

## Lecture 15

*Lecturer: Daniel A. Spielman*

## 15.1 Related Reading

- Fan, pp. 108–110.

## 15.2 Remarks on Convolutional Codes

Most of this lecture will be devoted to Turbo codes and their variants. However, since these are built from convolutional codes, I'll begin with a few more remarks on them.

To begin, I'll point out that the algorithm that we discussed last class, alternately called the BCJR algorithm, MAP decoding, forwards-backwards algorithm, or sum-product algorithm, solves the probability estimation problem exactly. In this way, it is analogous to the computations that we could do for the repetition code, the parity code, and LDPC codes on trees. For each message bit, we exactly compute the probability that it was 1 given the probability that every other message and check bit was 1.

The next point that I would like to make is that the algorithm described last lecture can run into numerical trouble. In particular, it keeps computing values that are proportional to the probabilities in question. However, these values will get lower and lower as the algorithm progresses through the trellis. It is quite possible for them to become so low that they cannot be accurately represented in floating point. To compensate for this problem, one may re-scale the values. For example, say that we are interested in the probabilities that some state  $s_i$  is equal to one of  $\{00, 01, 10, 11\}$ . Our algorithm computes values that are proportional to these probabilities. It is natural to re-scale these so that they actually become probabilities. To do that, one just multiplies them by a constant so that their sum becomes 1. If you keep doing this as the algorithm moves down the trellis, you will avoid some of the potential numerical problems.

## 15.3 Turbo Codes

Turbo codes make use of a systematic recursive convolutional code and a random permutation, and are encoded by a very simple algorithm:

1. Form message bits  $w \in \{0, 1\}^n$  (the algorithm is so simple, I include this as a step)
2. Output  $w$ .

3. Encode  $w$  using the systematic recursive convolutional code. Output the check bits generated.
4. Permute the bits of  $w$  according to the random permutation, and call the result  $w'$ .
5. Encode  $w'$  using the systematic recursive convolutional code. Output the check bits generated.

If the systematic recursive convolution code is the one described in the last class, which has one stream of check bits in its output, then this is a rate  $1/3$  code.

Let me make one thing clear: the code is specified by the convolutional coder and the permutation. The permutation must be known to both the encoder and the decoder. The permutation does not really have to be random, but random ones do seem to work very well.

## 15.4 Decoding

Turbo codes are exciting because they have a fast decoding algorithm that performs exceptionally well. There are many ways of describing the decoding algorithm, but Fan's is the easiest. So, I will follow his description. In particular, I will reproduce his figures.

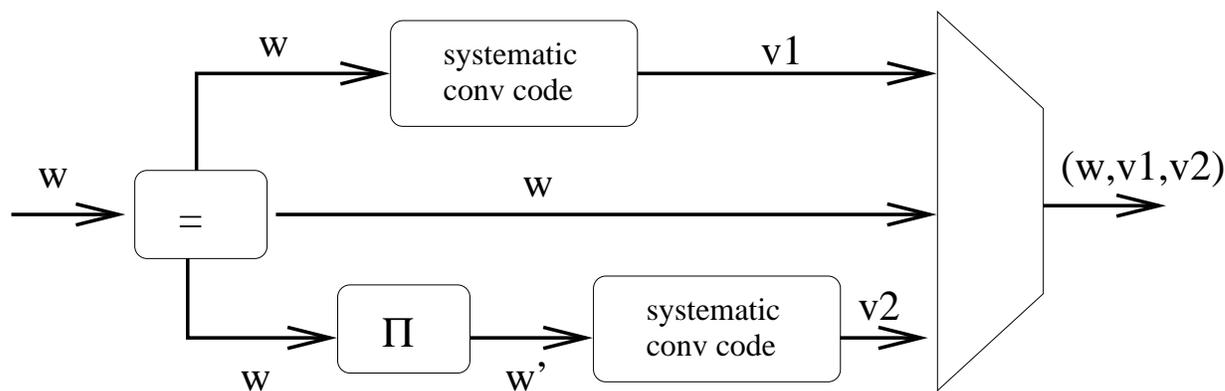


Figure 15.1: Turbo Encoder. The “=” node can be viewed as a repetition code that sends its input 3 ways. The  $\Pi$  node represents the permutation. Note that the block length must be fixed so that  $\Pi$  can be fixed. The object on the right just interleaves the three streams. This is Figure 3.8 in Fan.

In Fan's depiction, the input string,  $w$ , goes through a repetition code, denoted by the “=” node in the figure. One of the outputs of the “=” node goes straight into the output, making this a systematic code. The next output of the “=” node goes into a systematic convolutional encoder, and the check bits  $v^1$  output by this encoder go into the output. Finally, the third output of the “=” node goes through a permutation, denoted  $\Pi$ , and then into a systematic convolutional encoder, and the check bits  $v^2$  output by this encoder go into the output.

To understand the decoding, we first observe that the output of this encoder contains the information necessary to apply the forward-backward decoder to either convolutional code, as we obtain channel outputs for both the inputs and outputs of each code. However, here the convolutional

codes share their inputs. This sharing allows for a more powerful decoding algorithm, in which we take the extrinsic outputs produced by each decoder and feed them into the other.

In the first stage of the decoder, depicted in Figure 15.4, the channel output for  $w$  and  $v^1$  are fed into a node that is used to decode the upper conv code. In the figure, we depict this by passing the channel output for  $w$  through the “=” node, and bringing a null message up from the other decoder. As when we initialized the LDPC decoders, combining with a null message at a “=” node has no effect on the other message being sent, so the channel output for  $w$  goes straight through.

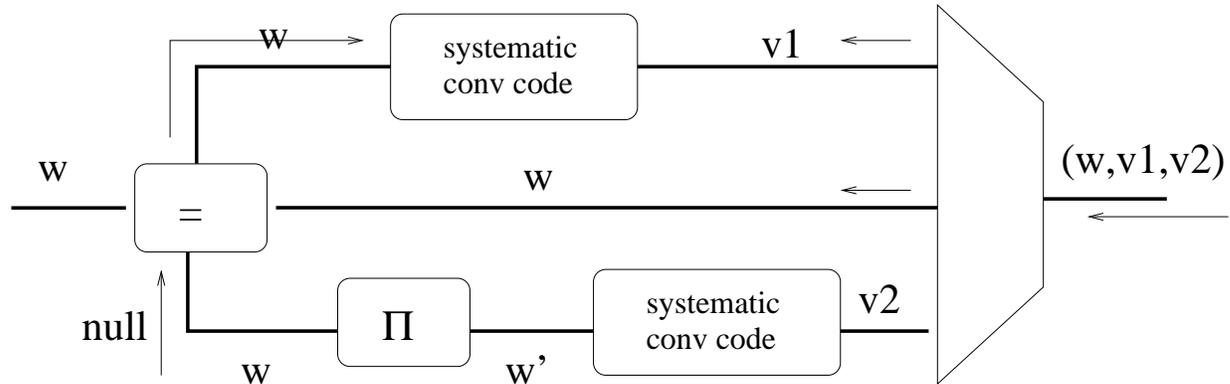


Figure 15.2: Decoding step 0.

On receiving the channel outputs for  $w$  and  $v^1$ , the top convolutional decoder will output the *extrinsic* probabilities that each bit of  $w$  was 1. It is very important that these be the extrinsic, not the posterior, probabilities. That is, the probability that  $w_i = 1$  will not take into account the channel output for  $w_i$ . The channel output will be factored in later. This is just like what happened at the parity nodes in LDPC codes.

In the next stage of the decoding, depicted in Figure 15.4,

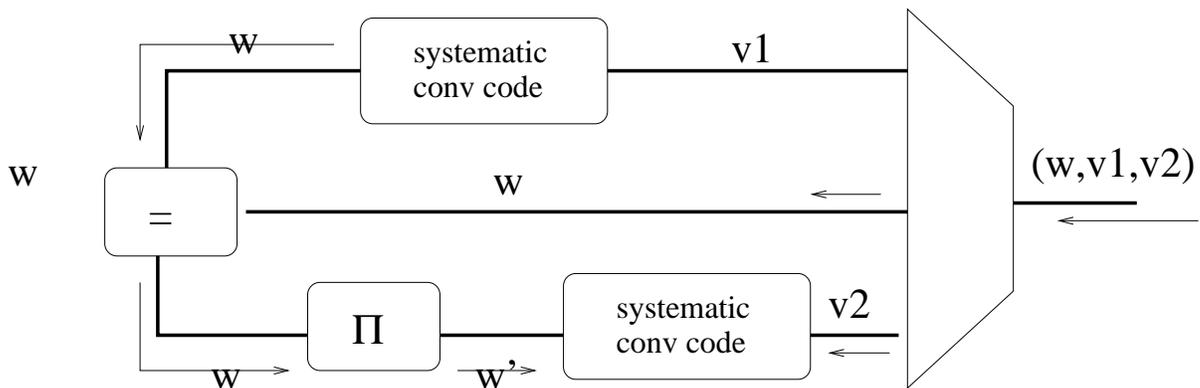


Figure 15.3: Decoding step 1.

The extrinsic outputs from the top decoder are passed through the “=” node, where they are combined with the channel outputs for  $w$ , and are treated as improved estimates for the probability that each bit of  $w = 1$ . These probabilities are then passed into the lower decoder, which also takes

as input the channel outputs for  $v^2$ . This decoder will also produce *extrinsic* outputs for the probability that each input bit is 1.

So far, this is all pretty conventional.

Where things get interesting is in the next step. In the next step, depicted in Figure 15.4, the extrinsic outputs from the bottom decoder are passed up through the “=” node, where they are again combined with the outputs from the channel for  $w$ , and are fed as inputs into the top decoder, where they are again combined with the channel outputs for  $v^2$ .

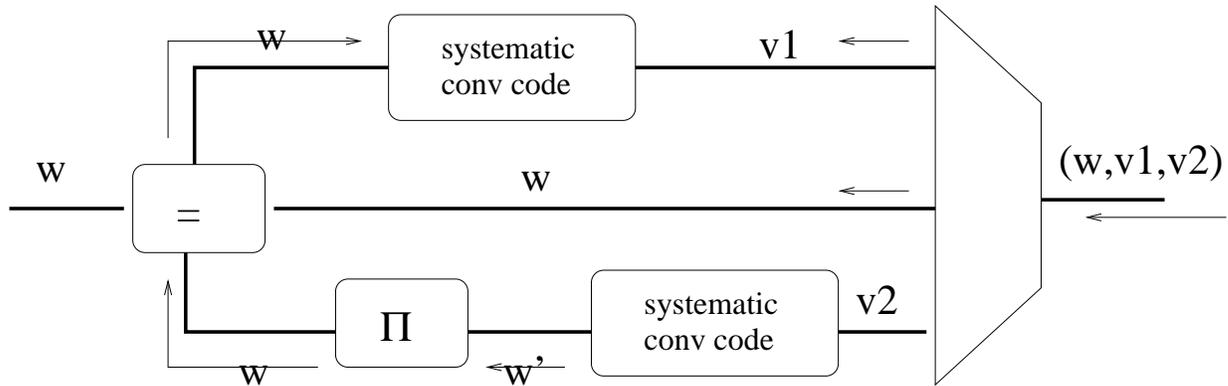


Figure 15.4: Decoding step 2.

This process is then repeated for a few iterations: bouncing from one decoder to the next.

When we finally want to get an output, we combine the channel outputs and the last outputs from each decoder at the “=” node, and pass the outputs out on the left. This is depicted in Figure 15.5.

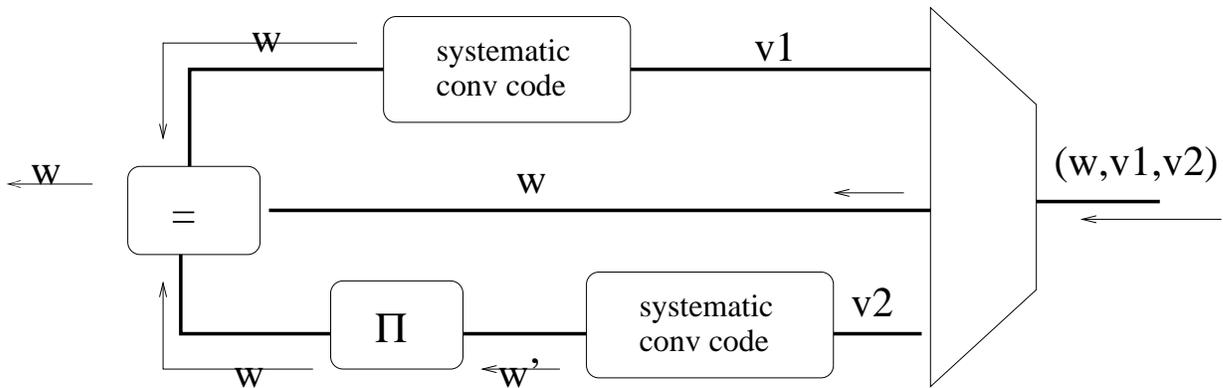


Figure 15.5: Obtaining the outputs.

Empirically, this algorithm does incredibly well, providing good performance even at 0.6 dB!

## 15.5 Exit charts

For the first couple of years after Turbo codes were invented and experimentally observed to work well, no one had a good explanation for why they worked so well. The first explanation that I thought helped, although it was still not completely rigorous, was provided by the EXIT charts of Stephan ten Brink. These EXIT charts are essentially a heuristic version of the analysis that we did of LDPC codes on the erasure channel. Here's the idea:

We are going to form a chart that we hope will explain the performance of each convolutional code individually. By then making the heuristic assumptions

- that each is behaving independently, and
- that all messages being sent in the system are Gaussian distributed of a given capacity, (even though they are not)

we will predict the behavior of the Turbo decoder. It turns out that even though these assumptions are false, they give a very good approximation of what actually happens.

Let me explain more concretely what we will measure. Consider just one convolutional decoder, as in Figure 15.5. This part of the decoder can be viewed as having three inputs: channel outputs for  $w$ , channel outputs for  $v_1$ , and the messages being passed up from the other decoder,  $x$ .

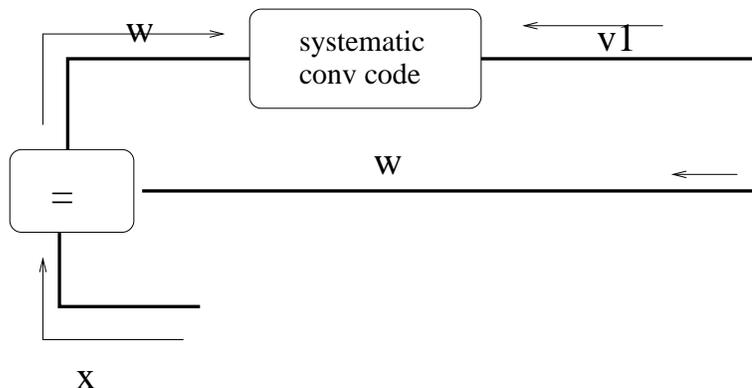


Figure 15.6: A part of the decoder

Begin by fixing some random codeword, say  $w, v_1$ . Assume that we fix the channel noise level, and thereby fix the distributions of the channel outputs for each of the symbols in  $w$  and  $v_1$ . At each iteration of the decoding algorithm, what is changing is the messages  $x$  coming from the lower decoder. We are going to view these as being like passing the input  $w$  through another channel, and by ignoring the correlations among the bits, view it as a memoryless channel. In particular, we will take  $x$  to be the output of passing  $w$  through some Gaussian channel. To try to understand the effects of successive iterations, we will see what happens when we vary the noise of this Gaussian channel. For each noise level of the  $x$  channel, we will perform a simulation. We will then measure, empirically, the capacity of the meta-channel that we see if we measure the extrinsic outputs of the top decoder. That is, form  $x$  by passing  $w$  through a Gaussian channel, apply the decoder, look at

the outputs, treat the whole thing as a meta-channel, as estimate the capacity empirically as we did in Project 1. By plotting the capacity of the channel through which we passed  $x$  against the observed capacity of the meta-channel, we obtain a point on our exit chart. For an example, see Figure 15.5.

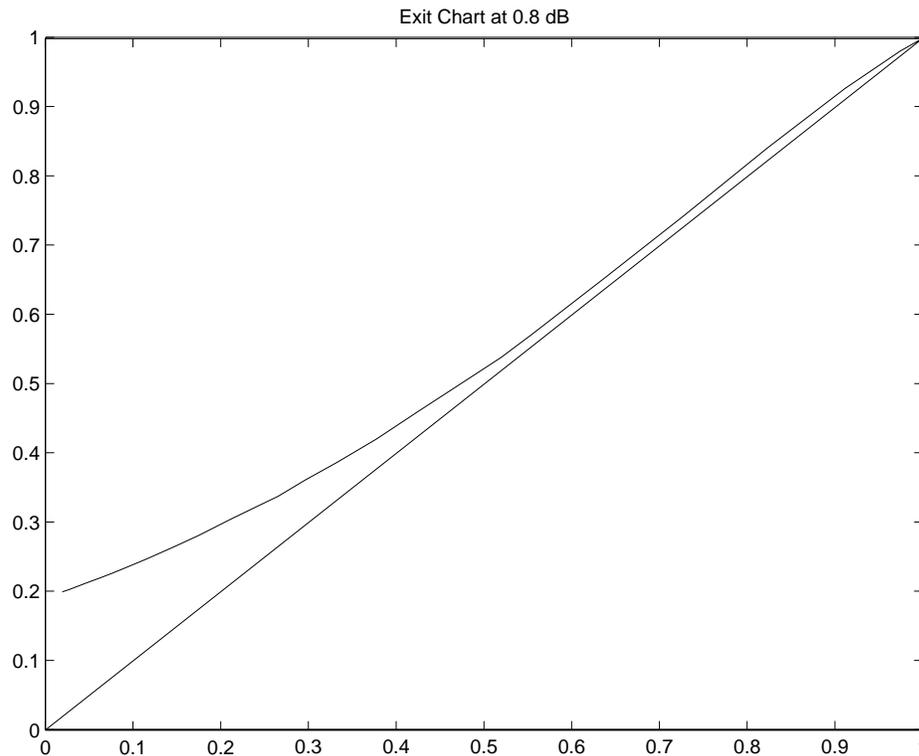


Figure 15.7: An exit chart for the convolutional code from small project 3

From the exit charts, we would predict that if the curve stays above the  $x = y$  line, then the decoder should converge.

The reasoning is as follows: at the first stage of the decoding, a null message is transmitted. This corresponds to obtaining  $x$  by sending  $w$  through a channel that just erases  $w$ . So, the capacity of the output of the first decoder will correspond to where the curve intersects the y-axis in the figure. The messages output by the first decoder are then passed to the second decoder, which is identical. Now, the messages output by the first decoder are not necessarily Gaussian distributed. But, we can empirically measure their capacity, and pretend that they came from a Gaussian distribution of the same capacity. We then look again at the EXIT chart to figure out what the capacity should be of the messages output by the bottom decoder, etc.

So, this heuristic analysis works by measuring the capacities of the outputs of decoders, pretending that they were Gaussian of that capacity, and then looking on the chart for what the next capacity should be. If all of these pretend assumptions were right, we could just follow the charts to find out how the decoding should behave. Empirically, we find that this works very well.

If we vary the original channel through which  $w$  and  $v^1$  are passed, we get a different chart:

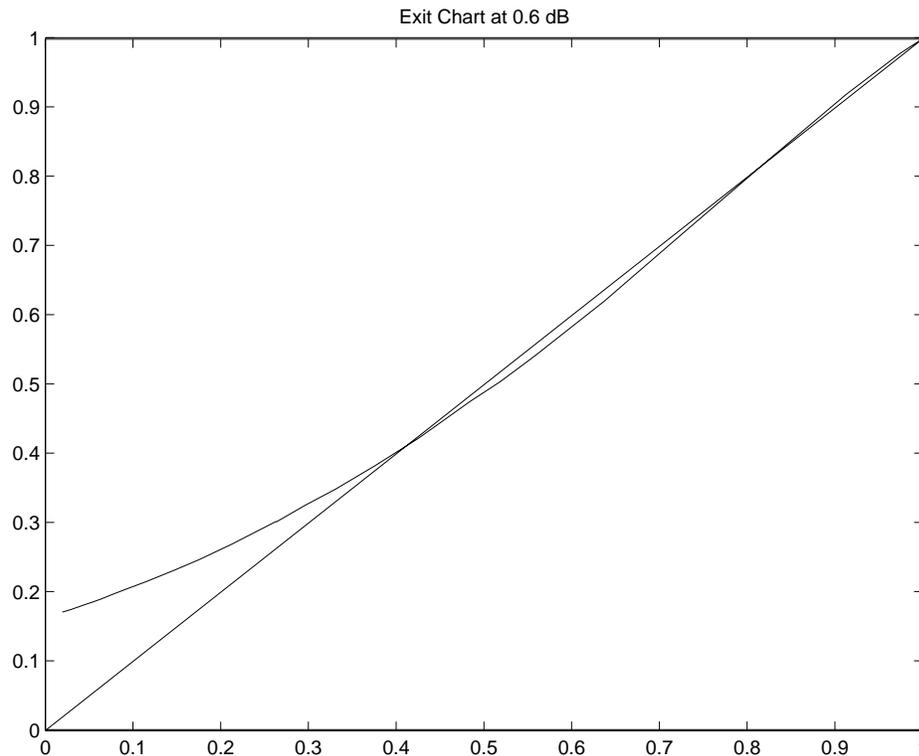


Figure 15.8: An exit chart for the convolutional code from small project 3

But, you might wonder, why are we going to the trouble of making all these heuristic assumptions? Why not just use the distributions that actually occur? The reason is that we do not have any nice characterization of what these distributions will be, and cannot find any way of determining them other than by simulating the Turbo code decoding process. However, simulating the full Turbo code decoding is computationally expensive, while simulating the decoding of the convolutional codes is cheap. So, we prefer the cheaper simulation.

More importantly, this simulation seems to predict the behavior of the full system from just the behavior of its components. If this simulation always yields a good prediction, then it should be possible to design better codes by computing EXIT charts of various components, and seeing how they fit together. In particular, it allows us to predict how the system will behave if we use two different convolutional codes.