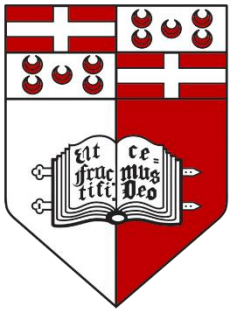

University of Malta
L-Università ta' Malta

CCE5101
Information Theory and Coding

Daniel Bonanno
141295(M)
M.Sc. in ICT (Telecommunications)

Note

All code can be found in the attached Python script. The code is also commented such that it is easy to follow. Furthermore, the comments help to describe the processes at every stage.

Convolutional Encoders

The convolutional encoders defined correspond to the schematics shown in Figure 1 and Figure 2. They were implemented in Python, were 2 global list variables were used to hold the state of the shift registers and were updated after every iteration by shifting the bits. The output operations were computed using the XOR gate. Terminating 0's were added to the given input stream to 'tail off'. Since we only have one input and the maximum length of the shift registers are 2 and 5 respectively, then the input stream was appended with 2 terminating 0s when the Convolutional Code in Figure 1 was used and with 5 terminating 0s when the Convolutional Code in Figure 2 was used.

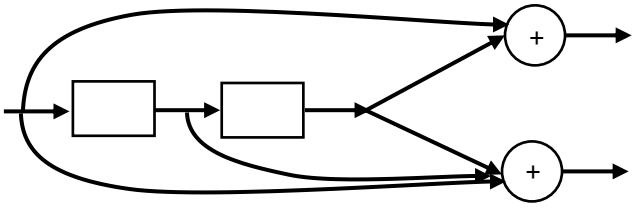

Figure 1: Convolutional Code with K=2

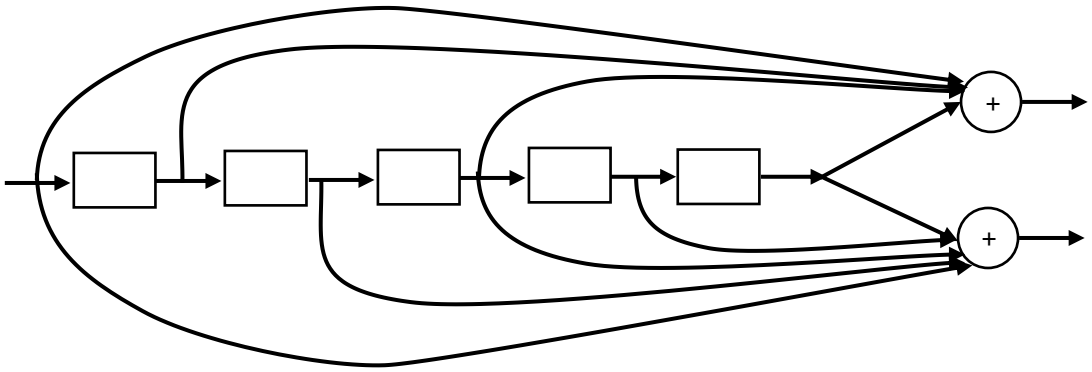

Figure 2: Convolutional Code with K=5

Channel

In order to introduce errors in the system, a channel has to be simulated. Figure 3 shows an example of the Gaussians that would be used to simulate the channel.

In this case, the mean is equal to 1 if a binary 0 was to be transmitted and the mean is equal to -1 if a binary 1 was to be transmitted. The sigma is dependent on the $\frac{E_b}{N_0}$ of the channel. For BPSK Modulation with matched filter detection over the AWGN channel, it can be shown that $\sigma = \frac{1}{\sqrt{2 \times R \times \frac{E_b}{N_0}}}$, where R is the rate of the code. From this equation, is can be observed that, for low values of $\frac{E_b}{N_0}$, the value of sigma would be larger, increasing the possibility of going over to the other symbol.

This makes sense, since, if the channel is very noisy, we would expect that there are more errors. In this case, both code rates are equal to 0.5 and $\frac{E_b}{N_0}$ varied such that different Bit Error Rates can be observed. The plots will be discussed in a later section.

With these 2 parameters, real valued numbers were generated, which simulate the channel. If soft decoding is being done, these values would be passed on to the decoder. However, if hard decoding is being done, the generated values would be compared to 0. If the value is >0, it is assumed that a 0 should enter the decoder whereas if it is <0, then it is assumed that a 1 should be received at the decoder.

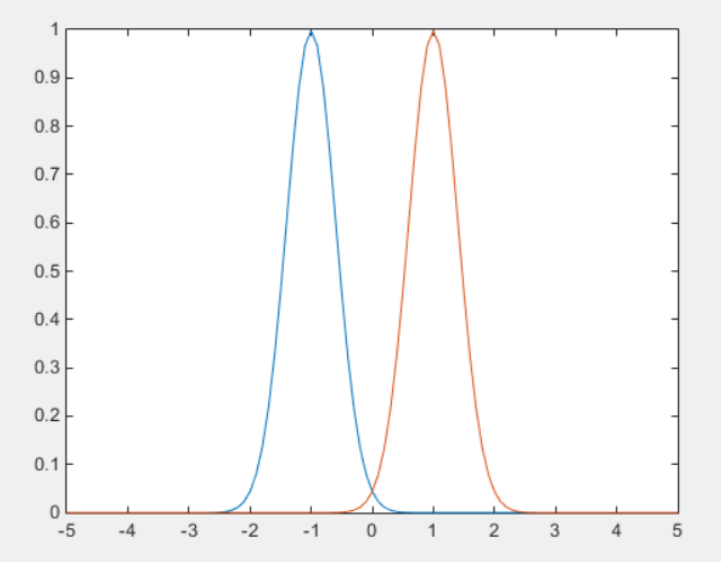

Figure 3: 2 Gaussian Distributions to simulate the channel

Viterbi Decoder

In this case, a Viterbi Decoder was implemented which is capable of performing both hard and soft decision decoding. The Viterbi decoder is based on a trellis. An example of a trellis can be found in Figure 4 below (not related to this work). In this work, the trellis was viewed as a series of layers, with each layer containing 2^K states. Each state has various information related to it, including the transitions that are possible from it. These are also depicted in Figure 4. Classes were created for each of these 3 main components, as can be observed in the attached script.

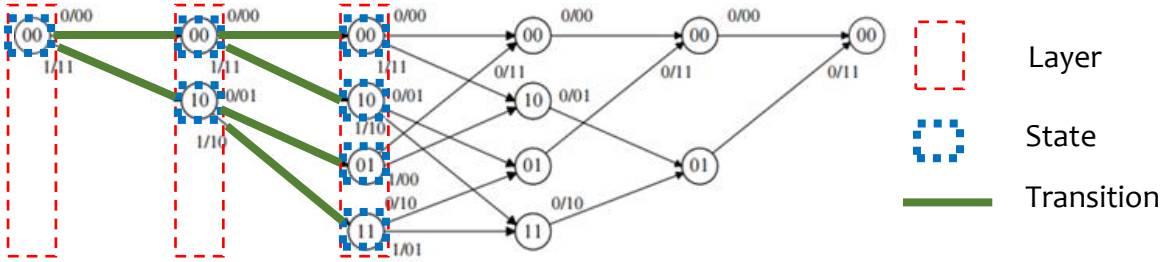

Figure 4: Trellis example, with the different components highlighted

The decoding is done by creating the trellis layer by layer. Part of the information of every state in the layer is the running cost and the previous state. In Viterbi decoding, only the transition with the lowest running cost is retained. Thus, the code checks every state in the previous layer and checks whether the present state is a possible transition from the previous layer's state. If the current state is a possible transition, it takes the output that would have been made in the encoding process, had that transition taken place, and calculates the running cost between that output and what the decoder has received. To obtain the cost, we have to take into consideration whether we are performing hard decoding or soft decoding. If we are performing soft decoding, then we must keep in mind that we changed the representation such that a 1 represents a binary 0 and a -1 represents a binary 1. The cost in both hard and soft decoding is always calculated by the square error, that is:

$$current\ cost = \sum_{all\ values} \left(output\ value_{previous\ state} - received\ decoder\ value\right)^2$$

With the above formula, the cost will be an integer if hard decoding takes place or a real value if soft decoding takes place. In both cases, the cost is always positive, due to the squaring function, which is what is required, since the cost must always keep increasing as we decode further. The total running cost is the previous running cost (from the previous layer) + the current cost. The lowest running cost is maintained in the current state, as well as a link to the state from which we obtained that cost.

Once the above is done for all the states, due to the tailing off that we did in the encoding, we know that we must end up in the all zeros state. Thus, we go to it and back-trace through the links that we kept, as mentioned above. Whilst doing so, we also take the first bit of the state name, since we know that this must be the input bit due to the shifting nature of the shift registers and the fact that there is only one input. Finally, we reverse the list to obtain the output in the correct order and remove the tailing off 0's which we added initially.

Bit Error Rate Plots

Figure 5 below shows the Bit Error Rate (BER) plots show the performance of the 2 decoders, using both hard and soft decision decoding, under various noise conditions. It also shows the theoretical performance of uncoded BPSK modulation on the AWGN channel.

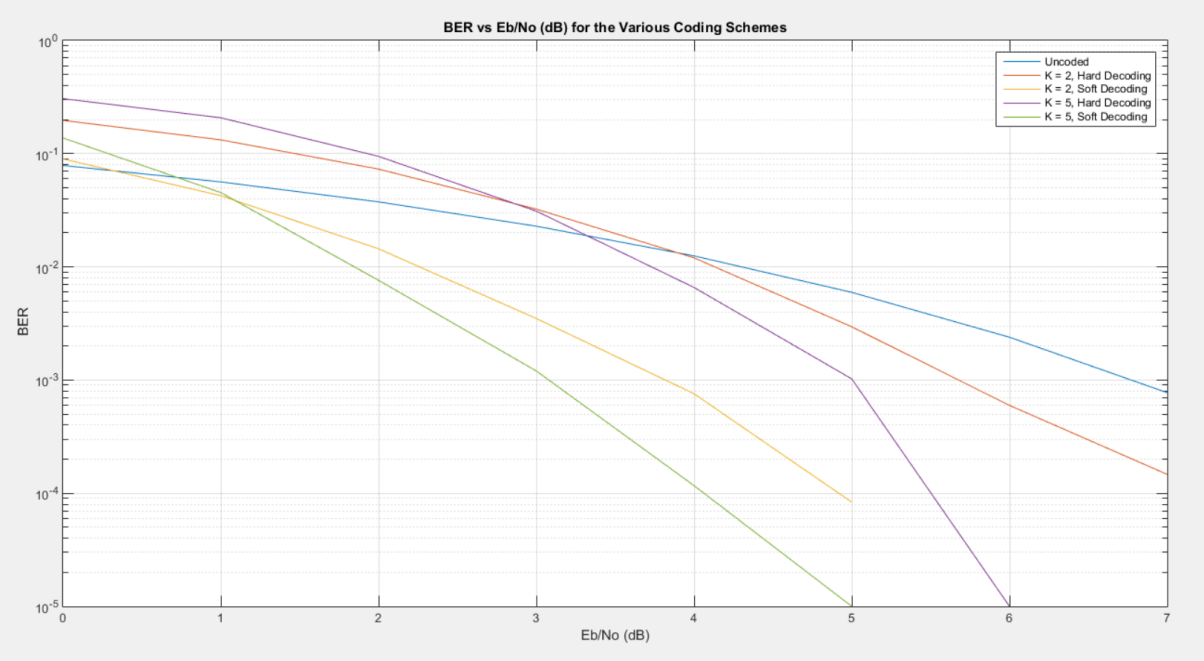

Figure 5: The BER plots obtained for the different coding schemes used and the uncoded case

It is clear that, coding has helped to reduce the bit error rate graphs. However, we must keep in mind that we are now sending 2 bits on the channel for every one information bit. Furthermore, we notice that, for very poor conditions, it is impractical to perform error coding using. In fact, the uncoded case performs better. This is because the conditions are so bad that the code's capabilities have been exceeded. This means that the code is not capable of correcting the erroneous bits, since there are too many of them. However, it still tries to do so and, in the process, ends up flipping correct bits, which increases the BER. What defines poor conditions is dependent on the coding scheme used. The $\frac{E_b}{N_0}$ threshold for the hard decision decoding cases are larger than those for soft decision decoding.

From these curves, it is also clear that soft decision decoding is better than hard decision decoding. This is because soft decision decoding takes into consideration the confidence of the data that is received. In this case, for example, receiving a -1.2 is a much more reliable representation of a binary 1, when compared to receiving a -0.05. This therefore, introduces additional reliability information in the running cost of the trellis and thus, the decoding better.

Furthermore, the code having longer constraint length K equal to 5 performs better than the one with K equal to 2, when comparing hard and soft decision decoding separately. This is because the code with constraint length 5 has more memory and makes the code

more powerful. However, it is also important to note that as K increases, the complexity at the decoder increases exponentially, since it would have 2^K states at every layer.

Modifying the System

The system can be modified to achieve a higher transmission rate. This can be done by modifying the code rate R. Since R is the ratio between the input and output bits, increasing the number of input bits would give us a better R. For example, we could input 2 bits and have the code output 3 bits, which would give us a code rate of $\frac{2}{3}$. This would increase the transmission rate. However, this would also weaken our code. To obtain similar performance at a more favourable R, it is possible to increase the constraint length K. Increasing the constraint length will, however, increase the complexity.

Another way to increase R would be to delete some bits from the encoder's output. This is called code puncturing. Bits are not deleted at random, but according to a puncturing matrix.