Analysis and simulation of a digital transmission system

Alessandro Trigolo

2023/2024

Index

Introduction

This document analyses and simulates the behavior of a digital transmission system to have a better understanding of the concept behind these types of telecommunication. The document is split into two main parts:

I **Analysis.** The first part of the document analyzes the transmission system properties following the tasks of the course project guidelines. The numerical outcome of these tasks is summarized in the raw results at the end of the analysis part of the document.

II **Simulation.** The second part of the document focuses on the functions used to properly simulate a digital transmission system. In the simulation part, there will be the explained MATLAB code produced to perform all the processes described in the general schematic.

The full project can be found and downloaded on the public GitHub repository `imAlessas/transmission-simulation.git` where it is possible to find the full MATLAB code of the project and the LaTeX code of the documentation.

General schematic

The full schematic - containing every step - of a transmission system is presented in figure 1. Before exploring the mathematical background hidden between the steps, it is crucial to understand what every phase of the system means.

- *Source.* The source device is whichever device is sending a signal; it could be a television, a computer, a smartphone, or anything else.
- *Formatting Device.* The formatting device's task is to translate the information from analogic to digital which translates into sampling the continuous analogic signal and creating a discrete digital signal that can be transmitted through digital devices.
- *Source Coding.* The source coding goal is lossless data compression. Sure enough, through the Shannon-Fano source coding, the symbols transmitted are encoded to reduce the average codeword length.
- *Channel Coding.* The channel coding goal is to add some control bits that will help detect and eventually correct the errors that occurred during the transmission.
- *Interleaving.* The interleaver is needed to transform package errors into independent errors. This is achieved by changing the ordering of the symbols that will be transmitted.
- *Scrambling.* The scrambling procedure helps with the synchronization between the two devices and improves the security of the transmission. This is achieved by adding a *pseudo-random sequence* to the symbols before the transmission.
- *Modulation.* The modulation process' goal is to match the spectrum of the transmitted signal with the transmission channel bandwidth making the signal

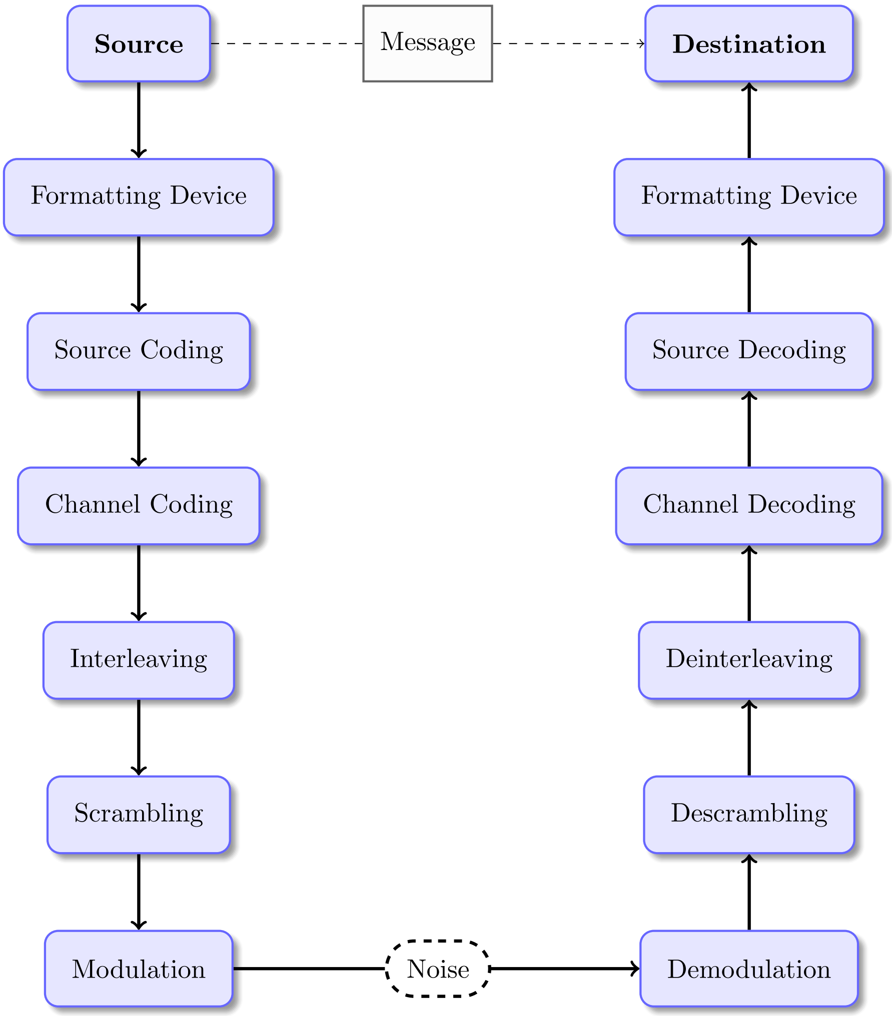

Figure 1: The diagram of the digital information transmission system.

more noise-immune and increasing the data-transfer rate; these operations are performed by the modulator. There are different types of modulation, the one utilized in this project is the *Binary Phase Shift Keying*, which is one of the most effective modulations against noise.

◇ *Noise.* The noise is a crucial obstacle to overcome to have a successful transmission; the noise is the main reason for a wrongly transmitted symbol. There are different types of noise, some of them are generated by other transmissions, others are due to the physical medium and others are caused by the intermediate devices between the transmission. Nevertheless, in every transmission, there will be the *Gaussiam White Noise* which is a thermal noise caused by the Big Bang.

◇ *Demodulation.* In this phase the demodulator device, after receiving the disturbed signal, will try to detect the signal to regenerate the original one. Some-times the noise energy will be stronger than the signal energy generating errors that will be corrected in the next steps.

◇ *Descrambling*. The descrambling procedure is the opposite of the scrambling. The added *pseudo-random sequence*, after the reception is subtracted by the descrambler.

◇ *Deinterleaving*. The deinterleaver reorders the transmitted symbols in the opposite way that the interleaver did. In such a way the *burst* errors that occurred during the transmission will become single errors that can be easily recovered.

◇ *Channel Decoding*. The channel decoding process uses the added bits during the channel encoding to perform an error correction algorithm that will drastically decrease the error rate of the transmission.

◇ *Source Decoding*. The source decoding procedure decompresses the received data into the original symbols. This is achieved by one of the source coding properties: symbols are easily detected because there are no shorter codes at the beginning of longer codes.

◇ *Formatting Device*. During the transmission this device converts the signal from analogic to digital, during the reception of the signal the formatting device translates the discrete digital signal into a continuous analogic signal.

◇ *Destination*. The destination device is whichever device will receive the signal. Likewise the source one, the destination device could be a satellite, a smartphone, a server, or anything else.

Initial parameters

The parameters used in this project have been assigned in a datasheet and are reported in the following list:

- *Symbol duration*: 60 ns, also called τ;
- *SNR*: 8.1 dB;
- *Source code*: Shannon-Fano coding;
- *Error correction code*: cyclic coding with codeword length $m = 31$ and generator polynomial $z^5 \oplus z^2 \oplus 1$;
- *Carrier frequency*: 2.5 GHz;
- *Modulation*: Binary Amplitude Shift Keying (BPSK) with the phase shift of π.

In addition, the source data (alphabet) and the symbols' respective probabilities are summarized in the following table.
Afterwards the parameters have been transcripted in the MATLAB program, as shown in the following snippet.

```
% source number 7
ALPHABET = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
PROBABILITY_VECTOR = [11, 7, 9, 1, 6, 6, 13, 14, 13, 5, 11, 4]/100;

TAU = 60e-9; % symbol duration time, [s]
SNR = 8.1; % Signal-to-Noise-Ration, [dB]
```

Source 7	
a_1	0.11
a_2	0.07
a_3	0.09
a_4	0.01
a_5	0.06
a_6	0.06
a_7	0.13
a_8	0.14
a_9	0.13
a_{10}	0.05
a_{11}	0.11
a_{12}	0.04

```
% Source Code: Shannon-Fano
% Error correction code: Cyclic
CODEWORD_LENGTH = 31; % m
%
F_0 = 2.5e+9; % carrier frequency [Hz]
% Modulation: BPSK
PHASE_SHIFT = pi; % [rad]
U = 1; % amplitude BPSK signal [V]

transmitted_symbol_number = 20;
```

Part I
Analysis

1 Source data

The source data analysis provides a general overview of how the data are generated and how this will impact the encoding scheme. Specifically, the source data analysis is achieved by calculating two important values: the source entropy and the redundancy coefficient.

Source entropy

The source entropy H and the maximum source entropy H_{max}. The entropy of a sequence of symbols is a number that summarizes the randomness of the selection of the symbols in the source sequence. The more uncertain the symbols are, the higher the entropy is and the higher the information the symbols carry. The ideal entropy is when the source symbols are `1 0 1 0 1 0 ...` while the worst entropy is when all the symbols are `1` or `0`. Given a sequence S of N symbols, where each of them has its probability P_i to occur, the entropy of the sequence is:

$$H(S) = -\sum_{i=1}^{N} P_i \log_2 P_i$$

The entropy calculation can be simply achieved with the following MATLAB code. The only thing to note is that P is the probability vector that assigns to every symbol of the alphabet its probability.

```
% sum(V .* log2(V))
H = - dot(PROBABILITY_VECTOR, log2(PROBABILITY_VECTOR));
```

Secondly, in order to calculate the maximum entropy H_{max}, two conditions have to be met: all of the symbols have the same probability $P_i = \frac{1}{N}$ and, of course, they do not correlate one another. Consequently:

$$H_{max}(S) = -\sum_{i=1}^{N} \frac{1}{N} \log_2 \frac{1}{N} = \frac{1}{N} \sum_{i=1}^{N} \log_2 N = \log_2 N$$

Also in this case the MATLAB script to calculate the maximum entropy is trivial.

```
% Number of symbols in the alphabet
N  = length(PROBABILITY_VECTOR);

% Maximum source entropy
H_max = log2(N);
```

By running the scripts, the value obtained are $H = 3.3995$ while $H_{max} = 3.5850$. Reasonably $H < H_{max}$ because the given probabilities in the datasheet weren't equal to each other.

Redundancy coefficient

The redundancy coefficient ρ summarizes in a number how much additional information is present inside the sequence. Essentially, the lower the redundancy coefficient is, the better, because it means that the source entropy is very high. Mathematically, the coefficient ρ can be obtained as follows:

$$\rho = 1 - \frac{H}{H_{max}}(S)$$

which translates into the following code snippet:

```
    % Calculate the redundancy coefficient 'rho'
    source_redoundancy = 1 - H/H_max;
```

Expectedly, the redundancy coefficient is not zero because $H < H_{max}$: by running the script, $\rho = 0.0517$.

2 Source encoding

The source coding analysis provides the necessary tools to evaluate the source coding algorithms for efficient data representation and compression. In this case, the analysis calculates and uses different values to provide a better understanding of the efficiency of the Shannon-Fano source coding. Particularly the values that will be analyzed are the average codeword length $\overline{m}$, the probability of `1` and `0` (P_1 and P_0), the binary entropy H_{bin}, the source data generation rate R and the compression ratio K.

Shannon-Fano algorithm

Before calculating the values it is important to encode the symbols of the alphabet through the Shannon-Fano algorithm. A brief recursive description of it is reported below.

1. Sort the symbol of the alphabet by descending probability;
2. Divide the sets of symbols into two continuous subsets with the same probability (or the lowest difference between the two);
3. Assign to one subset the symbol 1 and the other 0;
4. Repeat until every subset consists of one symbol;
5. Read the codeword from left to right.

By applying the Shannon-Fano algorithm to the given source, the result should be the following.

S	P	Shannon-Fano					Code	m	m_0	m_1
a_8	0.14	1	1	1			111	3	0	3
a_7	0.13			0			110	3	1	2
a_9	0.13		1	1			101	3	1	2
a_1	0.11			0			100	3	2	1
a_{11}	0.11	0	1	1			011	3	1	2
a_3	0.09			0	1		0101	4	2	2
a_2	0.07				0		0100	4	3	1
a_5	0.06		0	1	1		0011	4	2	2
a_6	0.06				0		0010	4	3	1
a_{10}	0.05			0	1		0001	4	3	1
a_{12}	0.04				0	1	00001	5	4	1
a_4	0.01					0	00000	5	5	0

After computing the Shannon-Fano algorithm to the given source, the results should be inserted into the MATLAB program, as follows.

```
    % Probability vector sorted from highest to lowest
    P = sort(PROBABILITY_VECTOR, 'descend');


    % Values obtained with Shannon-Fano code algorithm

    % Symbols codeword length
    m = [3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5];

    % Number of 0s inside the symbols codeword
    m_0 = [0, 1, 1, 2, 1, 2, 3, 2, 3, 3, 4, 5];

    % Number of 1s inside the symbols codeword
m_1 = [3, 2, 2, 1, 2, 2, 1, 2, 1, 1, 1, 0];
```

Binary entropy

At this point, there is all the needed information to calculate the required data for the analysis. First of all, to calculate the average codeword length $\overline{m}$ of N symbols, the following formula should be computed:

$$\overline{m} = \sum_{i=1}^{N} m_i \cdot P_i$$

Additionally, to calculate P_0 and P_1, it is necessary to calculate also the average number of 0 and 1 symbols. The formula is the same as for the average codeword length:

$$\overline{m_0} = \sum_{i=1}^{N} m_{0_i} \cdot P_i \qquad \overline{m_1} = \sum_{i=1}^{N} m_{1_i} \cdot P_i$$

After inserting these formulas in the MATLAB script, the values are $\overline{m} = 3.4300$, $\overline{m_0} = 1.6400$ and $\overline{m_1} = 1.7900$.

```
% Average codeword length
m_average = dot(P, m);

% Average number of 0 symbols
m_0_average = dot(P, m_0);

% Average number of 1 symbols
m_1_average = dot(P, m_1);
```

Moreover, by dividing the number of 0 or 1 symbols by the average length of the codeword the two probabilities, P_0 and P_1, can be computed:

$$P_0 = \frac{\overline{m_0}}{\overline{m}} \qquad\qquad P_1 = \frac{\overline{m_1}}{\overline{m}}$$

The two probabilities values are $P_0 = 0.4781$ and $P_1 = 0.5219$. Ideally, the probabilities should be $P_0 = P_1 = 0.5$; nonetheless, the two values are still very close to each other. Finally, with the two probability values the binary entropy H_{bin} can be obtained using the following calculation.

$$H_{bin}(S) = -P_0 \log_2 P_0 - P_1 \log_2 P_1$$

By running the following MATLAB script, the value of the binary entropy is $H_{bin} = 0.9986$ which is very close to 1. The higher the entropy is, the more uncertainty is associated with every symbol: this means that encoding the initial data with the Shannon-Fano algorithm provides a great value, information-wise.

```
% Probability of 0 symbol
P_0 = m_0_average / m_average;

% Probability of 1 symbol
P_1 = m_1_average / m_average;

% Binary source entropy after coding
H_bin = -P_0 * log2(P_0) - P_1 * log2(P_1);
```

Data rate and compression ratio

After encoding the source data with the Shannon-Fano algorithm, it is important to evaluate the source data generation rate R, which can be calculated as follows:

$$R = \frac{H(S)}{\overline{m}\tau}, \text{ where } \tau \text{ is the symbol duration}$$

The data compression ratio K is important as well: it helps evaluate how much the initial data has been compressed after the source coding. The following formula will help to obtain this value.

$$K = \frac{\overline{m}}{H(S)}$$

After running the MATLAB script displayed below, the data rate $R = 16.519 Mbit/s$ which should be lower than the channel capacity C_{chan} with noise. Moreover, the compression ratio $K = 1.0090$ which is very close to 1, means that the overall compression is low: this is still not a bad result because the overall entropy is increased significantly after the source coding.

```
% Calculate Data Rate
R = H * (m_average * TAU) ^ (-1);

% Calculate Compression Ratio
K = m_average / H;
```

3 Shannon's theorem condition

Shannon's theorem asserts that for reliable communication two important conditions should be verified: using a strong error correction code for a specified SNR value and $R \leq C_{chan} - \epsilon$, $\epsilon \to 0$ meaning that the source data rate R should be less (or, at most equal) that the channel capacity C_{chan}.

It is already possible to compare the data rate R with the noiseless channel capacity C_{bin} by computing this formula:

$$C = \frac{1}{\tau}$$

Expectedly, the result is $C_{bin} = 16.667$ Mbit/s and reasonably meet the Shannon's theorem contidion : $R = 16.5 \text{ Mbps} \leq 16.7 \text{ Mbps} = C_{bin}$.

Bit Error Rate

Before calculating the channel capacity noise, the error probability P_{err}, also called BER (Bit Error Rate), shall be calculated. To do so, by reversing the SNR formula, the energy per bit to noise power spectral density ratio $\frac{E_b}{N_0}$ needs to be calculated:

$$\text{SNR} = 10 \log_{10}(\frac{E_b}{N_0}) \quad \Longrightarrow \quad \frac{E_b}{N_0} = 10^{\frac{\text{SNR}}{10}}$$

Which translates in the following MATLAB line:

```
% Energy per bit to noise power spectral density ratio
Eb_N0 = 10^(SNR / 10);
```

To calculate the error probability of the BPSK modulation time, the following formula should be used:

$$P_{err} = 1 - \Phi\left(\sqrt{2\frac{E_b}{N_0}}\right)$$

Additionally the Φ function can be created using the `erf` function in MATLAB as follows:

$$\Phi(x) = \frac{1}{2}\left[1 + \texttt{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

With this information the MATLAB script can be produced and the BER value can be calculated: $P_{err} = 1.6315 \cdot 10^{-4}$.

```
% define the phi function
phi = @(x) 1/2 * ( 1 + erf(x / sqrt(2)) );

% error probability
P_err = 1 - phi( sqrt( 2 * Eb_N0) );

% no error probability
P_err_comp = 1 - P_err;
```

Channel capacity with noise

All the information needed for the calculation of C_{chan} is now ready for use. To calculate the channel capacity with noise the following formula should be utilized:

$$C_{chan} = \frac{1}{\tau}\left[1 + P_{err}\log_2\left(P_{err}\right) + \left(1 - P_{err}\right)\log_2\left(1 - P_{err}\right)\right]$$

The formula translates in the following MATLAB code line:

```
% Channel capacity with noise
C_chan = ( 1 + P_err * log2(P_err) + P_err_comp *
    log2(P_err_comp) ) * C;
```

By running the script the result is $C_{chan} = 16.629\text{Mbps} \geq 16.519\text{Mbps} = R$ meaning that the Shannon's Theorem condition is fulfilled. Consequently, it is possible to find a coding approach that will recover the errors that occurred during the transmission. If the SNR value was, hypothetically, lower, there was a chance that $R > C_{chan}$ would've translated into the unpossibility of finding an error-correcting code for the transmission.

4 Error correction

The error correction analysis is important to understand how powerful and yet dangerous the error correction codes are. In this document, the analysis focuses on the *cyclic Hamming code* error correction properties even though the conclusions are still valid for the *group Hamming code* (both systematic and non-systematic).

Before analyzing the error correction code, it is necessary to properly implement it. The first thing to do is to generate a binary sequence and the encoding and decoding matrix. To do so it has been used the `cyclgen` function which should be imported from the communication package as follows: `import communications.*`. These steps are summarized in the code snipped below.

```
% generate a sequence of k binary symbols
binary_sequence = randi(2, 1, k) - 1;

% generate decoding and encoding matrix
[cyclic_decoding_matrix, cyclic_encoding_matrix] =
    cyclgen(codeword_length, generation_polynomial);

% reorder the matrixes
% [6 -> 31, 1 ->]
reorder = [6:codeword_length, 1:5];

cyclic_encoding_matrix = cyclic_encoding_matrix (:, reorder);

cyclic_decoding_matrix = (cyclic_decoding_matrix (:,
    reorder))';
```

One crucial thing to do is to redefine the associations between the syndrome values and the error position. The vector that is shown below is not random at all but it has been calculated using the algorithm shown in the chapter 11 and the copy-pasted it. This vector is very important because if it is not defined the correction algorithm won't work at all but will increase the error rate.

```
% Associates the syndrome to the bit.
% This vector has been calculated in the hamming_decoding function
    and copy-pasted here.
associations = [0 31 30 13 29 26 12 20 28 2 25 4 11 23 19 8 27
    21 1 14 24 9 3 5 10 6 22 15 18 17 7 16];
```

After setting up the error correction algorithm, it is possible to begin the analysis by encoding the codeword and studying the behavior of the cyclic Hamming code. Reasonably, by introducing no errors the decoded codeword is the same as the initial codeword.

```
% encode the codeword
codeword = mod(binary_sequence * cyclic_encoding_matrix, 2);
initial_codeword = codeword;

% decode without errors
syndrome_no_error = mod(codeword * cyclic_decoding_matrix, 2);
```

For the next analysis, to properly understand the functioning of the error correction cyclic code, it will be used the following 26-symbols randomly-generated binary sequence:

1 0 1 0 0 1 0 1 0 1 1 0 1 0 1 0 0 1 1 1 1 0 0 1 1 0

The above sequence, after the cyclic hamming encoding will have 31 symbols as follows:

1 0 1 0 0 1 0 1 0 1 1 0 1 0 1 0 0 1 1 1 1 0 0 1 1 0 0 0 1 0 0

One error

By introducing one error to a random position it is necessary to calculate the decimal syndrome value and then use the `associations` vector to detect and correct the error. The code snippet presented below sums up the error correction after one error is displayed below.

```
% introduce one error
error_position = randi(31, 1);
codeword(error_position) =~ codeword(error_position);

codeword_one_error = codeword;

% get the error syndrome
syndrome_one_error = mod(codeword_one_error *
    cyclic_decoding_matrix, 2);

% convert the syndrome into decimal
syndrome_one_error_decimal =
    bin2dec(num2str(syndrome_one_error));

% get the index of the wrong symbol
wrong_symbol_position =
    associations(syndrome_one_error_decimal + 1);

% correct the error
codeword_one_error(wrong_symbol_position) =
    ~codeword_one_error(wrong_symbol_position);
```

By introducing one error in a random position, like position 30, the wrong sequence would be the following:

`1 0 1 0 0 1 0 1 0 1 1 0 1 0 1 0 0 1 1 1 1 0 0 1 1 0 0 0 0 0 0`

Nonetheless the code manages to spot the error using the syndrome value:

`0 0 0 1 0`

It must be highlighted again the importance of the `associations` vector because the decimal value of the syndrome is not 30 but it is 2. The vector bonds the decimal value 2 to the position error 30 succesfully managing to perform the error correction:

`1 0 1 0 0 1 0 1 0 1 1 0 1 0 1 0 0 1 1 1 1 0 0 1 1 0 0 0 1 0 0`

Two errors

By using the below-displayed code, very similar to the previous one, it is possible to introduce a second error to the codeword to analyze the effect of two errors in the codeword.

```
% introduce second error
error_position = randi(31, 1);
```

```
    codeword(error_position) =~ codeword(error_position);

    codeword_two_errors = codeword;

    % get the error syndrome
    syndrome_two_errors = mod(codeword_two_errors *
        cyclic_decoding_matrix, 2);

    % convert the syndrome into decimal
    syndrome_two_errors_decimal =
        bin2dec(num2str(syndrome_two_errors));

    % get the index of the wrong symbol
    wrong_symbol_position =
        associations(syndrome_two_errors_decimal + 1);

    % correct the error
    codeword_two_errors(wrong_symbol_position) =
        ~codeword_two_errors(wrong_symbol_position);
```

By running the code and generating a second error position, like position 11, the codeword becomes the following:

`1 0 1 0 0 1 0 1 0 1 0 0 1 0 1 0 0 1 1 1 1 0 0 1 1 0 0 0 0 0 0`

Unfortunately in this case the syndrome value will not be helpful:

`0 1 1 1 0`

which its decimal value is 14, meaning that, using the `associations` vector, the error position is 19 not corresponding in either the two errors but creating a third error:

`1 0 1 0 0 1 0 1 0 1 0 0 1 0 1 0 0 1 0 1 1 0 0 1 1 0 0 0 0 0 0`

For this reason it is important to analyze the probability of two errors occurring in the codeword (see 7): because with the error correction code the two errors not only not be correct but also a third error will be generated in the attempt.

Three erros particoular situation

If three errors occur in specific positions the algorithm may not even detect the errors because the syndrome is zero. This happens when the three error syndromes cancel each other out. In this case, if the errors are at positions 30 and 11, the critical error position is 14. In this situation, the error syndrome is 0, preventing the algorithm from detecting and correcting any of the three errors. The following code calculates the critical position for any two random error positions using a simple `brute force` algorithm:

```
    % Three errors experiment
    codeword = initial_codeword;

```

```
    % introduce error
    error_position = randi(31, 1)
    codeword(error_position) = ~codeword(error_position);

    % introduce error
    error_position = randi(31, 1)
    codeword(error_position) = ~codeword(error_position);

    critical_position = -1;

    for i = 1 : 31
        % introduce error
        codeword(i) = ~codeword(i);

        % memorize the codeword
        codeword_three_errors = codeword;

        % get the error syndrome
        syndrome_three_errors = mod(codeword_three_errors *
            cyclic_decoding_matrix, 2);

        % convert the syndrome into decimal
        syndrome_three_errors_decimal =
            bin2dec(num2str(syndrome_three_errors));

        % get the index of the wrong symbol
        wrong_symbol_position =
            associations(syndrome_three_errors_decimal + 1);

        if ~wrong_symbol_position
            critical_position = i;
        end
    end
```

5 Bit Error Rate plot

Another type of analysis that is important to make for the Hamming Code is its overall advantages during the transmission. Particularly, it is important to make a comparison between an encoded transmission (with error correction) and a not encoded transmission. To do so it is important to plot an important graph describing the relationship between the BER in relationship with the SNR value.

To plot such a graph it is necessary to evaluate the Bit Error Rate of the transmission of a random binary sequence with the given modulation (BPSK) for different Signal-to-Noise Ratio. The first thing to do is generate the random sequence and generate the BPSK carrier signal using the given initial parameters:

```
% Initialize SNR vector
SNR_vector = 0 : 1/2 : 15;

% Generation of binary sequence
N = 1e4; % number of bits to be sent
N = floor(N / k) * k; % match information block size
binary_sequence = randi(2, 1, N) - 1;

% Generate carrier signal

% Define the time-step
delta_t = tau / samples_per_symbol;

% Time intervals for one symbol
time_intervals = 0: delta_t: tau - delta_t;

% Create the carrier signal
carrier_signal = sin(2 * pi * f0 * time_intervals); % Carrier
    signal

% Calculate the energy per symbol
Eb = dot(carrier_signal, carrier_signal);
```

After creating the carrier signal it is necessary to encode and modulate the randomly generated sequence. To perform the Hamming encoding algorithm it is necessary to create a matrix that will be used in the algorithm. The functioning of the encoding is carefully explained in chapter 11. After Hamming-encoding the sequence the BPSK modulation is performed by transforming the sequence into a Non-Return-to-Zero signal and performing the *Kronecke* multiplication with the carrier signal.

```
% Hamming encoding
% Create the matrix for the Hamming-encoding algorithm
binary_matrix = reshape(binary_sequence, k, N / k)';

% Hamming-encode the matrix
hamming_encoded_matrix = hamming_encoding(binary_matrix,
    codeword_length, k, generation_polynomial); % encode data
hamming_encoded_matrix = hamming_encoded_matrix';

% Unwrap the matrix into a single row
hamming_encoded_sequence = hamming_encoded_matrix(:)';

% Update the number of bits
M = N;
N = length(hamming_encoded_sequence);

% Modulate the sequence with BPSK
BPSK_signal = kron(-2 * hamming_encoded_sequence + 1,
```

```
        carrier_signal);
```

Before calculating the different BER values it is necessary to generate the noise power and standard deviation as follows:

```
    % Generate noise power

    % Reversed SNR formula
    EbN0 = 10.^(SNR_vector / 10);

    % Obtain noise spectral power density
    N0 = Eb./EbN0;

    % Calculate sigma for BPSK
    sigma = sqrt(N0 / 2);


    % Prepare the vectors for the for-loop
    BER_no_hamming = 1 : length(SNR_vector);
    BER_with_hamming = 1 : length(SNR_vector);
```

The crucial section of the analysis is presented in the following `for` loop. First of all the Gaussian White Noise is generated for every entry of the `SNR_vector` variable and then added to the BPSK modulated signal. At this point, the detection is performed using the *optimal correlation receiver* and the BPSK threshold which is zero. Before performing the error correction, the detected sequence is compared with the initial sequence to keep calculating the Bit Error Rate without performing the decoding (`BER_no_hamming`). Secondly, the error correction is performed using the detection algorithm, thoughtfully explained in chapter 11, and then the second BER value is computed (`BER_with_hamming`).

```
    for i = 1 : length(SNR_vector)
        % Calculate the GWN for a specific SNR value
        noise = sigma(i) * randn(1, N * samples_per_symbol);

        % Add the noise
        signal_with_noise = BPSK_signal + noise; % add noise in
            transmitted channel;


        % Use CORRELATION RECEIVER to detect symbols

        % Slice recieved signal into segments in each column
        sliced_signal_with_noise = reshape(signal_with_noise,
            samples_per_symbol, N);

        % Detect the signal with the BPSK threshold
        detected_signal = carrier_signal * sliced_signal_with_noise
            < 0;


```

```
        errors_number_no_hamming = sum(detected_signal ~=
            hamming_encoded_sequence);

        % Calculate BER value
        BER_no_hamming(i) = errors_number_no_hamming / N;


        % Reshape the sequence into a matrix, every row is a codeword
        detected_sequence_matrix = reshape(detected_signal,
            codeword_length, N/codeword_length)';

        % Perform Hamming decoding
        decoded_data_matrix =
            hamming_decoding(detected_sequence_matrix,
            codeword_length, k, generation_polynomial); % encode data
        decoded_data_matrix = decoded_data_matrix';

        % Unwrap the matrix into a sequence
        decoded_data_sequence = decoded_data_matrix(:)';

        % Check number of erros
        errors_number_with_hamming = sum (decoded_data_sequence ~=
            binary_sequence);

        % Calculate BER value
        BER_with_hamming(i) = errors_number_with_hamming/M;
    end
```

All the information to plot the BER curve is known. The following script will provide the plots needed to properly analyze the impact of the cyclic Hamming coding during the noisy transmission. Noticeably, the BER graphics are not linear but should be plotted with the logarithmic scale.

```
    % creates figure and settings
    f = figure(1);
    f.Name = 'Analysis of BER curve';
    f.NumberTitle = 'off';
    f.Position = [450, 100, 700, 600];

    % Draw plot without Hamming code
    semilogy(SNR_vector, BER_no_hamming, 'b'), grid on;

    % Draw plot with Hamming code
    hold on, semilogy(SNR_vector, BER_with_hamming, 'r'), hold off;

    % Draw theoretical plot
    hold on, semilogy(SNR_vector, error_propability, 'm'), hold
        off;

    % Draw SNR project value
```

```
    hold on, plot([SNR SNR], [1e-4, 1e-1], 'g--'), hold off;


    xlabel('Signal-to-Noise Ratio, [dB]'), ylabel('Bit Error
        Rate'); % lables
    ylim([1e-4, 1e-1]), xlim([0, 10]); % limits
    legend('Uncoded', 'Coded', 'Theoretical', 'Given SNR value'); %
        legend
```

By running the MATLAB script the plot obtained is displayed in figure 2. As expected the red plot decreases faster than the blue plot. This is a reasonable and expected result because the error correction code decreases the error rate by correcting the errors occurring during the transmission. Additionally, the given SNR value is plotted with a dotted green line: the red and the green curves do not meet meaning that the given SNR value, the given modulation technique and the given channel coding algorithm are acceptable and valid to successfully perform a digital transmission.

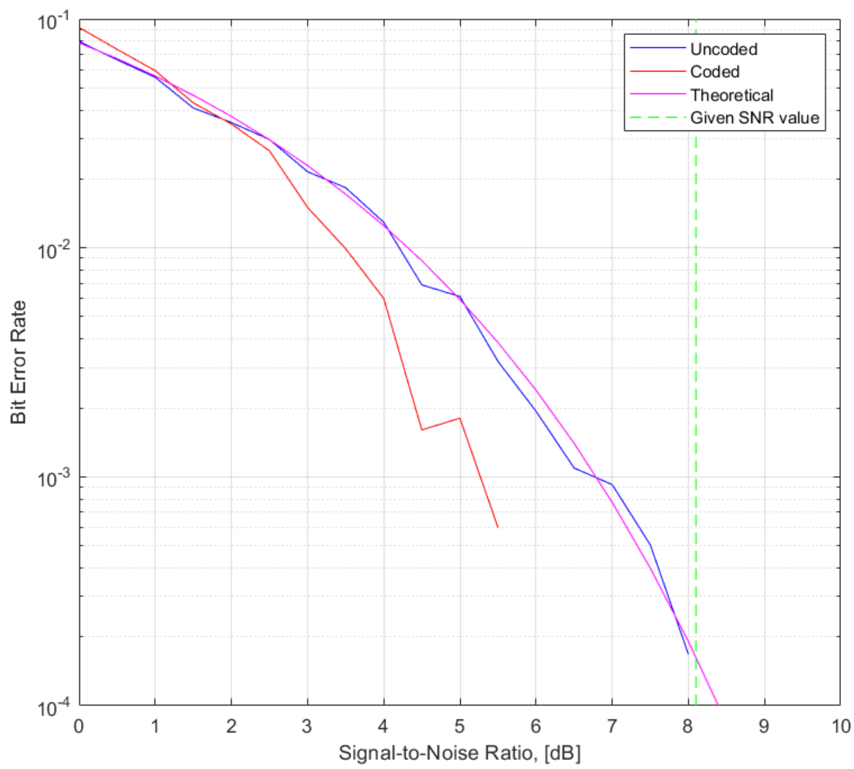

Figure 2: BER vs SNR plot of a randomly generated sequence.

One important thing to notice in figure 2 is the beginning of the three plots: the red one is above the blue one meaning that with a low SNR value (meaning that the power of the signal is almost the same as the noise) the error rate of the coded source is higher compared to the uncoded one. This is because when there are 2 or more errors in the codeword the Hamming code is not able to perform the error correction (as explained in the chapter 4) and creates an additional error in the codeword, consequently raising the error probability (or the BER).

6 Modulated signal spectrum

The spectrum analysis is helpful for a better understanding of the behavior of the BPSK modulation technique. Analyzing the spectrum provides insights into the distribution of signal power across different frequencies. In this section, there will be the analysis an the plots of a periodic `1010` sequence and a randmoly generated sequence.

Periodic `1010` sequence signal

To analyze and plot the spectrum of a periodic `1010` sequence signal some important values should be calculated. The ω_0 value, which is the angular carrier frequency, the value, representing the base harmonic frequency and k, which is the range in which the spectrum will be calculated. The first two values may be calculated with the following expression:

$$\omega_0 = 2\pi f_0 \qquad \Omega = \frac{\pi}{\tau}$$

The k range is a range of n indexes around the carrier frequency central index, $k_0 = \frac{\omega_0}{\Omega}$. These values can be easily obtained by running the below-displayed code snippet.

```
% anguolar carrier frequency
omega_0 = 2 * pi * f0;

% base harmonic angoular frequency
OMEGA = pi / tau;

% Carrier frequency central index
k_0 = omega_0 / OMEGA;

% Define range of indexes for spectrum
k = k_0 + (-10 : 10);
```

At this point, the BPSK spectrum can be calculated. To do so it is necessary to calculate the Fourier series coefficient for the BASK[1] modulation type using the following equation:

$$C_{BASK}(k) = j\frac{U}{4}\frac{\sin\left[(k\Omega - \omega_0)\frac{\tau}{2}\right]}{(k\Omega - \omega_0)\frac{\tau}{2}}$$

Now that the BASK coefficients are calculated, the BPSK coefficients are easy to be computed:

[1]Which stands for Binary Amplitude Shift Keying

$$C_{BPSK}(k) = C_{BASK}(k)\left[e^{+jk\Omega\frac{\tau}{2}} - e^{-jk\Omega\frac{\tau}{2}}\right]$$

These two complex equations can be computed in MATLAB with the help of the `sinc` function as follows:

```
% Phase value of the spectral function
phase = (k * OMEGA - omega_0) * tau / 2;

C_BASK = sinc(phase / pi) * U / 4 * 1j; % fourirer series
    coefficient, BASK

% BPSK spectrum for periodocal '1' and '0' sequence (...1 0 1 0 1 0 1
    0 1 0...)
C_BPSK = C_BASK .* ( exp(1j * k * OMEGA * tau / 2) - exp(- 1j
    * k * OMEGA * tau / 2));
```

The code for plotting the spectrum of the BPSK for a periodic sequence signal is shown below.

```
% creates figure and settings
f = figure(2);
f.Name = 'Analysis of BPSK spectrum';
f.NumberTitle = 'off';
f.Position = [450, 100, 700, 600];

% plot 1st result
subplot(2, 1, 1), stem( k * OMEGA / (2 * pi), abs(C_BPSK), 'b'
    ), grid on,
xlabel('Frequency [GHz]'), ylabel('Amplitude, [V]'),
    title('Amplitude Spectrum of periodic signal')
ylim([-0.05, 0.35]);
```

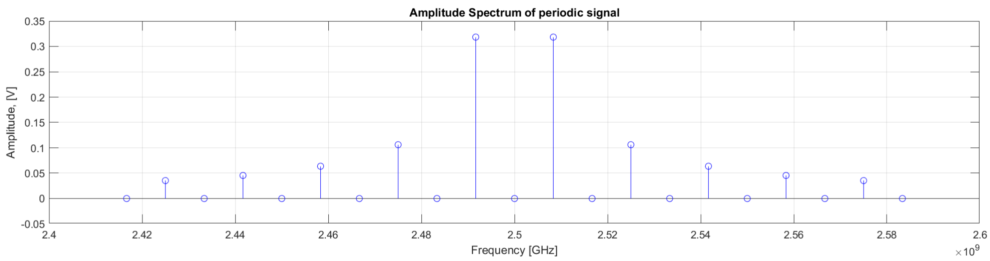

Figure 3: BPSK periodic signal spectrum.

Random sequence signal

we have twice as great value compared for the BASK.

```
% Power Spectral Density (PSD) for random input signal
omega = ( k(1) : 1/100 : k(end) ) * OMEGA; % angoular frequency
```

```

    phase = (omega - omega_0 ) * tau / 2; % continuous phase 2

    S_BASK = 2 * tau * sinc(phase / pi ) * U / 4 * 1j;


    % PSD as a normalized squarred spectral function
    G_BPSK = 1/ tau * abs(S_BASK) .^2;


    subplot(2, 1, 2), plot( omega / (2 * pi), G_BPSK, 'b' ), grid
        on,
    xlabel('Frequency [GHz]'), ylabel('PSD'), title('PSD of random
        signal')
    ylim([-0.1e-8, 1.6e-8]);
```

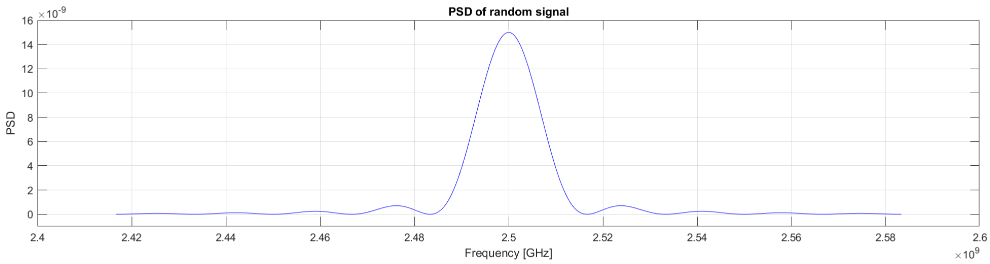

Figure 4: BPSK random signal spectrum.

7 Uncorractable errors

The reason for calculating the probability of 2 or more errors occurring in the same codeword has been explained in the previous chapters. The main reason is that the Hamming code is not able to correct 2 or more errors in the codeword: this does not mean that at least one of them is correct but it leads to the creation of another error. Consequently, the probability of an uncorrectable error is strictly bonded to the fact that with that type of error a new error will be almost surely generated by the Hamming code. For this reason, this probability should be as near to zero as possible.

To calculate such a value the mathematization equation below-displayed should be computed:

$$P_{\geq 2\,err} = 1 - (1 - P_{err})^m - \sum_{i=1}^{g} C_m^i\, P_{err}^i (1 - P_{err})^{m-i}$$

In this particular case, $g = 1$ and $C_m^i = m$ so the equation can be simplified as follows:

$$P_{\geq 2\, err} = 1 - (1 - P_{err})^m - m\, P_{err}^i (1 - P_{err})^{m-1}$$

Which translates in the following MATLAB line:

```
% probability of the case when it is not possible to correct errors
    with the Hamming code (>= 2 errors)
P_uncor = 1 - (P_err_comp)^(codeword_length) - codeword_length
    * P_err * (P_err_comp)^(codeword_length - 1);
```

After running the script the probability of an uncorrectable error $P_{\geq 2\, err} = 1.2338 \cdot 10^{-5}$, which is a low value but, with a high mole of transmitted data there is the possibility to still occur in uncorrectable errors that may lead to an unsuccessful transmission. The probability is still rather low but it is a still possible scenario that should be taken into consideration.

Raw results

In this section, the project's numerical results will be displayed with the purpose of having a direct and straightforward summary of the course project outcomes.

Symbol	Description	Value
	Task 2	
$H(S)$	Source entropy	3.3995
$H(S)_{max}$	Maximum source entropy	3.5850
ρ	Source redundancy	0.0517
	Task 3	
$\overline{m}$	Average codeword length	3.4300
P_0	Probability of "0"	0.4781
P_1	Probability of "1"	0.5219
$H_{bin}(S)$	Binary entropy	0.9986
R	Source data rate	16.519 Mbps
K	Compression ratio	1.0090
	Task 4	
C_{bin}	Noiseless channel capacity	16.667 Mbps
P_{err}	Error probability (BER)	$1.6315 \cdot 10^{-4}$

Conitnue on next page ...

... continued from previous page

Symbol	**Description**	**Value**
C_{chan}	Channel capacity with noise	16.629 Mbps
	Task 8	
$P_{\geq 2\, err}$	Probability of ≥ 2 errors occurring	$1.2338 \cdot 10^{-5}$

Part II
Simulation

8 Data generation algorithm

To simulate the transmission using the given initial parameters it is crucial to generate the symbols following the probabilities specified in the `Source 7` data sheet. To achieve such generation specifics a generation algorithm shall be implemented in MATLAB. The following MATLAB functions will generate a sequence of `n` symbols in the alphabet following their probability distribution.

Comulative distribution probabilities calculator

The first function - `distribution_probability_matrix` - will take as input the `symbol_matrix` whose first row contains the symbols in the alphabet and the second row contains their respective probabilities. The function will return a matrix whose first row is the same but the second row contains cumulative distribution meaning that the second probability value is added to the first, the third is added to the second and so on. Consequently, the last probability value will be one.

```
function result = distribution_probability_matrix(symbol_matrix)
    % Extract probability vector from the symbol matrix
    probability_vector = symbol_matrix(2, :);

    % Get the number of possible symbols
    alphabet_length = length(probability_vector);

    % Calculate the cumulative probability matrix
    cumulative_probability = 0;
    sum_probability_vector = zeros(1, alphabet_length);
    for i = 1:alphabet_length
        % Calculate cumulative probability
        cumulative_probability = cumulative_probability +
            probability_vector(i);

        % Store cumulative probability in the vector
        sum_probability_vector(i) = cumulative_probability;
    end

    % Combine symbols and cumulative probabilities into the result matrix
    result = [symbol_matrix(1, :); sum_probability_vector];
end
```

This helper function will be useful when a random number r between 0 and 1 is generated: the symbol associated with r will be the i-th symbol where $P_{i-1} < r \leq P_i$. In such a way the symbols will have the same probability to be associated with the number r as specified in the datasheet.

Sequence generator

The `distribution_probability_matrix` function will be used in the actual generation function called `symbol_sequence_generator` which generates `n` symbols conforming with their specified probabilities. The below-displayed function will generate a random number r between 0 and 1 and associate it with the i-th symbol whose probability is $P_{i-1} < r \leq P_i$. This is achieved by subtracting the random number from the cumulative probability vector and choosing the symbol with the lowest positive probability value. This procedure is computed `n` time: every temporal association is added to the final result which will be the generated symbol sequence.

```
function result = symbol_sequence_generator(symbol_matrix, n)
    % Initialize an empty vector for the symbol sequence with length n
    result = zeros(1, n);

    % Calculate the cumulative probability matrix using the provided function
    sum_probability_matrix = distribution_probability_matrix(symbol_matrix);

    % Generate the symbol sequence
    for i = 1:n
        % Generate a random number between 0.00 and 1.00
        random_number = round(rand(), 2);

        % Calculate the distance of each cumulative probability from the random number
        distance_from_random_number = sum_probability_matrix(2, :) - random_number;
        distance_from_random_number(distance_from_random_number < 0) = +Inf;

        % Get the index of the symbol with the minimum distance
        [~, symbol] = min(distance_from_random_number);

        % Assign the selected symbol to the result vector
        result(i) = symbol;
    end
end
```

9 Source coding and decoding

The second step is to implement an algorithm that will encode and decode the newly generated symbols using the Shannon-Fano algorithm. The two algorithms are based on the results obtained and displayed in the `Code` column of the table in the "Source encoding" section.

Shannon-Fano encoding

The Shannon-Fano encoding can be achieved by using a very simple switch. Sure enough, the helper function `encode_symbol` displayed below associates with every symbol in the alphabet and its respective codeword.

```
function result = encode_symbol(symbol)
    % Use a switch statement to assign the encoded representation based
        on the input symbol
    switch symbol
        case 1
            result = [ 1 0 0 ];
        case 2
            result = [ 0 1 0 0 ];
        case 3
            result = [ 0 1 0 1 ];
        case 4
            result = [ 0 0 0 0 0 ];
        case 5
            result = [ 0 0 1 1 ];
        case 6
            result = [ 0 0 1 0 ];
        case 7
            result = [ 1 1 0 ];
        case 8
            result = [ 1 1 1 ];
        case 9
            result = [ 1 0 1 ];
        case 10
            result = [ 0 0 0 1 ];
        case 11
            result = [ 0 1 1 ];
        case 12
            result = [ 0 0 0 0 1 ];
    end
end
```

The `shannon_fano_encoding` function takes the `symbol_sequence` as input and encodes it symbol-by-symbol using the aforementioned `encode_symbol` function.

```
function encoded_sequence = shannon_fano_encoding(symbol_sequence)
    encoded_sequence = [];

    % Iterate through the symbol sequence and encode each symbol
    for i = 1:length(symbol_sequence)
        encoded_sequence = [encoded_sequence,
            encode_symbol(symbol_sequence(i))];
    end
end
```

Shannon-Fano decoding

The decoding algorithm performs the exact reverse operation of the encoding algorithm. Sure enough, there is the `decode_symbol` function which translates the binary sequence into its respective symbol.

```
function symbol = decode_symbol(code)
    % Use a switch statement to check each possible code and return the
        corresponding symbol
    switch code
        case '[1 0 0]'
            symbol = 1;
        case '[0 1 0 0]'
            symbol = 2;
        case '[0 1 0 1]'
            symbol = 3;
        case '[0 0 0 0 0]'
            symbol = 4;
        case '[0 0 1 1]'
            symbol = 5;
        case '[0 0 1 0]'
            symbol = 6;
        case '[1 1 0]'
            symbol = 7;
        case '[1 1 1]'
            symbol = 8;
        case '[1 0 1]'
            symbol = 9;
        case '[0 0 0 1]'
            symbol = 10;
        case '[0 1 1]'
            symbol = 11;
        case '[0 0 0 0 1]'
            symbol = 12;
        otherwise
            symbol = []; % Return empty if the code does not match any
                known code
    end
end
```

This function is used in the `shannon_fano_decoding` function wich performs the decoding of the input `encoded_sequence`. The body of the function is a little more complicated than the encoding function because the length of the encoded symbol is not fixed - it can be 3, 4 or 5 - and, as such, every time a new symbol is read from the decoded data, a check should be done to understand if the symbol can be decoded or not.

```
function decoded_sequence = shannon_fano_decoding(encoded_sequence)
    decoded_sequence = [];

    % Iterate through the encoded sequence and decode each symbol
```

```
    current_code = [];
    for i = 1:length(encoded_sequence)
        current_code = [current_code, encoded_sequence(i)];

        % Check if the current code matches any known code
        symbol = decode_symbol(string(mat2str(current_code)));

        % If a symbol is found, add it to the decoded sequence and reset
            the current code
        if ~isempty(symbol)
            decoded_sequence = [decoded_sequence, symbol];
            current_code = [];
        end
    end
end
```

10 Padding bits

One important thing to notice is that the cyclic Hamming encoding is that the algorithm requires an input matrix whose number of elements is a divisor of the information symbols, in this case, 26. The problem is that it is not granted that the generated and encoded sequence is a perfect divisor of 26, so it is crucial to add some padding bits to round the length of the sequence to the closest bigger multiplier of 26. The idea behind this process is to count how many padding bits are needed to round the length of the sequence and store the value into a 5-bit sequence containing the binary number of bits to remove during the reception phase.

Add padding bits

To add the padding bits, the first thing to know is how many padding bits are needed to round the length of the sequence. The purpose of the following helper function is to calculate the number of bits to add. The important thing to notice about the function is the meaning behind the `storage_bits` variable: its value is 5 because with five bits it is possible to store the number between 0 and 31, which is more than the maximum number of padding bits. This is because the worst-case scenario is when the last row of the matrix has 22 elements, meaning that it is not possible to insert the 5-bit sequence and it is necessary to add a new row just to insert the sequence. The new row will have 26 symbols which, in addition to the 4 symbols of the previous row will add to 30, which can be represented with a 5-bit sequence.

```
function number_of_padding_bits =
    count_padding_bits(compact_sequence)

    % Define the number of bits reserved for storing the padding
        information
```

```
    storage_bits = 5;

    % Define the divisor used in determining the number of padding bits
    divisor = 26;

    % Calculate the number of padding bits required
    number_of_padding_bits = divisor -
        rem(length(compact_sequence), divisor);

    % Adjust the number of padding bits if it is less than the
        storage_bits
    if number_of_padding_bits < storage_bits
        number_of_padding_bits = number_of_padding_bits + divisor;
    end
end
```

After obtaining the number of bits to add, the only thing to do is to fill the sequence and at the end incorporate the binary sequence containing the number of added bits as it is shown in the `add_padding_bits` function below.

```
function padded_sequence = add_padding_bits(compact_sequence)
    % Determine the number of bits needed for padding
    bits_to_pad = count_padding_bits(compact_sequence);

    % Convert the number of bits to pad to binary representation
    binary_padding_value = str2num(dec2bin(bits_to_pad, 5)')';

    % Add padding bits to the end of the sequence
    padded_sequence = [compact_sequence, zeros(1, bits_to_pad -
        5), binary_padding_value];
end
```

Remove padding bits

On the other hand during the reception, the only thing to do is to get the last 5 symbols of the sequence and convert them into a decimal number, as shown in the `get_padding_bits` function below.

```
function padding_bits = get_padding_bits(padded_sequence)
    % Extract the last 5 bits from the padded sequence
    bits = padded_sequence(end-4 : end);

    % Convert the binary representation of the bits to a string
    str = '';
    for i = 1 : length(bits)
        str = append(str, num2str(bits(i)));
    end

    % Convert the binary string to decimal to obtain the number of
        padding bits
```

```
    padding_bits = bin2dec(str);
end
```

After getting the number n of padding bits added, the last step is to remove the last n symbols of the sequence to get the original data.

```
function compact_sequence = remove_padding_bits(padded_sequence)
    % Call the get_padding_bits function to determine the number of padding bits
    padding = get_padding_bits(padded_sequence);

    % Remove the padding bits from the end of the sequence
    compact_sequence = padded_sequence(1 : end - padding);
end
```

11 Channel coding and decoding

The Hamming encoding and decoding is the most crucial part of the transmission because it is the one responsible for the error correction of the transmission. There are two types of hamming encoding, group coding and cyclic coding: in this case, cyclic coding has been utilized to perform the error correction. It is important to note that the channel coding adds some bits to the codeword: precisely during the encoding phase to the codeword are added 5 symbols, making the sequence length a perfect divisor of $26+5 = 31$, meanwhile after the decoding the five symbols at the end of the codeword are removed making it again a perfect divisor of 26. It is important to highlight that the Hamming coding (both the group and cyclic) is able to correct only one error per codeword. If there is more than one error, the code will not only not correct the error but it will create other errors by attempting the correction, as already explained on section 4. This is one of the reasons the interleaving process is needed.

Cyclic Hamming encoding

To perform the Hamming encoding to the sequence it is necessary to generate the `encoding_matrix` using the `cyclgen` function[2] that uses the codeword length and the generation polynomial defined at the beginning of the code. The `hamming_encoding` function, uses the generated encoding matrix to perform a matrix multiplication and encode the codeword.

```
function encoded_data_matrix = hamming_encoding(binary_data_matrix, codeword_length, k, generation_polynomial)
    % Calculate the number of redundant symbols (parity symbols)
    r = codeword_length - k;
```

[2]Note that the function needs to be imported: `import communications.*`.

```
    % Generate the cyclic encoding matrix based on the generator
        polynomial
    [~, cyclic_encoding_matrix] = cyclgen(codeword_length,
        generation_polynomial);

    % Reorder the encoding matrix to match Hamming code requirements
    reorder = [r + 1 : codeword_length, 1 : r];
    cyclic_encoding_matrix = cyclic_encoding_matrix(:, reorder);

    % Calculate control symbol values using matrix multiplication
    % Use rem() function to find modulo 2 sum as a remainder of division
        by 2
    encoded_data_matrix = rem(binary_data_matrix *
        cyclic_encoding_matrix, 2);
end
```

Cyclic Hamming decoding

The decoding algorithm is slightly more complicated due to its error-correction properties. The below-displayed `hamming_decoding` function has the purpose of analyzing the encoded data and, by calculating the `syndrome_value`, performing the error correction. After the decoding, the codeword length won't be 31 anymore but will return to 26.

```
function decoded_data_matrix =
    hamming_decoding(encoded_data_matrix, codeword_length, k,
    generation_polynomial)
    % Determine the number of control symbols
    r = codeword_length - k;

    % Specify syndrome calculation matrix
    [~, cyclic_encoding_matrix] = cyclgen(codeword_length,
        generation_polynomial);
    syndrome_matrix = cyclic_encoding_matrix(:, (1:r));
    syndrome_matrix = [syndrome_matrix; eye(r)];

    % Calculate syndrome for each codeword
    syndrome_value = rem(encoded_data_matrix * syndrome_matrix, 2);
    syndrome_value = syndrome_value * 2.^(r - 1 : -1 : 0)';

    % Calculate error indexes based on syndrome values
    error_indexes = get_error_indexes(syndrome_matrix,
        syndrome_value, codeword_length);

    % Define error vector table
    error_vector = [zeros(1, codeword_length);
                    eye(codeword_length)];

```

```
    % Perform correction by adding the error vector to the received
        codeword
    codeword = rem(encoded_data_matrix +
        error_vector(error_indexes + 1, :), 2);

    % Read the columns of data symbols to form decoded data
    decoded_data_matrix = codeword(:, 1:k);
end
```

Noticeably, a crucial part of the cyclic error correction algorithm is the calculation of the `error_indexes`. In the cyclic coding, the syndrome value and the error position are not linearly associated. For example, the syndrome value `0 0 0 0 1`$_2 = 1$ does not mean that the error is at index 1, but it is in position 31 instead and the syndrome `1 1 1 1 1`$_2 = 31$ is not associated with the position 31 but with the index 16. The association between the syndrome value and the error position is deterministic and the `get_error_indexes` function helps to associate the error-index and the syndome decimal value.

```
function error_indexes = get_error_indexes(syndrome_matrix,
    syndrome_value, codeword_length)
    % Initialize vector to store decimal values of binary syndrome
        patterns
    syndrome_decimal_value_vector = [];

    % Convert binary syndrome values to decimal for association
    for i = 1 : codeword_length
        syndrome_decimal_value_vector =
            [syndrome_decimal_value_vector;
            bin2dec(num2str(syndrome_matrix(i, :)))];
    end

    % Combine positions and corresponding decimal values for sorting
    associations = [transpose(1:codeword_length),
        syndrome_decimal_value_vector];

    % Sort associations based on decimal values for efficient error
        detection
    associations = sortrows(associations, 2);

    % Create correction index for mapping syndrome values to error
        positions
    correction_index = [0, associations(:, 1)'];

    % Map syndrome values to error positions using correction index
    error_indexes = correction_index(syndrome_value + 1);
end
```

12 Interleaving and deinterleaving

The interleaving and deinterleaving process is needed to prevent group errors, also called *burst errors*. This is achieved by deterministically mixing the sequence before the transmission and recomposing it by performing the initial algorithm in reverse. In such a way, if during the communication a burst error happens when the sequence is restored in the initial order, the possibility of having these types of errors drastically decreases. Noticeably this algorithm can be performed multiple times in the same sequence to minimize the probability of burst errors.

Interleaving

The `interleaving` function uses a matrix transpose algorithm to mix the sequence. Noticeably the unmixed sequence is compressed into the `interleaver_matrix`: the first 31 symbols of the sequence are inserted into the first column, the second 31 symbols are inserted into the second column and so on. After filling the matrix, to create the interleaved sequence it is necessary to read the matrix through the rows: this is the purpose of the second `for` loop.

```
function mixed_sequence = interleaving(unmixed_sequence)
    % Define the length of each column (also the number of rows)
    column_length = 31;

    % Calculate the length of each row (also the number of columns) based on the input sequence length
    row_length = length(unmixed_sequence) / column_length;

    % Write on the columns and read on the rows to create the interleaved matrix
    interleaver_matrix = [];

    % Iterate through the rows
    for i = 1 : row_length
        % Extract the current column from the unmixed sequence
        current_column = unmixed_sequence(column_length * (i-1) + 1 : column_length * i)';

        % Append the current column to the matrix
        interleaver_matrix = [interleaver_matrix, current_column];
    end

    % Initialize the interleaved sequence
    mixed_sequence = [];

    % Iterate through the columns of the matrix
    for i = 1 : column_length
        % Append the elements from each row of the current column to the interleaved sequence
        mixed_sequence = [mixed_sequence, interleaver_matrix(i, 1 : end)];
```

```
    end

end
```

Deinterleaving

The `deinterleaving` function is the exact opposite of the interleaving function. In this case, the sequence is inserted into the rows of the `deinterleaver_matrix` and then read through the column. Reasonably the body of the function is very similar to its reverse function.

```
function unmixed_sequence = deinterleaving(mixed_sequence)
    % Define the length of each column (also the number of rows)
    column_length = 31;

    % Calculate the number of columns (also the number of rows) based on
        the input sequence length
    row_length = length(mixed_sequence) / column_length;

    % Initialize the matrix for deinterleaving
    deinterleaver_matrix = [];

    % Iterate through the columns of the interleaved sequence
    for i = 1 : column_length
        % Extract the current column from the mixed sequence
        current_column = mixed_sequence(row_length * (i-1) + 1 :
            row_length * i)';

        % Append the current column to the matrix
        deinterleaver_matrix = [deinterleaver_matrix,
            current_column];
    end

    % Initialize the deinterleaved sequence
    unmixed_sequence = [];

    % Iterate through the rows of the matrix
    for i = 1 : row_length
        % Append the elements from each column of the current row to the
            deinterleaved sequence
        unmixed_sequence = [unmixed_sequence,
            deinterleaver_matrix(i, 1 : end)];
    end

end
```

13 Scrambling and descrambling

The scrambling process helps with the synchronization between the sender machine and the receiver device. More precisely, when there is a long sequence of `"0"` or `"1"` symbols it becomes rather difficult to understand how many of them are being transmitted. For this purpose, before modulating and transmitting the signal, is extremely helpful to perform a logical `XOR` to every codeword with a pseudo-random sequence, called `scambling_key`. This is the purpose of the `scrambling` function: it applies an exclusive `OR` bitwise with a key that is known both from the source and the destination.

```
function scrambled_sequence = scrambling(unscrambled_sequence,
    scrambler_key)
    % Initialize the output sequence
    scrambled_sequence = [];

    % Determine the length of the scrambler key
    codeword_length = length(scrambler_key);

    % Loop through the input sequence in codeword-sized chunks
    for i = 1 : length(unscrambled_sequence) / codeword_length
        % Extract the current codeword from the input sequence
        current_codeword = unscrambled_sequence(codeword_length *
            (i - 1) + 1 : codeword_length * i);

        % Perform XOR operation with the scrambler key
        scrambled_codeword = xor(current_codeword, scrambler_key);

        % Append the scrambled codeword to the output sequence
        scrambled_sequence = [scrambled_sequence,
            scrambled_codeword];
    end
end
```

There is no need to create the `descrambling` function because the `XOR` operation is cyclical: performing this operation two times to a sequence of binary numbers will return the initial sequence. In this case, the function has been created just for better clarity and readability.

```
function unscrambled_sequence = descrambling(scrambled_sequence,
    scrambler_key)
    % Utilize the scrambling function in reverse to perform descrambling
    unscrambled_sequence = scrambling(scrambled_sequence,
        scrambler_key);
end
```

14 Modulation and noise

After translating, coding, mixing and changing the sequence it is finally time to simulate the signal transmission process. In this case, the Binary Phase Shift Keying has been utilized to modulate and demodulate the binary sequence. To create this type of signal, it is necessary to create the `carrier_signal` which is a sine wave with the properties given in the initial parameters. Then the carrier signal energy and its length are stored to calculate the Gaussian White Noise afterwards. The BPSK modulation is performed by creating an *NRZ* signal of the sequence and then using the *Kronecker* product as shown below.

```
% Define the time-step
delta_t = tau / samples_per_symbol;

% Time intervals for one symbol
time_intervals = 0: delta_t: tau - delta_t;

% Create the carrier signal
carrier_signal = sin(2 * pi * f0 * time_intervals);

% Calculate the energy per symbol
Eb = dot(carrier_signal, carrier_signal);

% Save length of encoded sequence
N = length(binary_sequence);

% Perform BPSK modulation
BPSK_signal = kron(-2 * binary_sequence + 1, carrier_signal);
```

To add the GWN is necessary to calculate the noise signal by reversing the SNR formula to obtain the noise spectral power density N_0. With this value, it is possible to obtain the standard deviation, σ and then generate the `noise_signal` which will be added to the modulated signal to compute the `disturbed_signal`.

```
% Reversed SNR formula
EbN0 = 10^(SNR / 10);

% Obtain noise spectral power density
N0 = Eb./EbN0;

% Calculate sigma for BPSK
sigma = sqrt(N0 / 2);

% Create noise signal
noise_signal = sigma * randn(1, N * samples_per_symbol);

% Create disturbed signal by adding noise to the modulated signal
disturbed_signal = BPSK_signal + noise_signal;
```

The signal detection is performed using the *correlation receiver* approach. The disturbed signal is sliced and then it is compared to its respective modulation threshold, which is 0 in this case.

```
% Slice the received signal into segments in each column
sliced_disturbed_signal = reshape(disturbed_signal,
    samples_per_symbol, N);

% Detect the signal with the BPSK threshold
detected_signal = carrier_signal * sliced_disturbed_signal < 0;
```

Tests

All the above-displayed functions have been meticulously tested using newly random-generated sequences hundreds of times to spot and correct any type of logical errors. The tests are public and can be found under the folder `/src/func/test` in the public repository mentioned at the very beginning of this document.