Analysis and simulation of a digital transmission system

Alessandro Trigolo

2023/2024

Index

Todo list

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.** `Raw results` section

ii **Simulation.**

The full project can be found and downloaded on the public GitHub repository `imAlessas/transmission-simulation.git`.

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 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,

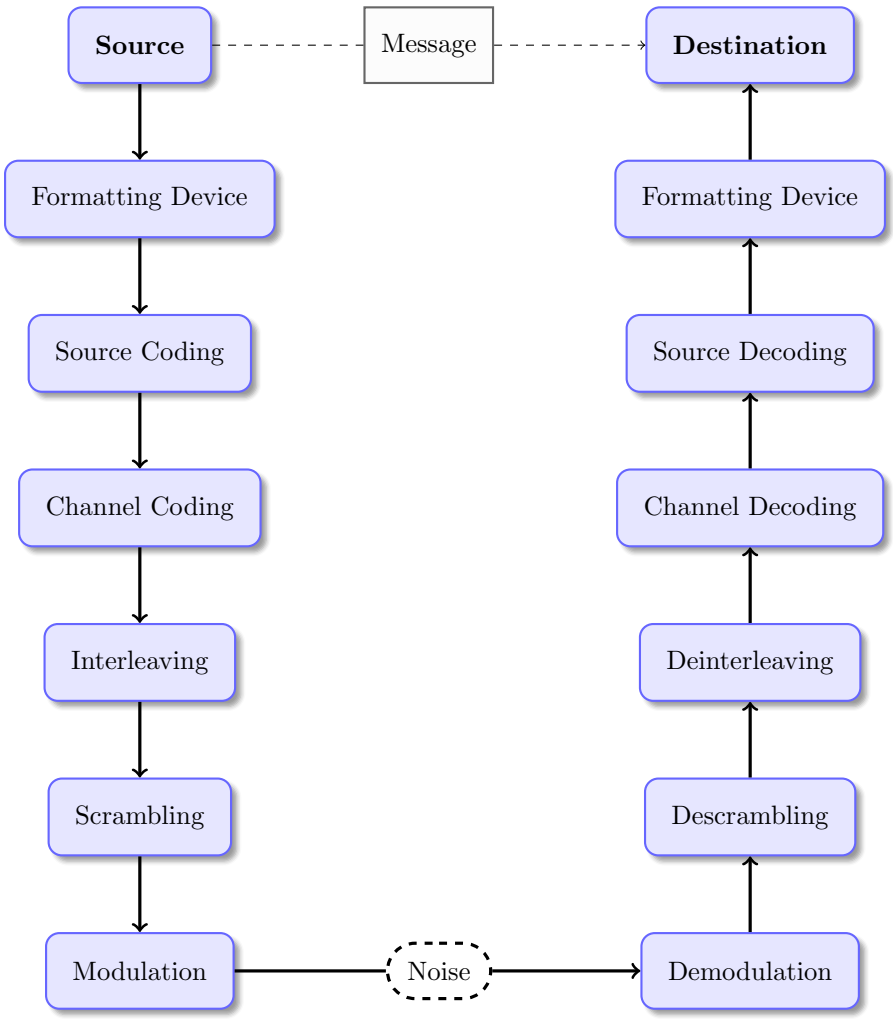

Figure 1: The diagram of the digital information transmission system.

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. Sometimes 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 op-

posite 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.

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

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 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
Simulation

1 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
```

2 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
```

3 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
```

4 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. 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[1] 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;

```

[1]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
```

5 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
```

6 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
```

7 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 and *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.