Turbo Coder - Implementation and Testing

Jędrzej Stefańczyk, Klemens Zalewski (Translated by ChatGPT4)

03.07.2024

1 Introduction

Turbo codes are a class of high-quality error-correcting codes characterized by the feedback loop between two decoders during message decoding.
To implement such a code, we will create an encoder 2 and the necessary interleaver 2.2. We will describe the algorithm of a single decoder 3 and the way they are interconnected 3.1. For testing, a signal generator 4.1, error generator 4.2, and error detector 4.3 are needed. Testing the algorithm involves generating a bit sequence by the generator, encoding it by the encoder, corrupting the sequence by the error generator, and attempting to decode it by the decoder. The error detector compares the sent sequence with the received one and calculates the BER.

2 Encoder

The main function of the encoder is to generate additional information that will allow for error correction during decoding. In this implementation, we will use recursive systematic convolutional codes. These are codes with memory (shift register) and unlike non-recursive ones, the bit added to the register depends on those already stored. Encoders are characterized by: memory amount - v; number of states - $S = 2^v$; generating polynomials - $G = A_8, B_8$ - their binary representation indicates which memory bits should be connected to the input and output; input to output bit ratio (*rate*) - $R = \frac{a}{b}$.

2.1 Single Encoder

We will use an encoder that gave good results in turbo code implementations by Hagenauer et al. (1996) - with memory $v = 4$ and polynomials $G = \{21_8, 37_8\}$. The scheme of this encoder is shown in 1.
To increase the information that the decoder can use, the encoder always starts in state $S = 0$ (all memory bits are 0) and ends in it. To ensure it finishes in a specific state, we must add at least v bits. The encoder computes these bits and also encodes and outputs them on a separate output.
In the encoders we used, the aforementioned zeroing bits are equal to the memory bits in reverse order, although we cannot guarantee this is true for every encoder.

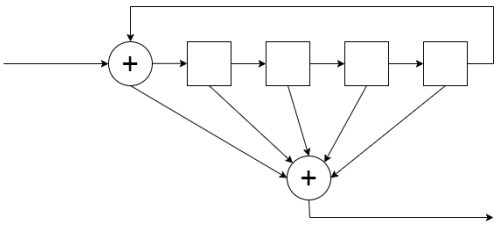

Figure 1: Encoder scheme $G = \{21_8, 37_8\}$

2.2 Interleaver

To increase the amount of additional information, a turbo encoder uses two encoders. Simply adding a second encoder does not help much, as it will give the same result. Using a different encoder is slightly better. To maximize the effects of the second encoder, it must operate on different data. Therefore, its input will pass through an interleaver, an element that changes the order of bits. Its design is very important for maximizing decoding effects. We have defined two conditions for a good interleaver:

- Each v neighboring bits must be as far apart as possible, and if after interleaving any of them are less than v apart, they should at least be in a different order. Encoders generally create information about each v neighboring bits, so we want to separate them as much as possible.
- The first and last bits should be as far from the edges as possible. Since our encoders start and end in a known state, we have the most certainty about the initial and final bits during decoding, and thus we want these bits to be as far apart as possible.

We created the following interleaving procedure:

1. The input is a sequence of length divisible by v.
2. Append every v-th bit of the input to the output.
3. If you exceed the input length, repeat the previous step starting from one bit further than last time.
4. If you repeat the previous step v times, proceed to the step below.
5. Divide the created output in half and reverse both halves, then concatenate them.

Diagram 2 shows the result of this procedure for a sequence of length 8 and $v = 4$.

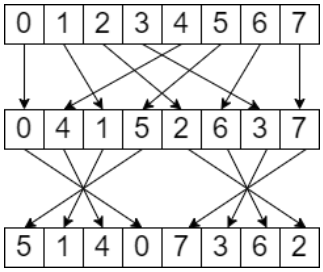

Figure 2: Interleaving diagram for a sequence of length 8 and $v = 4$

2.3 Turbo Encoder

Let us define our Data (*plaintext*) that we want to receive as d, its number of bits as D, and d_k as the k-th data bit.
The entire turbo encoder consists of two encoders, which do not have to be identical, but in our implementation, they are. The first encoder encodes the Data and appends its zeroing bits to them. Then this sequence passes through the interleaver and is encoded by the second encoder. Transmitting the second zeroing bits is problematic. We managed to do it, but it significantly worsened the decoder's performance. Ultimately, we abandoned this solution and don't send the zeroing information of the second encoder at all. Instead, during decoding, we assume that it ends in any state with equal probability. The sent packet consists of the Data, the first, and the second encoded sequence. Diagram 3 graphically shows this procedure along with the number of bits at each moment. As can be seen, the final packet has $3(D+v)$ bits, which gives the overall encoder rate $R = \frac{D}{3(D+v)}$. For large D, it approaches $\frac{1}{3}$, although lower rates give better results.

3 Decoder

Convolutional code decoding is usually done using the Viterbi algorithm. It uses state transition probabilities in the encoder to find the most probable input sequence. In our implementation, we will use an optimized modification of the BAHL algorithm, by Pietrobon and Barbulescu (1994).
The algorithm takes x - received Data and y - received encoded sequence. It returns L - probabilities for each bit. The algorithm involves calculating two three-dimensional probability tables: $\alpha[i][k][m]$ and $\beta[i][k][m]$. α represents probabilities from start to finish, while β from finish to start. We calculate them in two directions due to a problem in the original algorithm resulting from several paths with equal probability. In the algorithm, we define the following notations: $i \in (0, 1)$ - predicted bit, $k \in (D + v)$ - considered bit, $m \in S$ - considered state. For simplicity of implementation, the state number corresponds to the decimal representation of the bits in the encoder memory ($m = 3$ corresponds to memory 0011).
The first step is to calculate the general probabilities for each possible state:

$$\delta[i][k][m] = exp(\frac{1}{2}(x_k * i + y_k * Y(i, m)))$$

For the equation to work correctly, before substitution, we change 0 and 1 to -1 and 1 using the formula $b = (2a - 1)$. The Y function returns the bit that would be encoded by the encoder if it were in state m and received i as the input. The result of this function is also converted to -1 and 1. In the original paper, it is multiplied by a value depending on the channel error, but for values other than $\frac{1}{2}$, we obtained significantly worse results. The next step is to determine α and β using the formulas:

$$\alpha[i][k][m] = \delta[i][k][m] * (\alpha[0][k-1][S_{\leftarrow}(0, m)]+$$

$$+\alpha[1][k-1][S_{\leftarrow}(1, m)])$$

$$\beta[i][k][m] = \beta[0][k+1][S_{\rightarrow}(i, m)]*\delta[0][k+1][S_{\rightarrow}(i, m)]+$$

$$+\beta[1][k+1][S_{\rightarrow}(i, m)] * \delta[1][k+1][S_{\rightarrow}(i, m)]$$

The function $S_{\leftarrow}(i, m)$ returns the state the encoder was in if it is currently in state m and previously received bit i. In contrast, the function $S_{\rightarrow}(i, m)$ returns the state the encoder will be in if it is currently in state m and receives bit i. Both of these functions are recursive, so we start calculating them from the second (or penultimate in the case of β) element. Initially, the arrays are entirely filled with zeros and we pre-fill them as follows: $\alpha[0][1][0] = \delta[0][0][1], \alpha[1][1][0] = \delta[1][0][1], \beta[0][D + v][S_{\leftarrow}(0, 0)] = 1, \beta[1][D + v][S_{\leftarrow}(1, 0)] = 1$.

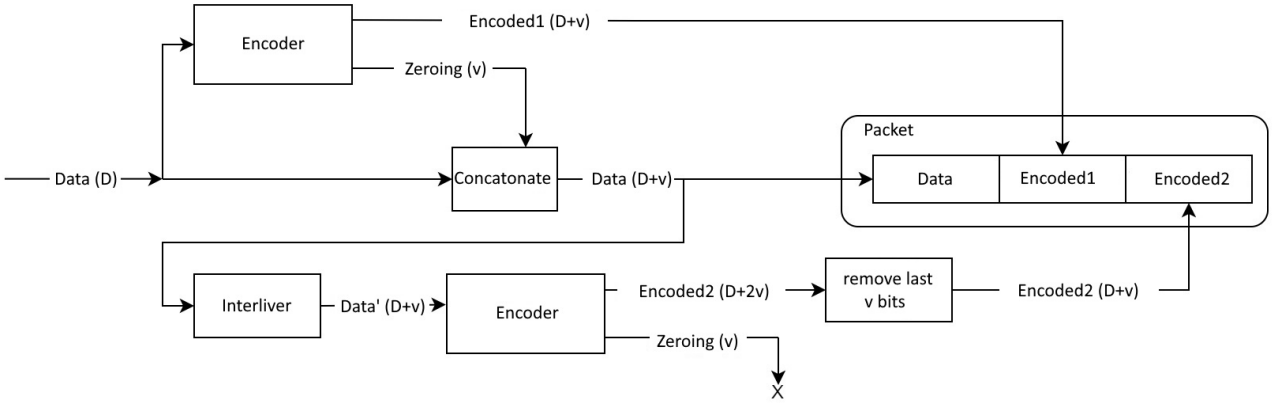

Figure 3: Turbo encoder diagram

We do this to ensure starting/ending in state 0. Since the second decoder does not end in state 0, when decoding its data, we set $\beta[i][D+v][m]$ for each m and i to be equally probable. The final step is to determine the probability for each bit using the formula:

$$L[k] = \frac{\sum_m \alpha[1][k][m] * \beta[1][k][m]}{\sum_m \alpha[0][k][m] * \beta[0][k][m]}$$

Thus, the more positive $L[k]$ is, the higher the probability that $d_k = 1$, while the more negative it is, the higher the probability that $d_k = 0$. Therefore, for the final decoding, we use the following formula:

$$d'_k = \begin{cases} 1 & \text{if} \quad L[k] \geq 0 \\ 0 & \text{if} \quad L[k] < 0 \end{cases}$$

3.1 Feedback

The turbo encoder gets its name from the feedback between two decoders. To connect the two decoders, we want them to exchange information correctly. For this purpose, we define "z" as additional information generated by the encoder that the decoder has recovered, and "W" as the information recovered from the original data. The decoder's output can be written as $L = W + z_1 + z_2$. Next, we will subtract the results of both decoders so that the j-th one receives $W + z_{!j}$ as x. The diagram 4 shows these operations. $z_2 = 0$ in the first iteration.

4 Additional Components

4.1 Signal Generator

To avoid missing any edge cases during tests, the sequences to be encoded are generated pseudo-randomly. For this purpose, we implemented a simple shift register with linear feedback (LSFR) with a generating polynomial $x^{15} + x^{14} + 1$. Its period is 32767, and since we plan to conduct 1000 tests for messages of up to 16 bits in length, we will not go through all states.
We borrowed the implementation from Wikipedia (2024). We use the built-in random function of the language to select the initial state.

4.2 Error Generator

To control the channel's BER, we want to be able to corrupt a specific number of bits in each message. For this purpose, we create an array of length $3(D + v)$ and set a specific number of elements to 1, with the remaining elements set to 0. Then, in each message, we flip the bit at the position where a 1 appears in our array. Before each such corruption, we call the built-in *shuffle()* function on the array, which randomly rearranges the elements.
To ensure the randomness of the functions we use, we checked their implementation. The authors use PCG family algorithms, which provide good distribution uniformity - Godot (2024).

4.3 Error Detector

Since we conduct the tests in a program rather than physically, we can simply compare the message decoded by the decoder with the one generated initially. Therefore, we only check the first D bits. Errors in the additional bits do not affect our data, so we do not count them.
The detector counts the number of messages in which any error occurred and the number of corrupted bits. This allows us to calculate the Bit Error Rate (BER) and Packet Error Rate (PER) at the end of the test.

5 Tests

We conducted tests for messages of length 8. For each setting, we performed 1000 repetitions of the test. We increased the number of corrupted bits by one from 0

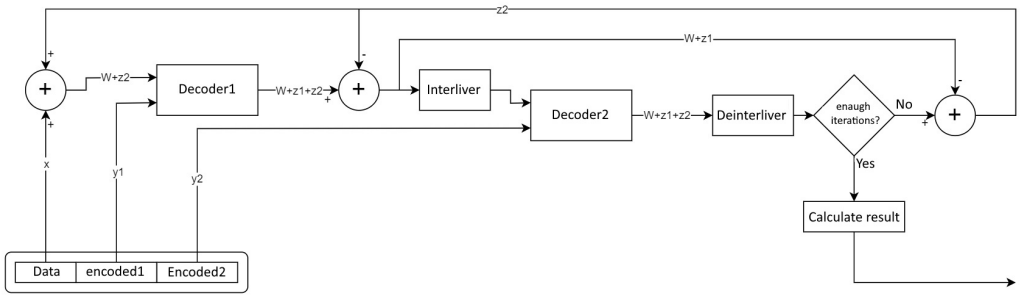

Figure 4: Turbo decoder diagram

to 12. Generally, we performed a small number of iterations due to the long runtime for many iterations. The graph 5 shows the results we obtained.
As can be seen, the encoder works very well when less than $\frac{1}{5}$ of the message is corrupted, and when we exceed $\frac{1}{3}$, it starts to worsen rather than improve. The number of iterations has some impact on the quality of decoding.

References

Godot (2024). Godot engine documentation - random number generation. Wersion: 4.11.

Hagenauer, J., E. Offer, and L. Papke (1996). Iterative decoding of binary block and convolutional codes. *IEEE Transactions on information theory 42*(2), 429–445.

Pietrobon, S. S. and A. S. Barbulescu (1994). A simplification of the modified bahl decoding algorithm for systematic convolutional codes. In *National Conferenc Publication-Institution of Engeneers Australia NCP*, pp. 1073–1077. Citeseer.

Wikipedia (2024). Linear-feedback shift register. Accessed: 17.03.2024.

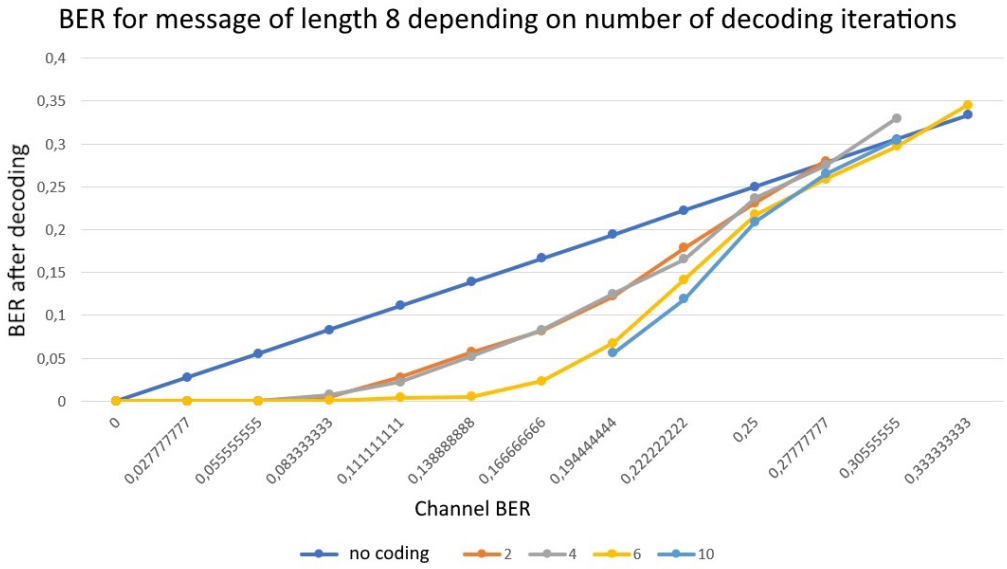

Figure 5: Test results