Error Correction Coding: Weeks 11-12

Convolutional and Turbo Codes

Eric DeFelice
Department of Electrical Engineering
Rochester Institute of Technology
ebd9423@rit.edu

***Abstract* – Convolutional codes are a special case of error-correcting codes. These codes, unlike block codes, have some memory because they rely on multiple bits of the data stream during the encoding process. They are good at correcting random errors because of this fact. There are a couple ways to represent a convolutional coder, in polynomial form or in trellis form. Trellis form is helpful, as it more represents a digital system and it easy to convert to digital hardware for implementation. A convolutional code can be decoded using the Viterbi algorithm.**

Turbo codes are created by combining two convolutional coders using a puncturing scheme. The encoder in this system uses parallel concatenation so the output of the encoder is a combination of both convolutional codes. These two convolutional codes are recursive systematic convolutional codes (RSC codes). This means that the original input bits are maintained in the output, and the encoder can be constructed using sums of different delays of the input stream. Turbo codes are decoded using two serial decoders, with an interleaver in between.

In this paper, an introduction of convolutional codes will be presented, and the properties of this code will be looked at. The operation of the encoder and the decoder will be described. Matlab examples of the convolutional codes will be looked at to show the codes performance. Turbo codes will also be investigated, mainly through Matlab simulations.

I. Introduction

Convolutional codes are error correction codes that are particularly well suited to correct random errors. These codes use memory to produce output bits that are summations of bits from the input data stream. The rate of convolutional codes is related to how many output bits are generated per input bit. This is where the coding gain comes in, as each input bit produces a number of parity bits. The encoder can be described using a polynomial representation or a trellis representation. We will briefly look at the polynomial representation, but our main focus will be on the trellis encoder. This is because the decoder we will use, the Viterbi decoder, is also a trellis based decoder.

Turbo codes are a class of convolutional code, where two recursive systematic convolutional codes (RSC codes) are combined. This produces an output data stream that has a greater distance from the input stream than a single convolutional encoder. In this paper, Matlab examples of the turbo encoder and an example turbo decoder will be analyzed.

II. Properties of Convolutional Codes

A polynomial description of a convolutional coder represents the connections between shift registers and modulo-2 adders, which would be used to implement the encoder in hardware. In general, the encoder can be described as follows:

$$y[n] = \sum h_i x_{N-i}$$

This encoder can be a feedforward encoder or a feedback encoder. The following is an example of a convolutional encoder with feedback paths:

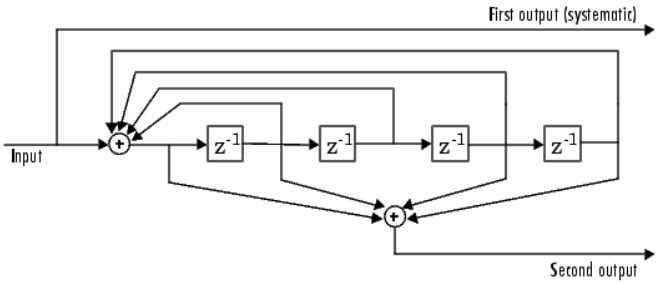

Figure 1: Convolutional encoder with feedback

It is easily seen in this figure that the rate of this encoder is 1/2. This means that the encoder takes in 1-bit and outputs 2-bits. The encoder will reduce the data rate of the system by ½, but the BER should be reduced because of the coding.

The system can also be described in trellis form. This form is more of a state machine representation of the system. The trellis form shows all of the possible current states of the system and shows the possible transitions to the next states. For example, in the system shown above, if the current state is [00], meaning both of the previous input bits were 0, the next state could either be [00] or [10], depending on the next input bit. The output would be represented in the transition to the next state. Below is an example trellis of a convolutional code:

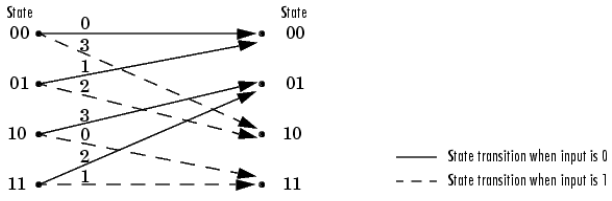

Figure 2: Trellis represenation of a convolutional encoder

In figure 2, the numbers seen on the state transitions are the bits that are output from the encoder. This representation is useful because it describes the input versus output relationship more clearly, which is useful for decoding. For example, if the decoder is at state [00], the output should either be [00] or [11]. If we decode to either of the other two possible outputs, we know that this is incorrect, since it is not possible from the encoder. This is a known error, and the decoder must now figure out the trellis path that is closest to the received bits.

The operation described above is the basic operation of the Viterbi decoder. The Viterbi decoder follows the trellis with the incoming received bits, and tries to determine the most likely path of the trellis. This decoding scheme is vulnerable to bursts of errors, because it gets onto the wrong trellis path and isn't able to find a valid path back to the original transmitted bits.

Below is a typical trellis diagram from a Viterbi decoding algorithm:

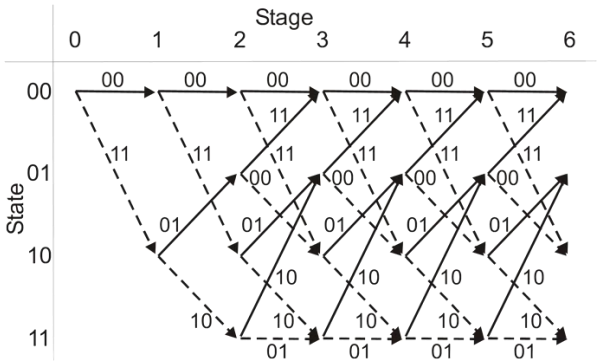

Figure 3: Example Viterbi decoding trellis

It can be seen in the figure above that there are some paths that aren't valid. For example, in the encoder that this trellis describes, when the encoder begins at stage 0, there are only two possible symbols that can be received, [00] and [11]. If the decoder receives [01], for instance, the decoder must now start looking at both paths. For the subsequent receive symbols, the decoder continues along the path with the lowest distance, since that path is more likely. At this point in the decoding process, there are N paths being analyzed. When these paths meet back up at a single node, the decoder can then choose the bits from the one that has the lowest distance from the received symbols. This is how the Viterbi decoder works. This operation also explains why the decoder can allow burst errors to remain in the code. If the burst is longer than the largest traceback length, then the decoder is likely to decode the whole path incorrectly, leading to a burst of errors.

A sample rate-1/2 convolutional encoder, along with a soft-decision Viterbi algorithm decoder was simulated in Matlab. Its bit error rate performance was compared to an uncoded data stream. The results from this simulation are seen in figure 4.

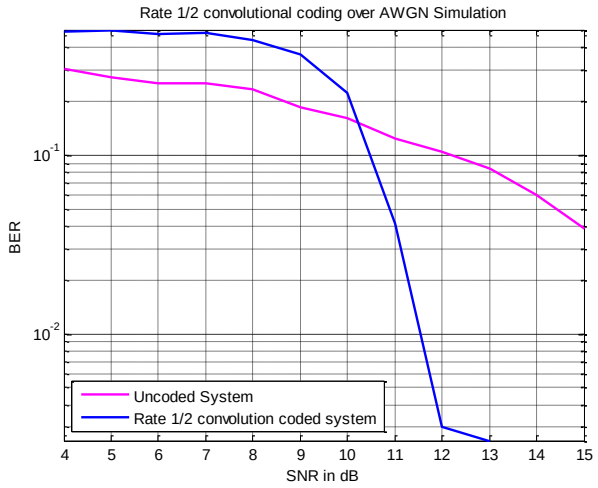

Figure 4: Example convolution coded system simulation

From this figure, it is seen that the bit error rate of the coded system drops quickly as the SNR increases. This is because there are no longer long strings of errors, so the code can correct most of the random bit errors. At low SNR, the system performance is worse because there are now parity bits in the system, reducing the data rate.

III. Properties of Turbo-Codes

Turbo codes are created by implementing two parallel convolutional encoders, and combining their outputs using an interleaver. The following is an example system where the encoder is comprised of two identical recursive systematic encoders:

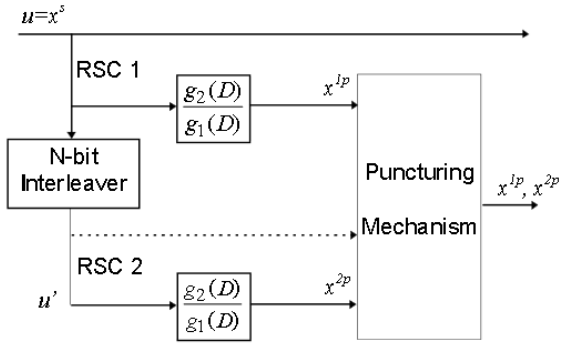

Figure 5: Sample Turbo-code encoder system

The encoding side of a Turbo-code is fairly simple and intuitive. It is the decoder that is more complicated, as it needs to de-interleave the data and decode it accurately and efficiently, since this should be able to be implemented in a real-time system.

The main decoding algorithm for Turbo-codes is the BCJR algorithm, which is named for its founders (Bahl, Cocke, Jelinik, and Raviv). This algorithm is a MAP algorithm, which stands for "Maximum Aposteriori Probability". The BCJR algorithm is designed to estimate random parameters with prior distributions from an input data stream. This

algorithm is similar to the Viterbi algorithm but does have some key differences:

- The Viterbi algorithm computes the most probable information sequence, while the BCJR algorithm computes the most probable bit from a data sequence.
- The BCJR algorithm is inherently a soft-decision algorithm, where the Viterbi algorithm is inherently a hard-decision engine.
- The BCJR decoder is much more complex than the Viterbi algorithm, and is better suited for iterative decoding.

The use of the BCJR decoding algorithm is a recent trend because of the computational ability of modern systems. Without the performance of modern systems, this algorithm would not be practical as it would be too intensive. The following figure shows an example decoder system:

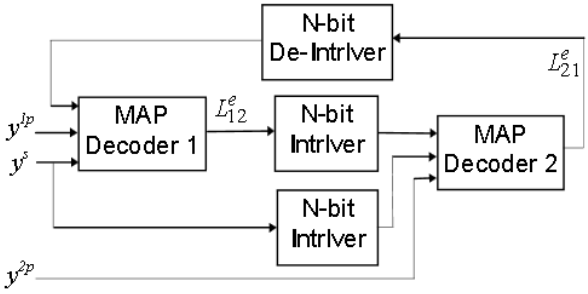

Figure 6: Example BCJR decoder

The final output of the previous decoder can either come from decoder 1 or decoder 2. The algorithm is complete when the two decoders agree on the output, and this can occur after any iteration. The interleavers and de-interleavers used in the decoder are the same as the encoding interleaver and its inverse, respectively.

The performance of the Turbo-code is better than the performance of a single convolutional code, and is similar to the performance of LDPC (low-density parity check) codes. The bit error rate of a Turbo-coded system is subject to an error floor, which is caused because the code contains some low-weight codewords. At low SNR, these codewords are insignificant, but at high SNR, these low-weight codewords begin to dominate the errors. These random low-weight codewords are created by the particular encoder and interleaver combination. There is ongoing research into how to reduce this error floor for both Turbo-codes and LDPC codes. Figure 7 shows the typical performance of a Turbo-coded system.

IV. Conclusion

In this paper, convolutional codes were introduced and their operation was described. A brief overview of their properties was presented, and the Viterbi algorithm was presented. The Viterbi algorithm is an efficient way to decode convolutional codes, so its operation is relevant to this topic. Finally, simulations were run to explore the effects of a convolutional code on a practical communications system. It was seen that the convolutional code does improve the bit error rate over an AWGN communication channel when the SNR is reasonably high. Convolutional codes are also well suited for use in communications systems because they are good at correcting random bit errors, which are likely in this type of system.

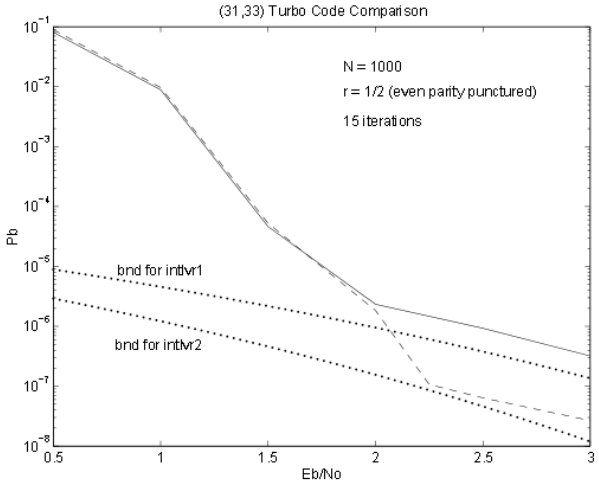

Figure 7: Typical performance for a Turbo-coded system

Over the course of my studies into error correction coding, I've learned a great deal about the theory behind their operation and the implementation trade-offs that need to be investigated when using these codes in practical systems. The theory behind linear block codes is much clearer to me now, and how the construction of these codes relates to finite field theory is also much clearer. I was very interested in researching convolutional codes and the Viterbi algorithm as it is quite relevant in many communications systems that I would deal with in my career. I understand the Viterbi algorithm in much greater detail now. I also understand the concepts behind Turbo-codes much more now, and some of the theory behind the decoding algorithms used with this code.

In the future, I would like to continue studying the implementation of Turbo-codes. I wasn't quite able to design a full Turbo-code Matlab simulation over the past week, but I do have an initial framework. I would like to continue studying this topic so that I can complete my simulation environment, and understand the MAP decoding further. This would allow me to better contribute to this type of decoder in communications systems I may encoder in my career.

References

[1] San Ling, Chaoping Xing, *Coding Theory: A First Course*. Cambridge University Press, 2004

[2] Andre Neubauer, Jurgen Freudenberger, Volker Kuhn, *Coding Theory: Algorithms, Architectures, and Applications*. John Wiley & Sons Ltd., 2007

[3] Scott A. Vanstone, Paul C. van Oorschot, *An Introduction to Error Correcting Codes with Applications*. Kluwer Academic Publishers., 1989

[4] William, Ryan E., *A Turbo Code Tutorial*, New Mexico State University.

APPENDIX

```
%% Error correction coding : Weeks 11-12
%% Convolutional codes
close all;
clear all;
clc;

% Compute the SNR over a range
snr_db = 4:15;
snr_linear = 10.^(snr_db/10);

number_code = zeros(1,length(snr_db));
ratio_code = zeros(1,length(snr_db));
number_uncode = zeros(1,length(snr_db));
ratio_uncode = zeros(1,length(snr_db));
for i=1:length(snr_linear)
  % Random data to be encoded
  msg = randint(4000,1,2,139);
  % Define the decoded uncoded-msg
  rx_u_msg = zeros(length(msg),1);

  % Define the trellis from the polynomial encoder representation
  t = poly2trellis(7,[171 133]);
  % Encode the data
  code = convenc(msg,t);
  % Send the data through an AWGN channel
  rx_code = awgn(code,snr_db(i),4);
  % Send the uncoded data through the channel also
  rx_uncode = awgn(msg,snr_db(i),4);

  % Quantize to prepare for soft-decision decoding
  qcode = quantiz(rx_code,[0.001,.1,.3,.5,.7,.9,.999]);

  % Traceback length for the Viterbi decoder
  % This is how far the decoder can trace in the bit stream
  tblen = 48; delay = tblen;
  % Decode the data with the Viterbi algorithm using soft-decision decoding
  rx_c_msg = vitdec(qcode,t,tblen,'cont','soft',3);
  % Decode the uncoded data
  rx_u_msg(rx_uncode>0.5) = 1;

  % Compute bit error rate
  [number_code(i),ratio_code(i)] = biterr(rx_c_msg(delay+1:end),msg(1:end-delay));
  [number_uncode(i),ratio_uncode(i)] = biterr(rx_u_msg,msg);
end
% Plot the BER vs SNR
semilogy(snr_db,ratio_uncode,'m-','linewidth',2.0);
hold on
semilogy(snr_db,ratio_code,'b-','linewidth',2.0);
title('Rate 1/2 convolutional coding over AWGN Simulation');xlabel('SNR in
dB');ylabel('BER');
legend('Uncoded System','Rate 1/2 convolution coded system');
axis tight
grid
```