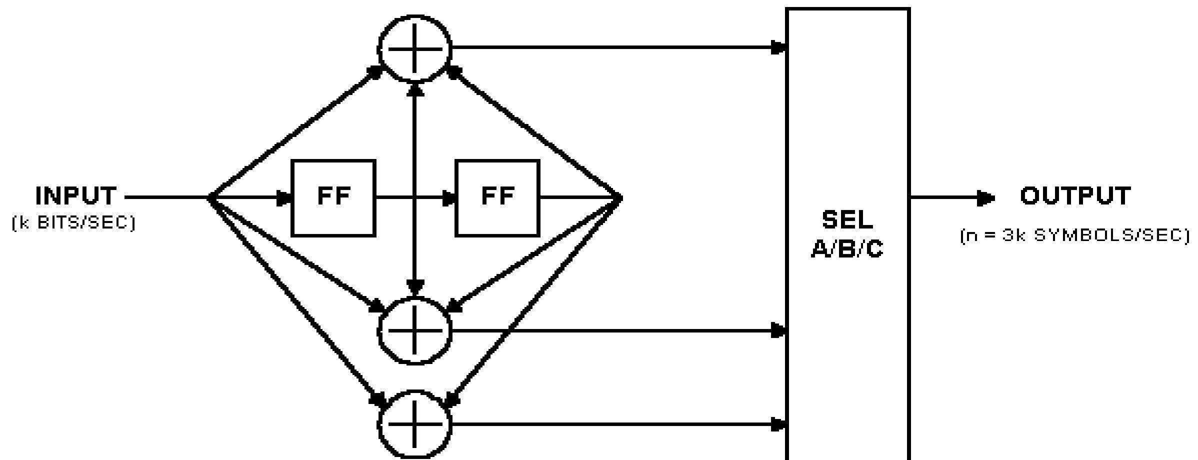
1	0	1
1	1	0

Where a one indicates that a channel symbol is to be transmitted, and a zero indicates that a channel symbol is to be deleted. To see how this make the rate be 3/4, think of each column of the above table as corresponding to a bit input to the encoder, and each one in the table as corresponding to an output channel symbol. There are three columns in the table, and four ones. You can even create a rate 2/3 code using a rate 1/2 encoder with the following puncturing pattern:

1	1
1	0

which has two columns and three ones.

To decode a punctured code, one must substitute null symbols for the deleted symbols at the input to the Viterbi decoder. Null symbols can be symbols quantized to levels corresponding to weak ones or weak zeroes, or better, can be special flag symbols that when processed by the ACS circuits in the decoder, result in no change to the accumulated error metric from the previous state. Of course, n does not have to be equal to two. For example, a rate 1/3, K = 3, (7, 7, 5) code can be encoded using the encoder shown below:



This encoder has three modulo-two adders, so for each input bit, it can produce three channel symbol outputs. Of course, with suitable puncturing patterns, you can create higher-rate codes using this encoder as well.

I don't have good data to share with you right now about the traceback depth requirements for Viterbi decoders for punctured codes. I have been told that instead of K x 5, depths of K x 7, K x 9, or even more are required to reach the point of diminishing returns. This would be a good topic around which to design some experiments using a modified version of the example simulation code I provide.

Return

Examples

Running for K = 5

Polynomials: (35, 23) in Octal
Upper Stream: (1 1 1 0 1); => Tab bits: 1101
Lower Stream: (1 0 0 1 1); => Tab bits: 0011

11101		Input	X	X		X	Output	Symbols	
10011					X	X			
I							=>	II	
0		0	0	0	0	0	0	0	0
1		1	0	0	0	0	0	1	1
0		0	1	0	0	0	0	1	0
1		1	0	1	0	0	0	0	1
1		1	1	0	1	0	0	0	0
1		1	1	1	0	1	0	0	0
0		0	1	1	1	0	1	0	1
0		0	0	1	1	1	0	0	0
1		1	0	0	1	1	1	0	1
0		0	1	0	0	1	1	0	1
1		1	0	1	0	0	1	0	1
0		0	1	0	1	0	0	1	1
0		0	0	1	0	1	0	0	1
0		0	0	0	1	0	1	0	1
1		1	0	0	0	1	0	0	0
0		0	1	0	0	0	1	1	0
1		1	0	1	0	0	0	0	1
0		0	1	0	1	0	0	1	1
1		1	0	1	0	1	0	1	0
1		1	1	0	1	0	1	0	0
1		1	1	1	0	1	0	0	0
0		0	1	1	1	0	1	0	1
0		0	0	1	1	1	0	0	0
1		1	0	0	1	1	1	0	1
0		0	1	0	0	1	1	0	1
1		1	0	1	0	0	1	0	1
0		0	1	0	1	0	0	1	1
0		0	0	1	0	1	0	0	1
0		0	0	0	1	0	1	0	1
1		1	0	0	0	1	0	0	0
1		1	1	0	0	0	1	0	1
0		0	1	1	0	0	0	0	0
0		0	0	1	1	0	0	1	1
0		0	0	0	1	1	0	1	0
0		0	0	0	0	1	1	1	1
0		0	0	0	0	0	1	0	0

Shift registers define the “state” of the virtual Viterbi machine. At each clock step they shift leaving a vacant place that is occupied by the “input bit”, the bit to be coded as a two bits symbol. Bits of the symbols must satisfy at each step the following relations:

Input bit + Tab bits Upper Polynomial = Upper bit
Input bit + Tab bits Lower Polynomial = Lower bit

Where the operator + stands for XOR. This is the procedure for a 1:2 Convolution, and in general for a 1:m Convolution we will have m bits symbols, each one staysfuing a similar relation, namely

Input bit + Tab bits j-Polynomial Tab bits = j-bit

Other Polynomials set:

Note: only changes Upper Stream Polynomial,
(11001) => (31 in octal),
(10011) => (23 in octal)
Warning: as used in our Prototype (programmed in DSP board)

11001			X			X			
10011					X	X			
I							=>	**II**	
0	1	0	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	1	1
0	3	0	1	0	0	0	0	1	0
1	4	1	0	1	0	0	0	1	1
1	5	1	1	0	1	0	0	0	0
1	6	1	1	1	0	1	0	1	0
0	7	0	1	1	1	0	1	1	1
0	8	0	0	1	1	1	0	1	0
1	9	1	0	0	1	1	1	0	1
0	10	0	1	0	0	1	1	0	1
1	11	1	0	1	0	0	1	1	1
0	12	0	1	0	1	0	0	1	1
0	13	0	0	1	0	1	0	1	1
0	14	0	0	0	1	0	1	0	1
1	15	1	0	0	0	1	0	0	0
0	16	0	1	0	0	0	1	1	0
1	17	1	0	1	0	0	0	1	1
0	18	0	1	0	1	0	0	1	1
1	19	1	0	1	0	1	0	0	0
1	20	1	1	0	1	0	1	0	0
1	21	1	1	1	0	1	0	1	0
0	22	0	1	1	1	0	1	1	1
0	23	0	0	1	1	1	0	1	0
1	24	1	0	0	1	1	1	0	1
0	25	0	1	0	0	1	1	0	1
1	26	1	0	1	0	0	1	1	1
0	27	0	1	0	1	0	0	1	1
0	28	0	0	1	0	1	0	1	1
0	29	0	0	0	1	0	1	0	1
1	30	1	0	0	0	1	0	0	0
1	31	1	1	0	0	0	1	0	1
0	32	0	1	1	0	0	0	1	0
0	33	0	0	1	1	0	0	0	1
0	34	0	0	0	1	1	0	1	0
0	35	0	0	0	0	1	1	1	1
0	36	0	0	0	0	0	1	0	0

Return

Convolution seen as from a States Machine

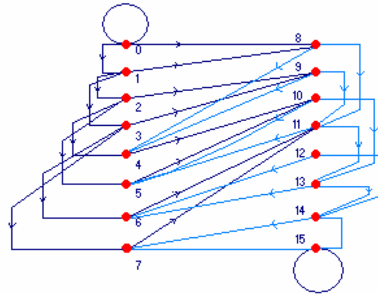
We may see that at large in each step j the symbols value depends on the pair [S(j), input j-bit]. As a corollary the whole sequence could be obtained by clocking stepwise an equivalent Convolution “states machine”. The state transformation at its turn depends on convolution polynomials and they could be easily tabulated as follows.

States Changes

[B0 B1 B2 B3]	Input bit = 0	Input bit = 1
0000	0000	1000
0001	0000	1000
0010	0001	1001
0011	0001	1001
0100	0010	1010
0101	0010	1010
0110	0011	1011
0111	0011	1011
1000	0100	1100
1001	0100	1100
1010	0101	1101
1011	0101	1101
1100	0110	1110
1101	0110	1110
1110	0111	1111
1111	0111	1111

Symbols Generation

g(1) [**B0** B1 B2 **B3**] g(2) [B0 B1 **B2 B3**]	Input bit = 0	Input bit = 1
0000	00	11
0001	11	00
0010	01	10
0011	10	01
0100	00	11
0101	11	00
0110	01	10
0111	10	01
1000	10	01
1001	01	10
1010	11	00
1011	00	11
1100	10	01
1101	01	10
1110	11	00
1111	00	11



[Return](#)

Appendices

i) Going Backward in case of No errors condition

Fast way utility

Using again our example we have the following symbols:

00 11 10 00 01 10 01 11 11 10 00 10 11 00 11 10 11 00

And the recursion direct algorithm for F´ gives us:

Time step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Symbol	00	11	10	00	01	10	01	11	11	10	00	10	11	00	11	10	11	00
State	00	00	10	01	10	11	11	01	00	10	01	10	01	00	00	10	01	00
Inferred bit	0	1	0	1	1	1	0	0	1	0	1	0	0	0	1	0	0	0

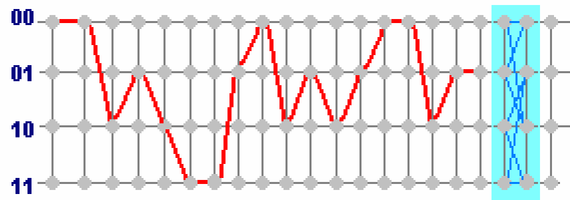
0 1 0 1 1 1 0 0 1 0 1 0 0 0 1 0 0 0

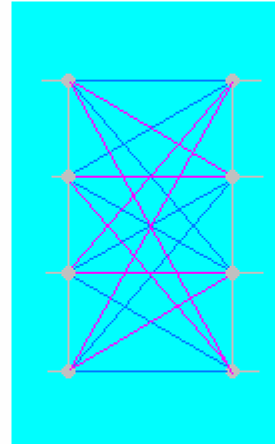
The algorithm involves two table lookups:

a) Given the symbol (initially 00), and being the initial state equal to 00, going to the **Output Symbol Table** we “infer” that the bit “cause” should be 0. Otherwise an error accounts.

b) Now we proceed to change the state of the convolution “machine” going to the **Next State Table** looking for the recently inferred bit (0), and it said that next state must be 00 again.

We iterate this procedure along t, symbol per symbol. In our example, as next state is 00 and generated symbols “was” 11, the first table tell us that next inferred bit should be 1, and this bit change state from 00 to 10. And so on and so forth.





Light Blue Lines: permitted transitions
Fuchsia Lines: Forbidden Transitions

[Return]

II) Another Sources of Viterbi related soft:

Library of Comm issues in C++ **SPUC** Org: http://spuc.sourceforge.net/
http://spuc.sourceforge.net/iterbi Class (Full): http://spuc.sourceforge.net/

SPUC::viterbi Class Reference
[Communication Classes, Forward Error Correcting Codes]

A Viterbi decoder (for DVB). More...

#include <**viterbi.h**>

Public Member Functions

	viterbi ()
void	**reset** ()
bool	**clock** (long value)
bool	**decode** (long s0, long s1)
void	**minimize_metrics** ()
bool	**depuncture** (const long steal, long soft_in)

Public Attributes

bool	**decoded**
bool	**enable_output**
bool	**output_ready**
long	**prev_value**

viterbi_state	**state0** [64]
viterbi_state	**state1** [64]
viterbi_state *	**state**
viterbi_state *	**next**
int	**bitcnt**
int	**beststate**
long	**depuncture_bit_number**
bool	**phase**

Detailed Description

A Viterbi decoder (for DVB).
Author: Tony Kirke, Copyright(c) 2001

Constructor & Destructor Documentation

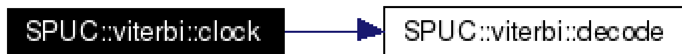
SPUC::viterbi::viterbi( ) [inline]

Here is the call graph for this function:

Member Function Documentation

bool SPUC::viterbi::clock(long *value*) [inline]

Here is the call graph for this function:



bool viterbi::decode(long *s0*,
long *s1*
)

bool viterbi::depuncture(const long *steal*,
long *soft_in*
)

void SPUC::viterbi::minimize_metrics( ) [inline]

void SPUC::viterbi::reset(void) [inline]

Member Data Documentation

int SPUC::viterbi::beststate

int SPUC::viterbi::bitcnt

bool SPUC::viterbi::decoded

long SPUC::viterbi::depuncture_bit_number

bool SPUC::viterbi::enable_output

viterbi_state * SPUC::viterbi::next

bool SPUC::viterbi::output_ready

bool SPUC::viterbi::phase

long SPUC::viterbi::prev_value

viterbi_state * SPUC::viterbi::state

viterbi_state SPUC::viterbi::state0[64]

viterbi_state SPUC::viterbi::state1[64]

The documentation for this class was generated from the following files:

- comm/**viterbi.h**

comm/**viterbi.cpp**

Return

III) Convolution - Viterbi Source Codes

Source: http://home.netcom.com/%7Echip.f/viterbi/tutorial.html

It comprises the following parts:

- Header
- Convolution Encoder
- Data Generator
- Test Driver (Main)
- Channel Simulator
- Viterbi Decoder
- Experiences

Header:

```
#define K 3             /* constraint length */
#define TWOTOTHEM 4      /* 2^(K - 1) -- change as required */
#define PI 3.141592654   /* circumference of circle divided by diameter */
#define MSG_LEN 100000l  /* how many bits in each test message */
#define DOENC           /* test with convolution encoding/Viterbi decoding */
#undef  DONOENC          /* test with no coding */
#define LOESN0 0.0       /* minimum Es/No at which to test */
#define HIESN0 3.5       /* maximum Es/No at which to test */
#define ESN0STEP 0.5     /* Es/No increment for test driver */
```

Convolution Encoder:

```
/* CONVOLUTIONAL ENCODER                                  */
/* Copyright (c) 1999, Spectrum Applications, Derwood, MD, USA     */
/* All rights reserved                                    */
/* Version 2.0 Last Modified 1999.02.17                   */

#include <alloc.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include "vdsim.h"

void cnv_encd(int g[2][K], long input_len, int *in_array, int *out_array) {

    int m;                  /* K - 1 */
    long t, tt;             /* bit time, symbol time */
    int j, k;               /* loop variables */
    int *unencoded_data;    /* pointer to data array */
    int shift_reg[K];       /* the encoder shift register */
    int sr_head;            /* index to the first elt in the sr */
    int p, q;               /* the upper and lower xor gate outputs */

    m = K - 1;

    /* allocate space for the zero-padded input data array */
    unencoded_data = malloc( (input_len + m) * sizeof(int) );
    if (unencoded_data == NULL) {
        printf("\ncnv_encd.c: Can't allocate enough memory for unencoded data!  Aborting...");
        exit(1);
    }
    /* read in the data and store it in the array */
    for (t = 0; t < input_len; t++)
        *(unencoded_data + t) = *(in_array + t);

    /* zero-pad the end of the data */
    for (t = 0; t < m; t++) {
        *(unencoded_data + input_len + t) = 0;
    }
```

```
    /* Initialize the shift register */
    for (j = 0; j < K; j++) {
        shift_reg[j] = 0;
    }

    /* To try to speed things up a little, the shift register will be operated
       as a circular buffer, so it needs at least a head pointer. It doesn't
       need a tail pointer, though, since we won't be taking anything out of
       it--we'll just be overwriting the oldest entry with the new data. */
    sr_head = 0;

    /* initialize the channel symbol output index */
    tt = 0;

    /* Now start the encoding process */
    /* compute the upper and lower mod-two adder outputs, one bit at a time */
    for (t = 0; t < input_len + m; t++) {
        shift_reg[sr_head] = *( unencoded_data + t );
        p = 0;
        q = 0;
        for (j = 0; j < K; j++) {
            k = (j + sr_head) % K;
            p ^= shift_reg[k] & g[0][j];
            q ^= shift_reg[k] & g[1][j];
        }

        /* write the upper and lower xor gate outputs as channel symbols */
        *(out_array + tt) = p;
        tt = tt + 1;
        *(out_array + tt) = q;
        tt = tt + 1;


        sr_head -= 1;    /* equivalent to shifting everything right one place */
        if (sr_head < 0) /* but make sure we adjust pointer modulo K */
            sr_head = m;
    }

    /* free the dynamically allocated array */
    free(unencoded_data);

}
```

Data Generator:

```
/* 0/1 DATA GENERATOR                                          */
/* Copyright (c) 1999, Spectrum Applications, Derwood, MD, USA      */
/* All rights reserved                                    */
/* Version 2.0 Last Modified 1999.02.17                        */
 #include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "vdsim.h"
void gen01dat( long data_len, int *out_array ) {
    long t;           /* time */
    /* re-seed the random number generator */
    randomize();
    /* generate the random data and write it to the output array */
    for (t = 0; t < data_len; t++)
        *( out_array + t ) = (int)( rand() / (RAND_MAX / 2) > 0.5 );
}
```

Test Driver:

```
/* Soft decision Viterbi Decoder Test Driver                    */
/* Copyright (c) 1999, Spectrum Applications, Derwood, MD, USA      */
/* All rights reserved                                    */
```

```
/* Version 2.0 Last Modified 1999.02.20                          */

#include <alloc.h>
#include <conio.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "vdsim.h"

extern void gen01dat(long data_len, int *out_array);
extern void cnv_encd(int g[2][K], long data_len, int *in_array, int *out_array);
extern void addnoise(float es_ovr_n0, long data_len, int *in_array, float *out_array);
extern void sdvd(int g[2][K], float es_ovr_n0, long channel_length,
          float *channel_output_vector, int *decoder_output_matrix);

void testsdvd(void) {

   long iter, t, msg_length, channel_length; /* loop variables, length of I/O files */

   int *onezer;
   int *encoded;                 /* original, encoded, & decoded data arrays */
   int *sdvdout;

   int start;

   float *splusn;                /* noisy data array */

   int i_rxdata, m;              /* int rx data , m = K - 1*/
   float es_ovr_n0, number_errors_encoded, number_errors_unencoded,
         e_threshold, ue_threshold, e_ber, ue_ber; /* various statistics */

   #if K == 3       /* polynomials for K = 3 */
   int g[2][K] = {{1, 1, 1},     /* 7 */
               {1, 0, 1}};    /* 5 */
   #endif

   #if K == 5       /* polynomials for K = 5 */
   int g[2][K] = {{1, 1,  1, 0, 1},  /* 35 */
               {1, 0,  0, 1, 1}}; /* 23 */
   #endif

   #if K == 7       /* polynomials for K = 7 */
   int g[2][K] = {{1,  1, 1, 1,  0, 0, 1},  /* 171 */
               {1,  0, 1, 1,  0, 1, 1}}; /* 133 */
   #endif

   #if K == 9       /* polynomials for K = 9 */
   int g[2][K] = {{1, 1, 1,  1, 0, 1,  0, 1, 1}, /* 753 */
               {1, 0, 1,  1, 1, 0,  0, 0, 1}}; /* 561 */
   #endif

   clrscr();

   printf("\nK = %d", K);

   #if K == 3
   printf("\ng1 = %d%d%d", g[0][0], g[0][1], g[0][2] );
   printf("\ng2 = %d%d%d\n", g[1][0], g[1][1], g[1][2] );
   #endif

   #if K == 5
   printf("\ng1 = %d%d %d%d%d", g[0][0], g[0][1], g[0][2], g[0][3], g[0][4] );
   printf("\ng2 = %d%d %d%d%d\n", g[1][0], g[1][1], g[1][2], g[1][3], g[1][4] );
   #endif

   #if K == 7
   printf("\ng1 = %d %d%d%d %d%d%d", g[0][0], g[0][1], g[0][2], g[0][3], g[0][4],
                 g[0][5], g[0][6] );
```

```
 printf("\ng2 = %d %d%d%d %d%d%d\n", g[1][0], g[1][1], g[1][2], g[1][3], g[1][4],
             g[1][5], g[1][6] );
#endif

#if K == 9
printf("\ng1 = %d%d%d %d%d%d %d%d%d", g[0][0], g[0][1], g[0][2], g[0][3], g[0][4],
             g[0][5], g[0][6], g[0][7], g[0][8] );
printf("\ng2 = %d%d%d %d%d%d %d%d%d\n", g[1][0], g[1][1], g[1][2], g[1][3], g[1][4],
             g[1][5], g[1][6], g[1][7], g[1][8] );
#endif

m = K - 1;
msg_length = MSG_LEN;
channel_length = ( msg_length + m ) * 2;

onezer = malloc( msg_length * sizeof( int ) );
if (onezer == NULL) {
   printf("\n testsdvd.c:  error allocating onezer array, aborting!");
   exit(1);
}

encoded = malloc( channel_length * sizeof(int) );
if (encoded == NULL) {
   printf("\n testsdvd.c:  error allocating encoded array, aborting!");
   exit(1);
}

splusn = malloc( channel_length * sizeof(float) );
if (splusn == NULL) {
   printf("\n testsdvd.c:  error allocating splusn array, aborting!");
   exit(1);
}

sdvdout = malloc( msg_length * sizeof( int ) );
if (sdvdout == NULL) {
   printf("\n testsdvd.c:  error allocating sdvdout array, aborting!");
   exit(1);
}


for (es_ovr_n0 = LOESN0; es_ovr_n0 <= HIESN0; es_ovr_n0 += ESN0STEP) {

   start = time(NULL);

   number_errors_encoded = 0.0;
   e_ber = 0.0;
   iter = 0;

   #ifdef DOENC
   if (es_ovr_n0 <= 9)
      e_threshold = 100; /* +/- 20% */
   else
      e_threshold = 20; /* +/- 100 % */

   while (number_errors_encoded < e_threshold) {
      iter += 1;


       /*printf("Generating one-zero data\n");*/
      gen01dat(msg_length, onezer);

      /*printf("Convolutionally encoding the data\n");*/
      cnv_encd(g, msg_length, onezer, encoded);

      /*printf("Adding noise to the encoded data\n");*/
      addnoise(es_ovr_n0, channel_length, encoded, splusn);


        /*printf("Decoding the BSC data\n");*/
        sdvd(g, es_ovr_n0, channel_length, splusn, sdvdout);
```

```
        for (t = 0; t < msg_length; t++) {
            if ( *(onezer + t) != *(sdvdout + t) ) {
               /*printf("\n error occurred at location %ld", t);*/
                   number_errors_encoded += 1;
            } /* end if */
        } /* end t for-loop */

       if (kbhit()) exit(0);
        /*printf("\nDone!");*/

    }

    e_ber = number_errors_encoded / (msg_length * iter);

    printf("\nThe elapsed time was %d seconds for %d iterations",
              time(NULL) - start, iter);
    #endif

    number_errors_unencoded = 0.0;
    ue_ber = 0.0;
    iter = 0;

    #ifdef DONOENC
    if (es_ovr_n0 <= 12)
        ue_threshold = 100;
    else
        ue_threshold = 20;


    while (number_errors_unencoded < ue_threshold) {
        iter += 1;

        /*printf("Generating one-zero data\n");*/
        gen01dat(msg_length, onezer);

        /*printf("Adding noise to the unencoded data\n");*/
        addnoise(es_ovr_n0, msg_length, onezer, splusn);

        for (t = 0; t < msg_length; t++) {

            if ( *(splusn + t) < 0.0 )
                i_rxdata = 1;
            else
                i_rxdata = 0;

            if ( *(onezer + t) != i_rxdata )
                number_errors_unencoded += 1;
        }

        if (kbhit()) exit(0);
        /*printf("\nDone!");*/

    }

    ue_ber = number_errors_unencoded / (msg_length * iter);
    #endif

    printf("\nAt %1.1fdB Es/No, ", es_ovr_n0);

    #ifdef DOENC
    printf("the e_ber was %1.1e ", e_ber);
    #ifdef DONOENC
    printf("and ");
    #endif
    #endif
    #ifdef DONOENC
    printf("the ue_ber was %1.1e", ue_ber);
    #endif
}
```

```
    free(onezer);
    free(encoded);
    free(splusn);
    free(sdvdout);

    while ( !kbhit() ) {
    }

    exit(0);
}
```

Channel Simulator:

```
/* BPSK BINARY SYMMETRIC CHANNEL SIMULATOR                          */
/* Copyright (c) 1999, Spectrum Applications, Derwood, MD, USA      */
/* All rights reserved                                              */
/* Version 2.0 Last Modified 1999.02.17                             */

#include <alloc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "vdsim.h"

float gngauss(float mean, float sigma);

void addnoise(float es_ovr_n0, long channel_len, int *in_array, float *out_array) {

    long t;
    float mean, es, sn_ratio, sigma, signal;

    /* given the desired Es/No (for BPSK, = Eb/No - 3 dB), calculate the
    standard deviation of the additive white gaussian noise (AWGN). The
    standard deviation of the AWGN will be used to generate Gaussian random
    variables simulating the noise that is added to the signal. */

    mean = 0;
    es = 1;
    sn_ratio = (float) pow(10, ( es_ovr_n0 / 10) );
    sigma =  (float) sqrt (es / ( 2 * sn_ratio ) );

    /* now transform the data from 0/1 to +1/-1 and add noise */
    for (t = 0; t < channel_len; t++) {

        /*if the binary data value is 1, the channel symbol is -1; if the
        binary data value is 0, the channel symbol is +1. */
        signal = 1 - 2 * *( in_array + t );

        /*  now generate the gaussian noise point, add it to the channel symbol,
            and output the noisy channel symbol */

        *( out_array + t ) = signal + gngauss(mean, sigma);
    }

}

float gngauss(float mean, float sigma) {

    /* This uses the fact that a Rayleigh-distributed random variable R, with
    the probability distribution F(R) = 0 if R < 0 and F(R) =
    1 - exp(-R^2/2*sigma^2) if R >= 0, is related to a pair of Gaussian
    variables C and D through the transformation C = R * cos(theta) and
    D = R * sin(theta), where theta is a uniformly distributed variable
    in the interval (0, 2*pi()). From Contemporary Communication Systems
    USING MATLAB(R), by John G. Proakis and Masoud Salehi, published by
```

```
   PWS Publishing Company, 1998, pp 49-50. This is a pretty good book. */

   double u, r;           /* uniform and Rayleigh random variables */

   /* generate a uniformly distributed random number u between 0 and 1 - 1E-6*/
   u = (double)_lrand() / LRAND_MAX;
   if (u == 1.0) u = 0.999999999;

   /* generate a Rayleigh-distributed random number r using u */
   r = sigma * sqrt( 2.0 * log( 1.0 / (1.0 - u) ) );

   /* generate another uniformly-distributed random number u as before*/
   u = (double)_lrand() / LRAND_MAX;
   if (u == 1.0) u = 0.999999999;

   /* generate and return a Gaussian-distributed random number using r and u */
   return( (float) ( mean + r * cos(2 * PI * u) ) );
}
```

VITERBI DECODER:

```
/* SOFT-DECISION VITERBI DECODER                              */
/* Copyright (c) 1999, 2001 Spectrum Applications, Derwood, MD, USA */
/* All rights reserved                                 */
/* Version 2.2 Last Modified 2001.11.28                       */

#include <alloc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <values.h>
 #include "vdsim.h"

#undef SLOWACS
#define FASTACS
#undef NORM
#define MAXMETRIC 128

void deci2bin(int d, int size, int *b);
int bin2deci(int *b, int size);
int nxt_stat(int current_state, int input, int *memory_contents);
void init_quantizer(void);
void init_adaptive_quant(float es_ovr_n0);
int soft_quant(float channel_symbol);
int soft_metric(int data, int guess);

int quantizer_table[256];

void sdvd(int g[2][K], float es_ovr_n0, long channel_length,
          float *channel_output_vector, int *decoder_output_matrix) {

   int i, j, l, ll;                       /* loop variables */
   long t;                                /* time */
   int memory_contents[K];                /* input + conv. encoder sr */
   int input[TWOTOTHEM][TWOTOTHEM];       /* maps current/nxt sts to input */
   int output[TWOTOTHEM][2];              /* gives conv. encoder output */
   int nextstate[TWOTOTHEM][2];           /* for current st, gives nxt given input */
   int accum_err_metric[TWOTOTHEM][2];    /* accumulated error metrics */
   int state_history[TWOTOTHEM][K * 5 + 1];  /* state history table */
   int state_sequence[K * 5 + 1];         /* state sequence list */

   int *channel_output_matrix;            /* ptr to input matrix */
   int binary_output[2];                  /* vector to store binary enc output */
   int branch_output[2];                  /* vector to store trial enc output */
   int m, n, number_of_states, depth_of_trellis, step, branch_metric,
      sh_ptr, sh_col, x, xx, h, hh, next_state, last_stop; /* misc variables */
/* ************************************************************************ */
    /* n is 2^1 = 2 for rate 1/2 */
   n = 2;
```

```
/* m (memory length) = K - 1 */
m = K - 1;

/* number of states = 2^(K - 1) = 2^m for k = 1 */
number_of_states = (int) pow(2, m);

/* little degradation in performance achieved by limiting trellis depth
   to K * 5--interesting to experiment with smaller values and measure
   the resulting degradation. */
depth_of_trellis = K * 5;

/* initialize data structures */
for (i = 0; i < number_of_states; i++) {
    for (j = 0; j < number_of_states; j++)
        input[i][j] = 0;

    for (j = 0; j < n; j++) {
        nextstate[i][j] = 0;
        output[i][j] = 0;
    }

    for (j = 0; j <= depth_of_trellis; j++) {
        state_history[i][j] = 0;
    }

    /* initial accum_error_metric[x][0] = zero */
    accum_err_metric[i][0] = 0;
    /* by setting accum_error_metric[x][1] to MAXINT, we don't need a flag */
    /* so I don't get any more questions about this:  */
    /* MAXINT is simply the largest possible integer, defined in values.h */
    accum_err_metric[i][1] = MAXINT;

}

/* generate the state transition matrix, output matrix, and input matrix
   - input matrix shows how FEC encoder bits lead to next state
   - next_state matrix shows next state given current state and input bit
   - output matrix shows FEC encoder output bits given current presumed
   encoder state and encoder input bit--this will be compared to actual
   received symbols to determine metric for corresponding branch of trellis
*/

for (j = 0; j < number_of_states; j++) {
    for (l = 0; l < n; l++) {
        next_state = nxt_stat(j, l, memory_contents);
        input[j][next_state] = l;

        /* now compute the convolutional encoder output given the current
           state number and the input value */
        branch_output[0] = 0;
        branch_output[1] = 0;

        for (i = 0; i < K; i++) {
            branch_output[0] ^= memory_contents[i] & g[0][i];
            branch_output[1] ^= memory_contents[i] & g[1][i];
        }

        /* next state, given current state and input */
        nextstate[j][l] = next_state;
        /* output in decimal, given current state and input */
        output[j][l] = bin2deci(branch_output, 2);

    } /* end of l for loop */

} /* end of j for loop */

#ifdef DEBUG
printf("\nInput:");
for (j = 0; j < number_of_states; j++) {
```

```
        printf("\n");
        for (l = 0; l < number_of_states; l++)
            printf("%2d ", input[j][l]);
    } /* end j for-loop */

    printf("\nOutput:");
    for (j = 0; j < number_of_states; j++) {
        printf("\n");
        for (l = 0; l < n; l++)
            printf("%2d ", output[j][l]);
    } /* end j for-loop */

    printf("\nNext State:");
    for (j = 0; j < number_of_states; j++) {
        printf("\n");
        for (l = 0; l < n; l++)
            printf("%2d ", nextstate[j][l]);
    } /* end j for-loop */
    #endif

    channel_output_matrix = malloc( channel_length * sizeof(int) );
    if (channel_output_matrix == NULL) {
        printf(
        "\nsdvd.c: Can't allocate memory for channel_output_matrix! Aborting...");
        exit(1);
    }

    /* now we're going to rearrange the channel output so it has n rows,
       and n/2 columns where each row corresponds to a channel symbol for
       a given bit and each column corresponds to an encoded bit */
    channel_length = channel_length / n;

    /* interesting to compare performance of fixed vs adaptive quantizer */
    /* init_quantizer(); */
    init_adaptive_quant(es_ovr_n0);

    /* quantize the channel output--convert float to short integer */
    /* channel_output_matrix = reshape(channel_output, n, channel_length) */
    for (t = 0; t < (channel_length * n); t += n) {
        for (i = 0; i < n; i++)
            *(channel_output_matrix + (t / n) + (i * channel_length) ) =
                soft_quant( *(channel_output_vector + (t + i) ) );
    } /* end t for-loop */

/* ************************************************************************** */

    /* End of setup. Start decoding of channel outputs with forward
        traversal of trellis!  Stop just before encoder-flushing bits.  */
    for (t = 0; t < channel_length - m; t++) {

       if (t <= m)
           /* assume starting with zeroes, so just compute paths from all-zeroes state */
           step = pow(2, m - t * 1);
        else
            step = 1;

        /* we're going to use the state history array as a circular buffer so
           we don't have to shift the whole thing left after each bit is
           processed so that means we need an appropriate pointer */
        /* set up the state history array pointer for this time t */
        sh_ptr = (int) ( ( t + 1 ) % (depth_of_trellis + 1) );

        /* repeat for each possible state */
        for (j = 0; j < number_of_states; j+= step) {
            /* repeat for each possible convolutional encoder output n-tuple */
            for (l = 0; l < n; l++) {
                branch_metric = 0;

                /* compute branch metric per channel symbol, and sum for all
                    channel symbols in the convolutional encoder output n-tuple */
```

```
            #ifdef SLOWACS
            /* convert the decimal representation of the encoder output to binary */
            deci2bin(output[j][l], n, binary_output);

            /* compute branch metric per channel symbol, and sum for all
               channel symbols in the convolutional encoder output n-tuple */
            for (ll = 0; ll < n; ll++) {
               branch_metric = branch_metric + soft_metric( *(channel_output_matrix +
               ( ll * channel_length + t )), binary_output[ll] );

            } /* end of 'll' for loop */
            #endif

            #ifdef FASTACS
            /* this only works for n = 2, but it's fast! */

            /* convert the decimal representation of the encoder output to binary */
            binary_output[0] = ( output[j][l] & 0x00000002 ) >> 1;
            binary_output[1] = output[j][l] & 0x00000001;

            /* compute branch metric per channel symbol, and sum for all
               channel symbols in the convolutional encoder output n-tuple */
            branch_metric = branch_metric + abs( *( channel_output_matrix +
               ( 0 * channel_length + t ) ) - 7 * binary_output[0] ) +
                                  abs( *( channel_output_matrix +
               ( 1 * channel_length + t ) ) - 7 * binary_output[1] );
            #endif

            /* now choose the surviving path--the one with the smaller accumlated
               error metric... */
            if ( accum_err_metric[ nextstate[j][l] ] [1] > accum_err_metric[j][0] +
               branch_metric ) {

               /* save an accumulated metric value for the survivor state */
               accum_err_metric[ nextstate[j][l] ] [1] = accum_err_metric[j][0] +
                  branch_metric;

               /* update the state_history array with the state number of
                 the survivor */
               state_history[ nextstate[j][l] ] [sh_ptr] = j;

            } /* end of if-statement */
         } /* end of 'l' for-loop */
      } /* end of 'j' for-loop -- we have now updated the trellis */

      /* for all rows of accum_err_metric, move col 2 to col 1 and flag col 2 */
      for (j = 0; j < number_of_states; j++) {
         accum_err_metric[j][0] = accum_err_metric[j][1];
         accum_err_metric[j][1] = MAXINT;
      } /* end of 'j' for-loop */


      /* now start the traceback, if we've filled the trellis */
      if (t >= depth_of_trellis - 1) {

         /* initialize the state_sequence vector--probably unnecessary */
         for (j = 0; j <= depth_of_trellis; j++)
            state_sequence[j] = 0;

         /* find the element of state_history with the min. accum. error metric */
         /* since the outer states are reached by relatively-improbable runs
           of zeroes or ones, search from the top and bottom of the trellis in */
         x = MAXINT;

         for (j = 0; j < ( number_of_states / 2 ); j++) {

            if ( accum_err_metric[j][0] < accum_err_metric[number_of_states - 1 - j][0] ) {
               xx = accum_err_metric[j][0];
               hh = j;
```

```
            }
            else {
                xx = accum_err_metric[number_of_states - 1 - j][0];
                hh = number_of_states - 1 - j;
            }
            if ( xx < x) {
                x = xx;
                h = hh;
            }

        } /* end 'j' for-loop */

        #ifdef NORM
        /* interesting to experiment with different numbers of bits in the
           accumulated error metric--does performance decrease with fewer  bits? */
        /* if the smallest accum. error metric value is > MAXMETRIC, normalize the
           accum. errror metrics by subtracting the value of the smallest one from
           all of them (making the smallest = 0) and saturate all other metrics
           at MAXMETRIC */
        if (x > MAXMETRIC) {
            for (j = 0; j < number_of_states; j++) {
                accum_err_metric[j][0] = accum_err_metric[j][0] - x;
                if (accum_err_metric[j][0] > MAXMETRIC)
                    accum_err_metric[j][0] = MAXMETRIC;
            } /* end 'j' for-loop */
        }
        #endif

        /* now pick the starting point for traceback */
        state_sequence[depth_of_trellis] = h;

        /* now work backwards from the end of the trellis to the oldest state
           in the trellis to determine the optimal path. The purpose of this
           is to determine the most likely state sequence at the encoder
            based on what channel symbols we received. */
        for (j = depth_of_trellis; j > 0; j--) {
            sh_col = j + ( sh_ptr - depth_of_trellis );
            if (sh_col < 0)
               sh_col = sh_col + depth_of_trellis + 1;

            state_sequence[j - 1] = state_history[ state_sequence[j] ] [sh_col];
        } /* end of j for-loop */

        /* now figure out what input sequence corresponds to the state sequence
         in the optimal path */
        *(decoder_output_matrix + t - depth_of_trellis + 1) =
            input[ state_sequence[0] ] [ state_sequence[1] ];

      } /* end of if-statement */

   } /* end of 't' for-loop */

/* ************************************************************************** */

   /* now decode the encoder flushing channel-output bits */
   for (t = channel_length - m; t < channel_length; t++) {

      /* set up the state history array pointer for this time t */
      sh_ptr = (int) ( ( t + 1 ) % (depth_of_trellis + 1) );

      /* don't need to consider states where input was a 1, so determine
       what is the highest possible state number where input was 0 */
      last_stop = number_of_states / pow(2, t - channel_length + m);

      /* repeat for each possible state */
      for (j = 0; j < last_stop; j++) {

         branch_metric = 0;
         deci2bin(output[j][0], n, binary_output);
```

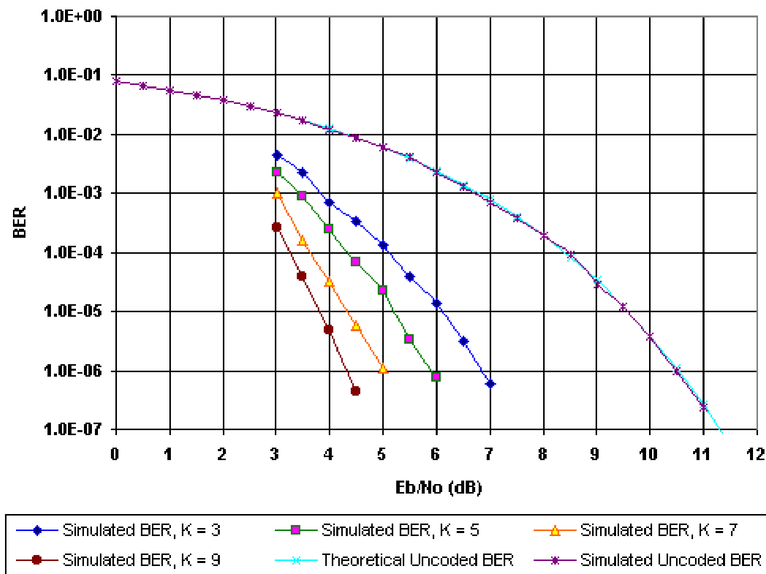
```
        /* compute metric per channel bit, and sum for all channel bits
            in the convolutional encoder output n-tuple */
        for (ll = 0; ll < n; ll++) {
            branch_metric = branch_metric + soft_metric( *(channel_output_matrix +
            (ll * channel_length + t)), binary_output[ll] );
        } /* end of 'll' for loop */

        /* now choose the surviving path--the one with the smaller total
            metric... */
        if ( (accum_err_metric[ nextstate[j][0] ][1] > accum_err_metric[j][0] +
            branch_metric) /*|| flag[ nextstate[j][0] ] == 0*/) {

            /* save a state metric value for the survivor state */
            accum_err_metric[ nextstate[j][0] ][1] = accum_err_metric[j][0] +
                branch_metric;

            /* update the state_history array with the state number of
                the survivor */
            state_history[ nextstate[j][0] ][sh_ptr] = j;

        } /* end of if-statement */

    } /* end of 'j' for-loop */

    /* for all rows of accum_err_metric, swap columns 1 and 2 */
    for (j = 0; j < number_of_states; j++) {
        accum_err_metric[j][0] = accum_err_metric[j][1];
        accum_err_metric[j][1] = MAXINT;
    } /* end of 'j' for-loop */

    /* now start the traceback, if i >= depth_of_trellis - 1*/
    if (t >= depth_of_trellis - 1) {

        /* initialize the state_sequence vector */
        for (j = 0; j <= depth_of_trellis; j++) state_sequence[j] = 0;

        /* find the state_history element with the minimum accum. error metric */
        x = accum_err_metric[0][0];
        h = 0;
        for (j = 1; j < last_stop; j++) {
            if (accum_err_metric[j][0] < x) {
                x = accum_err_metric[j][0];
                h = j;
            } /* end if */
        } /* end 'j' for-loop */

        #ifdef NORM
        /* if the smallest accum. error metric value is > MAXMETRIC, normalize the
           accum. errror metrics by subtracting the value of the smallest one from
           all of them (making the smallest = 0) and saturate all other metrics
           at MAXMETRIC */
        if (x > MAXMETRIC) {
            for (j = 0; j < number_of_states; j++) {
                accum_err_metric[j][0] = accum_err_metric[j][0] - x;
                if (accum_err_metric[j][0] > MAXMETRIC) {
                 accum_err_metric[j][0] = MAXMETRIC;
                } /* end if */
            } /* end 'j' for-loop */
        }
        #endif

        state_sequence[depth_of_trellis] = h;

        /* now work backwards from the end of the trellis to the oldest state
            in the trellis to determine the optimal path. The purpose of this
            is to determine the most likely state sequence at the encoder
            based on what channel symbols we received. */
        for (j = depth_of_trellis; j > 0; j--) {

            sh_col = j + ( sh_ptr - depth_of_trellis );
```

```
            if (sh_col < 0)
               sh_col = sh_col + depth_of_trellis + 1;

            state_sequence[j - 1] = state_history[ state_sequence[j] ][sh_col];
         } /* end of j for-loop */

         /* now figure out what input sequence corresponds to the
            optimal path */

          *(decoder_output_matrix + t - depth_of_trellis + 1) =
            input[ state_sequence[0] ][ state_sequence[1] ];

      } /* end of if-statement */
   } /* end of 't' for-loop */

   for (i = 1; i < depth_of_trellis - m; i++)
      *(decoder_output_matrix + channel_length - depth_of_trellis + i) =
         input[ state_sequence[i] ] [ state_sequence[i + 1] ];

   /* free the dynamically allocated array storage area */
   free(channel_output_matrix);

   return;
} /* end of function sdvd */


/* ********************* END OF SDVD FUNCTION ******************************* */
/* this initializes a 3-bit soft-decision quantizer optimized for about 4 dB Eb/No.
*/
void init_quantizer(void) {

   int i;
   for (i = -128; i < -31; i++)
      quantizer_table[i + 128] = 7;
   for (i = -31; i < -21; i++)
      quantizer_table[i + 128] = 6;
   for (i = -21; i < -11; i++)
      quantizer_table[i + 128] = 5;
   for (i = -11; i < 0; i++)
      quantizer_table[i + 128] = 4;
   for (i = 0; i < 11; i++)
      quantizer_table[i + 128] = 3;
   for (i = 11; i < 21; i++)
      quantizer_table[i + 128] = 2;
   for (i = 21; i < 31; i++)
      quantizer_table[i + 128] = 1;
   for (i = 31; i < 128; i++)
      quantizer_table[i + 128] = 0;
}


/* this initializes a quantizer that adapts to Es/No */
void init_adaptive_quant(float es_ovr_n0) {

   int i, d;
   float es, sn_ratio, sigma;

   es = 1;
   sn_ratio = (float) pow(10.0, ( es_ovr_n0 / 10.0 ) );
   sigma =  (float) sqrt( es / ( 2.0 * sn_ratio ) );

   d = (int) ( 32 * 0.5 * sigma );

   for (i = -128; i < ( -3 * d ); i++)
      quantizer_table[i + 128] = 7;

   for (i = ( -3 * d ); i < ( -2 * d ); i++)
      quantizer_table[i + 128] = 6;

   for (i = ( -2 * d ); i < ( -1 * d ); i++)
```

```
        quantizer_table[i + 128] = 5;

    for (i = ( -1 * d ); i < 0; i++)
        quantizer_table[i + 128] = 4;

    for (i = 0; i < ( 1 * d ); i++)
        quantizer_table[i + 128] = 3;

    for (i = ( 1 * d ); i < ( 2 * d ); i++)
        quantizer_table[i + 128] = 2;

    for (i = ( 2 * d ); i < ( 3 * d ); i++)
        quantizer_table[i + 128] = 1;

    for (i = ( 3 * d ); i < 128; i++)
        quantizer_table[i + 128] = 0;
}


/* this quantizer assumes that the mean channel_symbol value is +/- 1,
   and translates it to an integer whose mean value is +/- 32 to address
   the lookup table "quantizer_table". Overflow protection is included.
*/
 int soft_quant(float channel_symbol) {
    int x;

    x = (int) ( 32.0 * channel_symbol );
    if (x < -128) x = -128;
    if (x > 127) x = 127;

    return(quantizer_table[x + 128]);
}


/* this metric is based on the algorithm given in Michelson and Levesque,
   page 323. */
int soft_metric(int data, int guess) {

    return(abs(data - (guess * 7)));
}


/* this function calculates the next state of the convolutional encoder, given
   the current state and the input data.  It also calculates the memory
   contents of the convolutional encoder. */
int nxt_stat(int current_state, int input, int *memory_contents) {

    int binary_state[K - 1];              /* binary value of current state */
    int next_state_binary[K - 1];         /* binary value of next state */
    int next_state;                       /* decimal value of next state */
    int i;                                /* loop variable */

    /* convert the decimal value of the current state number to binary */
    deci2bin(current_state, K - 1, binary_state);

    /* given the input and current state number, compute the next state number */
    next_state_binary[0] = input;
    for (i = 1; i < K - 1; i++)
        next_state_binary[i] = binary_state[i - 1];

    /* convert the binary value of the next state number to decimal */
    next_state = bin2deci(next_state_binary, K - 1);

    /* memory_contents are the inputs to the modulo-two adders in the encoder */
    memory_contents[0] = input;
    for (i = 1; i < K; i++)
        memory_contents[i] = binary_state[i - 1];

    return(next_state);
}
```

```
/* this function converts a decimal number to a binary number, stored
   as a vector MSB first, having a specified number of bits with leading
   zeroes as necessary */
void deci2bin(int d, int size, int *b) {
    int i;

    for(i = 0; i < size; i++)
        b[i] = 0;

    b[size - 1] = d & 0x01;

    for (i = size - 2; i >= 0; i--) {
        d = d >> 1;
        b[i] = d & 0x01;
    }
}


/* this function converts a binary number having a specified
   number of bits to the corresponding decimal number
   with improvement contributed by Bryan Ewbank 2001.11.28 */
int bin2deci(int *b, int size) {
    int i, d;

    d = 0;

    for (i = 0; i < size; i++)
        d += b[i] << (size - i - 1);

    return(d);
}
```

Simulation Results:



I obtained the results shown in this chart using the example simulation code, with the trellis depth set to K x 5, using the adaptive quantized with three-bit channel symbol quantization. For each data point, I ran the simulation until 100 errors (or possibly more) occurred. With this number of errors, I have 95% confidence that the true number of errors for the number of data bits through the simulation lies between 80 and 120. Notice how the simulation results for BER on an uncoded channel closely track the theoretical BER for an uncoded channel, which is given by the equation $P(e) = 0.5 * erfc(sqrt(E_b/N_0)) = Q(sqrt(2E_b/N_0)$. This validates the uncoded BER algorithm and the Gaussian noise generator. The coded BER results appear to agree well with those obtained by others.

The obvious next step for you to take is to start varying some of the parameters. For example, you can try different trellis depths, the fixed quantized instead of the adaptive quantized, more or fewer bits in the quantized, and so on.

Click on one of the links below to go to the beginning of that section:

Return

IV) Something about efficiency

By Steve Gardner and Tom Hardin both work with Istari Design, Inc. in San Diego. They deal with systems design and implementation of modems and FEC codecs for wireless and wireline systems. They can be reached at shg@istari-design.com and cth@istari-design.com.

Accelerating Viterbi Decoder Simulations.

Today's communication systems are more complex than ever, and few systems engineers can commit to a design without extensive Monte Carlo simulation to prove their approach. Designing a Viterbi decoder accelerator to integrate with system simulation packages can help solve the simulation bottleneck.

Once the realm of an obscure branch of mathematics, forward error correction (FEC) codes are now used in satellite links, magnetic recording, cell phones, compact disc players, wire line modems, terrestrial microwave links, and virtually anywhere that errors can be introduced by an imperfect channel. Today, no communication systems engineer can afford to ignore the consideration of coding for any data transfer problem.

FEC codes map a set of desired user data bits to another, larger set of code bits, which are then transmitted on the channel. Because not all code bit sequences can occur in the transmitted stream, errors introduced by the channel can be corrected by replacing the received sequence with the most likely possible sequence of code bits (given what was received), and then by inverting the transmitter mapping. This is called maximum likelihood decoding.

Most communications channels add noise, distortion, and other impairments to the received signal. The bit error rate (BER) degrades as the ratio of the signal power to the power of the impairments decreases. Good performance usually prevails if the available transmitter power is limitless, but in a cost-effective design, transmitting with as little power as possible is almost always preferred.

A system's measure of effectiveness is often expressed as the ratio of the energy required per user data bit to the noise power spectral density (the Eb/No) required producing a given BER. Typical codes commonly in use today reduce the required Eb/No by 5 dB for Gaussian noise channels.

Codes are classified as either block codes or convolutional codes. A rate *m/n* block code maps a group of *m* user data bits into a block of *n* code bits. Each block is independent of each other block.

A rate *m/n* convolutional code maps a continuous running bit stream at *m* bits/sec into a continuous running stream at *n* bits/sec. Although such codes can be modified to operate on blocks of data, the decoding process is most easily understood by thinking of the operation as continuous.

In this article, we'll focus on convolutional codes, and particularly, on the implementation and simulation of maximum likelihood decoders for commonly used convolutional codes.

Convolutional codes and Viterbi decoding
In 1967, Andrew Viterbi introduced an algorithm for maximum likelihood decoding (often referred to as the Viterbi algorithm). In brief, the algorithm requires that for each bit time, we compute the set of encoder probabilities for each of its possible states (the encoder state is simply the contents of its shift register).

Figure 1 shows a convolutional encoder for a typical rate 12 code. For each bit time, the encoder shift register is clocked, and an output *i* ,*j* symbol pair is created. The code shown has a constraint length *k* of 7, indicating that each input bit affects a time span of 7 bits. Longer constraint lengths yield better performance, but increasing *k* by an increment of one doubles decoder complexity. The code's performance is also strongly affected by the choice of shift register taps connected to the exclusive-OR summers.

For the types of codes discussed in this article, a given encoder state can only be reached from two other states. Figure 2 shows a butterfly decoding diagram — there are $2^{(k-2)}$ such butterflies in a constraint length *k* decoder. In the following discussion, we'll use the vector X=(x5,x4,x3,x2,x1,x0) to enable a shorthand notation for the state. The state 0X is interpret- ed to mean the state (0,x5,x4,x3, x2,x1,x0) while the state X0 is (x5,x4,x3,x2,x1,x0,0).

The two states on the right (corresponding to the current bit time), 0X and 1X, can only be reached from the states X0 or X1 on the left (at the previous bit time). If we know P(X0) and P(X1) (the probabilities of states X0 and X1), and we can determine the probability of receiving the values of *i* and *j* , assuming that the encoder made the transitions from X0 to 0X or from X1 to 0X, then P(0X) for the current bit time must be either: P(X0)*P(*ij* given X0 goes to 0X), or P(X1)*P(*ij* given X1 goes to 0X).

Viterbi observed that a maximum likelihood decoder needs only to choose the larger of these two probabilities, and can discard the smaller. To produce decoded bits, we simply retrace the state paths that led up to the most likely state in the current bit time. Only one path must be remembered for each state.

In most designs, the decoder computations are performed using values proportional to the logarithm of the reciprocal of the probability, called log-likelihood functions or metrics. Log-likelihood functions allow us to replace the multiplication operation with an adder. Instead of choosing the largest probability, we choose the smallest metric. In summary, for each bit time, we hypothesize each of the $2^{(k-1)}$ possible encoder states. We determine what bits the encoder would have had to output, for each of the two paths leading into our hypothesized state. We then compute the log-likelihood of getting what was received, if those bits had been what the encoder really output (this value is called the branch metric). For each of the two paths, we add the branch metric to the log-likelihood function for the state from which the branch originates (the state metric). We compare this sum to that of both the branch metric and state metric for the other path. We then set the state metric for the hypothesized state to the smaller of the two sums.

Next, we update the path history for the destination. We set the decision bit to 0 if the smaller metric came from the state X0, and to 1 if the smaller metric came from X1. The path history is a sequence of decision bits. The path for a hypothesized state is formed by appending the decision bit to the path, which is saved for the state that produced the smaller state metric. In a theoretical design, the decoder waits until the transmission is complete, and then outputs the bits on the path leading to the most probable state.

Punctured codes

It is possible to design a convolutional encoder that shifts *n* bits at a time, and computes *m* different parity sums to create a rate *n/m* code. In practice, however, it is more common to use code puncturing. In this technique, a standard rate 12 convolutional encoder is the basis for the code, but 2 *n-m* of every 2 *n* bits output by the encoder are discarded prior to transmission. The actual code rate (the ratio of information bits to transmitted bits) is thus *n/m* .

To decode the punctured code, we use the standard rate 12 Viterbi decoder, but the positions where bits have been discarded are filled by *erasure* bits, which do not contribute to any metrics. As you might guess, the choice of which bits to discard is important for good performance.

Simulating Viterbi decoding

Today's communication systems are more complex than ever, and few systems engineers can commit to a design without extensive Monte Carlo simulation to prove their approach. The computational power required for good statistical significance is often staggering. Running a simulation overnight or even for weeks at a time is not that unusual. These run-time requirements often make extensive "what if?" modeling problematic.

Because the bit manipulation operations required for Viterbi decoding are not well suited for efficient implementation on most computer platforms, a decoder model can often dominate simulation time. For example, a reasonably efficient k=9 simulation, coded in C and running on a 300-MHz Pentium II PC, can achieve about 12 kbps. Receiving good statistical significance requires approximately 1,000 errors, which for 1e-6 BER, takes nearly 24 hours. Going to k=10 doubles the BER time.

Modem simulation speeds vary substantially depending on complexity, but efficient modem simulations coded in C will typically run at 10 kbps to 100 kbps. Eliminating the decoder's computation requirements could thus improve the speed of an integrated modem/ FEC simulation by a factor of about 2 kbps to 10 kbps for the k=9 de- coder, and much more for higher constraint lengths.

A Viterbi decoder accelerator card was designed for a PC to solve this bottleneck in our own simulations. The card decodes with any constraint length from 5 to 14. The user can select any code generators, and use any puncture pattern. The decoder is called via a subroutine in a way that, from external appearances, is no different from calling a software Viterbi decoder routine, except that the result is returned some 30 times faster. The decoder is easily integrated with system simulation packages that have interfaces for externally generated C code.

Decoder implementation

Figure 3 shows the block diagram of a Viterbi decoder. The recovered data from the modem is mapped into a set of symbols for input to the decoder. The symbol mapper performs serial-to-parallel conversion, and inserts erasures as necessary to provide the decoder with a complete set of symbols for each information bit (two symbols for rate 12 based codes, three symbols for rate 13-based codes, and so on).

The sign bit of the symbol gives a hard decision estimate of the encoder output, while the symbol magnitude gives a soft decision measure of the confidence in the hard decision. The larger the magnitude, the more reliable the hard decision estimate. Increasing the number of bits used to represent each symbol improves the decoder's performance, but also raises gate count. In practice, most systems use two or three soft decision bits in addition to the sign bit.

The symbol set is presented to the branch metric calculator, which computes branch metrics for all possible encoder state transitions. For linear, bi-phase modulation and an additive white Gaussian noise (AWGN) channel, metric computation is simple, because the desired log-likelihood functions are directly proportional to the symbol magnitude. Metric computation in other instances can be much more difficult.

With linear modulation in AWGN, the branch metric calculation can be thought of as a negative correlation between the received symbol set and the ideal output of the encoder making the hypothesized state transition. The correlation can be calculated as the sum of the soft decision magnitudes of the symbols whose hard decisions disagree with the hypothesized output of the encoder, minus the soft decision magnitudes of the symbols that do agree.

Figure 4 shows an example branch metric calculator for a k=7 rate 12 decoder. The received symbols are represented by *ri* and *rj* . The state counter represents the presumed contents of the encoder shift register, excluding the first and last bits.

If we assume that the taps used to generate the outputs of the encoder shift register include the newest and oldest bits (which is the case for most good codes), then the branch metric for the presumed encoder contents 0X0 is the same as that for 1X1. This is true because the encoder is unaffected by the addition of two 0s or two 1s. Like- wise, the branch metrics for 0X1 and 1X0 are the same. Taking advantage of this symmetry simplifies the metric computation.

A more subtle symmetry argument allows the metric computation to include only contributions from symbols whose sign bits disagree with the hypothesized encoder output. Finally, erasure symbols have magnitudes of zero, and, therefore, make no contribution to any metric. The branch metrics then perform the state metric update as shown in Figure 2.

The process of adding state metrics with branch metrics, and selecting the lesser, is referred to as the add-compare-select (ACS) operation. The ACS takes advantage of symmetry to make the process more efficient. Consider the new state 0X. The two possible predecessor states are X0 and X1. But this is also true for state 1X. The only difference is in the assumption of the polarity of the newest bit of the shift register.

Because only the newest bit is different in the two states, the same pair of branch metrics is required to compute each state. For this reason, an ACS pair is often implemented in a butterfly structure that performs the operations for two states at once.

More generally, gate count can be traded against speed in three ways. One way is by implementing the state update block using a single ACS, which updates the states serially in $2^{(k-1)}$ clocks. Another way is to use a complete set of ACSes, which updates all $2^{(k-1)}$ states in a single clock cycle, and the third method is as a hybrid with N ACSes requiring $2^{(k-1)}$/N clock cycles.

For example, during each clock cycle, an implementation using two butterflies retrieves the previous state metrics from four consecutive states, X00 through X11, and creates the new state metrics and decision bits for states 0X0, 0X1, 1X0, and 1X1.

Note that in any serial or mixed serial/parallel state update, if the previous state metrics are used in order (from 0 to $2^{(k-1)}$), then the new state metrics will be created out of order [0X, 1X, 0(X+1), 1(X+1)]. This complicates the storage and retrieval of the state metrics. The easiest solution is to use two separate RAMs for holding the state metrics — one to read from and another to write to. The RAM requirement is doubled as a result, but is often still a good trade-off compared to the problem of managing a single memory.

Path memory and decision traceback

Once the state update is done, the path memory for all states must be updated. The path memory is defined as the set of sequences of decision bits that lead to each state at the current time. The path for each new state consists of the path of its predecessor state, along with the decision bit for the new state appended to its end.

In theory, the decoder is usually assumed to save all the bits of each path until the transmission is complete. It then outputs the bits on the path leading to the most probable state. In practice, however, because the paths all tend to converge to the same sequence as we trace backwards down the path in time, the path memory only needs to be deep enough to allow convergence. The required depth is typically several constraint lengths.

Because the paths are quite long, reading and writing entire paths for each state is an inefficient approach to use in a hardware implementation. Completion of all activities needed for each state in a single clock cycle is preferred, and is not easy to do with long paths.

Rather than moving entire paths from one memory location to another, a more efficient technique is to store all of the decisions as they occur for all of the states into RAM. The RAM is organized as a rectangular page where each column contains all the decisions made for a given bit time. Moving back one column along a row is equivalent to backing up a bit time. The state forms the row address, and the bit time forms the column address.

Each time a set of ACS operations is completed, the resulting decisions are written into the appropriate column of the RAM. At the end of a state update cycle, the traceback RAM contains all of the decisions from the current bit time, as well as those from the previous N bit times.

To determine the path that led to a given state, the state is used as the row address, and the current bit time is used as the column address. We read the bit stored at that row/ column address. A new row address is then formed by left shifting the old row address and adding the bit just read as the least significant bit (LSB). The column address is decremented, and another bit is read. This process is repeated until the traceback has gone far enough back so that the path can be assumed to have converged. At that point the data bits on the path are output as the corrected data.

The path converges most rapidly if the traceback starts at the state with the best (lowest) metric. Usually, the traceback is limited to about five to ten constraint lengths, at which point the degradation from a theoretical design is only one or two tenths of a dB. Higher code rates generally require more path memory depth than low rates.

One or more decoded bits can be output during each traceback cycle. Fewer operations are required when outputting multiple bits per traceback. The performance is slightly worse though, than when recovering each bit from a path traced all the way to the end of the path memory.

Results

Besides the choice of constraint length, code generators, and code rate, the performance of the decoder depends upon the symbol width (number of bits of soft-decision magnitude), traceback depth, puncture pattern, and the number of bits used to represent the state metric.

Our Viterbi decoder accelerator card was designed for rate 12 and rate 13 codes using any constraint length from five up to fourteen, and any desired puncture pattern. It uses a field programmable gate array (FPGA) with external RAM to implement the decoder.

The upper graph in Figure 5 shows a BER curve of this decoder for rate 12 for various constraint lengths. The performance of an ideal modem without coding is also shown here for reference. The coding gain for k=5 is about 4 dB compared to the uncoded channel. For k=14, the coding gain is approximately 6.25 dB. For each increment in k, the performance improves about 0.25 dB.

Increasing k further would result in even greater performance increase, but complexity limits this option. Using a 4-ACS architecture, the hardware implementation of the k=14 decoder operating at 25 MHz achieves a throughput of about 13 kbps. For k=15, this would be reduced to 6.5 kbps.

The lower graph in Figure 5 shows the performance of a k=9 decoder for various code rates. The fractional rates are achieved by puncturing the rate 12 code. Rate 13 codes can also be punctured, allowing code rates that lie between 12 and 13. The increase in coding gain falls off rapidly as the code rate is reduced. Also, the signal-to-noise ratio (SNR) at which the modem operates decreases in proportion to the code rate (for a given Eb/No), which can make it difficult to keep the modem locked to the signal. More bandwidth is also needed to transmit the extra symbols. All of these problems combined make using rates of less than 13 uncommon.

Better performance

Advances in integrated circuit technology will enable the use of higher constraint length codes in future communication systems. These codes will allow better performance and improved system capacity. System engineers designing modem/FEC simulations for these new systems will need some assistance to make their designs practical. A hardware-based accelerator that provides a 30-fold speed increase over a software implementation offers a practical solution to this problem.

Return