Assignment 5
BCJR Decoding of Convolutional Codes

Surajkumar Harikumar (EE11B075)

November 30, 2014

1 PROBLEM STATEMENT

Implement a Bitwise-MAP BCJR decoder for a convolutional code of rate 1/2 and memory order 1. Plot the Block Error-SNR curve for the decoder, and compare with uncoded transmission.

2 ENCODING

The convolutional code used here has the base generator matrix

$$G = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \quad ; \quad G(D) = \begin{bmatrix} 1 \\ 1+D \end{bmatrix} \tag{2.1}$$

The state transition diagram is shown in Figure (2.1).

For the decoding, we assume the all-zero input message. We pass it through the trellis encoder to obtain the output. The BCJR algorithm implements a bitwise-MAP rule. Unlike the Viterbi case, where we sent one long codeword, here we send many many blocks of small length codewords. The message length used was $k = 7$.

For code simplicity, we store 2 arrays, one for the state-nextstate-output relations for input 0, and one for input 1. For the BCJR algorithm, it also makes sense to store the reverse state map (given a state and an input, find the previous state and output).

We then use BPSK modulation to obtain the codeword sent through the channel. We use the code in [1] to simulate the AWGN channel for a given SNR.

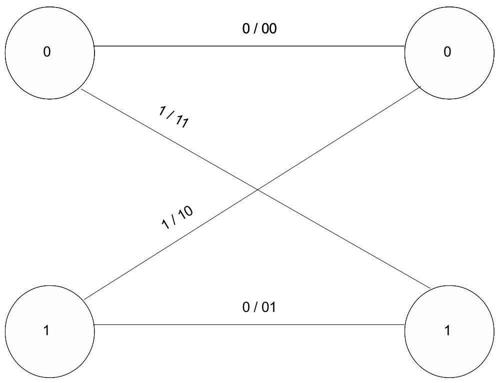

Figure 2.1: State Diagram for the convolutional code

3 BCJR Decoder

We implemented the Max-log-MAP decoding algorithm as shown in [2]. We first computed the branch metrics $\gamma_l^*(s,s') = \log\gamma_l(s,s') = \log p(s, r_l|s')$ for all branches in all iterations.

We then do a forward trellis parse to compute all the $\alpha_l^*(s') = \log p(s', r_{t<l})$ as a maximization based on previous (γ^*, α^*). We then use a bckward trellis parse (using *prev_state* as our trellis) to compute $\beta_l^*(s) = \log p(r_{t<l}|s)$ based on forward looking (γ^*, β^*). The max-log map rule helps us keep each of the terms small, as we have a neat way to evaluate $\log(e^x + e^y)$

Finally, we calculate the aposteriori-probabilities (bit-LLRs) and make a decision on each bit. For every bit, we need to look at all possible state transitions with $0,1$-input, and factor all of them into the aposteriori expression. Note that since we are making bit-decisions, there is no guarantee we get a codeword, and so we compute **Block Error** probability (instead of bit-error rate).

For a given SNR, we send about $5 * 10^4$ blocks of small length codewords. We run these through the BCJR algorithm. If the received vector does not decode correctly to the all-zero message vector, we add it to the count of block errors. In this way, we generate the Block Error Probability for each SNR. The script to implement this is **bcjr.m**.

Figure (3.1) shows the Block Error Probability curve for uncoded transmission over BPSK-AWGN, and that of the BCJR decoder.

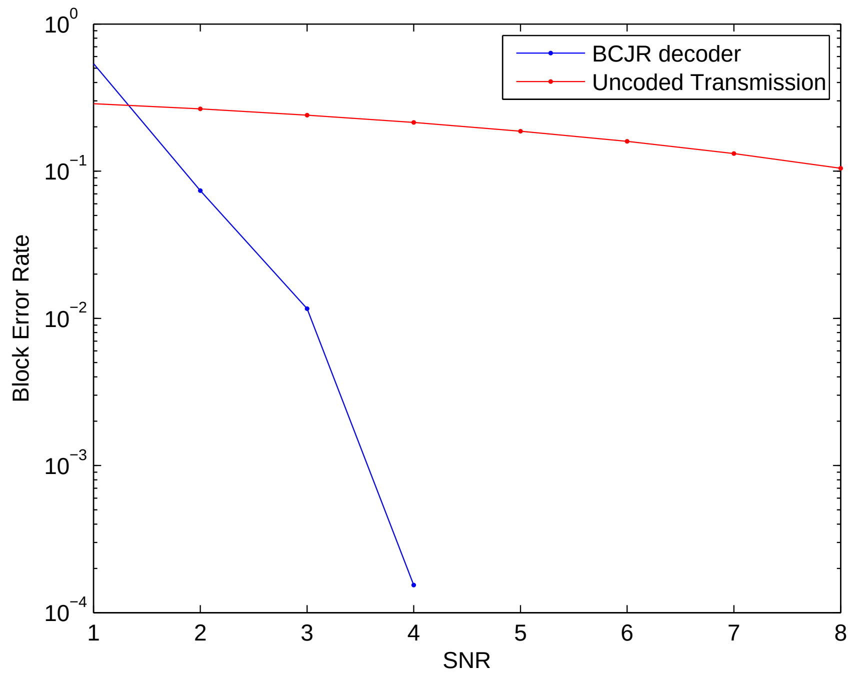

Figure 3.1: Comparison of Block error rate-SNR curves for the BCJR Decoder, and Uncoded BPSK transmission

References

[1] AWGN Generation in MATLAB *addAWGN.m* - http://stackoverflow.com/questions/23690766/proper-way-to-add-noise-to-signal

[2] Presentation on the BCJR algorithm - http://site.iugaza.edu.ps/ahdrouss/files/2011/03/SOVA-and-BCJR.pdf