**Communication Systems Project**



**Projects description.**

This project involves simulating and evaluating the performance of different line coding schemes and a binary phase shift-keying (BPSK) system.

In Part I, the experiment involves generating a stream of random bits and line coding the bits using various schemes such as Polar non-return to zero, Uni-polar return to zero, Bipolar return to zero, and Manchester coding. The eye diagram and spectral domains of the pulses are plotted, and a receiver is designed to calculate the bit error rate (BER) and count the number of errors. Noise is added to the received signal, and the experiment is repeated for different levels of noise.

In Part II, the experiment involves generating a stream of random bits and line coding the bits using Polar non-return to zero. The modulated BPSK signal is plotted in the time and frequency domains, and a receiver is designed to calculate the bit error rate (BER) and count the number of errors.

Overall, this project aims to provide a hands-on experience in simulating and evaluating the performance of communication systems using different line coding schemes and a BPSK system.

**GITHUB Link:** <https://github.com/Mahmoud-Abdelraouf/Fundamentals-of-Communications-Systems-Project>

ASU – ENG

[ECE252s] – Fundamentals of Communications Systems

Fundamentals of Communications Systems Project

# **TABLE OF CONTENTS**

[1 TABLE OF CONTENTS 1](#_Toc135651287)

[2 TEAM MEMBERS 4](#_Toc135651288)

[3 BRIEF ABOUT PROJECT PART ONE 4](#_Toc135651289)

[4 IMPORTANT NOTES 5](#_Toc135651290)

[5 PART I TRANSMITTER 6](#_Toc135651291)

[5.1 LINE CODING SYSTEMS 6](#_Toc135651292)

[5.2 SPECTRAL DOMAIN FUNCTION 13](#_Toc135651293)

[5.3 THE MAIN.M FILE 15](#_Toc135651294)

[5.4 HERE IS A SUMMARY OF THE CODE 15](#_Toc135651295)

[5.5 SNAPSHOTS 17](#_Toc135651296)

[5.5.1 Unipolar Non-Return to Zero 17](#_Toc135651297)

[5.5.2 Polar Non-Return to Zero 17](#_Toc135651298)

[5.5.3 Unipolar Return to Zero 18](#_Toc135651299)

[5.5.4 Bipolar Return To Zero 18](#_Toc135651300)

[5.5.5 Manchester Coding 19](#_Toc135651301)

[5.5.6 Spectral Domain of Unipolar Non-Return to Zero 19](#_Toc135651302)

[5.5.7 Spectral Domain of Polar Non-Return to Zero 20](#_Toc135651303)

[5.5.8 Spectral Domain of Unipolar Return to Zero 20](#_Toc135651304)

[5.5.9 Spectral Domain of Bipolar Return to Zero 21](#_Toc135651305)

[5.5.10 Spectral Domain of Manchester Coding 21](#_Toc135651306)

[5.5.11 Eye Diagram of Unipolar Non-Return to Zero 22](#_Toc135651307)

[5.5.12 Eye Diagram of Polar Non-Return to Zero 22](#_Toc135651308)

[5.5.13 Eye Diagram of Unipolar Return to Zero 23](#_Toc135651309)

[5.5.14 Eye Diagram of Bipolar Return to Zero 23](#_Toc135651310)

[5.5.15 Eye Diagram of Manchester Coding 24](#_Toc135651311)

[6 PART I RECEIVER 24](#_Toc135651312)

[6.1 USED FUNCTIONS 24](#_Toc135651313)

[6.2 UNIPOLAR NRZ 30](#_Toc135651314)

[6.2.1 CODE 30](#_Toc135651315)

[6.2.2 GRAPHS 31](#_Toc135651316)

[6.2.3 POLAR NRZ 33](#_Toc135651317)

[6.2.4 UNIPOLAR RZ 35](#_Toc135651318)

[6.2.5 BIPOLAR RZ 37](#_Toc135651319)

[6.2.6 MANCHESTER LINE CODING 39](#_Toc135651320)

[6.2.7 Plot ber\_values of different line coding with sigma values 41](#_Toc135651321)

[6.3 MAIN.M FILE 42](#_Toc135651322)

[6.4 Bonus (FOR THE CASE OF BIPOLAR RETURN TO ZERO, DESIGN AN ERROR DETECTION CIRCUIT. COUNT THE NUMBER OF DETECTED ERRORS IN CASE OF DIFFERENT NUMBER OF SIGMA (USE THE OUTPUT OF STEP 8). 45](#_Toc135651323)

[6.5 WORKSPACE 46](#_Toc135651324)

[6.6 FOCUS ON SOME VARIABLES 47](#_Toc135651325)

[6.6.1 NO. OF ERRORS SIGNALS 1 -> 5 without noise 47](#_Toc135651326)

[6.6.2 BER FOR SIGNALS 1 - > 5 without noise 47](#_Toc135651327)

[6.6.3 BER\_VALUES VS SIGMA FOR DIFFERENT VARIABLES 48](#_Toc135651328)

[6.6.4 NOISY SIGNALS AT SIGMA = 0.2 48](#_Toc135651329)

[6.6.5 CONCLUSIONS 48](#_Toc135651330)

[7 PART II 49](#_Toc135651331)

[7.1 THE USED FUNCTIONS 49](#_Toc135651332)

[7.2 PART II TRANSIMITTER 50](#_Toc135651333)

[7.2.1 Generate stream of random bits (100 bit) (This bit stream should be selected to be random, which means that the type of each bit is randomly selected by the program code to be either ‘1’ or ‘0’.) 50](#_Toc135651334)

[7.2.2 Line code the stream of bits (pulse shape) according to Polar non return to zero (Maximum voltage +1, Minimum voltage -1). 51](#_Toc135651335)

[7.2.3 Plot the spectral domains. 52](#_Toc135651336)

[7.2.4 Plot the time domain of the modulated BPSK signal (𝑓𝑐 = 1𝐺𝐻𝑧) 53](#_Toc135651337)

[7.2.5 Plot the spectrum of the modulated BPSK signal. 55](#_Toc135651338)

[7.3 PART II RECEIVER 56](#_Toc135651339)

[7.3.1 Design a receiver which consists of modulator, integrator (simply LPF) and decision device. 56](#_Toc135651340)

[7.3.2 Compare the output of decision level with the generated stream of bits in the transmitter. The comparison is performed by comparing the value of each received bit with the corresponding transmitted bit (step 1) and count number of errors. Then calculate bit error rate (BER) = number of error bits/ Total number of bits. 57](#_Toc135651341)

[7.4 PART II FULL CODE 57](#_Toc135651342)

[7.5 WORKSPACE 60](#_Toc135651343)

# TEAM MEMBERS

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Code | Name | Department |
| 1 | 2001436 | Mahmoud Abdelraouf Mahmoud Abdelall | CSE |
| 2 | 2000217 | Ahmed Samir Tharwat Mohamed | ECE |
| 3 | 2000218 | Ahmed Khaled Abdelmaksod ibrahim | CSE |
| 4 | 2001771 | Muhammed Ahmed Abdel-Gawad Nassif | ECE |
| 5 | 2001457 | Youssef Foda Mohamed | EPM |
| 6 | 2001023 | Abdullah Yasser Ahmed | EPM |
| 7 | 2000037 | Ahmed Mohammed Bakr Ahmed | CSE |
| 8 | 2001853 | Nada Ashraf Mohamed Abdullah | EPM |
| 9 | 2001278 | Mahmoud mohamed alsayd soliman | ECE |
| 10 | 2001760 | Abdelrahman Ahmed Abdelrahman Mahrous | ECE |

# BRIEF ABOUT PROJECT PART ONE

This part outlines a simulation experiment for evaluating the performance of different line coding schemes in a communication system. The experiment consists of two parts: transmitter and receiver. The transmitter generates a stream of random bits and line codes the bits using Uni-polar non return to zero scheme. The eye diagram and spectral domains of the pulses are plotted. The receiver consists of a decision device that compares the received waveform with the transmitted stream of bits and calculates the bit error rate (BER). The experiment is repeated for different line coding schemes, including Polar non return to zero, Uni-polar return to zero, Bipolar return to zero, and Manchester coding.

Additionally, noise is added to the received signal, and the experiment is repeated for different levels of noise (sigma). The BER is calculated for each value of sigma, and the results are plotted in a graph with the y-axis in log scale. Finally, for case of Bipolar return to zero, an error detection circuit is designed, and the number of detected errors is counted for different values of sigma.

# IMPORTANT NOTES

***Hint***

* Throughout the project, we divided project into a sub functions, and all functions were be built from scratch without using any built-in functions.
* The report contains all the functions with its implementation as a text.

**Note:**

In our report you can find the code of the bonus mark of Part I Receiver implemented.

# PART I TRANSMITTER

## LINE CODING SYSTEMS

1. unipolar\_nrz(bits, high\_voltage\_level, samples\_per\_bit): The `unipolar\_nrz` function generates a Unipolar Non-Return-to-Zero (NRZ) digital signal based on a sequence of binary bits. It takes three input arguments: `bits`, `high\_voltage\_level`, and `samples\_per\_bit`.

The function first checks if the `samples\_per\_bit` input argument is provided, and if not, it sets its default value to 100. It then initializes the output signal, sets the voltage levels for the signal, and generates the signal by iterating through the `bits` input vector.

After generating the signal, the function creates a new figure and plots the generated signal with the appropriate axis labels and limits. Finally, it adds grid lines and labels to the plot and gives it a title.

**function** **signal** = **unipolar\_nrz**(bits, high\_voltage\_level, samples\_per\_bit)  
 % Check the input arguments  
 **if** nargin < 3  
 samples\_per\_bit = 100;  
 **end**  
  
 % Initialize the output signal  
 signal = zeros(1, length(bits)\*samples\_per\_bit);  
  
 % Set the voltage level  
 v\_low = 0;  
 v\_high = high\_voltage\_level;  
  
 % Generate the signal  
 **for** i = 1:length(bits)  
 **if** bits(i) == 1  
 signal((i-1)\*samples\_per\_bit+1:i\*samples\_per\_bit) = v\_high;  
 **else**  
 signal((i-1)\*samples\_per\_bit+1:i\*samples\_per\_bit) = v\_low;  
 **end**  
 **end**  
  
 % Create a new figure  
 figure();  
  
 % Create the time axis  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
  
 % Plot the signal  
 plot(t, signal);  
 axis([0 t(**end**) -0.1\*high\_voltage\_level 1.1\*high\_voltage\_level]);  
  
 % Add grid and labels  
 grid on;  
 xlabel('Time (s)');  
 ylabel('Voltage (V)');  
 title('Unipolar NRZ Signal');  
**end**

1. polar\_nrz(bits, high\_voltage\_level, samples\_per\_bit): The `polar\_nrz` function generates a Polar Non-Return-to-Zero (NRZ) digital signal based on a sequence of binary bits. It takes three input arguments: `bits`, `high\_voltage\_level`, and `samples\_per\_bit`.

The function first checks if the `samples\_per\_bit` input argument is provided, and if not, it sets its default value to 100. It then initializes the output signal, sets the voltage levels for the signal, and generates the signal by iterating through the `bits` input vector.

After generating the signal, the function creates a new figure and plots the generated signal with the appropriate axis labels and limits. Finally, it adds grid lines and labels to the plot and gives it a title.

**function** **signal** = **polar\_nrz**(bits, high\_voltage\_level, samples\_per\_bit)  
 % Check the input arguments  
 **if** nargin < 3  
 samples\_per\_bit = 100;  
 **end**  
  
 % Initialize the output signal  
 signal = zeros(1, length(bits)\*samples\_per\_bit);  
  
 % Set the voltage level  
 v\_low = -high\_voltage\_level;  
 v\_high = high\_voltage\_level;  
  
 % Generate the signal  
 **for** i = 1:length(bits)  
 **if** bits(i) == 1  
 signal((i-1)\*samples\_per\_bit+1:i\*samples\_per\_bit) = v\_high;  
 **else**  
 signal((i-1)\*samples\_per\_bit+1:i\*samples\_per\_bit) = v\_low;  
 **end**  
 **end**  
  
 % Create a new figure  
 figure();  
  
 % Create the time axis  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
  
 % Plot the signal  
 plot(t, signal);  
 axis([0 t(**end**) 1.2\*v\_low 1.2\*v\_high]);  
  
 % Add grid and labels  
 grid on;  
 xlabel('Time (s)');  
 ylabel('Voltage (V)');  
 title('Polar NRZ Signal');  
**end**

1. unipolar\_rz(bits, high\_voltage\_level, samples\_per\_bit): The `unipolar\_rz` function generates a Unipolar Return-to-Zero (RZ) digital signal based on a sequence of binary bits. It takes three input arguments: `bits`, `high\_voltage\_level`, and `samples\_per\_bit`.

The function first checks if the `samples\_per\_bit` input argument is provided, and if not, it sets its default value to 100. It then initializes the output signal, computes the pulse width for the RZ pulse, and generates the RZ pulse waveform by iterating through the `bits` input vector.

After generating the signal, the function creates a new figure and plots the generated signal with the appropriate axis labels and limits. Finally, it adds grid lines and labels to the plot and gives it a title.

**function** **y** = **unipolar\_rz**(bits, high\_voltage\_level, samples\_per\_bit)  
% Bipolar RZ encoding of a binary sequence  
% bits: input binary sequence (row vector)  
% high\_voltage\_level: amplitude of the high voltage level for a logic high bit  
% samples\_per\_bit: number of samples per bit  
  
% Check the input arguments  
**if** nargin < 3  
 samples\_per\_bit = 100;  
**end**  
  
% Compute the number of samples in the waveform  
num\_samples = length(bits) \* samples\_per\_bit;  
  
% Create a waveform vector of zeros  
waveform = zeros(1, num\_samples);  
  
% Compute the pulse width for the RZ pulse  
pulse\_width = samples\_per\_bit / 2;  
  
  
% Generate the RZ pulse waveform  
**for** i = 1:length(bits)  
 **if** bits(i) == 1  
 % Set the amplitude to high voltage level for a logic high bit   
 waveform((i-1)\*samples\_per\_bit + 1:(i-1)\*samples\_per\_bit + pulse\_width) = high\_voltage\_level;   
 waveform((i-1)\*samples\_per\_bit + pulse\_width + 1:i\*samples\_per\_bit) = 0;  
 **else**  
 % Set the amplitude to zero for a logic low bit  
 waveform((i-1)\*samples\_per\_bit + 1:i\*samples\_per\_bit) = 0;  
 **end**  
**end**  
  
% Create a new figure  
figure();  
  
% Create the time axis  
t = linspace(0, length(waveform)/samples\_per\_bit, length(waveform));  
  
% Plot the signal  
plot(t, waveform);  
axis([0 t(**end**) -.1\*high\_voltage\_level 1.1\*high\_voltage\_level]);  
  
% Add grid and labels  
grid on;  
xlabel('Time (s)');  
ylabel('Voltage (V)');  
title('Unipolar RZ Signal');  
  
% Return the generated waveform  
y = waveform;  
**end**

1. bipolar\_rz(bits, high\_voltage\_level, samples\_per\_bit): The ‘bipolar\_rz’ function generates a Bipolar Return-to-Zero (RZ) digital signal based on a sequence of binary bits. The function takes three input arguments: `bits`, `high\_voltage\_level`, and `samples\_per\_bit`.

The function first checks if the `samples\_per\_bit` input argument is provided, and if not, it sets its default value to 100. It then initializes the output signal, computes the pulse width for the RZ pulse, and generates the RZ pulse waveform by iterating through the `bits` input vector.

After generating the signal, the function creates a new figure and plots the generated signal with the appropriate axis labels and limits. Finally, it adds grid lines and labels to the plot and gives it a title.

In summary, the `bipolar\_rz` function generates a Bipolar RZ digital signal based on a sequence of binary bits using either a positive or negative voltage level for logic high bits depending on the bit's position in the sequence and the previous bit value, and 0 voltage level for logic low bits. The generated signal is plotted in a new figure with the appropriate axis labels and limits.

**function** **y** = **bipolar\_rz**(bits, high\_voltage\_level, samples\_per\_bit)  
% Bipolar RZ encoding of a binary sequence  
% bits: input binary sequence (row vector)  
% high\_voltage\_level: amplitude of the high voltage level for a logic high bit  
% samples\_per\_bit: number of samples per bit  
  
% Check the input arguments  
**if** nargin < 3  
 samples\_per\_bit = 100;  
**end**  
  
% Compute the number of samples in the waveform  
num\_samples = length(bits) \* samples\_per\_bit;  
  
% Create a waveform vector of zeros  
waveform = zeros(1, num\_samples);  
  
% Compute the pulse width for the RZ pulse  
pulse\_width = samples\_per\_bit / 2;  
  
pos\_flag = 1;  
neg\_flag = 0;  
% Generate the RZ pulse waveform  
**for** i = 1:length(bits)  
 **if** bits(i) == 1  
 **if** neg\_flag == 0 && pos\_flag == 0  
 pos\_flag = 1;  
 **end**   
 **if** i > 1 && neg\_flag == 1   
 waveform((i-1)\*samples\_per\_bit + 1:(i-1)\*samples\_per\_bit + pulse\_width) = - high\_voltage\_level;  
 waveform((i-1)\*samples\_per\_bit + pulse\_width + 1:i\*samples\_per\_bit) = 0;  
 neg\_flag = 0;  
 **end**  
 **if** pos\_flag == 1  
 % Set the amplitude to high voltage level for a logic high bit  
 waveform((i-1)\*samples\_per\_bit + 1:(i-1)\*samples\_per\_bit + pulse\_width) = high\_voltage\_level;  
 waveform((i-1)\*samples\_per\_bit + pulse\_width + 1:i\*samples\_per\_bit) = 0;  
 pos\_flag = 0;  
 neg\_flag = 1;  
 **end**  
 **else**  
 % Set the amplitude to zero for a logic low bit  
 waveform((i-1)\*samples\_per\_bit + 1:i\*samples\_per\_bit) = 0;  
 **end**  
**end**  
  
% Create a new figure  
figure();  
  
% Create the time axis  
t = linspace(0, length(waveform)/samples\_per\_bit, length(waveform));  
  
% Plot the signal  
plot(t, waveform);  
axis([0 t(**end**) -1.2\*high\_voltage\_level 1.2\*high\_voltage\_level]);  
  
% Add grid and labels  
grid on;  
xlabel('Time (s)');  
ylabel('Voltage (V)');  
title('Bipolar RZ Signal');  
  
% Return the generated waveform  
y = waveform;  
**end**

1. manchester\_coding(bits, high\_voltage, sampling\_per\_bit): The `manchester\_coding` function performs Manchester encoding on a sequence of binary bits, which is a form of differential encoding used in digital communication systems. It takes three input arguments: `bits`, `high\_voltage`, and `sampling\_per\_bit`.

The `bits` argument is a vector of binary bits to be encoded, the `high\_voltage` argument is the voltage level for a logic high bit, and the `sampling\_per\_bit` argument is the number of samples per bit.

The function generates the Manchester pulse waveform by encoding each bit using a positive or a negative pulse, and generates an output signal vector containing the Manchester encoded signal.

Finally, the function plots the encoded signal in a new figure with grid and axis labels.

**function** **output\_signal** = **manchester\_coding**(bits, high\_voltage, sampling\_per\_bit)  
 % SPLIT\_PHASE\_ENCODING Encode a sequence of binary bits using Split Phase (Manchester) encoding  
 %  
 % INPUTS:  
 % bits: a vector of binary bits to be encoded (1s and 0s)  
 % high\_voltage: the voltage level for a logic high bit  
 % low\_voltage: the voltage level for a logic low bit  
 % sampling\_per\_bit: the number of samples per bit  
 %  
 % OUTPUTS:  
 % output\_signal: a vector containing the Split Phase (Manchester) encoded signal  
  
 % Check the input arguments  
 **if** nargin < 3  
 sampling\_per\_bit = 100;  
 **end**  
  
 % Compute the number of samples in the waveform  
 num\_samples = length(bits) \* sampling\_per\_bit;  
  
 % Initialize the output signal  
 output\_signal = zeros(1, num\_samples);  
  
 % Compute the pulse width for the Manchester pulse  
 pulse\_width = sampling\_per\_bit / 2;  
  
 % Generate the Manchester pulse waveform  
 **for** i = 1:length(bits)  
 **if** bits(i) == 1  
 % Encode a "1" bit as a positive pulse followed by a negative pulse  
 output\_signal((i-1)\*sampling\_per\_bit + 1:(i-1)\*sampling\_per\_bit+ pulse\_width) = high\_voltage;  
 output\_signal((i-1)\*sampling\_per\_bit + pulse\_width + 1:i\*sampling\_per\_bit) = -high\_voltage;  
 **else**  
 % Encode a "0" bit as a negative pulse followed by a positive pulse  
 output\_signal((i-1)\*sampling\_per\_bit + 1:(i-1)\*sampling\_per\_bit + pulse\_width) = -high\_voltage;  
 output\_signal((i-1)\*sampling\_per\_bit + pulse\_width + 1:i\*sampling\_per\_bit) = high\_voltage;  
 **end**  
 **end**  
  
 % Create a new figure  
 figure();  
   
 % Create the time axis  
 t = linspace(0, length(output\_signal)/sampling\_per\_bit, length(output\_signal));  
   
 % Plot the signal  
 plot(t, output\_signal);  
 axis([0 t(**end**) -1.2\*high\_voltage 1.2\*high\_voltage]);  
   
 % Add grid and labels  
 grid on;  
 xlabel('Time (s)');  
 ylabel('Voltage (V)');  
 title('Manchester Coding');  
**end**

## SPECTRAL DOMAIN FUNCTION

plot\_spectral\_domain(waveform): This function, ‘plot\_spectral\_domain’, takes a time-domain signal as input and generates a plot of its power spectral density (PSD) on a linear scale. The function applies a Hamming window to the input signal to reduce spectral leakage and improve the accuracy of the PSD estimate. It then computes the Fourier transform of the windowed signal, the absolute value of the Fourier transforms squared, and divides by the number of samples in the waveform to obtain the PSD. Negative values of the PSD are set to zero, and the square root of the PSD is computed and divided by 10 to obtain the root-mean-square (RMS) PSD.

The function checks for impulses in the RMS PSD and excludes them from the maximum value calculation used to set the y-axis limit of the plot. The frequency axis is defined as a vector of normalized frequencies ranging from -1/2 to 1/2, and the RMS PSD is plotted against the normalized frequency. The x-axis limit is set based on the maximum frequency of the signal and the number of samples in the waveform, and the y-axis limit is set based on the maximum value of the RMS PSD with some padding added.

The function provides a simple way to visualize the PSD of a given signal and identify any impulses or other irregularities in the PSD that may indicate noise or other issues with the signal.

**function** **plot\_spectral\_domain**(waveform)  
 % Apply a Hamming window to the input waveform  
 N = length(waveform);  
 window = hamming(N)';  
 waveform = waveform .\* window;  
  
 % Compute the Fourier transform of the input waveform  
 spectrum = fftshift(fft(waveform));  
  
 % Compute the power spectral density (PSD)  
 psd = abs(spectrum).^2 / (N);  
  
 % Set negative values of the PSD to zero and take the square root  
 psd(psd < 0) = 0;  
 rms\_psd = sqrt(psd)/10;  
  
 % Check for impulses in the PSD  
 threshold = 60 \* mean(rms\_psd); % set threshold as 10 times the mean RMS PSD  
 **if** any(rms\_psd > threshold)  
 % If there are impulses, exclude them from the max value calculation  
 max\_psd = max(rms\_psd(rms\_psd <= threshold)) \* 1.5;  
 disp('Impulse detected in the PSD');  
 **else**  
 % If there are no impulses, use the max value of the RMS PSD  
 max\_psd = max(rms\_psd) \* 1.5;  
 **end**  
  
 % Define the frequency axis for the plot in normalized frequency  
 f\_norm = linspace(-1/2, 1/2, N);  
  
  
 % Create a new figure  
 figure();  
  
 % Plot the RMS PSD on a linear scale  
 plot(f\_norm, rms\_psd);  
  
  
 % Set the axis labelsand title  
 xlabel('Normalized Frequency');  
 ylabel('Power/Frequency (V/Hz)');  
 title('Power Spectral Density');  
  
 % Set the x-axis limit based on the input waveform  
 f\_max = 1/2;  
 f\_step = 1/N;  
 x\_lim = [-(f\_max-f\_step)/6, (f\_max-f\_step)/6];  
 xlim(x\_lim);  
  
 % Set the y-axis limit based on the input waveform  
 y\_lim = [0, max\_psd\*1.1];  
 ylim(y\_lim);  
**end**

## THE MAIN.M FILE

% Load the Communications Toolbox package  
pkg load communications  
  
% Generate a sequence of 10000 random bits  
bits = generate\_bits(10000);  
  
% Generate a Unipolar NRZ signal from the bit sequence with a high voltage level of 1.2V  
signal\_1 = unipolar\_nrz(bits,1.2);  
  
%check polar\_nrz  
signal\_2 = polar\_nrz(bits,1.2);  
  
%check unipolar\_rz  
signal\_3 = unipolar\_rz(bits,1.2);  
  
%check bipolar\_rz  
signal\_4 = bipolar\_rz(bits,1.2);  
  
%check manchester\_coding  
signal\_5 = manchester\_coding(bits,1.2);  
  
plot\_spectral\_domain(signal\_2);  
% Plot the eye diagram and set the plot limits  
eyediagram(signal\_2, 300,1,1);  
xlim([-0.165, 0.5]);

## HERE IS A SUMMARY OF THE CODE

* The code loads the Communications Toolbox package in Octave, a numerical computing software. It then generates a sequence of 10000 random bits using the `generate\_bits` function, which is not shown in the code snippet.
* Next, the code generates several different types of baseband digital signals from the bit sequence using different encoding techniques:

1. unipolar\_nrz(bits,1.2): generates a Unipolar NRZ (Non-Return-to-Zero) signal from the bit sequence, using a high voltage level of 1.2V. Unipolar NRZ signal encodes a 1 bit as a high voltage level and a 0 bit as a low voltage level.
2. polar\_nrz(bits,1.2): generates a Polar NRZ signal from the bit sequence, which is similar to Unipolar NRZ except that it encodes a 0 bit as a negative voltage level.
3. unipolar\_rz(bits,1.2): generates a Unipolar RZ (Return-to-Zero) signal from the bit sequence, which encodes a 1 bit as a high voltage level followed by a zero-voltage level, and a 0 bit as a zero-voltage level.
4. bipolar\_rz(bits,1.2): generates a Bipolar RZ signal from the bit sequence, which encodes a 1 bit as a positive or negative voltage leveldepending on the previous bit, and a 0 bit as a zero-voltage level.
5. manchester\_coding(bits,1.2): generates a Manchester encoded signal from the bit sequence, which is a form of differential encoding that represents each bit using a transition between two voltage levels.

* The output signals from all the encoding techniques are assigned to different variables, `signal\_1` to `signal\_5`, respectively.
* The code also includes some commented lines that demonstrate different signal analysis techniques using the Communications Toolbox package:

1. plot\_spectral\_domain(signal\_2): is a commented line that would plot the spectral domain of the Polar NRZ signal, which would show the frequency content of the signal.
2. eyediagram(signal\_2, 300,1,1): is a commented line that would plot the eye diagram of the Polar NRZ signal, which would show the signal quality and the timing jitter. The `xlim` command sets the limits of the plot to focus on a specific part of the signal.

* Overall, the code demonstrates how to generate and analyze different types of digital baseband signals from a bit sequence, using different encoding techniques.

## SNAPSHOTS

### Unipolar Non-Return to Zero

A screenshot of a computer screen

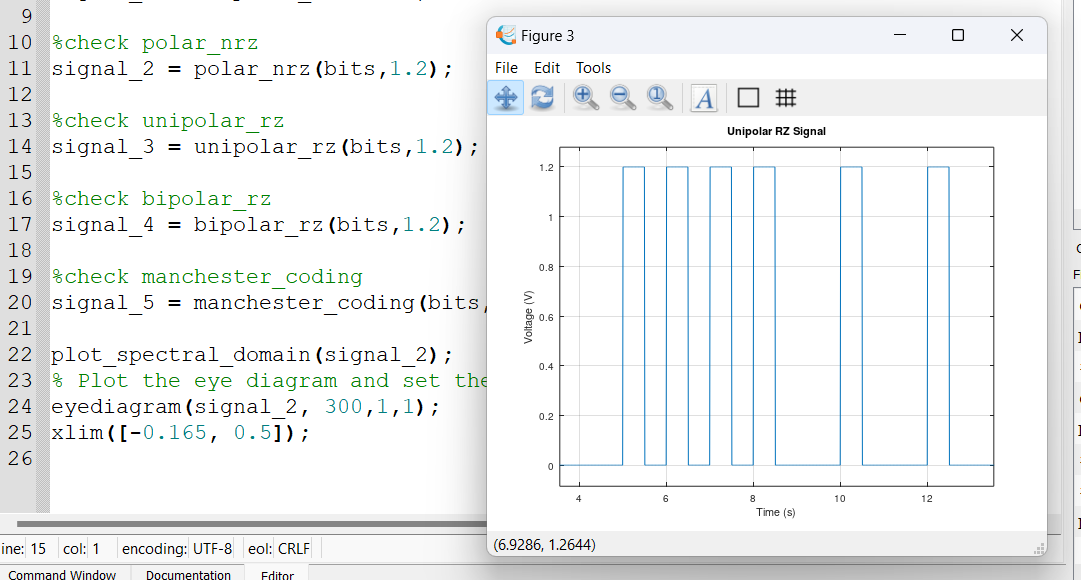
Description automatically generated with medium confidence

### Polar Non-Return to Zero

A screenshot of a computer

Description automatically generated

### Unipolar Return to Zero

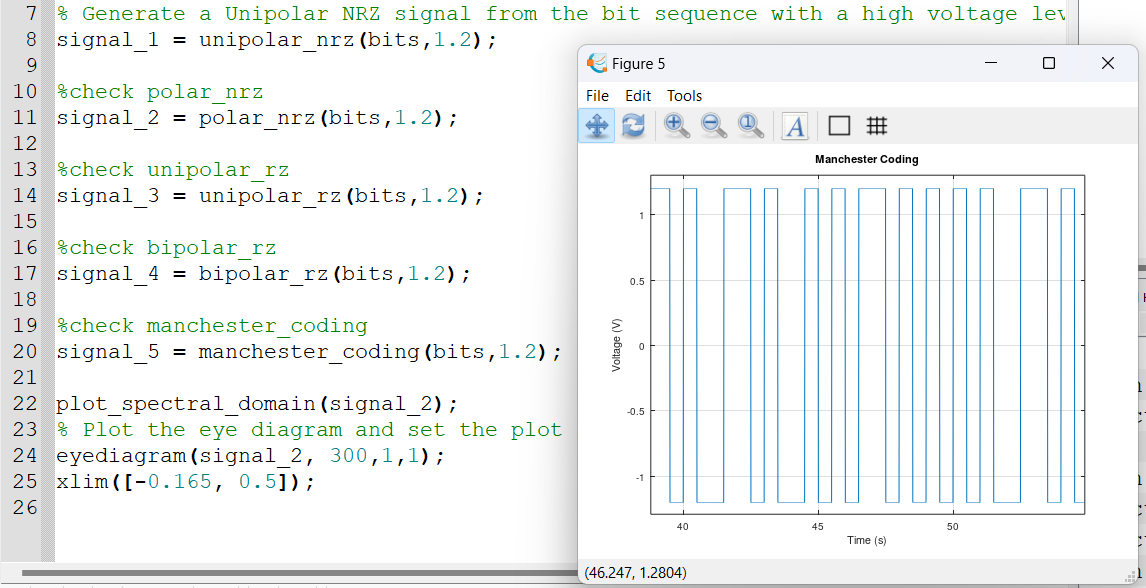


### Bipolar Return To Zero

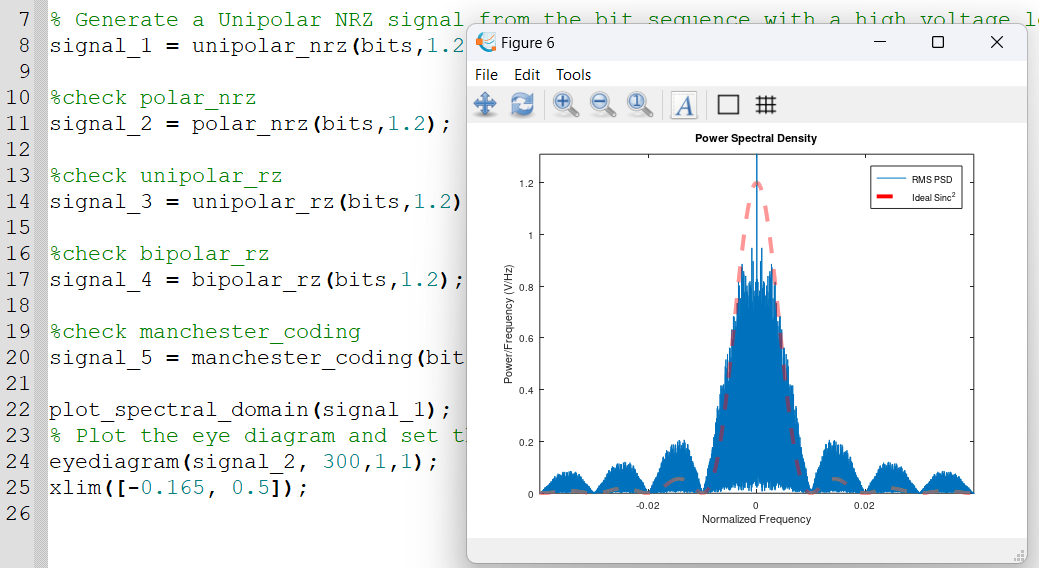
A screenshot of a computer

Description automatically generated

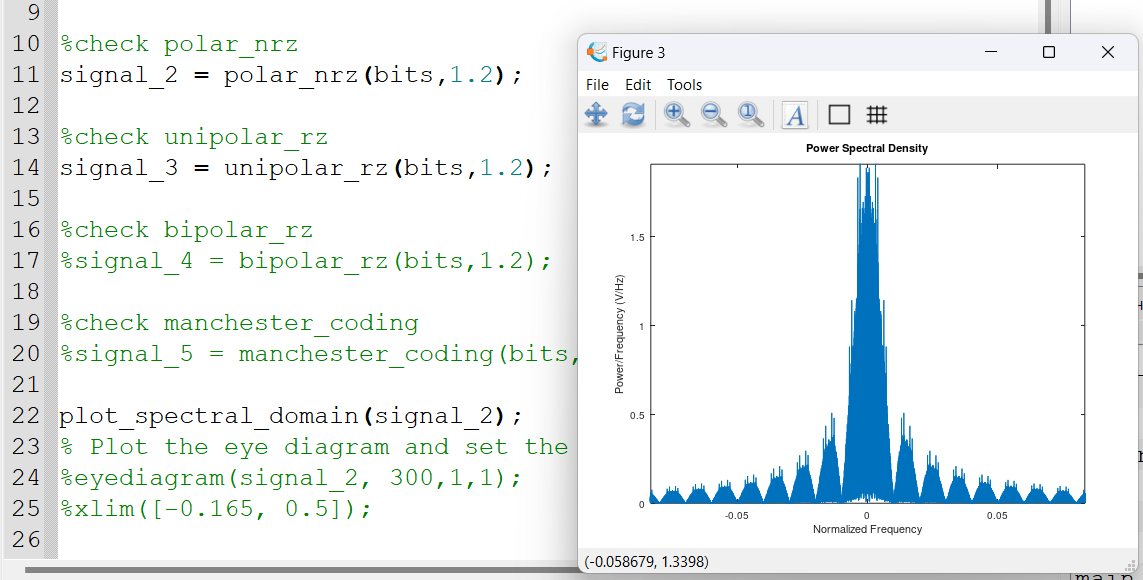
### Manchester Coding



### Spectral Domain of Unipolar Non-Return to Zero



### Spectral Domain of Polar Non-Return to Zero

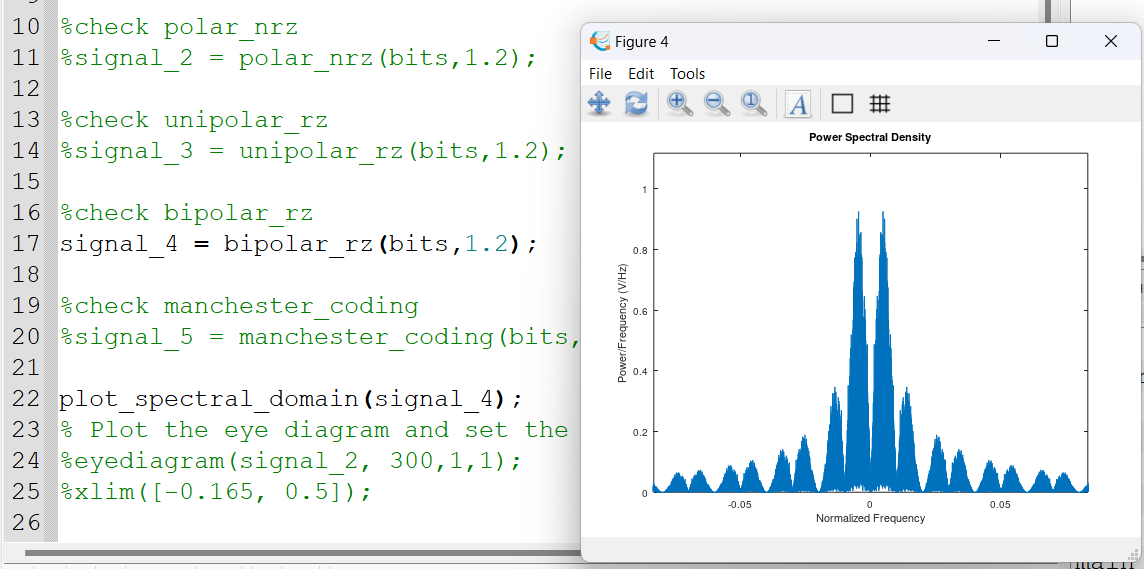


### Spectral Domain of Unipolar Return to Zero

A screenshot of a computer

Description automatically generated with medium confidence

### Spectral Domain of Bipolar Return to Zero

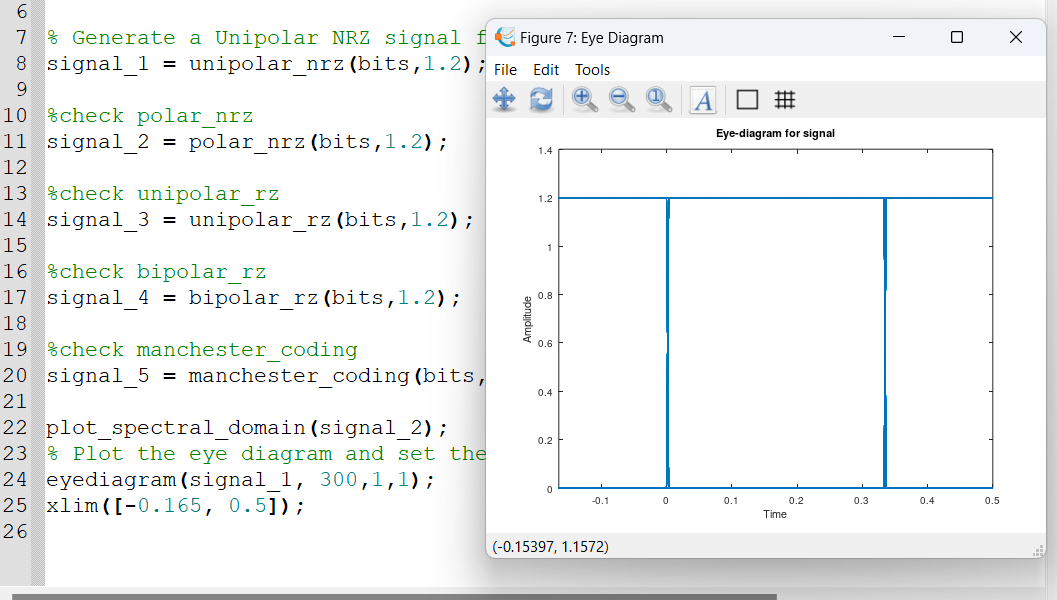


### Spectral Domain of Manchester Coding

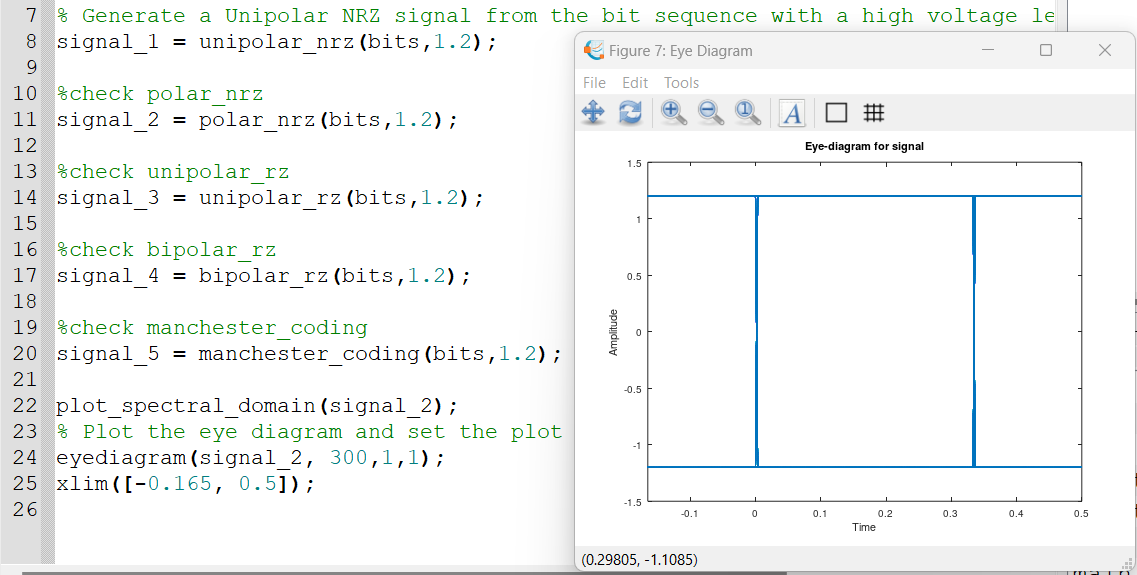
A screenshot of a computer

Description automatically generated with medium confidence

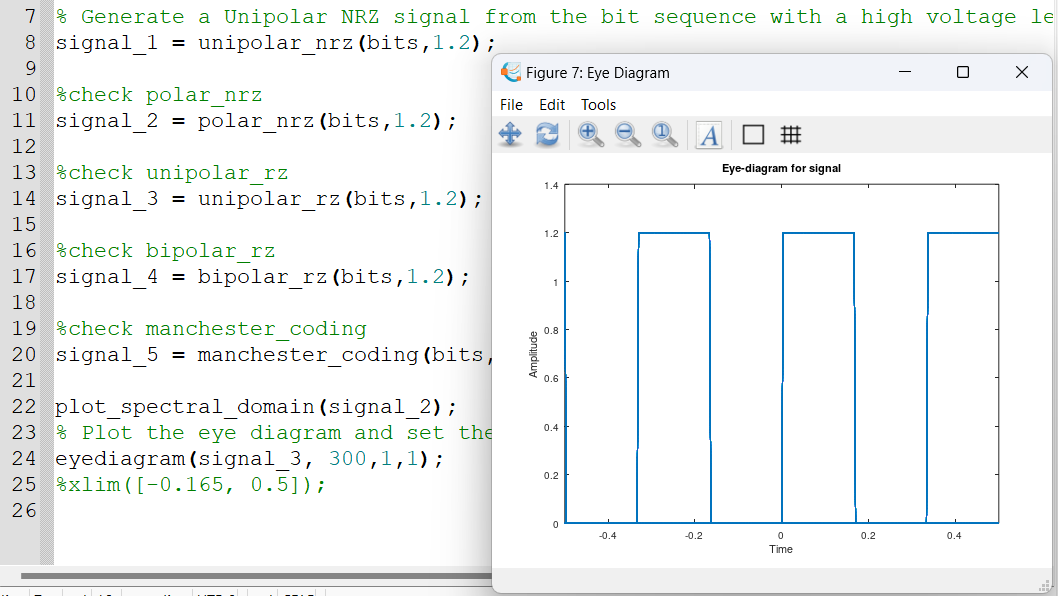
### Eye Diagram of Unipolar Non-Return to Zero



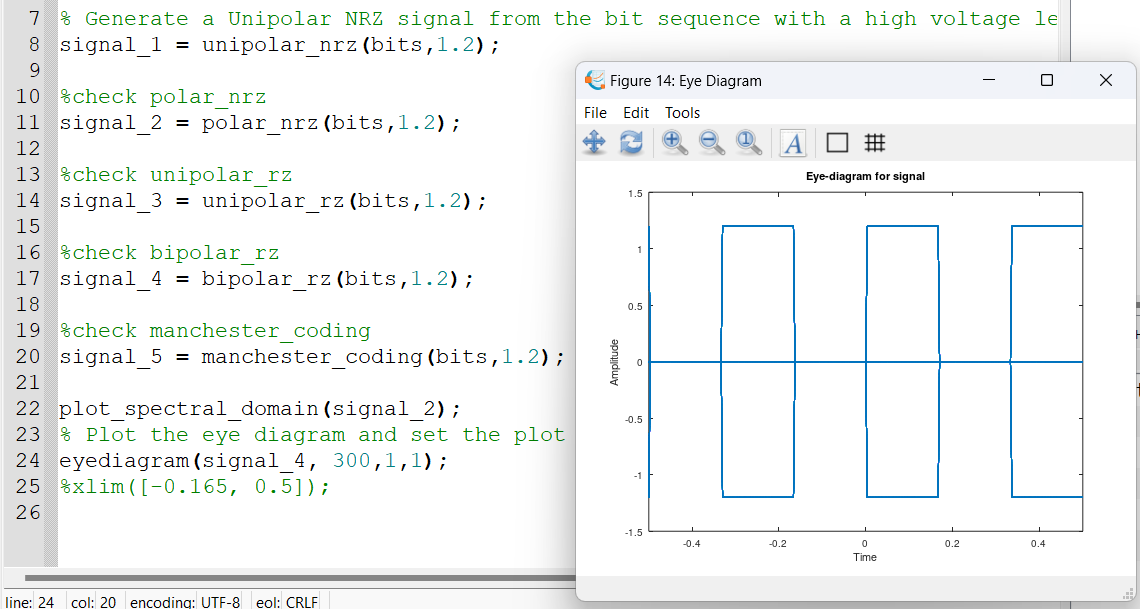
### Eye Diagram of Polar Non-Return to Zero



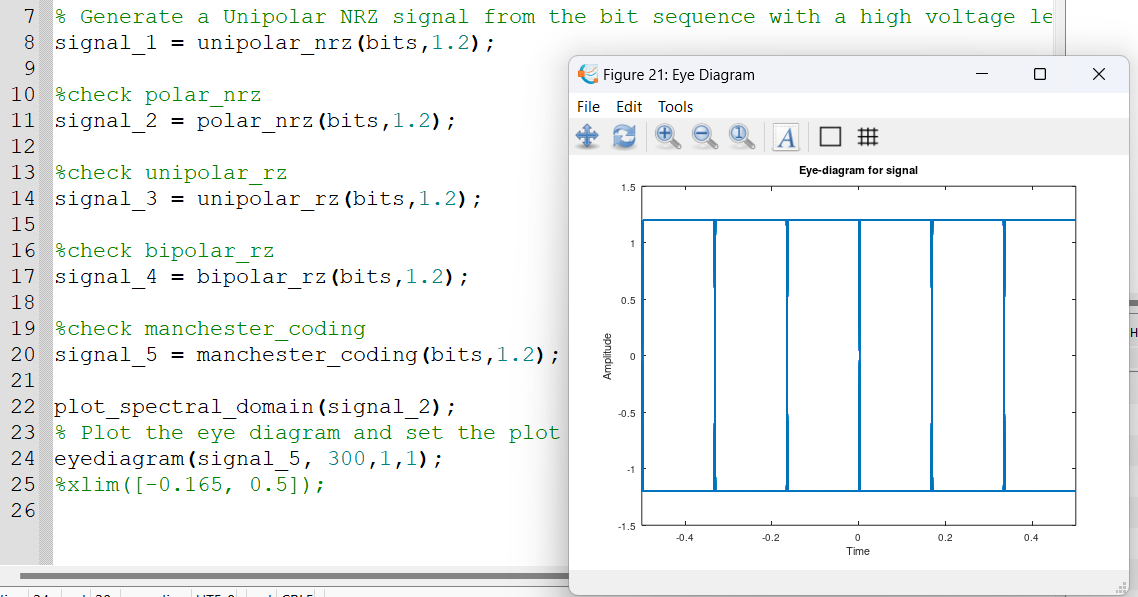
### Eye Diagram of Unipolar Return to Zero



### Eye Diagram of Bipolar Return to Zero



### Eye Diagram of Manchester Coding



# PART I RECEIVER

## USED FUNCTIONS

1. generate\_bits(num\_of\_bits) “Repeated”
2. unipolar\_nrz(bits, high\_voltage\_level, samples\_per\_bit) “Repeated”
3. plot\_spectral\_domain(waveform) “Repeated”
4. eyediagram “Repeated”
5. unipolar\_rz\_reciever(signal,N,decision)

% function to extract stream of bits in reciever  
**function** **recieved\_bits** = **unipolar\_rz\_reciever**(signal,N,decision)  
 index = 40;  
**if** nargin <3  
 decision = 0.6;  
**end**  
recieved\_bits = zeros(1,N);  
**for** i=1:N  
 **if**(signal(index) >= decision)  
 recieved\_bits(i) = 1;  
 index = index + 100;  
 **else**  
 recieved\_bits(i) = 0;  
 index = index + 100 ;  
 **end**  
**end**  
**end**

1. calculate\_ber(tx\_bits,rx\_bits)

% it is a function that calculate BER  
% it takes two parmeters tx\_bits  
% tx\_bits -> stream of bits of the transmitter  
% rx\_bits -> stream of bits of the Reciever  
**function** **BER** = **calculate\_ber**(tx\_bits,rx\_bits)  
 % calculate number of error bits  
 number\_of\_error\_bits = sum(tx\_bits ~= rx\_bits);  
 % calculate bit error rate  
 BER = number\_of\_error\_bits / length(tx\_bits);  
**end**  
% important !!!!  
% we have to calculate number of error in the main  
% use this formula  
% number\_of\_error\_bits = BER \* length(tx\_bits);

1. add\_noise(signal,sigma,samples\_per\_bit)

% function to add noise to the signal with a specified sigma  
**function** **noisy\_signal** = **add\_noise**(signal,sigma,samples\_per\_bit)  
 **if** nargin < 2  
 sigma = 0.2;  
 endif  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
 n = sigma \* randn(1,length(t));  
 noisy\_signal = signal + n;  
**end**

1. plot\_noisy\_signal(signal,samples\_per\_bit)

% function to plot nooisy signal with time  
**function** **plot\_noisy\_signal**(signal,samples\_per\_bit)  
 % Create a time vector for the signal  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
 figure  
 % Plot the Unipolar NRZ signal with time  
 plot(t, signal);  
 ylim([-3, 3]);  
 xlabel('Time (s)');  
 ylabel('Amplitude (V)');  
 title('Unipolar NRZ Signal');  
**end**

1. sweep\_sigma(signal,decision,bits,samples\_per\_bit)

% function to sweep values of sigma and calculate ber for each value of sigma  
**function** **ber\_values** = **sweep\_sigma**(signal,decision,bits,samples\_per\_bit)  
 N=10000;  
 sigma = linspace(0,1.2,10);  
  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
 ber\_values = zeros(1, 10);  
**for** i = 1:10  
 % Add noise to the received signal  
  
 n = sigma(i) .\* randn(1,length(t));  
 noisy\_signal = signal + n;  
  
 % Decode  
 index = 1;  
 received\_bits\_noise = zeros(1,N);  
 **for** j = 1:N  
 **if** noisy\_signal(index) >= decision  
 received\_bits\_noise(j) = 1;  
 index = index + 100;  
 **else**  
 received\_bits\_noise(j) = 0;  
 index = index + 100;  
 **end**  
 **end**  
 % Calculate BER  
 ber\_values(i) = calculate\_ber(bits, received\_bits\_noise);  
**end**  
**end**

1. polar\_nrz(bits, high\_voltage\_level, low\_voltage\_level, samples\_per\_bit) “Repeated”
2. polar\_nrz\_reciever(signal\_2,N,decision)

% function to extract stream of bits in reciever polar nrz  
**function** **recieved\_bits** = **polar\_nrz\_reciever**(signal\_2,N,decision)  
 index=50;  
**if** nargin <3  
 decision = 0;  
**end**  
 recieved\_bits = zeros(1,N);  
 **for** i=1:N  
 **if**(signal\_2(index)>=decision)  
 recieved\_bits(i)=1;  
 index = index +100;  
 **else**  
 recieved\_bits(i)=0;  
 index = index +100;  
 **end**  
 **end**  
**end**

1. plot\_noisy\_signal\_polar\_rz(signal,samples\_per\_bit)

**function** **plot\_noisy\_signal\_bipolar\_rz**(signal,samples\_per\_bit)  
 % Create a time vector for the signal  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
 figure  
 % Plot the Unipolar NRZ signal with time  
 plot(t, signal);  
 ylim([-3, 3]);  
 xlabel('Time (s)');  
 ylabel('Amplitude (V)');  
 title('Bipolar RZ Signal');  
**end**

1. unipolar\_rz(bits, high\_voltage\_level, samples\_per\_bit) “Repeated”
2. sweep\_sigma\_uni\_rz(signal,decision,bits,samples\_per\_bit)

**function** **ber\_values** = **sweep\_sigma\_uni\_rz**(signal,decision,bits,samples\_per\_bit)  
 N=10000;  
 sigma = linspace(0,1.2,10);  
  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
 ber\_values = zeros(1, 10);  
**for** i = 1:10  
 % Add noise to the received signal  
  
 n = sigma(i) .\* randn(1,length(t));  
 noisy\_signal = signal + n;  
  
 % Decode  
 index = 1;  
 received\_bits\_noise = zeros(1,N);  
 received\_bits\_noise = unipolar\_rz\_reciever(noisy\_signal,10000,0.6);  
 % Calculate BER  
 ber\_values(i) = calculate\_ber(bits, received\_bits\_noise);  
**end**  
  
**end**

1. bipolar\_rz(bits, high\_voltage\_level, samples\_per\_bit) “Repeated”
2. bipolar\_rz\_reciever(signal,N,decision,samples\_per\_bit)

**function** **recieved\_bits** = **bipolar\_rz\_reciever**(signal,N,decision,samples\_per\_bit)  
index = 1;  
 **if** nargin <2  
 decision = 0;  
**end**  
recieved\_bits = zeros(1,N);  
**for** i=1:N  
 **if**(signal(index) > -0.6 && signal(index) < 0.6)  
 recieved\_bits(i) = 0;  
 index = index+samples\_per\_bit;  
 **else**  
 recieved\_bits(i) = 1;  
 index = index + samples\_per\_bit ;  
 **end**  
**end**  
  
**end**

1. plot\_noisy\_signal\_bipolar\_rz(signal,samples\_per\_bit)

**function** **plot\_noisy\_signal\_bipolar\_rz**(signal,samples\_per\_bit)  
 % Create a time vector for the signal  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
 figure  
 % Plot the Unipolar NRZ signal with time  
 plot(t, signal);  
 ylim([-3, 3]);  
 xlabel('Time (s)');  
 ylabel('Amplitude (V)');  
 title('Bipolar RZ Signal');  
**end**

1. sweep\_sigma\_toggle(signal,bits,samples\_per\_bit)

**function** **ber\_values**=**sweep\_sigma\_toggle**(signal,bits,samples\_per\_bit)  
 N=10000;  
 sigma = linspace(0,1.2,10);  
  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
 ber\_values = zeros(1, 10);  
**for** i = 1:10  
 % Add noise to the received signal  
  
 n = sigma(i) .\* randn(1,length(t));  
 noisy\_signal = signal + n;  
  
 % Decode  
 index = 1;  
 received\_bits\_noise = zeros(1,N);  
 received\_bits\_noise = bipolar\_rz\_reciever(noisy\_signal,10000,0,samples\_per\_bit);  
 % Calculate BER  
 ber\_values(i) = calculate\_ber(bits, received\_bits\_noise);  
**end**  
**end**

1. manchester\_coding(bits, high\_voltage, sampling\_per\_bit) “Repeated”
2. manchester\_coding\_reciever(signal,sampling\_per\_bit)

**function** **recieved\_bits** = **manchester\_coding\_reciever**(signal,sampling\_per\_bit)  
 N =10000;  
  
 index = 1;  
recieved\_bits = zeros(1,N);  
**for** i=1:N  
 **if**(signal(index) >= 0)  
 recieved\_bits(i) = 1;  
 index = index + sampling\_per\_bit;  
 **else**  
 recieved\_bits(i) = 0;  
 index = index + sampling\_per\_bit;  
 **end**  
**end**  
**end**

1. plot\_noisy\_signal\_manchester(signal,samples\_per\_bit)

**function** **plot\_noisy\_signal\_manchester**(signal,samples\_per\_bit)  
 % Create a time vector for the signal  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
 figure  
 % Plot the Unipolar NRZ signal with time  
 plot(t, signal);  
 ylim([-3, 3]);  
 xlabel('Time (s)');  
 ylabel('Amplitude (V)');  
 title('Manchester Signal');  
**end**

1. sweep\_sigma\_manchester(signal,bits,samples\_per\_bit)

**function** **ber\_values** = **sweep\_sigma\_manchester**(signal,bits,samples\_per\_bit)  
 N=10000;  
 sigma = linspace(0,1.2,10);  
  
 t = linspace(0, length(signal)/samples\_per\_bit, length(signal));  
 ber\_values = zeros(1, 10);  
**for** i = 1:10  
 % Add noise to the received signal  
  
 n = sigma(i) .\* randn(1,length(t));  
 noisy\_signal = signal + n;  
  
 % Decode  
 index = 1;  
 received\_bits\_noise = zeros(1,N);  
 received\_bits\_noise = manchester\_coding\_reciever(noisy\_signal,samples\_per\_bit);  
 % Calculate BER  
 ber\_values(i) = calculate\_ber(bits, received\_bits\_noise);  
**end**  
**end**

## UNIPOLAR NRZ

### CODE

%clear memory  
clear  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% include some files  
# <include>sweep\_sigma\_toggle.m</include>  
# <include>sweep\_sigma\_uni\_rz.m</include>  
# <include>manchester\_coding\_reciever.m</include>  
# <include>sweep\_sigma\_manchester.m</include>  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% load package communications  
pkg load communications  
% define samples\_per\_bit  
samples\_per\_bit = 100;  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% Generate a sequence of 10000 random bits  
bits = generate\_bits(10000);  
  
% Generate a Unipolar NRZ signal from the bit sequence with a high voltage level of 1.2V  
signal\_1 = unipolar\_nrz(bits,1.2);  
  
% define vector time  
t = linspace(0, length(signal\_1)/samples\_per\_bit, length(signal\_1));  
  
% plot spectral diagram  
plot\_spectral\_domain(signal\_1);  
% Plot the eye diagram and set the plot limits  
eyediagram(signal\_1, 300,1,1);  
xlim([-0.165, 0.5]);  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% recieve unipolar nrz  
recieved\_bits\_1 = unipolar\_nrz\_reciever(signal\_1,10000,0.6);  
% bit error rate  
ber\_signal\_1 = calculate\_ber(bits,recieved\_bits\_1);  
% calculate number of errors  
number\_of\_errors\_signal\_1 = ber\_signal\_1 \* 10000;  
%add noise to tx signal and let sigma = 0.2 to plot the noisy signal  
noisy\_signal\_1 = add\_noise(signal\_1,0.2,samples\_per\_bit);  
% plot noisy\_signal  
plot\_noisy\_signal(noisy\_signal\_1,samples\_per\_bit);  
% sweep value of sigma  
sigma = linspace(0,1.2,10);  
ber\_values\_signal\_1 = sweep\_sigma(signal\_1,0.6,bits,samples\_per\_bit);

### GRAPHS

A screenshot of a bar code

Description automatically generated with medium confidence

Figure 1: Unipolar NRZ vs Time

A screen shot of a graph

Description automatically generated with medium confidence

Figure 2: Unipolar NRZ power spectral vs Time

A screenshot of a computer

Description automatically generated with medium confidence

Figure 3: Unipolar NRZ Eye diagram

A screenshot of a computer

Description automatically generated with medium confidence

Figure 4: Unipolar NRZ signal “with noise” vs time

### POLAR NRZ

#### CODE

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%recieve polar\_nrz line coding  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% line code the stream of bits to polar\_nrz  
signal\_2 = polar\_nrz(bits,1.2,samples\_per\_bit);  
recieved\_bits\_2 = polar\_nrz\_reciever (signal\_2,10000,0);  
  
ber\_signal\_2 = calculate\_ber(bits,recieved\_bits\_2);  
  
number\_of\_error\_signal\_2 =ber\_signal\_2\*10000;  
  
noisy\_signal\_2=add\_noise(signal\_2,0.2,samples\_per\_bit);  
plot\_noisy\_signal\_polar\_rz(noisy\_signal\_2,samples\_per\_bit);  
  
% sweep sigma 0 -> 1.2 and calculate ber  
ber\_values\_signal\_2=sweep\_sigma(signal\_2,0,bits,100);

#### GRAPHS

A screenshot of a bar code

Description automatically generated with medium confidence

Figure 5: Polar NRZ signal vs Time

A screenshot of a computer

Description automatically generated with medium confidence

Figure 6: Polar NRZ signal “with noise” vs Time

### UNIPOLAR RZ

#### CODE

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%recieve unipolar\_rz line coding  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% line code the stream of bits to unipolar return to zero  
signal\_3 = unipolar\_rz(bits,1.2);  
% recieve unipolar rz  
recieved\_bits\_3 = unipolar\_rz\_reciever(signal\_3,10000,0.6);  
% bit error rate  
ber\_signal\_3 = calculate\_ber(bits,recieved\_bits\_3);  
% calculate number of errors  
number\_of\_errors\_signal\_3 = ber\_signal\_3 \* 10000;  
%add noise to tx signal and let sigma = 0.2 to plot the noisy signal  
noisy\_signal\_3 = add\_noise(signal\_3,0.2,samples\_per\_bit);  
% plot noisy\_signal  
plot\_noisysignal\_unipolar\_rz(noisy\_signal\_3,samples\_per\_bit);  
% sweep sigma values from 0 -> 1.2  
ber\_values\_signal\_3 = sweep\_sigma\_uni\_rz(signal\_3,0.6,bits,samples\_per\_bit);

#### GRAPHS

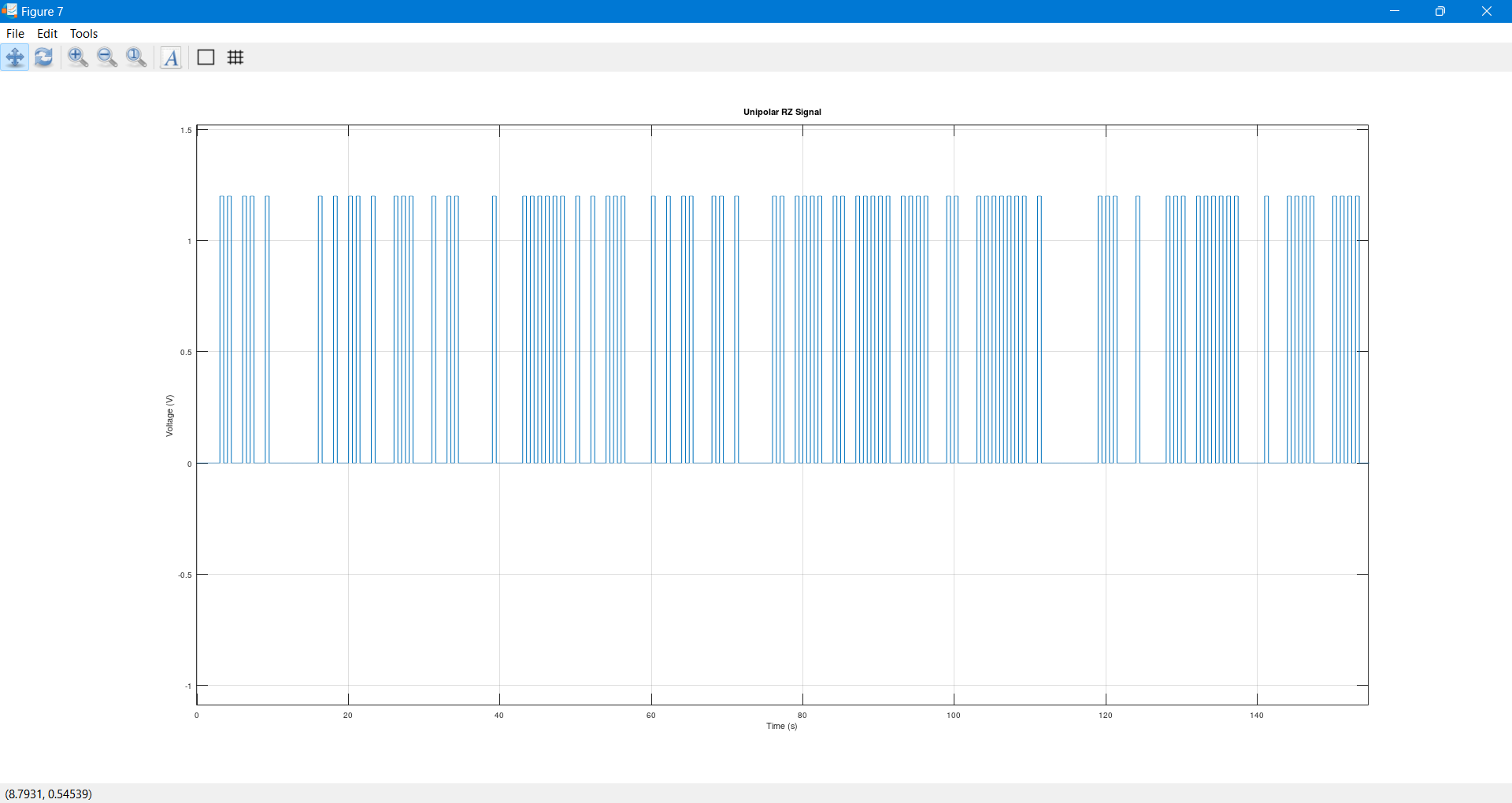


Figure 7: Unipolar RZ signal vs Time

A screenshot of a computer

Description automatically generated with medium confidence

Figure 8: Unipolar RZ signal “with noise” vs Time

### BIPOLAR RZ

#### CODE

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%recieve bipolar\_rz line coding  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% line code the stream of bits to bipolar return to zero  
signal\_4 = bipolar\_rz(bits,1.2);  
%recieve bipolar rz. note : decision levels is determind inside this function  
recieved\_bits\_4 = bipolar\_rz\_reciever(signal\_4,10000,0,100);  
% calculate ber for bipolar rz  
ber\_signal\_4 = calculate\_ber(bits,recieved\_bits\_4);  
% calculate number of errors  
number\_of\_errors\_signal\_4 = ber\_signal\_4 \* 10000;  
%add noise to tx signal and let sigma = 0.2 to plot the noisy signal  
noisy\_signal\_4 = add\_noise(signal\_4,0.2,samples\_per\_bit);  
% plot noisy\_signal  
plot\_noisy\_signal\_bipolar\_rz(noisy\_signal\_4,samples\_per\_bit);  
% sweep sigma 0 -> 1.2 and calculate ber  
ber\_values\_signal\_4 = sweep\_sigma\_toggle(signal\_4,bits,samples\_per\_bit);

#### GRAPHS

A screenshot of a barcode

Description automatically generated with low confidence

Figure 9: Bipolar RZ signal vs Time

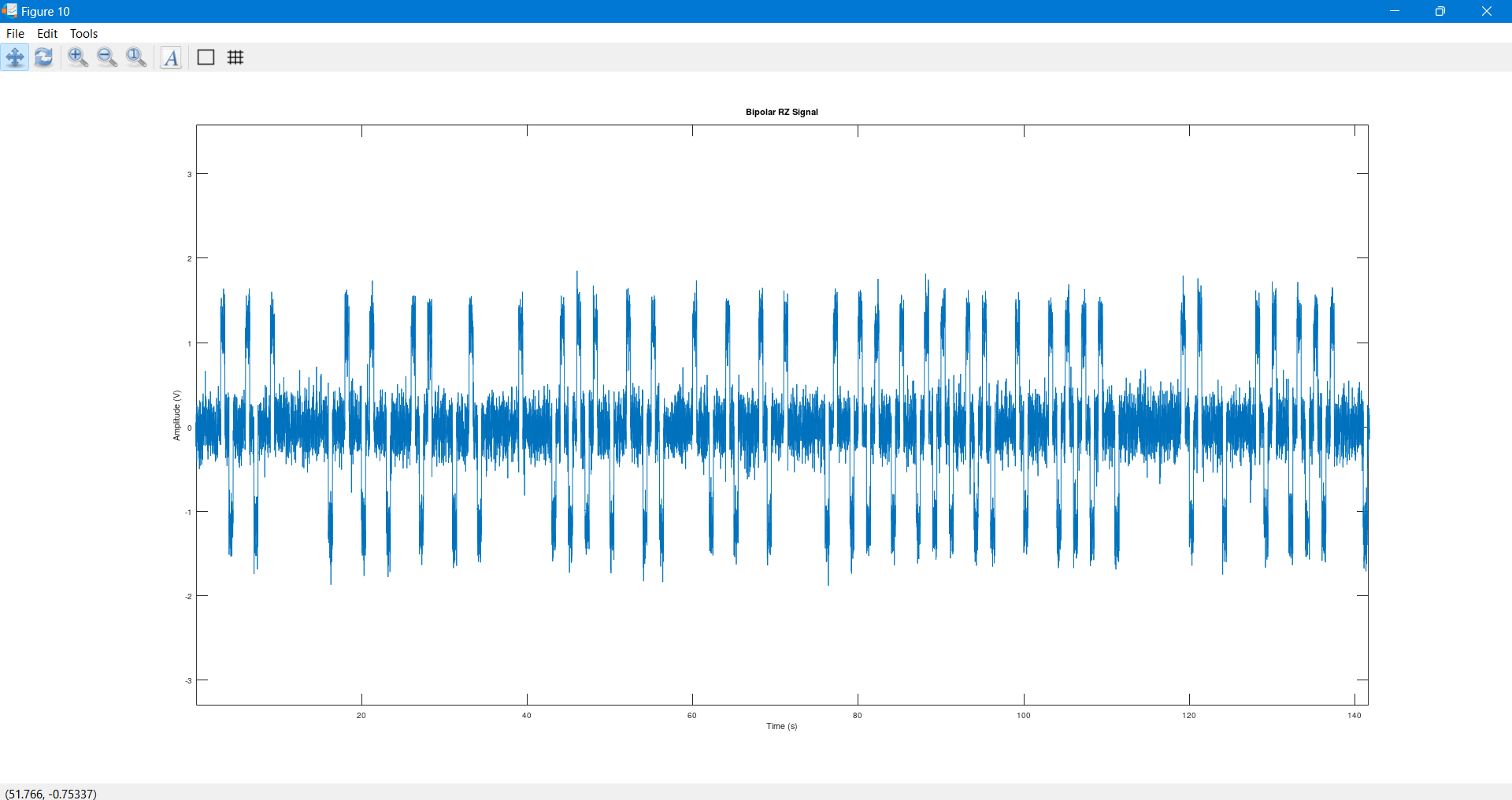


Figure 10: Bipolar RZ signal “with noise” vs Time

### MANCHESTER LINE CODING

#### CODE

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%recieve manchester line coding  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% start manchester code  
signal\_5 = manchester\_coding(bits,1.2,samples\_per\_bit);  
recieved\_bits\_5 = manchester\_coding\_reciever(signal\_5,samples\_per\_bit);  
ber\_signal\_5 = calculate\_ber(bits,recieved\_bits\_5);  
% calculate number of errors  
number\_of\_errors\_signal\_5 = ber\_signal\_5 \* 10000;  
%add noise to tx signal and let sigma = 0.2 to plot the noisy signal  
noisy\_signal\_5 = add\_noise(signal\_5,0.2,samples\_per\_bit);  
% plot noisy\_signal  
plot\_noisy\_signal\_manchester(noisy\_signal\_5,samples\_per\_bit);  
% sweep sigma 0 -> 1.2 and calculate ber  
ber\_values\_signal\_5 = sweep\_sigma\_manchester(signal\_5,bits,samples\_per\_bit);

#### GRAPHS

A screenshot of a computer

Description automatically generated with medium confidence

Figure 11: Manchester coding

A screenshot of a computer

Description automatically generated with medium confidence

Figure 12: Manchester Signal “with noise”

### Plot ber\_values of different line coding with sigma values

#### CODE

% plot ber\_values of different line codings with sigmas values  
figure  
hold on  
semiology(sigma,ber\_values\_signal\_1);  
semiology (sigma,ber\_values\_signal\_2);  
semiology(sigma,ber\_values\_signal\_3);  
semiology (sigma,ber\_values\_signal\_4);  
semiology(sigma,ber\_values\_signal\_5);  
legend(“Unipolar nrz”,”Polar nrz”,”Unipolar rz”,”Bipolar rz”,”Manchester”)  
xlabel(‘Sigma’)  
ylabel(‘BER’);  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_END\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### GRAPH

A screen shot of a graph

Description automatically generated with low confidence

Figure 13: ber\_values of different line coding with sigma values

## MAIN.M FILE

%clear memory  
clear all; close all;  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% include some files  
# <include>sweep\_sigma\_toggle.m</include>  
# <include>sweep\_sigma\_uni\_rz.m</include>  
# <include>manchester\_coding\_reciever.m</include>  
#<include>sweep\_sigma\_manchester.m</include>  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% load package communications  
pkg load communications  
% define samples\_per\_bit  
samples\_per\_bit = 100;  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% Generate a sequence of 10000 random bits  
bits = generate\_bits(10000);  
  
% Generate a Unipolar NRZ signal from the bit sequence with a high voltage level of 1.2V  
signal\_1 = unipolar\_nrz(bits,1.2);  
  
% define vector time  
t = linspace(0, length(signal\_1)/samples\_per\_bit, length(signal\_1));  
  
% plot spectral diagram  
plot\_spectral\_domain(signal\_1);  
% Plot the eye diagram and set the plot limits  
eyediagram(signal\_1, 300,1,1);  
xlim([-0.165, 0.5]);  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% recieve unipolar nrz  
recieved\_bits\_1 = unipolar\_nrz\_reciever(signal\_1,10000,0.6);  
% bit error rate  
ber\_signal\_1 = calculate\_ber(bits,recieved\_bits\_1);  
% calculate number of errors  
number\_of\_errors\_signal\_1 = ber\_signal\_1 \* 10000;  
%add noise to tx signal and let sigma = 0.2 to plot the noisy signal  
noisy\_signal\_1 = add\_noise(signal\_1,0.2,samples\_per\_bit);  
% plot noisy\_signal  
plot\_noisy\_signal(noisy\_signal\_1,samples\_per\_bit);  
% sweep value of sigma  
sigma = linspace(0,1.2,10);  
ber\_values\_signal\_1 = sweep\_sigma(signal\_1,0.6,bits,samples\_per\_bit);  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%recieve polar\_nrz line coding  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% line code the stream of bits to polar\_nrz  
signal\_2 = polar\_nrz(bits,1.2,samples\_per\_bit);  
recieved\_bits\_2 = polar\_nrz\_reciever (signal\_2,10000,0);  
  
ber\_signal\_2 = calculate\_ber(bits,recieved\_bits\_2);  
  
number\_of\_error\_signal\_2 =ber\_signal\_2\*10000;  
  
noisy\_signal\_2=add\_noise(signal\_2,0.2,samples\_per\_bit);  
plot\_noisy\_signal\_polar\_rz(noisy\_signal\_2,samples\_per\_bit);  
  
% sweep sigma 0 -> 1.2 and calculate ber  
ber\_values\_signal\_2=sweep\_sigma(signal\_2,0,bits,100);  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%recieve unipolar\_rz line coding  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% line code the stream of bits to unipolar return to zero  
signal\_3 = unipolar\_rz(bits,1.2);  
% recieve unipolar rz  
recieved\_bits\_3 = unipolar\_rz\_reciever(signal\_3,10000,0.6);  
% bit error rate  
ber\_signal\_3 = calculate\_ber(bits,recieved\_bits\_3);  
% calculate number of errors  
number\_of\_errors\_signal\_3 = ber\_signal\_3 \* 10000;  
%add noise to tx signal and let sigma = 0.2 to plot the noisy signal  
noisy\_signal\_3 = add\_noise(signal\_3,0.2,samples\_per\_bit);  
% plot noisy\_signal  
plot\_noisysignal\_unipolar\_rz(noisy\_signal\_3,samples\_per\_bit);  
% sweep sigma values from 0 -> 1.2  
ber\_values\_signal\_3 = sweep\_sigma\_uni\_rz(signal\_3,0.6,bits,samples\_per\_bit);  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%recieve bipolar\_rz line coding  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% line code the stream of bits to bipolar return to zero  
signal\_4 = bipolar\_rz(bits,1.2);  
%recieve bipolar rz. note : decision levels is determind inside this function  
recieved\_bits\_4 = bipolar\_rz\_reciever(signal\_4,10000,0,100);  
% calculate ber for bipolar rz  
ber\_signal\_4 = calculate\_ber(bits,recieved\_bits\_4);  
% calculate number of errors  
number\_of\_errors\_signal\_4 = ber\_signal\_4 \* 10000;  
%add noise to tx signal and let sigma = 0.2 to plot the noisy signal  
noisy\_signal\_4 = add\_noise(signal\_4,0.2,samples\_per\_bit);  
% plot noisy\_signal  
plot\_noisy\_signal\_bipolar\_rz(noisy\_signal\_4,samples\_per\_bit);  
% sweep sigma 0 -> 1.2 and calculate ber  
ber\_values\_signal\_4 = sweep\_sigma\_toggle(signal\_4,bits,samples\_per\_bit);  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%recieve manchester line coding  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% start manchester code  
signal\_5 = manchester\_coding(bits,1.2,samples\_per\_bit);  
recieved\_bits\_5 = manchester\_coding\_reciever(signal\_5,samples\_per\_bit);  
ber\_signal\_5 = calculate\_ber(bits,recieved\_bits\_5);  
% calculate number of errors  
number\_of\_errors\_signal\_5 = ber\_signal\_5 \* 10000;  
%add noise to tx signal and let sigma = 0.2 to plot the noisy signal  
noisy\_signal\_5 = add\_noise(signal\_5,0.2,samples\_per\_bit);  
% plot noisy\_signal  
plot\_noisy\_signal\_manchester(noisy\_signal\_5,samples\_per\_bit);  
% sweep sigma 0 -> 1.2 and calculate ber  
ber\_values\_signal\_5 = sweep\_sigma\_manchester(signal\_5,bits,samples\_per\_bit);  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
% plot ber\_values of different line codings with sigmas values  
figure  
hold on  
semilogy(sigma,ber\_values\_signal\_1);  
semilogy (sigma,ber\_values\_signal\_2);  
semilogy(sigma,ber\_values\_signal\_3);  
semilogy (sigma,ber\_values\_signal\_4);  
semilogy(sigma,ber\_values\_signal\_5);  
legend("Unipolar nrz","Polar nrz","Unipolar rz","Bipolar rz","Manchester")  
xlabel('Sigma')  
ylabel('BER');  
  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%%%%%%%%%%%%%%%%%%%%%%%%%%% BONUS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% BOUNS !!!!!!!!!!!! % sigma = 0.2 0.3 0.4  
sigma\_bouns = 0.2 : 0.1 : 0.4;  
number\_of\_error\_in\_recieved\_noise\_bipolar\_rz = detect\_number\_of\_errors(signal\_4,sigma\_bouns,samples\_per\_bit,bits)  
% \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_END\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Bonus (FOR THE CASE OF BIPOLAR RETURN TO ZERO, DESIGN AN ERROR DETECTION CIRCUIT. COUNT THE NUMBER OF DETECTED ERRORS IN CASE OF DIFFERENT NUMBER OF SIGMA (USE THE OUTPUT OF STEP 8).

* First, we made a function that detects the no. of errors.

**function** **number\_of\_errors** = **detect\_number\_of\_errors**(signal,sigma,samples\_per\_bit,bits)  
 number\_of\_errors = zeros(1,length(sigma));  
 **for** i=1:length(sigma)  
 noisy\_signal = add\_noise(signal,sigma(i),samples\_per\_bit);  
 recieved\_bits = bipolar\_rz\_reciever(noisy\_signal,10000,0,100);  
 ber\_noise\_bipolar\_rz = calculate\_ber(bits,recieved\_bits);  
 number\_of\_errors(i) = ber\_noise\_bipolar\_rz\*10000;  
 endfor  
**end**

* Then we calculated the error in main

%%%%%%%%%%%%%%%%%%%%%%%%%%% BONUS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% BOUNS!!!!!!!!!!!! % sigma = 0.2 0.3 0.4  
sigma\_bouns = 0.2 : 0.1 : 0.4;  
number\_of\_error\_in\_recieved\_noise\_bipolar\_rz = detect\_number\_of\_errors(signal\_4,sigma\_bouns,samples\_per\_bit,bits)  
% \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_END\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Result

A black background with white text

Description automatically generated with low confidence

## WORKSPACE

A picture containing text, screenshot, font, number

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidence

## FOCUS ON SOME VARIABLES

### NO. OF ERRORS SIGNALS 1 -> 5 without noise

A screenshot of a computer

Description automatically generated with medium confidence

### BER FOR SIGNALS 1 - > 5 without noise

A screenshot of a computer

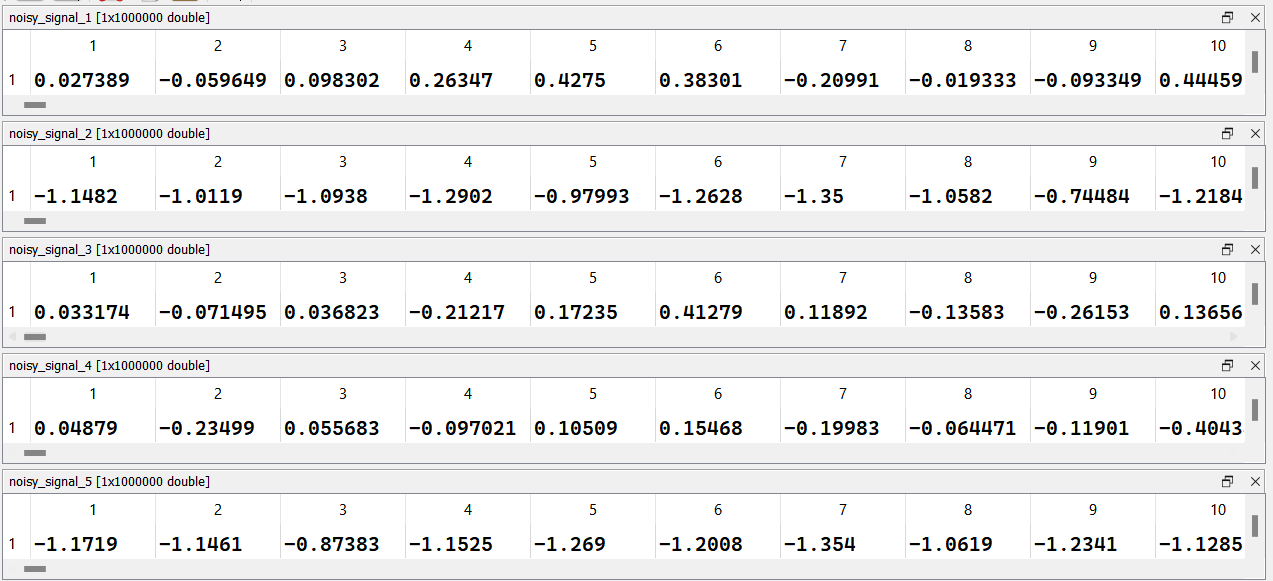
Description automatically generated with medium confidence

### BER\_VALUES VS SIGMA FOR DIFFERENT VARIABLES

A screenshot of a computer

Description automatically generated with low confidence

### NOISY SIGNALS AT SIGMA = 0.2



### CONCLUSIONS

From previous calculations we can say that **Manchester** coding is the best line coding while **Bipolar** line coding is the worst.

# PART II

## THE USED FUNCTIONS

1. generate\_bits(num\_bits) “Repeated”: This function generates a stream of random bits, where the num\_bits parameter specifies the number of bits to generate. This function would randomly select either a 1 or 0 for each bit.
2. line\_code (bits,voltage\_high,voltage\_low): This function is used to generate a stream of polar NRZ bits, where bits is the random bits generated from generate\_bits(num\_bits).

**function** **line\_coded** = **line\_code** (bits,voltage\_high,voltage\_low)  
 line\_coded = [];  
 **for** i = 1:length(bits);  
 **if** bits(i) == 1  
 line\_coded = [line\_coded ones(1,200)\*voltage\_high];  
 **elseif** bits(i) == 0  
 line\_coded = [line\_coded ones(1,200)\*voltage\_low];  
 endif   
 endfor  
**end**

1. decision (bits): This function takes a signal as a parameter and decides whether each bit is one or zero then returns the reconstructed bits

**function** **reconstructed** = **decision** (bits)  
 reconstructed = [];  
 **for** index=1:length(bits)  
 **if** bits(index) > 0  
 reconstructed(index) = 1;  
 **elseif** bits(index)<=0  
 reconstructed(index)=0;  
 endif  
 endfor  
endfunction

1. calculate\_ber(tx\_bits,rx\_bits): This function takes a stream of transimitted & received bits, compares between them then calculates the bit error rate from the formula:

% it is a function that calculate BER  
% it takes two parmeters tx\_bits  
% tx\_bits -> stream of bits of the transmitter  
% rx\_bits -> stream of bits of the Reciever  
**function** **BER** = **calculate\_ber**(tx\_bits,rx\_bits)  
 NumOfErrors = 0;  
 **for** index = 1:length(tx\_bits);  
 **if** tx\_bits != rx\_bits  
 NumOfErrors = NumOfErrors +1;  
 endif  
 endfor  
 BER = NumOfErrors/length(tx\_bits);  
**end**

## PART II TRANSIMITTER

### Generate stream of random bits (100 bit) (This bit stream should be selected to be random, which means that the type of each bit is randomly selected by the program code to be either ‘1’ or ‘0’.)

#### CODE

%%%%%%%%%%%%%%%%%%%%%%%%% Start Of Main %%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
clear all; close all;  
# <include>generate\_bits.m</include>  
# <include>line\_code.m</include>  
# <include>decision.m</include>  
# <include>calculate\_ber.m</include>  
fc = 1e9; % Carrier frequency  
Tb = 10/fc; % bit time  
Rb = 1/Tb; % bit rate  
ts = Tb/200; % sampling time  
numOfBits = 100; % no. of bits  
  
  
%%%%%%%%%%%%%%%%%%%%%% Part II Transmitter %%%%%%%%%%%%%%%%%%%%%%%%  
%%%%%%%%%%%%%%% 1) Generate stream of random bits %%%%%%%%%%%%%%%%%  
rand\_bits = generate\_bits(numOfBits);  
t\_bits = linspace(0,Tb\*numOfBits,numOfBits);  
  
% Graph 100 random bits  
figure  
stairs(t\_bits, rand\_bits);  
axis([0 t\_bits(end) -0.5 1.5]);

#### A screenshot of a computer Description automatically generated with medium confidenceGRAPH

### Line code the stream of bits (pulse shape) according to Polar non return to zero (Maximum voltage +1, Minimum voltage -1).

#### CODE

%%%%%%%%%%%%%%%%%%%%%%%% 2) Polar NRZ coding %%%%%%%%%%%%%%%%%%%%%%%%%  
  
line\_coded\_bits = line\_code(rand\_bits, 1, -1);  
Ns = length(line\_coded\_bits);  
time = 0:ts:ts\*(Ns-1);  
  
  
figure  
plot(time,line\_coded\_bits);  
axis([0 length(line\_coded\_bits)\*ts -1.5 1.5]);  
xlabel("Time");  
ylabel("Line coded bits");

#### GRAPH

A screenshot of a bar code

Description automatically generated with medium confidence

### Plot the spectral domains.

#### CODE

%%%%%%%%%%%%%%%%%%% 3) Plotting the spectral domain %%%%%%%%%%%%%%%%%%%  
  
df = 1/(Ns\*ts);  
fs = 1/ts;  
N = length(time);  
f = (-0.5\*fs):df:(0.5\*fs-df);  
% Calculate the spectrum  
line\_coded\_spectrum = abs((fftshift(fft(line\_coded\_bits)).^2)/N);  
  
figure  
plot(f, line\_coded\_spectrum);  
axis([-6e8 6e8 0 max(line\_coded\_spectrum)+20]);  
xlabel("Frequency");  
ylabel("line\_coded\_power\_spectrum");

#### A screenshot of a computer Description automatically generatedGRAPH

A screen shot of a graph

Description automatically generated with medium confidence

### Plot the time domain of the modulated BPSK signal (𝑓𝑐 = 1𝐺𝐻𝑧)

#### CODE

%%%%%%%%%%%%%%%%%% 4) BPSK modulation time domain %%%%%%%%%%%%%%%%%%%  
carrier = sin(2\*pi\*fc\*time); % carrier is a sine wave  
BPSK\_modulated\_signal = line\_coded\_bits.\*carrier; % modulating the signal  
  
% plotting BPSK\_modulated\_signal  
figure  
plot(time, BPSK\_modulated\_signal);  
axis([0 10/fc -1.5 1.5]);  
xlabel("Time");  
ylabel("BPSK\_modulated\_signal");

#### GRAPH

A screen shot of a computer screen

Description automatically generated with medium confidence

**ZOOMED IN**

A screen shot of a graph

Description automatically generated with medium confidence

### Plot the spectrum of the modulated BPSK signal.

#### CODE

%%%%%%%%%%%%%%%%%%%% 5) Plotting BPSK spectrum %%%%%%%%%%%%%%%%%%%%%%  
BPSK\_spectrum = abs(fftshift(fft(BPSK\_modulated\_signal)));  
figure  
plot(f, BPSK\_spectrum);  
xlabel("frequency");  
ylabel("BPSK\_spectrum");

#### GRAPH

A screen shot of a graph

Description automatically generated with medium confidence

## PART II RECEIVER

### Design a receiver which consists of modulator, integrator (simply LPF) and decision device.

#### CODE

%%%%%%%%%%%%%%%%%%%%%%%% Part II Receiver %%%%%%%%%%%%%%%%%%%%%%%%%%%  
%%%%%%%%%%%%%%%%%%%%%% 6) BPSK demodulation %%%%%%%%%%%%%%%%%%%%%%%%%  
BPSK\_demodulated\_signal = BPSK\_modulated\_signal.\*carrier;  
y=[];  
**for** index = 1:200:length (BPSK\_demodulated\_signal);  
 y = [y trapz(time(index:index+199), BPSK\_demodulated\_signal(index:index+199))];  
**end**  
  
  
figure  
  
reconstructed\_bits = line\_code(decision(y),1,-1);  
plot (time, reconstructed\_bits);  
axis([0 length(line\_coded\_bits)\*ts -1.5 1.5]);  
xlabel("Time");  
ylabel("rec");

#### GRAPH

A screenshot of a bar code

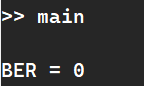
Description automatically generated with medium confidence

### Compare the output of decision level with the generated stream of bits in the transmitter. The comparison is performed by comparing the value of each received bit with the corresponding transmitted bit (step 1) and count number of errors. Then calculate bit error rate (BER) = number of error bits/ Total number of bits.

#### CODE

%%%%%%%%%%%%%%%%%%%%%% 7) Calculate BER %%%%%%%%%%%%%%%%%%%%%%%%%  
BER = calculate\_ber(line\_coded\_bits, reconstructed\_bits)

#### Result



## PART II FULL CODE

%%%%%%%%%%%%%%%%%%%%%%%%% Start Of Main %%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
clear all; close all;  
# <include>generate\_bits.m</include>  
# <include>line\_code.m</include>  
# <include>decision.m</include>  
# <include>calculate\_ber.m</include>  
  
  
fc = 1e9; % Carrier frequency  
Tb = 10/fc; % bit time  
Rb = 1/Tb; % bit rate  
ts = Tb/200; % sampling time  
numOfBits = 100; % no. of bits  
%%%%%%%%%%%%%%%%%%%%%% Part II Transmitter %%%%%%%%%%%%%%%%%%%%%%%%  
%%%%%%%%%%%%%%% 1) Generate stream of random bits %%%%%%%%%%%%%%%%%  
rand\_bits = generate\_bits(numOfBits);  
t\_bits = linspace(0,Tb\*numOfBits,numOfBits);  
  
% Graph 100 random bits  
figure  
stairs(t\_bits, rand\_bits);  
axis([0 t\_bits(end) -0.5 1.5]);  
  
%%%%%%%%%%%%%%%%%%%%%%%% 2) Polar NRZ coding %%%%%%%%%%%%%%%%%%%%%%%%%  
  
line\_coded\_bits = line\_code(rand\_bits, 1, -1);  
Ns = length(line\_coded\_bits);  
time = 0:ts:ts\*(Ns-1);  
  
  
figure  
plot(time,line\_coded\_bits);  
axis([0 length(line\_coded\_bits)\*ts -1.5 1.5]);  
xlabel("Time");  
ylabel("Line coded bits");  
  
%%%%%%%%%%%%%%%%%%% 3) Plotting the spectral domain %%%%%%%%%%%%%%%%%%%  
  
df = 1/(Ns\*ts);  
fs = 1/ts;  
N = length(time);  
f = (-0.5\*fs):df:(0.5\*fs-df);  
% Calculate the spectrum  
line\_coded\_spectrum = abs((fftshift(fft(line\_coded\_bits)).^2)/N);  
  
figure  
plot(f, line\_coded\_spectrum);  
axis([-6e8 6e8 0 max(line\_coded\_spectrum)+20]);  
xlabel("Frequency");  
ylabel("line\_coded\_power\_spectrum");  
  
%%%%%%%%%%%%%%%%%% 4) BPSK modulation time domain %%%%%%%%%%%%%%%%%%%  
carrier = sin(2\*pi\*fc\*time); % carrier is a sine wave  
BPSK\_modulated\_signal = line\_coded\_bits.\*carrier; % modulating the signal  
  
% plotting BPSK\_modulated\_signal  
figure  
plot(time, BPSK\_modulated\_signal);  
axis([0 10/fc -1.5 1.5]);  
xlabel("Time");  
ylabel("BPSK\_modulated\_signal");  
  
%%%%%%%%%%%%%%%%%%%% 5) Plotting BPSK spectrum %%%%%%%%%%%%%%%%%%%%%%  
BPSK\_spectrum = abs(fftshift(fft(BPSK\_modulated\_signal)));  
figure  
plot(f, BPSK\_spectrum);  
xlabel("frequency");  
ylabel("BPSK\_spectrum");  
  
%%%%%%%%%%%%%%%%%%%%%%%% Part II Receiver %%%%%%%%%%%%%%%%%%%%%%%%%%%  
%%%%%%%%%%%%%%%%%%%%%% 6) BPSK demodulation %%%%%%%%%%%%%%%%%%%%%%%%%  
BPSK\_demodulated\_signal = BPSK\_modulated\_signal.\*carrier;  
y=[];  
for index = 1:200:length(BPSK\_demodulated\_signal);  
 y = [y trapz(time(index:index+199), BPSK\_demodulated\_signal(index:index+199))];  
end  
  
figure  
reconstructed\_bits = line\_code(decision(y),1,-1);  
plot(time, reconstructed\_bits);  
axis([0 length(line\_coded\_bits)\*ts -1.5 1.5]);  
xlabel("Time");  
ylabel("rec");  
  
%%%%%%%%%%%%%%%%%%%%%% 7) Calculate BER %%%%%%%%%%%%%%%%%%%%%%%%%  
BER = calculate\_ber(line\_coded\_bits, reconstructed\_bits)  
  
%%%%%%%%%%%%%%%%%%%%%%%%% End Of Main %%%%%%%%%%%%%%%%%%%%%%%%%%%

## WORKSPACE

