

Analysis and simulation of a digital transmission system

Alessandro Trigolo

2023/2024

Index

Introduction	5
General schematic	5
Initial parameters	7
I Analysis	9
1 Source data	9
Source entropy	9
Redundancy coefficient	10
2 Source encoding	10
Shannon-Fano algorithm	10
Binary entropy	11
Data rate and compression ratio	12
3 Shannon's theorem condition	13
Bit Error Rate	13
Channel capacity with noise	14
4 Error correction	14
One error	16
Two errors	16
5 Bit Error Rate	19
6 Modulated signal spectrum	19
7 Uncorrectable errors	19
Raw results	19
II Simulation	21
1 Data generation algorithm	21
Cumulative distribution probabilities calculator	21
Sequence generator	22
2 Source coding and decoding	22
Shannon-Fano encoding	23
Shannon-Fano decoding	24
3 Padding bits	25
Add padding bits	25
Remove padding bits	26

4	Channel coding and decoding	27
	Cyclic Hamming encoding	27
	Cyclic Hamming decoding	28
5	Interleaving and deinterleaving	30
	Interleaving	30
	Deinterleaving	31
6	Scrambling and descrambling	32
7	Modulation and noise	33
	Tests	34

Todo list

Make a little introduction in the "Raw result" section 19

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 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 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.

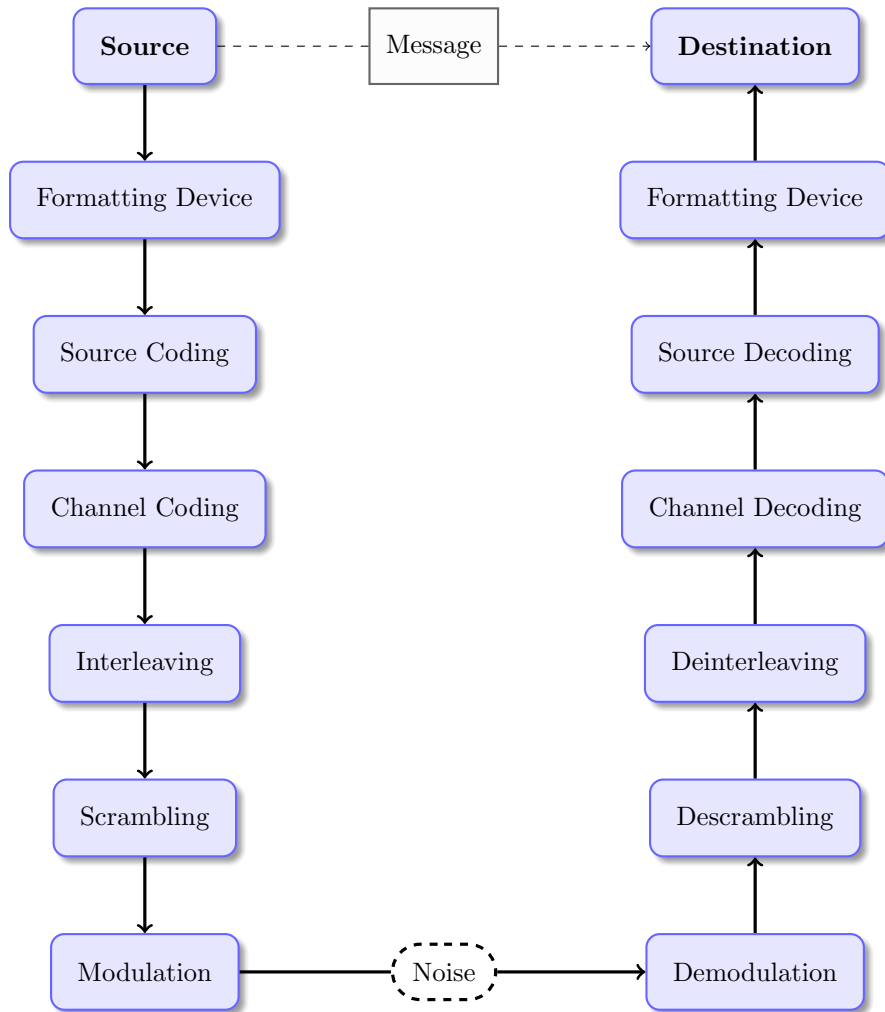


Figure 1: The diagram of the digital information transmission system.

- ◇ *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 *Gaussian 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 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 smart-phone, 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 transcribed in the MATLAB program, as shown in the following snippet.

```
1  % source number 7
2  ALPHABET = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
3  PROBABILITY_VECTOR = [11, 7, 9, 1, 6, 6, 13, 14, 13, 5, 11, 4]/100;
4
5  TAU = 60e-9; % symbol duration time, [s]
6  SNR = 8.1; % Signal-to-Noise-Ration, [dB]
7  % Source Code: Shannon-Fano
8  % Error correction code: Cyclic
9  CODEWORD_LENGTH = 31; % m
10 %
11 F_0 = 2.5e+9; % carrier frequency [Hz]
```

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

```

12 % Modulation: BPSK
13 PHASE_SHIFT = pi; % [rad]
14 U = 1; % amplitude BPSK signal [V]
15
16 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.

```
1 | % sum(V .* log2(V))
2 | 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.

```
1 | % Number of symbols in the alphabet
2 | N = length(PROBABILITY_VECTOR);
3 |
4 | % Maximum source entropy
5 | 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:

```
1 | % Calculate the redundancy coefficient 'rho'
2 | source_redondancy = 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 \bar{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		011	3	1	2
a_3	0.09		1	0	1	0101	4	2	2
a_2	0.07				0	0100	4	3	1
a_5	0.06			1	1	0011	4	2	2
a_6	0.06	0			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.

```

1  % Probability vector sorted from highest to lowest
2  P = sort(PROBABILITY_VECTOR, 'descend');
3
4
5  % Values obtained with Shannon-Fano code algorithm
6
7  % Symbols codeword length
8  m = [3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5];
9
10 % Number of 0s inside the symbols codeword
11 m_0 = [0, 1, 1, 2, 1, 2, 3, 2, 3, 3, 4, 5];
12
13 % Number of 1s inside the symbols codeword
14 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:

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

After inserting these formulas in the MATLAB script, the values are $\overline{m} = 3.4300$, $\overline{m}_0 = 1.6400$ and $\overline{m}_1 = 1.7900$.

```

1 % Average codeword length
2 m_average = dot(P, m);
3
4 % Average number of 0 symbols
5 m_0_average = dot(P, m_0);
6
7 % Average number of 1 symbols
8 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:

$$P_0 = \frac{\overline{m}_0}{\overline{m}} \quad P_1 = \frac{\overline{m}_1}{\overline{m}}$$

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.

```

1 % Probability of 0 symbol
2 P_0 = m_0_average / m_average;
3
4 % Probability of 1 symbol
5 P_1 = m_1_average / m_average;
6
7 % Binary source entropy after coding
8 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 $R = 16.519 \text{ Mbit/s}$ 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.

```

1 | % Calculate Data Rate
2 | R = H * (m_average * TAU) ^ (-1);
3 |
4 | % Calculate Compression Ratio
5 | 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 $R \leq C_{chan} - \epsilon$, $\epsilon \rightarrow 0$ meaning that the source data rate R should be less (or, at most equal) than 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 $C_{bin} = 16.667 \text{ Mbit/s}$ and reasonably meet the Shannon's theorem condition : $R = 16.5 \text{ Mbps} \leq 16.7 \text{ Mbps} = C_{bin}$.

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:

$$\text{SNR} = 10 \log_{10}\left(\frac{E_b}{N_0}\right) \implies \frac{E_b}{N_0} = 10^{\frac{\text{SNR}}{10}}$$

Which translates in the following MATLAB line:

```

1 | % Energy per bit to noise power spectral density ratio
2 | 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:

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

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

```

1  % define the phi function
2  phi = @(x) 1/2 * ( 1 + erf(x / sqrt(2)) );
3
4  % error probability
5  P_err = 1 - phi( sqrt( 2 * Eb_N0) );
6
7  % no error probability
8  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:

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

The formula translates in the following MATLAB code line:

```

1  % Channel capacity with noise
2  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 impossibility 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 snippet below.

```

1  % generate a sequence of k binary symbols
2  binary_sequence = randi(2, 1, k) - 1;
3
4  % generate decoding and encoding matrix
5  [cyclic_decoding_matrix, cyclic_encoding_matrix] =
6      cyclgen(codeword_length, generation_polynomial);
7
8  % reorder the matrixes
9  % [6 -> 31, 1 ->]
10 reorder = [6:codeword_length, 1:5];
11
12 cyclic_encoding_matrix = cyclic_encoding_matrix (:, reorder);
13
14 cyclic_decoding_matrix = (cyclic_decoding_matrix (:,
15     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 4 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.

```

1  % Associates the syndrome to the bit.
2  % This vector has been calculated in the hamming_decoding function
3  % and copy-pasted here.
4  associations = [0 31 30 13 29 26 12 20 28 2 25 4 11 23 19 8 27
5      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.

```

1  % encode the codeword
2  codeword = mod(binary_sequence * cyclic_encoding_matrix, 2);
3  initial_codeword = codeword;
4
5  % decode without errors
6  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.

```
1 % introduce one error
2 error_position = randi(31, 1);
3 codeword(error_position) = ~ codeword(error_position);
4
5 codeword_one_error = codeword;
6
7 % get the error syndrome
8 syndrome_one_error = mod(codeword_one_error *
9     cyclic_decoding_matrix, 2);
10
11 % convert the syndrome into decimal
12 syndrome_one_error_decimal = bin2dec(num2str(syndrome_one_error));
13
14 % get the index of the wrong symbol
15 wrong_symbol_position = associations(syndrome_one_error_decimal +
16     1);
17
18 % correct the error
19 codeword_one_error(wrong_symbol_position) =
20     ~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

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 successfully 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.

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



```

4
5 codeword_two_errors = codeword;
6
7 % get the error syndrome
8 syndrome_two_errors = mod(codeword_two_errors *
    cyclic_decoding_matrix, 2);
9
10 % convert the syndrome into decimal
11 syndrome_two_errors_decimal =
    bin2dec(num2str(syndrome_two_errors));
12
13 % get the index of the wrong symbol
14 wrong_symbol_position = associations(syndrome_two_errors_decimal +
    1);
15
16 % correct the error
17 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

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 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 errors particular 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:

```

1 % Three errors experiment
2 codeword = initial_codeword;
3
4 % introduce error

```

```

5 error_position = randi(31, 1)
6 codeword(error_position) = ~codeword(error_position);
7
8 % introduce error
9 error_position = randi(31, 1)
10 codeword(error_position) = ~codeword(error_position);
11
12 critical_position = -1;
13
14 for i = 1 : 31
15     % introduce error
16     codeword(i) = ~codeword(i);
17
18     % memorize the codeword
19     codeword_three_errors = codeword;
20
21     % get the error syndrome
22     syndrome_three_errors = mod(codeword_three_errors *
23         cyclic_decoding_matrix, 2);
24
25     % convert the syndrome into decimal
26     syndrome_three_errors_decimal =
27         bin2dec(num2str(syndrome_three_errors));
28
29     % get the index of the wrong symbol
30     wrong_symbol_position =
31         associations(syndrome_three_errors_decimal + 1);
32
33     if ~wrong_symbol_position
34         critical_position = i;
35     end
36 end

```

5 Bit Error Rate

6 Modulated signal spectrum

7 Uncorrectable errors

Raw results

Make a little introduction in the "Raw result" section

Symbol	Description	Value
<i>Task 2</i>		
$H(S)$	Source entropy	3.3995
$H(S)_{max}$	Maximum source entropy	3.5850
ρ	Source redundancy	0.0517
<i>Task 3</i>		
\bar{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
<i>Task 4</i>		
C_{bin}	Noiseless channel capacity	16.667 Mbps
P_{err}	Error probability (BER)	$1.6315 \cdot 10^{-4}$
C_{chan}	Channel capacity with noise	16.629 Mbps
<i>Task 5</i>		
<i>Task 6</i>		

Conitnue on next page ...

... continued from previous page

Symbol	Description	Value
<i>Task 7</i>		
<i>Task 8</i>		
$P_{\geq 2\,err}$	Probability of ≥ 2 errors occurring	$1.2338 \cdot 10^{-5}$

Part II

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.

Cumulative 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.

```
1 function result = distribution_probability_matrix(symbol_matrix)
2     % Extract probability vector from the symbol matrix
3     probability_vector = symbol_matrix(2, :);
4
5     % Get the number of possible symbols
6     alphabet_length = length(probability_vector);
7
8     % Calculate the cumulative probability matrix
9     cumulative_probability = 0;
10    sum_probability_vector = zeros(1, alphabet_length);
11    for i = 1:alphabet_length
12        % Calculate cumulative probability
13        cumulative_probability = cumulative_probability +
14            probability_vector(i);
15
16        % Store cumulative probability in the vector
17        sum_probability_vector(i) = cumulative_probability;
18    end
19
20    % Combine symbols and cumulative probabilities into the result matrix
21    result = [symbol_matrix(1, :); sum_probability_vector];
22 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.

```
1 function result = symbol_sequence_generator(symbol_matrix, n)
2     % Initialize an empty vector for the symbol sequence with length n
3     result = zeros(1, n);
4
5     % Calculate the cumulative probability matrix using the provided
6     % function
7     sum_probability_matrix =
8         distribution_probability_matrix(symbol_matrix);
9
10    % Generate the symbol sequence
11    for i = 1:n
12        % Generate a random number between 0.00 and 1.00
13        random_number = round(rand(), 2);
14
15        % Calculate the distance of each cumulative probability from the
16        % random number
17        distance_from_random_number = sum_probability_matrix(2, :)
18            - random_number;
19        distance_from_random_number(distance_from_random_number <
20            0) = +Inf;
21
22        % Get the index of the symbol with the minimum distance
23        [~, symbol] = min(distance_from_random_number);
24
25        % Assign the selected symbol to the result vector
26        result(i) = symbol;
27    end
28 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.

```
1 function result = encode_symbol(symbol)
2     % Use a switch statement to assign the encoded representation based
3     % on the input symbol
4     switch symbol
5         case 1
6             result = [ 1 0 0 ];
7         case 2
8             result = [ 0 1 0 0 ];
9         case 3
10            result = [ 0 1 0 1 ];
11        case 4
12            result = [ 0 0 0 0 0 ];
13        case 5
14            result = [ 0 0 1 1 ];
15        case 6
16            result = [ 0 0 1 0 ];
17        case 7
18            result = [ 1 1 0 ];
19        case 8
20            result = [ 1 1 1 ];
21        case 9
22            result = [ 1 0 1 ];
23        case 10
24            result = [ 0 0 0 1 ];
25        case 11
26            result = [ 0 1 1 ];
27        case 12
28            result = [ 0 0 0 0 1 ];
29    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.

```
1 function encoded_sequence = shannon_fano_encoding(symbol_sequence)
2     encoded_sequence = [];
3
4     % Iterate through the symbol sequence and encode each symbol
5     for i = 1:length(symbol_sequence)
6         encoded_sequence = [encoded_sequence,
7                             encode_symbol(symbol_sequence(i))];
8     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.

```
1 function symbol = decode_symbol(code)
2     % Use a switch statement to check each possible code and return the
3     % corresponding symbol
4     switch code
5         case '1 0 0',
6             symbol = 1;
7         case '0 1 0 0',
8             symbol = 2;
9         case '0 1 0 1',
10            symbol = 3;
11        case '0 0 0 0 0',
12            symbol = 4;
13        case '0 0 1 1',
14            symbol = 5;
15        case '0 0 1 0',
16            symbol = 6;
17        case '1 1 0',
18            symbol = 7;
19        case '1 1 1',
20            symbol = 8;
21        case '1 0 1',
22            symbol = 9;
23        case '0 0 0 1',
24            symbol = 10;
25        case '0 1 1',
26            symbol = 11;
27        case '0 0 0 0 1',
28            symbol = 12;
29        otherwise
30            symbol = []; % Return empty if the code does not match any
31            % known code
32    end
33 end
```

This function is used in the `shannon_fano_decoding` function which 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.

```
1 function decoded_sequence = shannon_fano_decoding(encoded_sequence)
2     decoded_sequence = [];
3
4     % Iterate through the encoded sequence and decode each symbol
```



```

5   current_code = [];
6   for i = 1:length(encoded_sequence)
7       current_code = [current_code, encoded_sequence(i)];
8
9       % Check if the current code matches any known code
10      symbol = decode_symbol(string(mat2str(current_code)));
11
12      % If a symbol is found, add it to the decoded sequence and reset
13      % the current code
14      if ~isempty(symbol)
15          decoded_sequence = [decoded_sequence, symbol];
16          current_code = [];
17      end
18  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.

```

1   function number_of_padding_bits =
2       count_padding_bits(compact_sequence)
3
4       % Define the number of bits reserved for storing the padding
5       % information

```

```

4     storage_bits = 5;
5
6     % Define the divisor used in determining the number of padding bits
7     divisor = 26;
8
9     % Calculate the number of padding bits required
10    number_of_padding_bits = divisor -
        rem(length(compact_sequence), divisor);
11
12    % Adjust the number of padding bits if it is less than the
        storage_bits
13    if number_of_padding_bits < storage_bits
14        number_of_padding_bits = number_of_padding_bits + divisor;
15    end
16 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.

```

1 function padded_sequence = add_padding_bits(compact_sequence)
2     % Determine the number of bits needed for padding
3     bits_to_pad = count_padding_bits(compact_sequence);
4
5     % Convert the number of bits to pad to binary representation
6     binary_padding_value = str2num(dec2bin(bits_to_pad, 5)');
7
8     % Add padding bits to the end of the sequence
9     padded_sequence = [compact_sequence, zeros(1, bits_to_pad -
        5), binary_padding_value];
10 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.

```

1 function padding_bits = get_padding_bits(padded_sequence)
2     % Extract the last 5 bits from the padded sequence
3     bits = padded_sequence(end-4 : end);
4
5     % Convert the binary representation of the bits to a string
6     str = '';
7     for i = 1 : length(bits)
8         str = append(str, num2str(bits(i)));
9     end
10
11    % Convert the binary string to decimal to obtain the number of
        padding bits

```

```

12 padding_bits = bin2dec(str);
13 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.

```

1 function compact_sequence = remove_padding_bits(padded_sequence)
2     % Call the get_padding_bits function to determine the number of
3     padding bits
4     padding = get_padding_bits(padded_sequence);
5
6     % Remove the padding bits from the end of the sequence
7     compact_sequence = padded_sequence(1 : end - padding);
8 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, 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` function¹ 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.

```

1 function encoded_data_matrix =
2     hamming_encoding(binary_data_matrix, codeword_length, k,
3     generation_polynomial)
4     % Calculate the number of redundant symbols (parity symbols)
5     r = codeword_length - k;

```

¹Note that the function needs to be imported: `import communications.*`.

```

4
5      % Generate the cyclic encoding matrix based on the generator
      polynomial
6      [~, cyclic_encoding_matrix] = cyclgen(codeword_length,
      generation_polynomial);
7
8      % Reorder the encoding matrix to match Hamming code requirements
9      reorder = [r + 1 : codeword_length, 1 : r];
10     cyclic_encoding_matrix = cyclic_encoding_matrix(:, reorder);
11
12     % Calculate control symbol values using matrix multiplication
13     % Use rem() function to find modulo 2 sum as a remainder of division
      by 2
14     encoded_data_matrix = rem(binary_data_matrix *
      cyclic_encoding_matrix, 2);
15 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.

```

1 function decoded_data_matrix =
      hamming_decoding(encoded_data_matrix, codeword_length, k,
      generation_polynomial)
2 % Determine the number of control symbols
3 r = codeword_length - k;
4
5 % Specify syndrome calculation matrix
6 [~, cyclic_encoding_matrix] = cyclgen(codeword_length,
      generation_polynomial);
7 syndrome_matrix = cyclic_encoding_matrix(:, (1:r));
8 syndrome_matrix = [syndrome_matrix; eye(r)];
9
10 % Calculate syndrome for each codeword
11 syndrome_value = rem(encoded_data_matrix * syndrome_matrix, 2);
12 syndrome_value = syndrome_value * 2.^(r - 1 : -1 : 0)';
13
14 % Calculate error indexes based on syndrome values
15 error_indexes = get_error_indexes(syndrome_matrix,
      syndrome_value, codeword_length);
16
17 % Define error vector table
18 error_vector = [zeros(1, codeword_length);
19                 eye(codeword_length)];
20

```

```

21 % Perform correction by adding the error vector to the received
    codeword
22 codeword = rem(encoded_data_matrix +
    error_vector(error_indexes + 1, :), 2);
23
24 % Read the columns of data symbols to form decoded data
25 decoded_data_matrix = codeword(:, 1:k);
26 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 syndrome decimal value.

```

1 function error_indexes = get_error_indexes(syndrome_matrix,
    syndrome_value, codeword_length)
2 % Initialize vector to store decimal values of binary syndrome
    patterns
3 syndrome_decimal_value_vector = [];
4
5 % Convert binary syndrome values to decimal for association
6 for i = 1 : codeword_length
7     syndrome_decimal_value_vector =
        [syndrome_decimal_value_vector;
        bin2dec(num2str(syndrome_matrix(i, :)))];
8 end
9
10 % Combine positions and corresponding decimal values for sorting
11 associations = [transpose(1:codeword_length),
    syndrome_decimal_value_vector];
12
13 % Sort associations based on decimal values for efficient error
    detection
14 associations = sortrows(associations, 2);
15
16 % Create correction index for mapping syndrome values to error
    positions
17 correction_index = [0, associations(:, 1)'];
18
19 % Map syndrome values to error positions using correction index
20 error_indexes = correction_index(syndrome_value + 1);
21 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.

```
1 function mixed_sequence = interleaving(unmixed_sequence)
2     % Define the length of each column (also the number of rows)
3     column_length = 31;
4
5     % Calculate the length of each row (also the number of columns) based
6     % on the input sequence length
7     row_length = length(unmixed_sequence) / column_length;
8
9     % Write on the columns and read on the rows to create the interleaved
10    % matrix
11    interleaver_matrix = [];
12
13    % Iterate through the rows
14    for i = 1 : row_length
15        % Extract the current column from the unmixed sequence
16        current_column = unmixed_sequence(column_length * (i-1) + 1
17            : column_length * i)';
18
19        % Append the current column to the matrix
20        interleaver_matrix = [interleaver_matrix, current_column];
21    end
22
23    % Initialize the interleaved sequence
24    mixed_sequence = [];
25
26    % Iterate through the columns of the matrix
27    for i = 1 : column_length
28        % Append the elements from each row of the current column to the
29        % interleaved sequence
30        mixed_sequence = [mixed_sequence, interleaver_matrix(i, 1 :
31            column_length)];
32    end
33 end
```

```

27     end
28
29 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.

```

1  function unmixed_sequence = deinterleaving(mixed_sequence)
2      % Define the length of each column (also the number of rows)
3      column_length = 31;
4
5      % Calculate the number of columns (also the number of rows) based on
6      % the input sequence length
7      row_length = length(mixed_sequence) / column_length;
8
9      % Initialize the matrix for deinterleaving
10     deinterleaver_matrix = [];
11
12     % Iterate through the columns of the interleaved sequence
13     for i = 1 : column_length
14         % Extract the current column from the mixed sequence
15         current_column = mixed_sequence(row_length * (i-1) + 1 :
16             row_length * i)';
17
18         % Append the current column to the matrix
19         deinterleaver_matrix = [deinterleaver_matrix,
20             current_column];
21     end
22
23     % Initialize the deinterleaved sequence
24     unmixed_sequence = [];
25
26     % Iterate through the rows of the matrix
27     for i = 1 : row_length
28         % Append the elements from each column of the current row to the
29         % deinterleaved sequence
30         unmixed_sequence = [unmixed_sequence,
31             deinterleaver_matrix(i, 1 : end)];
32     end
33 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 `scrambling_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.

```
1 function scrambled_sequence = scrambling(unscrambled_sequence,
    scrambler_key)
2     % Initialize the output sequence
3     scrambled_sequence = [];
4
5     % Determine the length of the scrambler key
6     codeword_length = length(scrambler_key);
7
8     % Loop through the input sequence in codeword-sized chunks
9     for i = 1 : length(unscrambled_sequence) / codeword_length
10        % Extract the current codeword from the input sequence
11        current_codeword = unscrambled_sequence(codeword_length *
            (i - 1) + 1 : codeword_length * i);
12
13        % Perform XOR operation with the scrambler key
14        scrambled_codeword = xor(current_codeword, scrambler_key);
15
16        % Append the scrambled codeword to the output sequence
17        scrambled_sequence = [scrambled_sequence,
            scrambled_codeword];
18    end
19 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.

```
1 function unscrambled_sequence = descrambling(scrambled_sequence,
    scrambler_key)
2     % Utilize the scrambling function in reverse to perform descrambling
3     unscrambled_sequence = scrambling(scrambled_sequence,
        scrambler_key);
4 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 an *NRZ* signal of the sequence and then using the *Kronecker* product as shown below.

```
1  % Define the time-step
2  delta_t = tau / samples_per_symbol;
3
4  % Time intervals for one symbol
5  time_intervals = 0: delta_t: tau - delta_t;
6
7  % Create the carrier signal
8  carrier_signal = sin(2 * pi * f0 * time_intervals);
9
10 % Calculate the energy per symbol
11 Eb = dot(carrier_signal, carrier_signal);
12
13 % Save length of encoded sequence
14 N = length(binary_sequence);
15
16 % Perform BPSK modulation
17 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`.

```
1  % Reversed SNR formula
2  EbN0 = 10^(SNR / 10);
3
4  % Obtain noise spectral power density
5  N0 = Eb./EbN0;
6
7  % Calculate sigma for BPSK
8  sigma = sqrt(N0 / 2);
9
10 % Create noise signal
11 noise_signal = sigma * randn(1, N * samples_per_symbol);
12
13 % Create disturbed signal by adding noise to the modulated signal
14 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.

```
1  % Slice the received signal into segments in each column
2  sliced_disturbed_signal = reshape(disturbed_signal,
3                                   samples_per_symbol, N);
4
5  % 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.