

# 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.

ASU – ENG

[ECE252s] – Fundamentals of Communications Systems



# Fundamentals of Communications Systems Project

## 1 TABLE OF CONTENTS

1	TABLE OF CONTENTS.....	1
2	TEAM MEMBERS.....	4
3	BRIEF ABOUT PROJECT PART ONE.....	4
4	IMPORTANT NOTES.....	5
5	PART I TRANSMITTER.....	6
5.1	LINE CODING SYSTEMS.....	6
5.2	SPECTRAL DOMAIN FUNCTION.....	13
5.3	THE MAIN.M FILE.....	15
5.4	HERE IS A SUMMARY OF THE CODE.....	15
5.5	SNAPSHOTS.....	17
5.5.1	Unipolar Non-Return to Zero.....	17
5.5.2	Polar Non-Return to Zero.....	17
5.5.3	Unipolar Return to Zero.....	18
5.5.4	Bipolar Return To Zero.....	18
5.5.5	Manchester Coding.....	19
5.5.6	Spectral Domain of Unipolar Non-Return to Zero.....	19
5.5.7	Spectral Domain of Polar Non-Return to Zero.....	20
5.5.8	Spectral Domain of Unipolar Return to Zero.....	20
5.5.9	Spectral Domain of Bipolar Return to Zero.....	21
5.5.10	Spectral Domain of Manchester Coding.....	21
5.5.11	Eye Diagram of Unipolar Non-Return to Zero.....	22
5.5.12	Eye Diagram of Polar Non-Return to Zero.....	22



5.5.13	Eye Diagram of Unipolar Return to Zero.....	23
5.5.14	Eye Diagram of Bipolar Return to Zero .....	23
5.5.15	Eye Diagram of Manchester Coding.....	24
6	PART I RECEIVER.....	24
6.1	USED FUNCTIONS .....	24
6.2	UNIPOLAR NRZ .....	30
6.2.1	CODE .....	30
6.2.2	GRAPHS.....	31
6.3	POLAR NRZ .....	33
6.3.1	CODE .....	33
6.3.2	GRAPHS.....	33
6.4	UNIPOLAR RZ.....	35
6.4.1	CODE .....	35
6.4.2	GRAPHS.....	35
6.5	BIPOLAR RZ.....	37
6.5.1	CODE .....	37
6.5.2	GRAPHS.....	37
6.6	MANCHESTER LINE CODING .....	39
6.6.1	CODE .....	39
6.7	PLOT BER_VALUES OF DIFFERENT LINE CODING WITH SIGMA VALUES ...	41
6.7.1	CODE .....	41
6.7.2	GRAPH.....	41
6.8	MAIN.M FILE.....	43
6.9	<b>Bonus</b> (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).....	46
6.10	WORKSPACE .....	47



6.11	FOCUS ON SOME VARIABLES .....	48
6.11.1	NO. OF ERRORS SIGNALS 1 -> 5 without noise .....	48
6.11.2	BER FOR SIGNALS 1 - > 5 without noise .....	48
6.11.3	BER_VALUES VS SIGMA FOR DIFFERENT VARIABLES.....	49
6.11.4	NOISY SIGNALS AT SIGMA = 0.2.....	49
6.12	CONCLUSIONS .....	49
7	PART II.....	50
7.1	THE USED FUNCTIONS .....	50
7.2	PART II TRANSIMITTER.....	51
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'.) .....	51
7.2.2	Line code the stream of bits (pulse shape) according to Polar non return to zero (Maximum voltage +1, Minimum voltage -1). .....	52
7.2.3	Plot the spectral domains.....	52
7.2.4	Plot the time domain of the modulated BPSK signal ( $f_c = 1GHz$ )... ..	54
7.2.5	Plot the spectrum of the modulated BPSK signal.....	56
7.3	PART II RECEIVER .....	56
7.3.1	Design a receiver which consists of modulator, integrator (simply LPF) and decision device. ....	56
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. ....	60
7.4	PART II FULL CODE .....	60
7.5	WORKSPACE .....	63



## 2 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

## 3 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  $\sigma$ .



## 4 IMPORTANT NOTES

---

### **Hint**

- Throughout the project, we divided the 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.
- In our report you can find the code of the bonus mark of Part I Receiver implemented.
- GitHub Repo: [Fundamentals-of-Communications-Systems-Project](#).
- Required Software and Tools
  - MATLAB/OCTAVE
  - Signal Processing Toolbox

## 5 PART I TRANSMITTER

---

### 5.1 LINE CODING SYSTEMS

- I. `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

```

- II. `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

```



```

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

```

- III. `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

```

- IV. `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
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

```

- V. `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

```

## 5.2 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 labels and 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

```

### 5.3 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]);

```

### 5.4 HERE IS A SUMMARY OF THE CODE

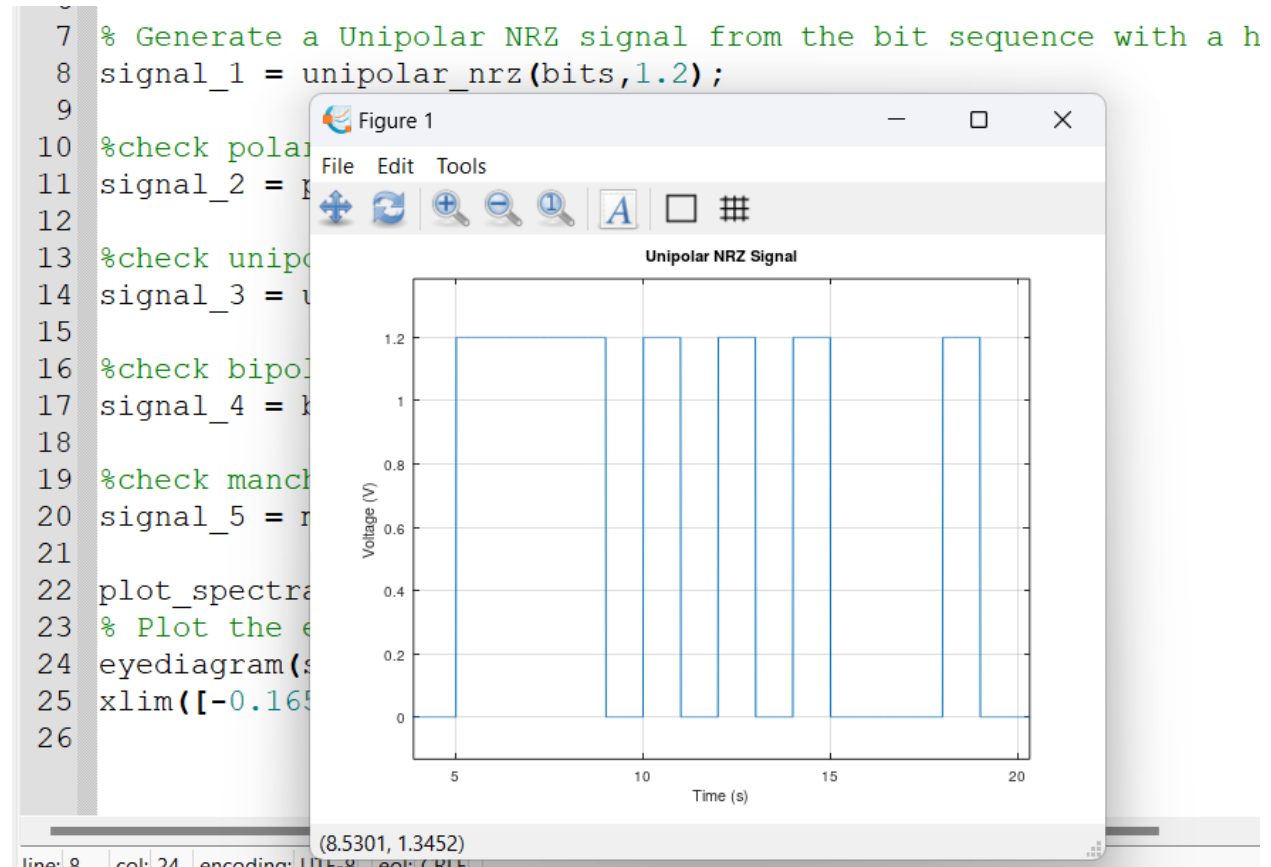
- The code loads the Communications Toolbox package in Octave, a numerical computing software. It then generates a sequence of 10000 random bits using the `generate\_bits` function, which is not shown in the code snippet.
- Next, the code generates several different types of baseband digital signals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



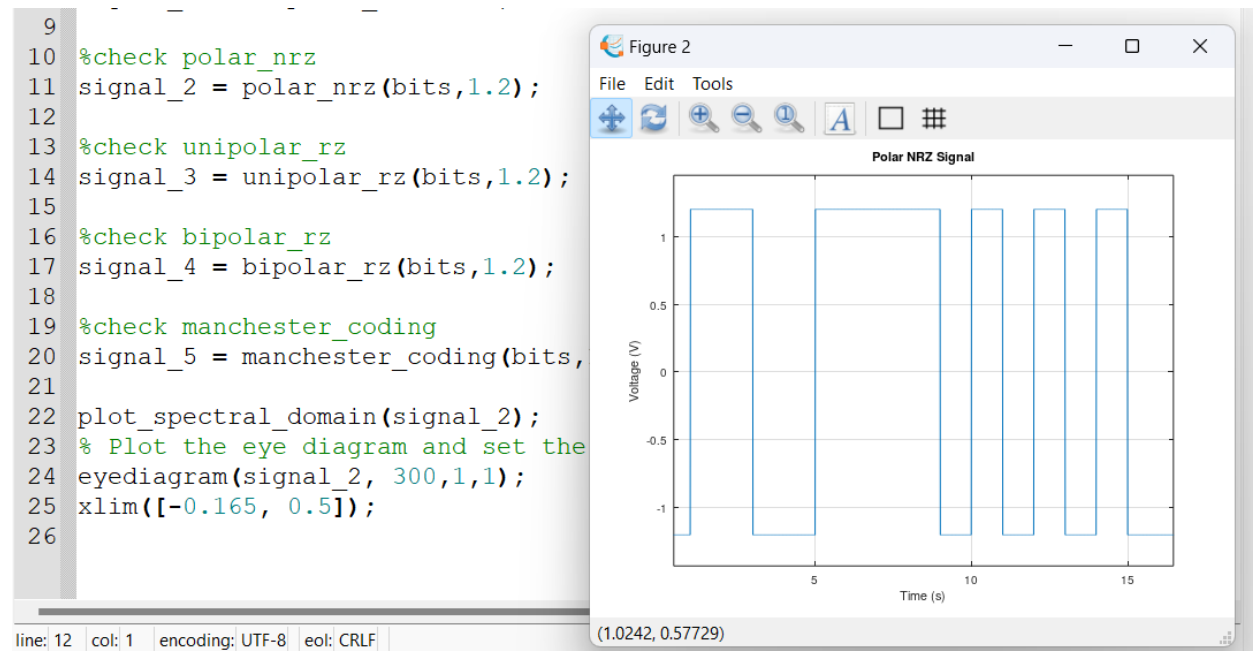
- 1.2V. Unipolar NRZ signal encodes a 1 bit as a high voltage level and a 0 bit as a low voltage level.
  2. `polar_nrz(bits,1.2)`: generates a Polar NRZ signal from the bit sequence, which is similar to Unipolar NRZ except that it encodes a 0 bit as a negative voltage level.
  3. `unipolar_rz(bits,1.2)`: 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 level depending 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.

## 5.5 SNAPSHOTS

### 5.5.1 Unipolar Non-Return to Zero

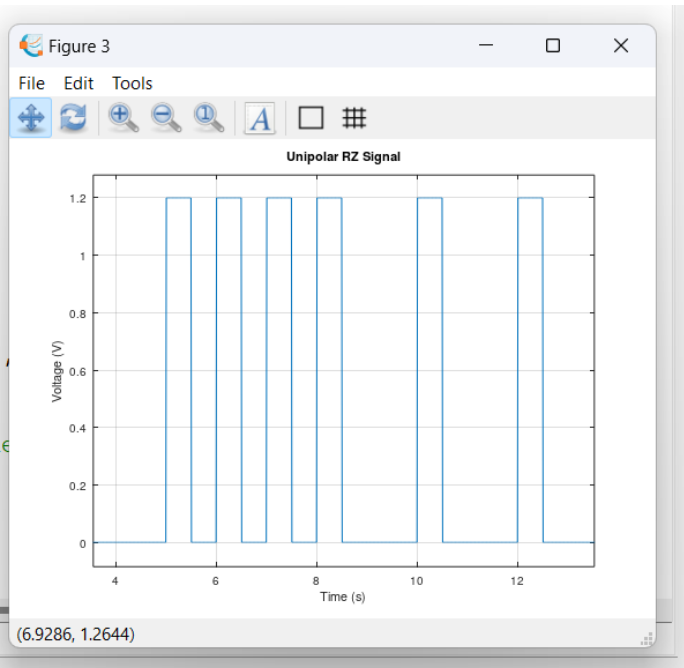


### 5.5.2 Polar Non-Return to Zero



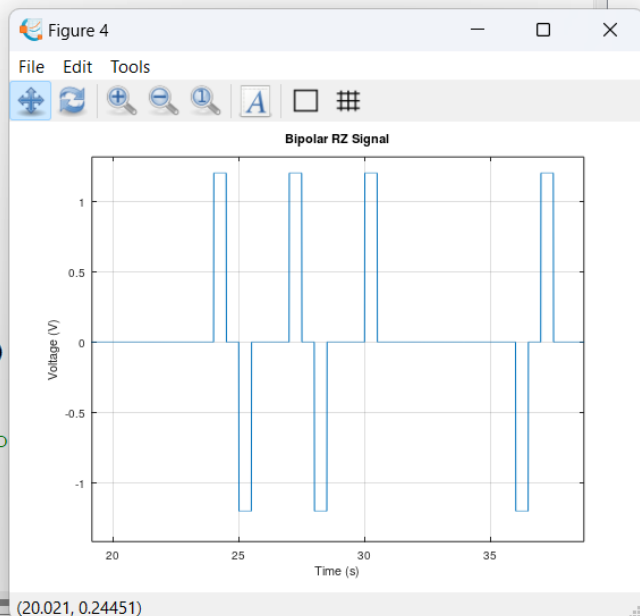
### 5.5.3 Unipolar Return to Zero

```
9
10 %check polar_nrz
11 signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 signal_5 = manchester_coding(bits,
21
22 plot_spectral_domain(signal_2);
23 % Plot the eye diagram and set the
24 eyediagram(signal_2, 300,1,1);
25 xlim([-0.165, 0.5]);
26
```



### 5.5.4 Bipolar Return To Zero

```
7 % Generate a Unipolar NRZ signal from the bit sequence with a high voltage level
8 signal_1 = unipolar_nrz(bits,1.2);
9
10 %check polar_nrz
11 signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 signal_5 = manchester_coding(bits,1.2)
21
22 plot_spectral_domain(signal_2);
23 % Plot the eye diagram and set the plot
24 eyediagram(signal_2, 300,1,1);
25 xlim([-0.165, 0.5]);
26
```

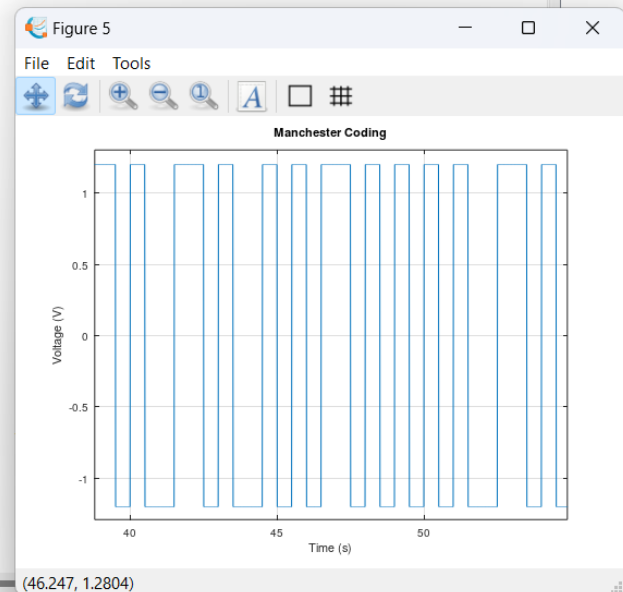


### 5.5.5 Manchester Coding

```

7 % Generate a Unipolar NRZ signal from the bit sequence with a high voltage level
8 signal_1 = unipolar_nrz(bits,1.2);
9
10 %check polar_nrz
11 signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 signal_5 = manchester_coding(bits,1.2);
21
22 plot_spectral_domain(signal_2);
23 % Plot the eye diagram and set the plot
24 eyediagram(signal_2, 300,1,1);
25 xlim([-0.165, 0.5]);
26

```

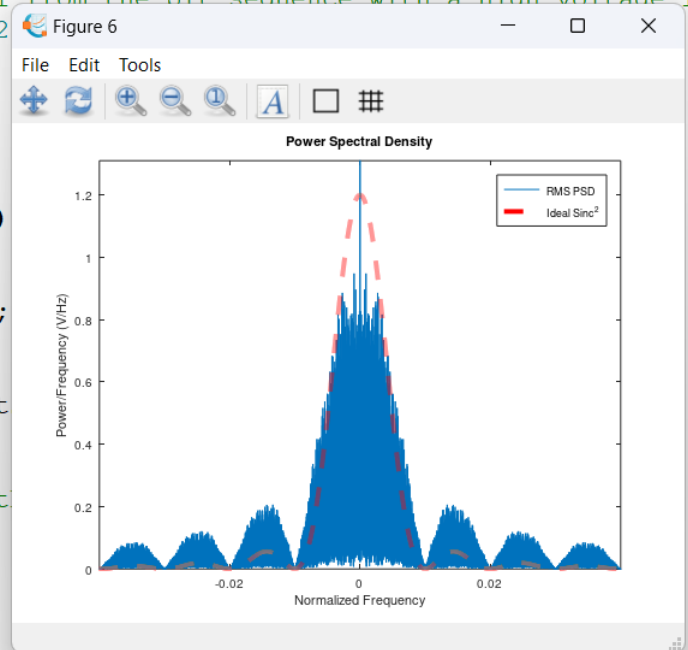


### 5.5.6 Spectral Domain of Unipolar Non-Return to Zero

```

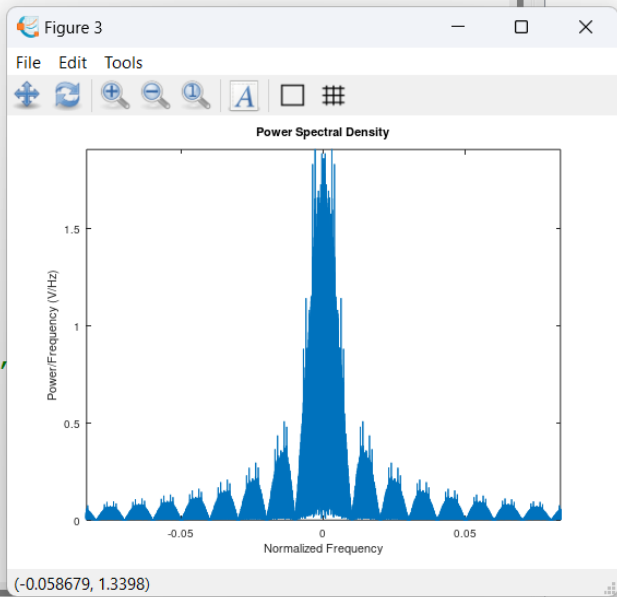
7 % Generate a Unipolar NRZ signal from the bit sequence with a high voltage level
8 signal_1 = unipolar_nrz(bits,1.2);
9
10 %check polar_nrz
11 signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 signal_5 = manchester_coding(bits,1.2);
21
22 plot_spectral_domain(signal_1);
23 % Plot the eye diagram and set the plot
24 eyediagram(signal_2, 300,1,1);
25 xlim([-0.165, 0.5]);
26

```



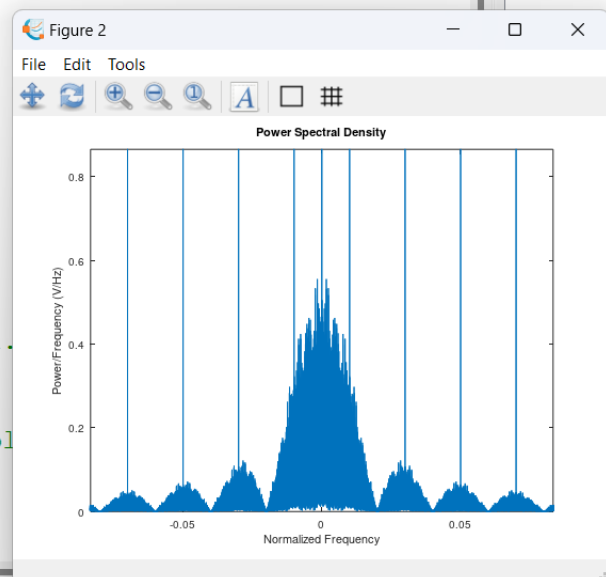
### 5.5.7 Spectral Domain of Polar Non-Return to Zero

```
9
10 %check polar_nrz
11 signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 %signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 %signal_5 = manchester_coding(bits,
21
22 plot_spectral_domain(signal_2);
23 % Plot the eye diagram and set the
24 %eyediagram(signal_2, 300,1,1);
25 %xlim([-0.165, 0.5]);
26
```



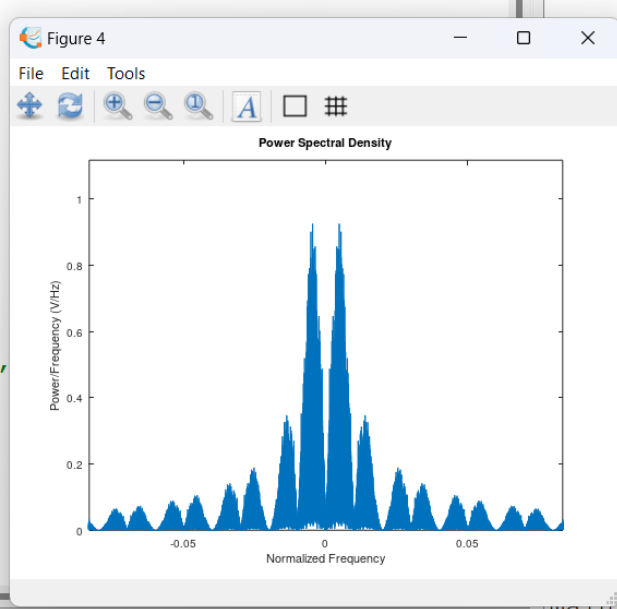
### 5.5.8 Spectral Domain of Unipolar Return to Zero

```
10 %check polar_nrz
11 %signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 %signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 %signal_5 = manchester_coding(bits,1.
21
22 plot_spectral_domain(signal_3);
23 % Plot the eye diagram and set the pl
24 %eyediagram(signal_2, 300,1,1);
25 %xlim([-0.165, 0.5]);
26
```



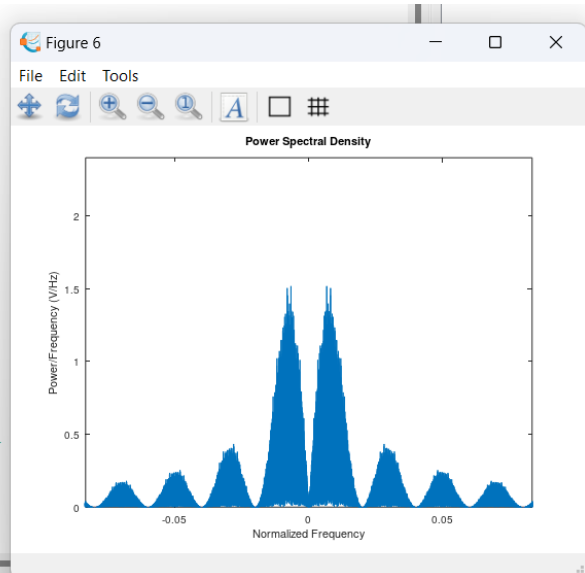
### 5.5.9 Spectral Domain of Bipolar Return to Zero

```
10 %check polar_nrz
11 %signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 %signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 %signal_5 = manchester_coding(bits,
21
22 plot_spectral_domain(signal_4);
23 % Plot the eye diagram and set the
24 %eyediagram(signal_2, 300,1,1);
25 %xlim([-0.165, 0.5]);
26
```



### 5.5.10 Spectral Domain of Manchester Coding

```
10 %check polar_nrz
11 %signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 %signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 %signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 signal_5 = manchester_coding(bits,1.2);
21
22 plot_spectral_domain(signal_5);
23 % Plot the eye diagram and set the plot
24 %eyediagram(signal_2, 300,1,1);
25 %xlim([-0.165, 0.5]);
26
```

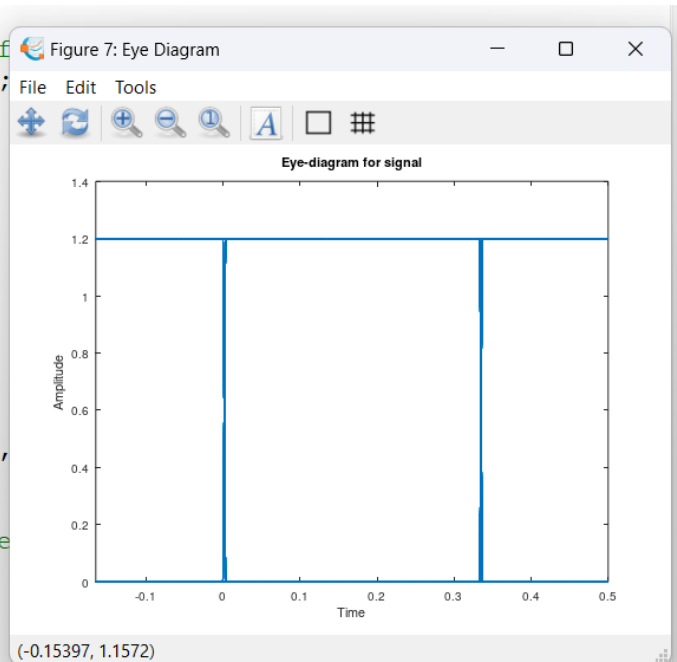


### 5.5.11 Eye Diagram of Unipolar Non-Return to Zero

```

6
7 % Generate a Unipolar NRZ signal from the bit sequence with a high voltage level
8 signal_1 = unipolar_nrz(bits,1.2);
9
10 %check polar_nrz
11 signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 signal_5 = manchester_coding(bits,1.2);
21
22 plot_spectral_domain(signal_2);
23 % Plot the eye diagram and set the plot parameters
24 eyediagram(signal_1, 300,1,1);
25 xlim([-0.165, 0.5]);
26

```

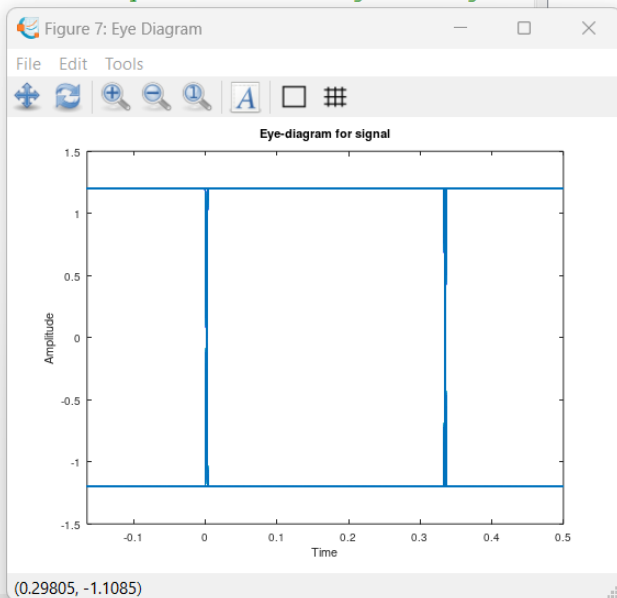


### 5.5.12 Eye Diagram of Polar Non-Return to Zero

```

7 % Generate a Unipolar NRZ signal from the bit sequence with a high voltage level
8 signal_1 = unipolar_nrz(bits,1.2);
9
10 %check polar_nrz
11 signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 signal_5 = manchester_coding(bits,1.2);
21
22 plot_spectral_domain(signal_2);
23 % Plot the eye diagram and set the plot parameters
24 eyediagram(signal_2, 300,1,1);
25 xlim([-0.165, 0.5]);
26

```

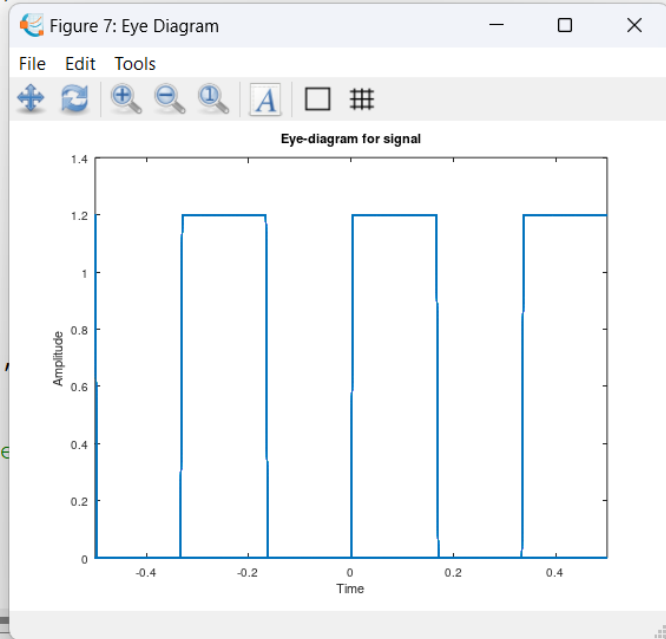


### 5.5.13 Eye Diagram of Unipolar Return to Zero

```

7 % Generate a Unipolar NRZ signal from the bit sequence with a high voltage level
8 signal_1 = unipolar_nrz(bits,1.2);
9
10 %check polar_nrz
11 signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 signal_5 = manchester_coding(bits,1.2);
21
22 plot_spectral_domain(signal_2);
23 % Plot the eye diagram and set the plot limits
24 eyediagram(signal_3, 300,1,1);
25 %xlim([-0.165, 0.5]);
26

```

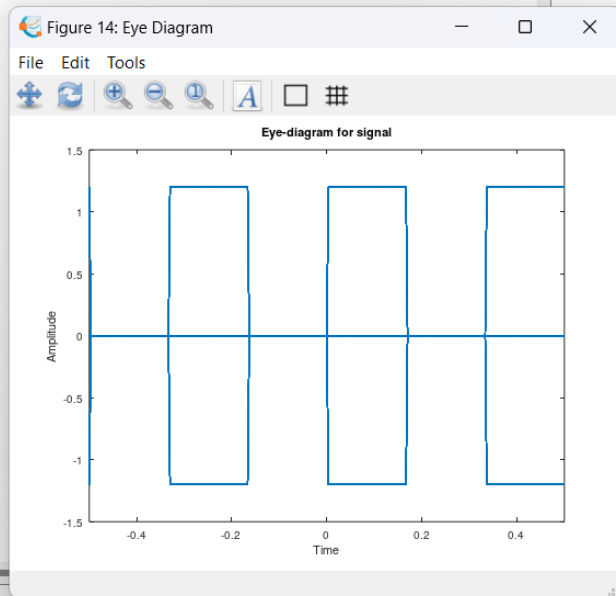


### 5.5.14 Eye Diagram of Bipolar Return to Zero

```

7 % Generate a Unipolar NRZ signal from the bit sequence with a high voltage level
8 signal_1 = unipolar_nrz(bits,1.2);
9
10 %check polar_nrz
11 signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 signal_5 = manchester_coding(bits,1.2);
21
22 plot_spectral_domain(signal_2);
23 % Plot the eye diagram and set the plot limits
24 eyediagram(signal_4, 300,1,1);
25 %xlim([-0.165, 0.5]);
26

```

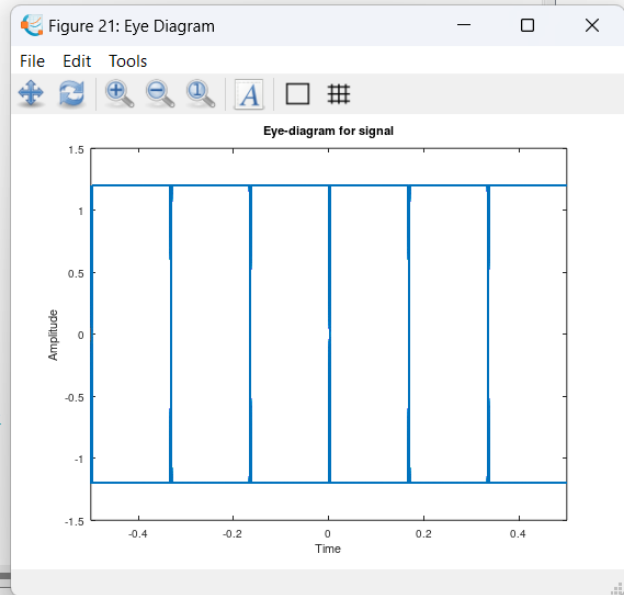


line: 24 col: 20 encoding: UTF-8 eol: CRLF



### 5.5.15 Eye Diagram of Manchester Coding

```
7 % Generate a Unipolar NRZ signal from the bit sequence with a high voltage level
8 signal_1 = unipolar_nrz(bits,1.2);
9
10 %check polar_nrz
11 signal_2 = polar_nrz(bits,1.2);
12
13 %check unipolar_rz
14 signal_3 = unipolar_rz(bits,1.2);
15
16 %check bipolar_rz
17 signal_4 = bipolar_rz(bits,1.2);
18
19 %check manchester_coding
20 signal_5 = manchester_coding(bits,1.2);
21
22 plot_spectral_domain(signal_2);
23 % Plot the eye diagram and set the plot
24 eyediagram(signal_5, 300,1,1);
25 %xlim([-0.165, 0.5]);
26
```



## 6 PART I RECEIVER

### 6.1 USED FUNCTIONS

- I. generate\_bits(num\_of\_bits) **“Repeated”**
- II. unipolar\_nrz(bits, high\_voltage\_level, samples\_per\_bit) **“Repeated”**
- III. plot\_spectral\_domain(waveform) **“Repeated”**
- IV. eyediagram **“Repeated”**
- V. 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
```

```
end
end
```

## VI. calculate\_ber(tx\_bits,rx\_bits)

```
% it is a function that calculate BER
% it takes two parameters tx_bits
% tx_bits -> stream of bits of the transmitter
% rx_bits -> stream of bits of the Receiver
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);
```

## VII. 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
```

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

```
% function to plot noisy 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
```

## IX. 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

```

X. polar\_nrz(bits, high\_voltage\_level, low\_voltage\_level, samples\_per\_bit)

**“Repeated”**

XI. 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
end

```

XII. 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

```

XIII. unipolar\_rz(bits, high\_voltage\_level, samples\_per\_bit) **“Repeated”**

XIV. 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

```

XV. bipolar\_rz(bits, high\_voltage\_level, samples\_per\_bit) **“Repeated”**

XVI. 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

```

#### XVII. 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

```

#### XVIII. 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

```

#### XIX. manchester\_coding(bits, high\_voltage, sampling\_per\_bit) “Repeated”

#### XX. 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

```

## XXI. 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

```

## XXII. 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

```

## 6.2 UNIPOLAR NRZ

### 6.2.1 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);
```

## 6.2.2 GRAPHS

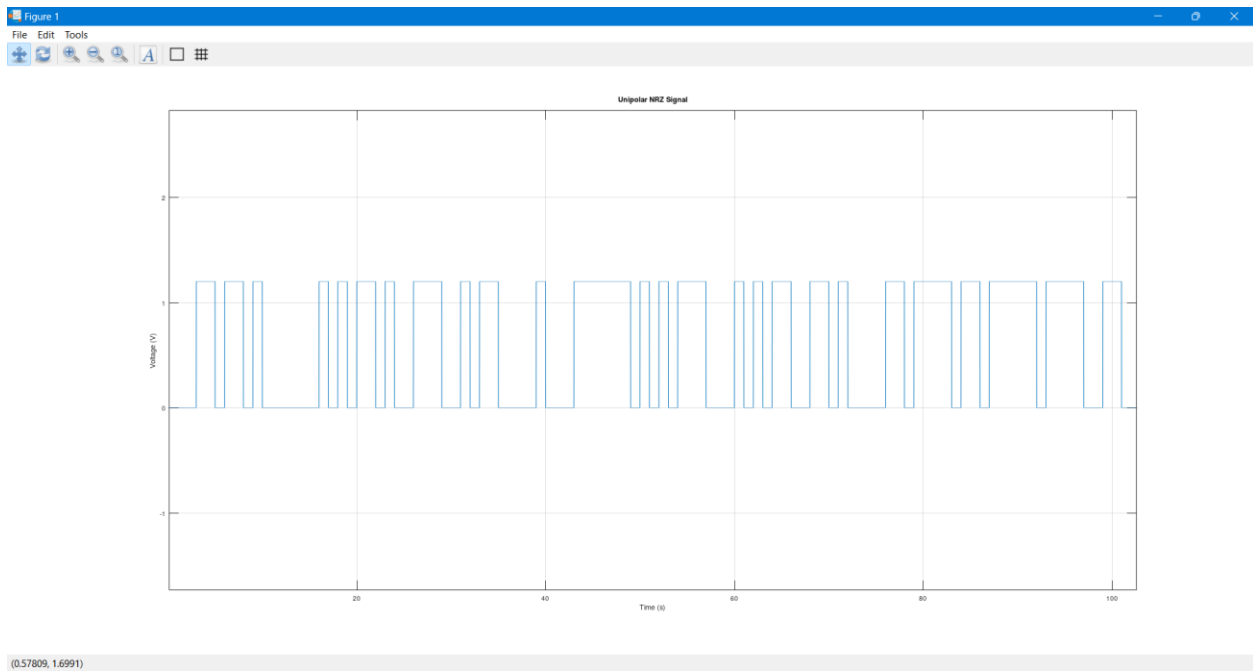


Figure 1: Unipolar NRZ vs Time

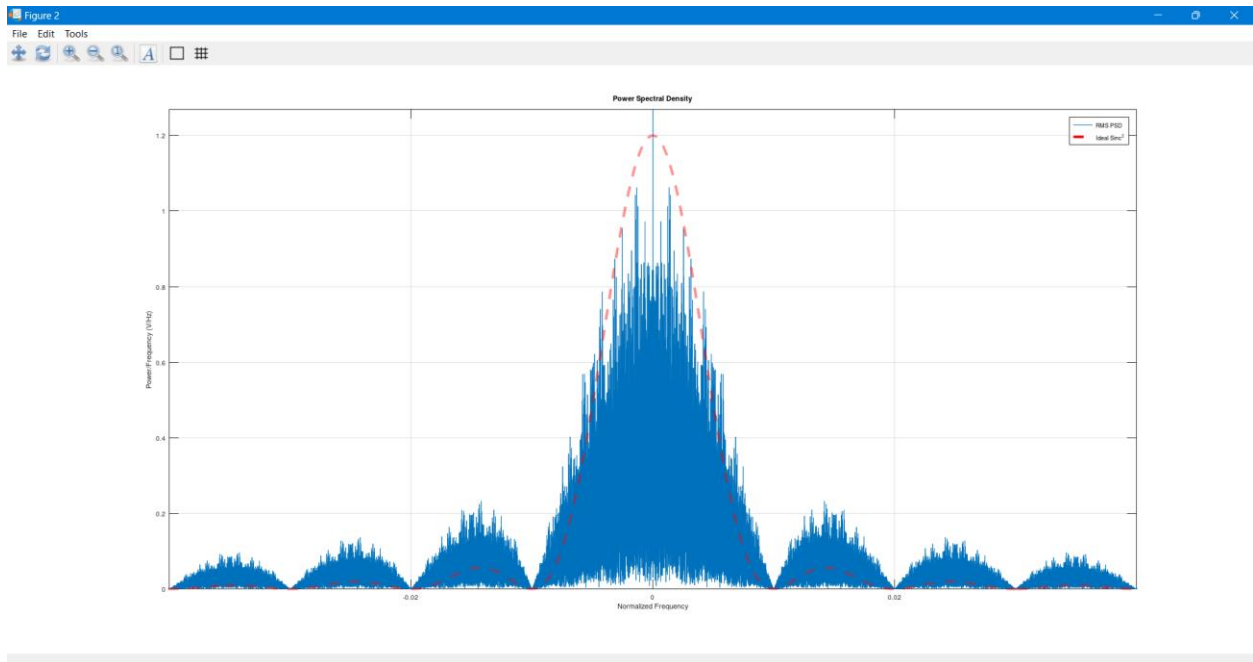


Figure 2: Unipolar NRZ power spectral vs Time



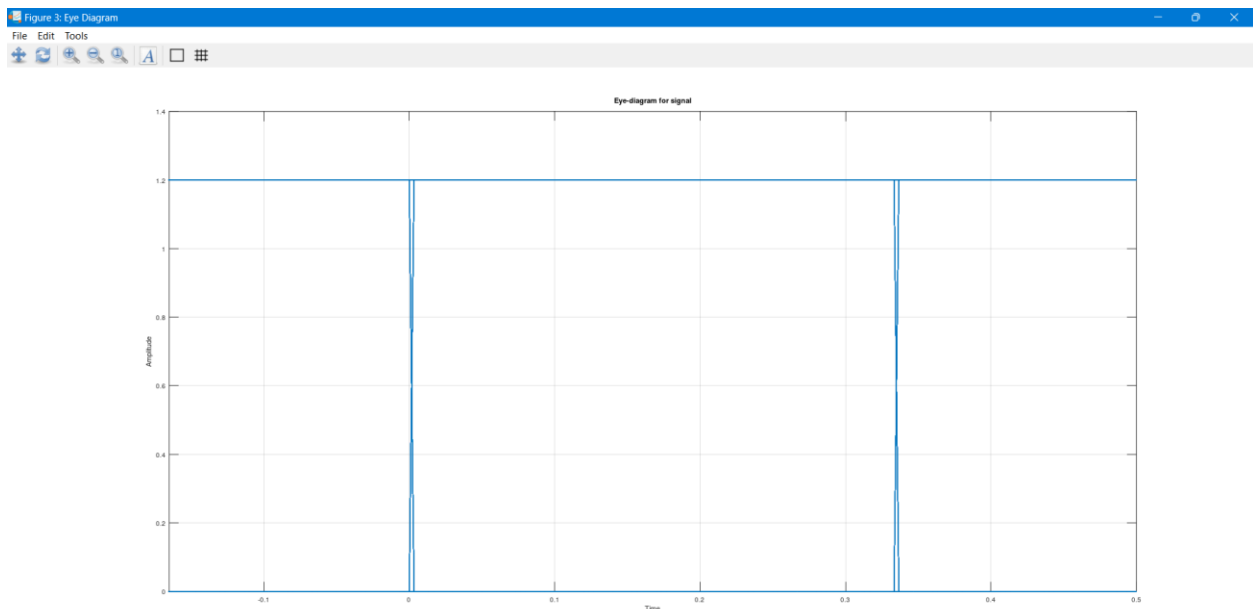


Figure 3: Unipolar NRZ Eye diagram

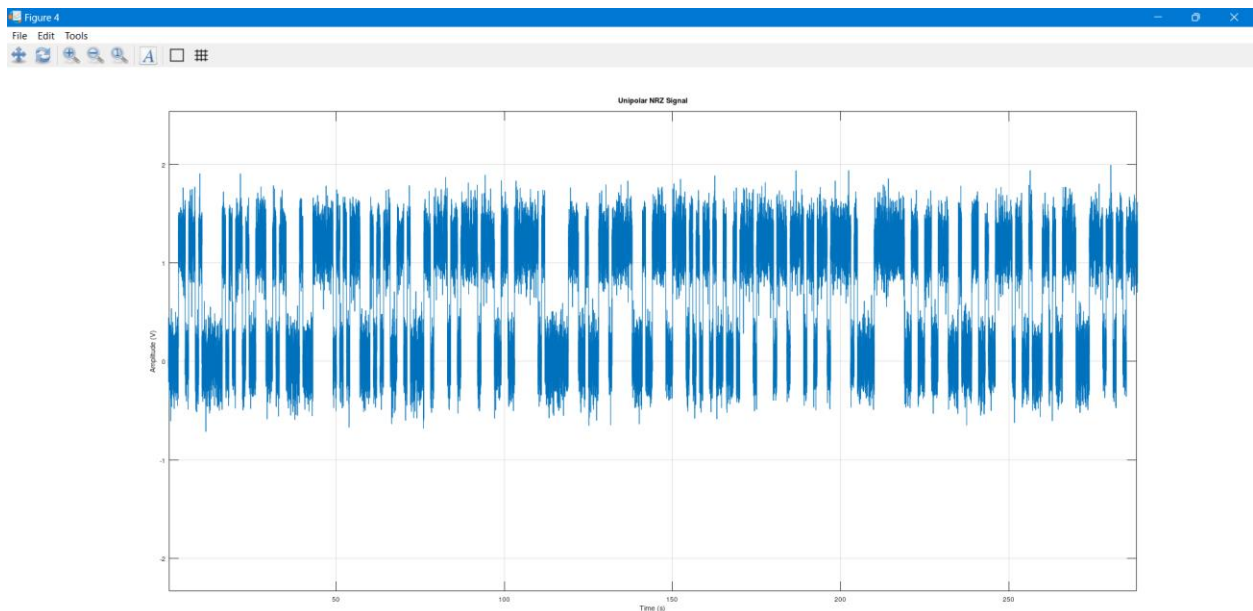


Figure 4: Unipolar NRZ signal "with noise" vs time

## 6.3 POLAR NRZ

### 6.3.1 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);
```

### 6.3.2 GRAPHS

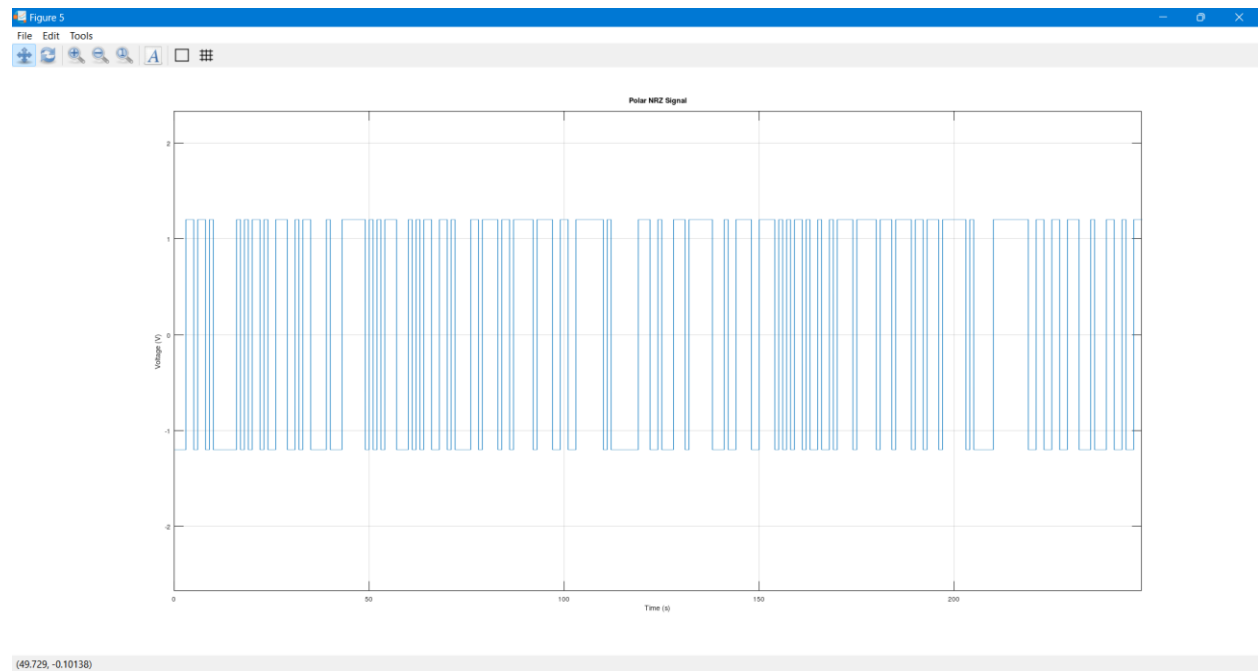


Figure 5: Polar NRZ signal vs Time

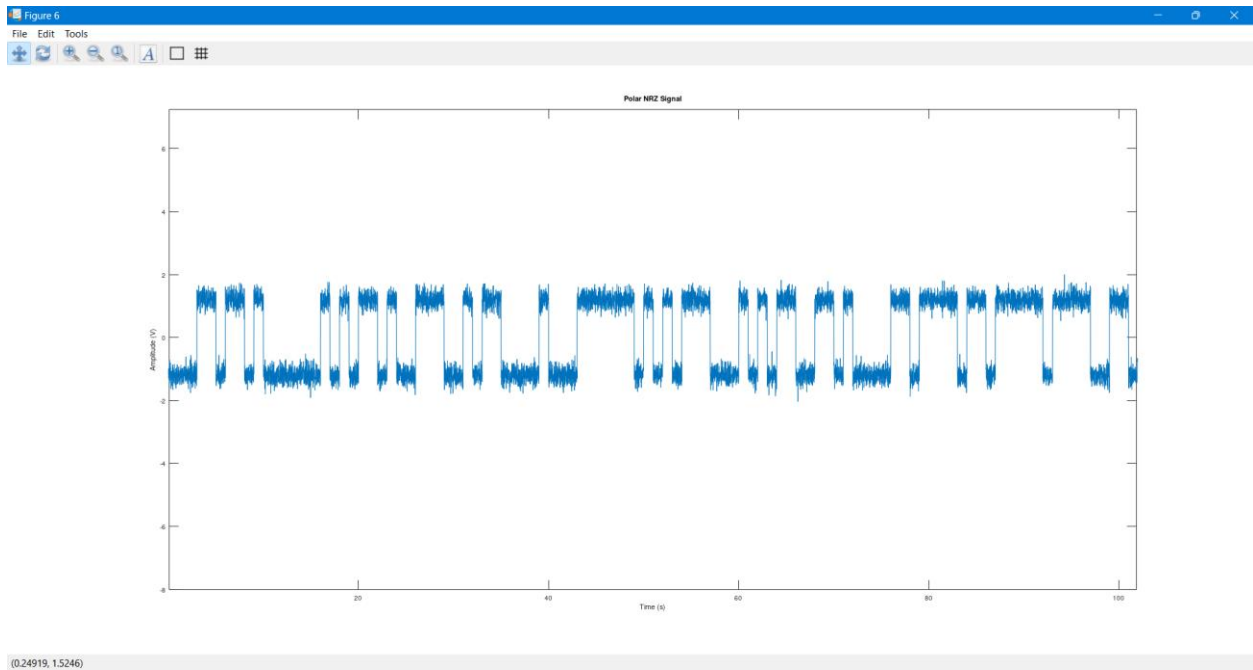


Figure 6: Polar NRZ signal "with noise" vs Time

## 6.4 UNIPOLAR RZ

### 6.4.1 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);
```

### 6.4.2 GRAPHS

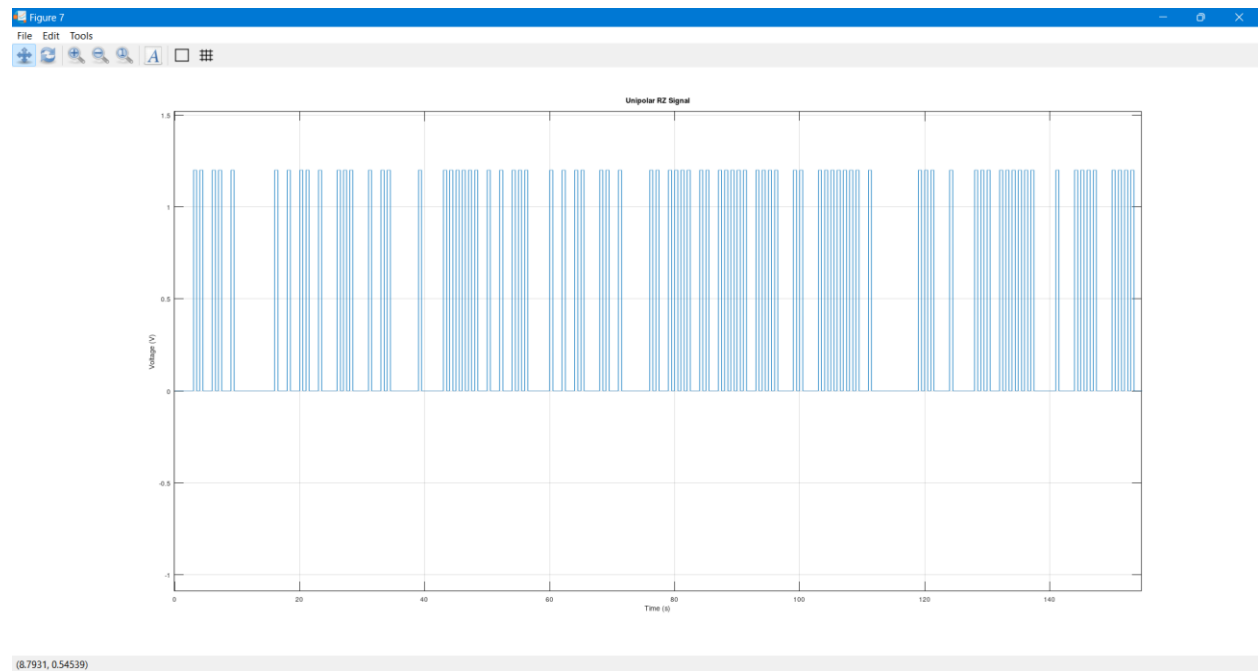


Figure 7: Unipolar RZ signal vs Time

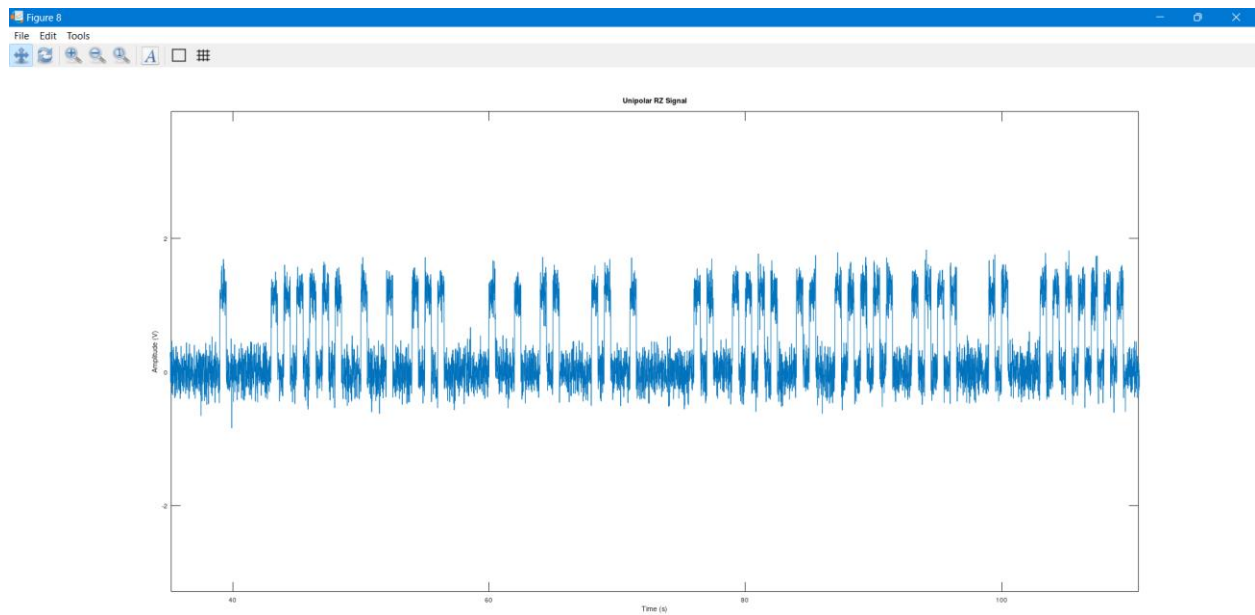


Figure 8: Unipolar RZ signal “with noise” vs Time

## 6.5 BIPOLAR RZ

### 6.5.1 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);
```

### 6.5.2 GRAPHS

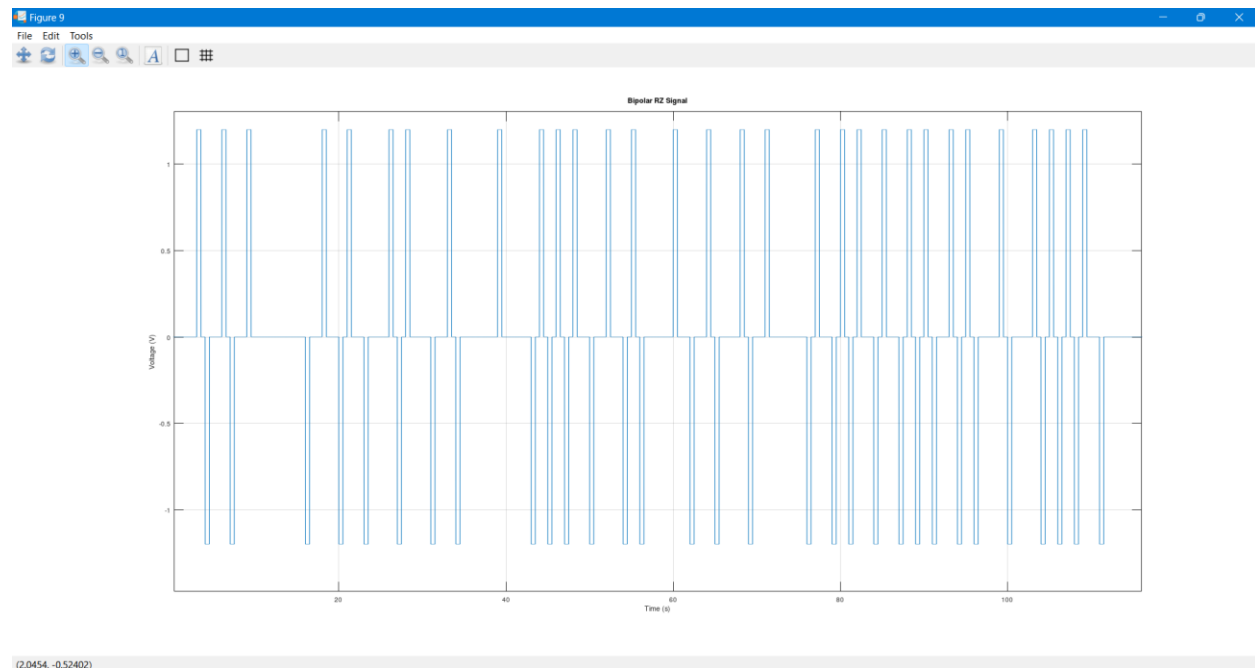


Figure 9: Bipolar RZ signal vs Time

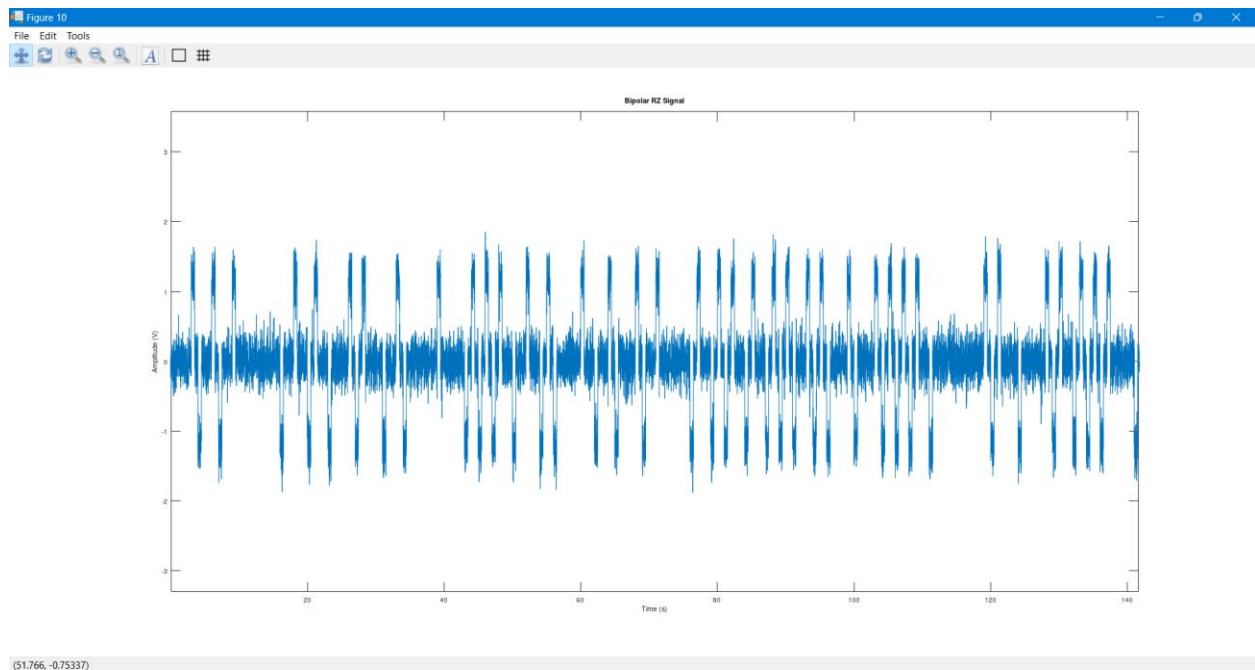


Figure 10: Bipolar RZ signal “with noise” vs Time

## 6.6 MANCHESTER LINE CODING

### 6.6.1 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);
```

#### 6.6.1.1 GRAPHS

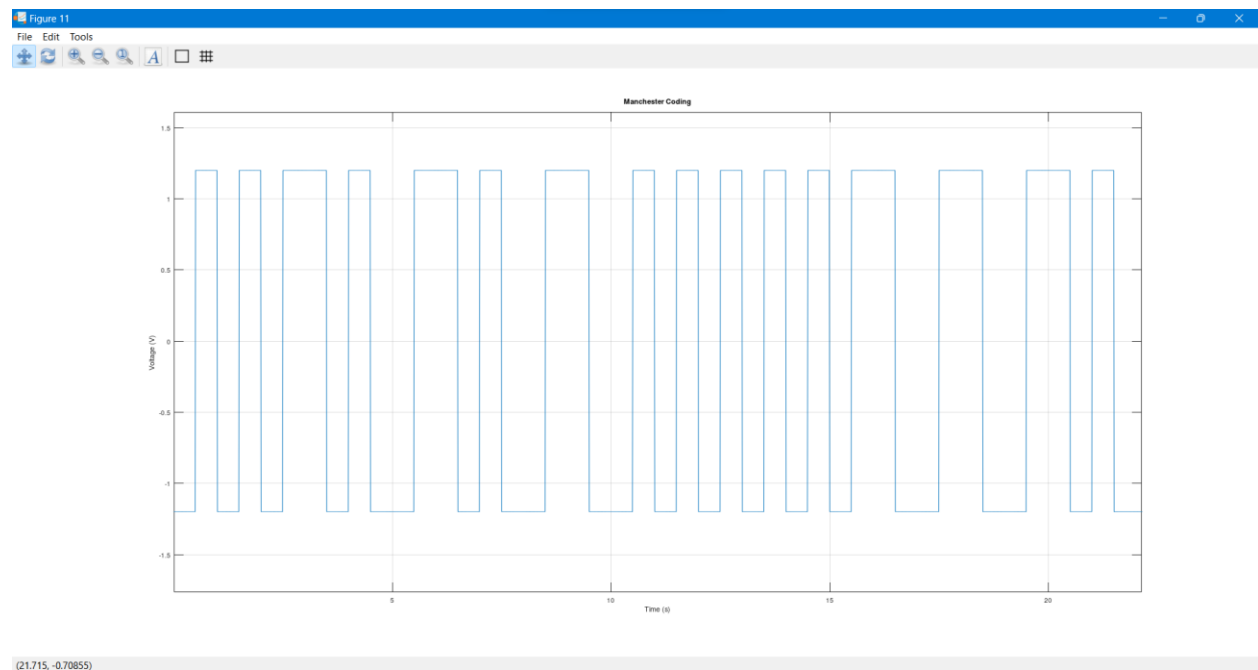


Figure 11: Manchester coding



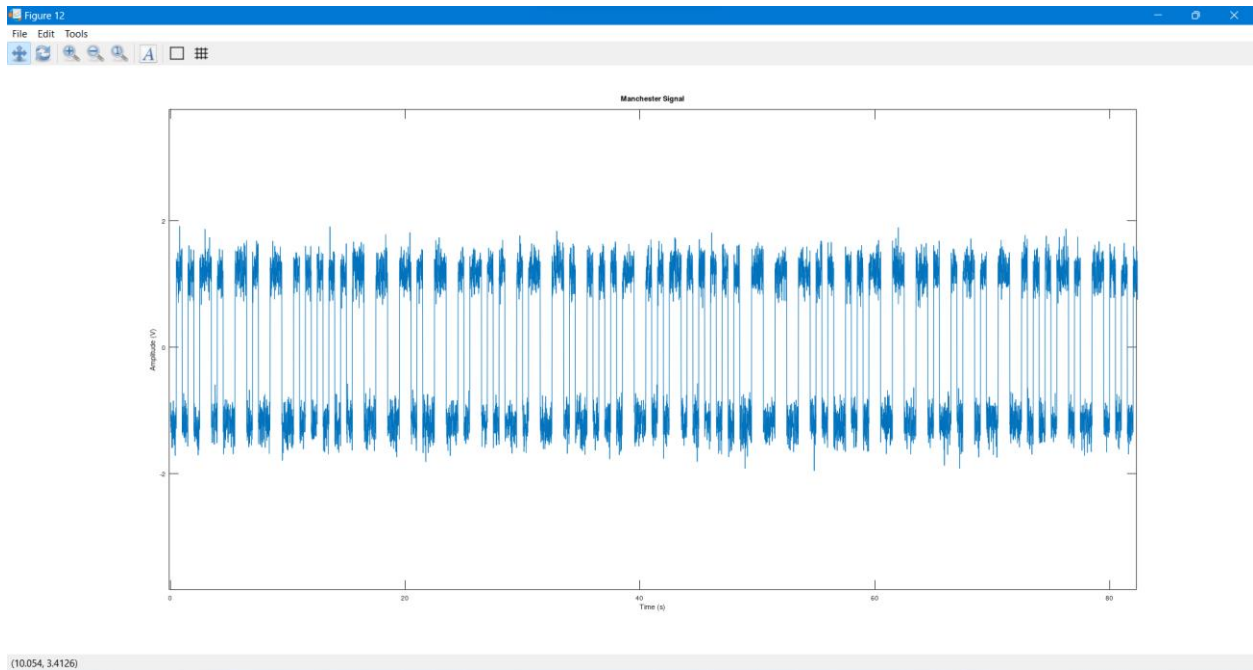


Figure 12: Manchester Signal “with noise”

## 6.7 PLOT BER\_VALUES OF DIFFERENT LINE CODING WITH SIGMA VALUES

### 6.7.1 CODE

```
% 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');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                               END
```

### 6.7.2 GRAPH

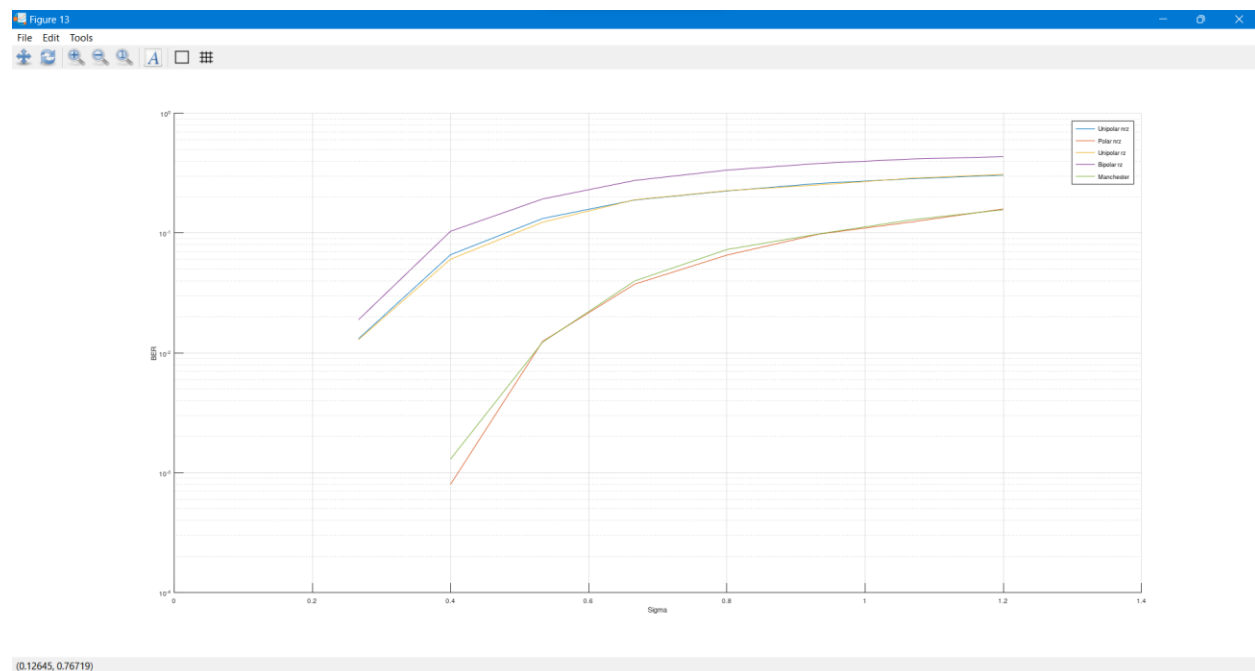


Figure 13: ber\_values of different line coding with sigma values

Figure 13

File Edit Tools

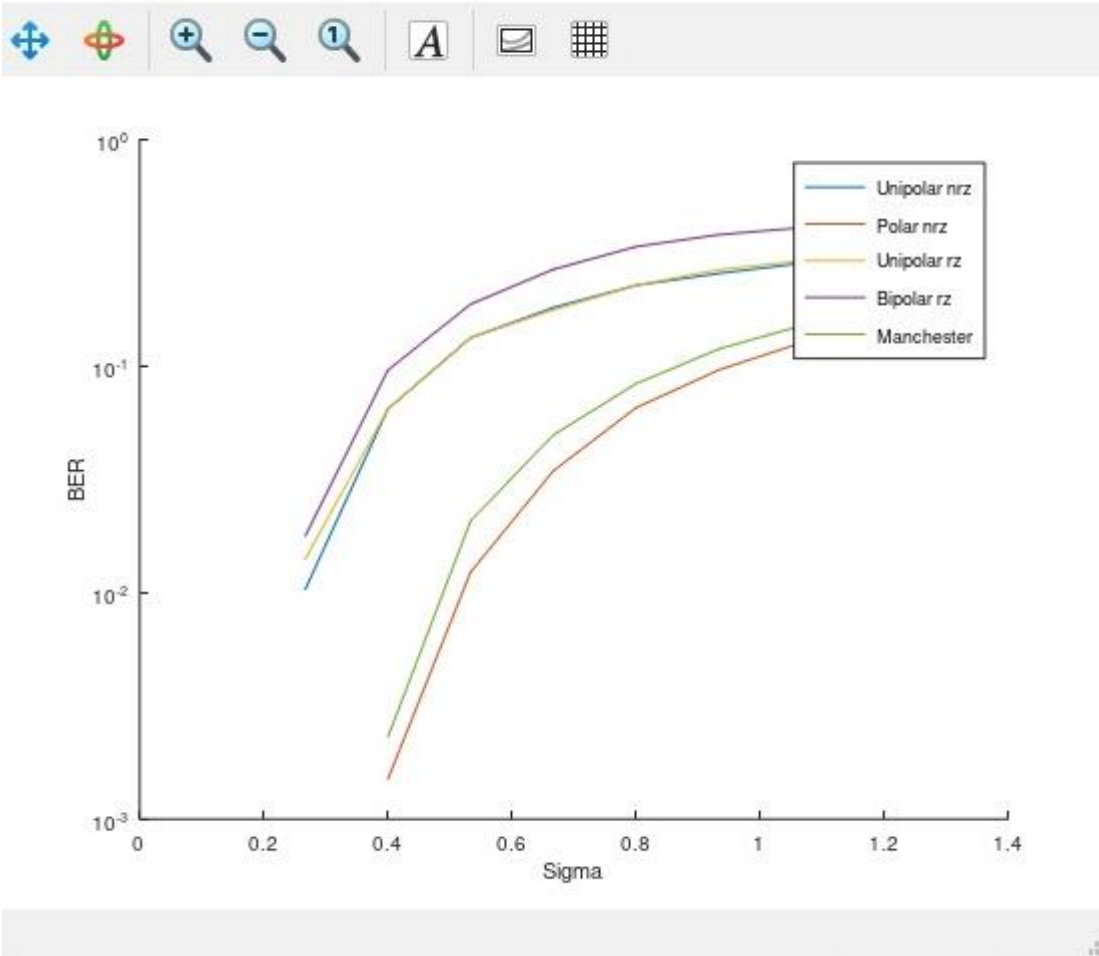


Figure 14: ber\_values of different line coding with sigma values

### Note

We add small noise to we can see the difference between Polar and Manchester and we find that Polar is better than Manchester code.

## 6.8 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_____

```

## 6.9 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

```
number_of_error_in_recieved_noise_bipolar_rz =  
  
    18    385   1046
```

## 6.10 WORKSPACE

Workspace			✕
Filter	<input type="text"/>		▼
Name	Class	Dimension	
ber_signal_1	double	1x1	
ber_signal_2	double	1x1	
ber_signal_3	double	1x1	
ber_signal_4	double	1x1	
ber_signal_5	double	1x1	
ber_values_sign...	double	1x10	
ber_values_sign...	double	1x10	
ber_values_sign...	double	1x10	
ber_values_sign...	double	1x10	
ber_values_sign...	double	1x10	
bits	double	1x10000	
noisy_signal_1	double	1x1000000	
noisy_signal_2	double	1x1000000	
noisy_signal_3	double	1x1000000	
noisy_signal_4	double	1x1000000	
noisy_signal_5	double	1x1000000	
number_of_erro...	double	1x1	
number_of_erro...	double	1x1	
number_of_erro...	double	1x1	
number_of_erro...	double	1x1	
number_of_erro...	double	1x1	
recieved_bits_1	double	1x10000	
recieved_bits_2	double	1x10000	
recieved_bits_3	double	1x10000	
recieved_bits_4	double	1x10000	
recieved_bits_5	double	1x10000	
samples per bit	double	1x1	

samples_per_bit	double	1x1
sigma	double	1x10
signal_1	double	1x1000000
signal_2	double	1x1000000
signal_3	double	1x1000000
signal_4	double	1x1000000
signal_5	double	1x1000000
t	double	1x1000000



## 6.11 FOCUS ON SOME VARIABLES

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

number_of_error_signal_2 [1x1 double]										
	1	2	3	4	5	6	7	8	9	10
1	0									

number_of_errors_signal_1 [1x1 double]										
	1	2	3	4	5	6	7	8	9	10
1	0									

number_of_errors_signal_3 [1x1 double]										
	1	2	3	4	5	6	7	8	9	10
1	0									

number_of_errors_signal_4 [1x1 double]										
	1	2	3	4	5	6	7	8	9	10
1	0									

number_of_errors_signal_5 [1x1 double]										
	1	2	3	4	5	6	7	8	9	10
1	0									

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

ber_signal_1 [1x1 double]										
	1	2	3	4	5	6	7	8	9	10
1	0									

ber_signal_2 [1x1 double]										
	1	2	3	4	5	6	7	8	9	10
1	0									

ber_signal_3 [1x1 double]										
	1	2	3	4	5	6	7	8	9	10
1	0									

ber_signal_4 [1x1 double]										
	1	2	3	4	5	6	7	8	9	10
1	0									

ber_signal_5 [1x1 double]										
	1	2	3	4	5	6	7	8	9	10
1	0									

### 6.11.3 BER\_VALUES VS SIGMA FOR DIFFERENT VARIABLES

ber_values_signal_1 [1x10 double]										
	1	2	3	4	5	6	7	8	9	10
1	0	0	0.0131	0.066	0.1324	0.1888	0.2247	0.26	0.2842	0.3048
ber_values_signal_2 [1x10 double]										
	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0.0008	0.0125	0.0375	0.0655	0.0981	0.1237	0.1591
ber_values_signal_3 [1x10 double]										
	1	2	3	4	5	6	7	8	9	10
1	0	0	0.0129	0.0606	0.1233	0.1898	0.226	0.2527	0.2871	0.3086
ber_values_signal_4 [1x10 double]										
	1	2	3	4	5	6	7	8	9	10
1	0	0	0.0189	0.1036	0.1927	0.2751	0.3355	0.3817	0.4153	0.4344
ber_values_signal_5 [1x10 double]										
	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0.0013	0.0123	0.0399	0.0729	0.0985	0.129	0.1567
sigma [1x10 double]										
	1	2	3	4	5	6	7	8	9	10
1	0	0.13333	0.26667	0.4	0.53333	0.66667	0.8	0.93333	1.0667	1.2

### 6.11.4 NOISY SIGNALS AT SIGMA = 0.2

noisy_signal_1 [1x1000000 double]										
	1	2	3	4	5	6	7	8	9	10
1	0.027389	-0.059649	0.098302	0.26347	0.4275	0.38301	-0.20991	-0.019333	-0.093349	0.44459
noisy_signal_2 [1x1000000 double]										
	1	2	3	4	5	6	7	8	9	10
1	-1.1482	-1.0119	-1.0938	-1.2902	-0.97993	-1.2628	-1.35	-1.0582	-0.74484	-1.2184
noisy_signal_3 [1x1000000 double]										
	1	2	3	4	5	6	7	8	9	10
1	0.033174	-0.071495	0.036823	-0.21217	0.17235	0.41279	0.11892	-0.13583	-0.26153	0.13656
noisy_signal_4 [1x1000000 double]										
	1	2	3	4	5	6	7	8	9	10
1	0.04879	-0.23499	0.055683	-0.097021	0.10509	0.15468	-0.19983	-0.064471	-0.11901	-0.4043
noisy_signal_5 [1x1000000 double]										
	1	2	3	4	5	6	7	8	9	10
1	-1.1719	-1.1461	-0.87383	-1.1525	-1.269	-1.2008	-1.354	-1.0619	-1.2341	-1.1285

### 6.12 CONCLUSIONS

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

## 7 PART II

---

### 7.1 THE USED FUNCTIONS

- I. 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.
- II. 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
```

- III. 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
```

- IV. calculate\_ber(tx\_bits,rx\_bits): This function takes a stream of transmitted & received bits, compares between them then calculates the bit error rate from the formula:  $BER = \frac{\text{no. of error bits}}{\text{Total no. of bits}}$  **“repeated”**

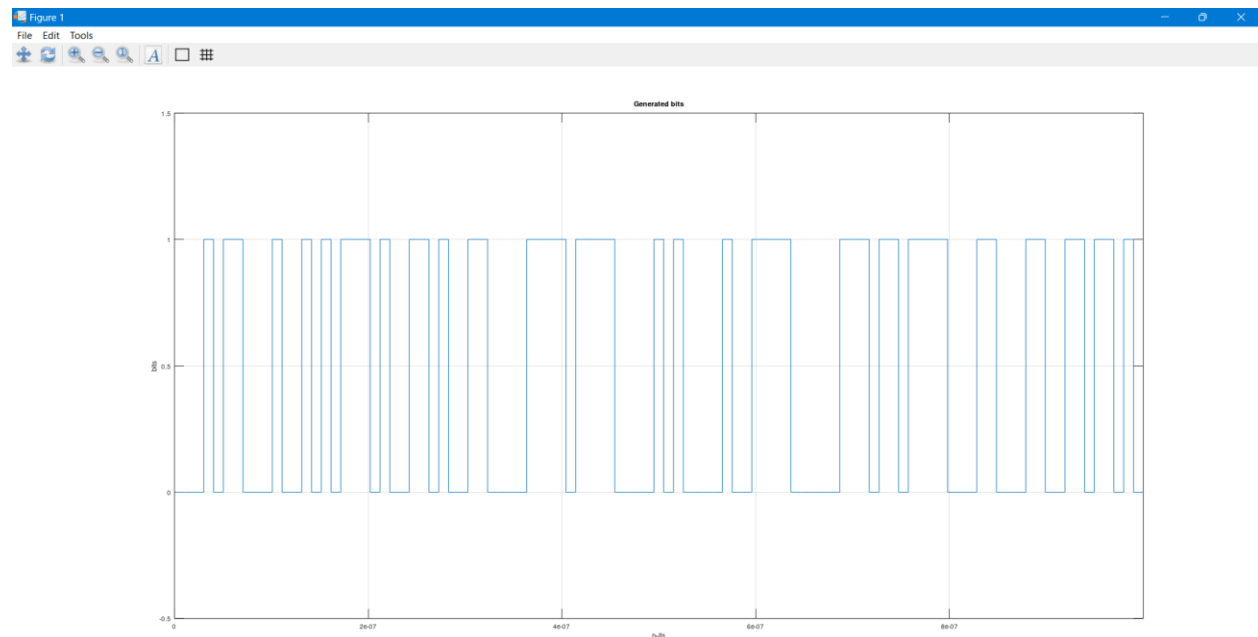
## 7.2 PART II TRANSMITTER

**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'.)

### 7.2.1.1 CODE

```
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);
title("Generated bits");
xlabel("n_bits");
ylabel("bits");
axis([0 t_bits(end) -0.5 1.5]);
```

### 7.2.1.2 GRAPH



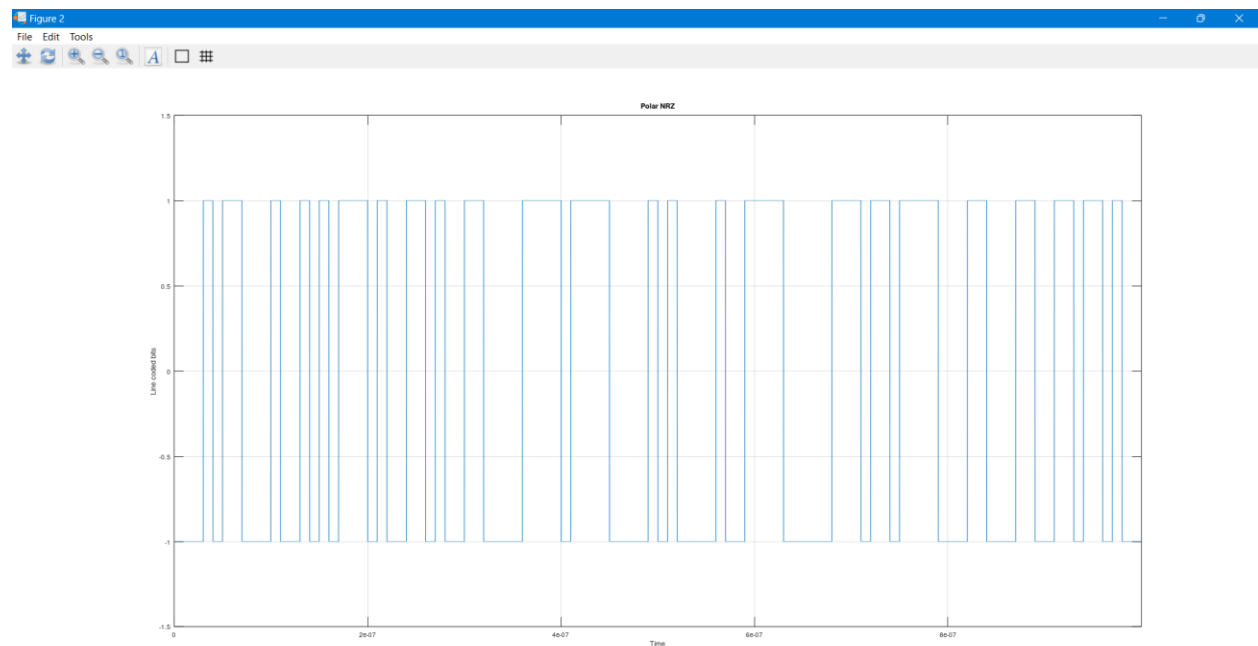
## 7.2.2 Line code the stream of bits (pulse shape) according to Polar non return to zero (Maximum voltage +1, Minimum voltage -1).

### 7.2.2.1 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]);
title("Polar NRZ");
xlabel("Time");
ylabel("Line coded bits");
```

### 7.2.2.2 GRAPH



## 7.2.3 Plot the spectral domains.

### 7.2.3.1 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)))/N);
line_coded_power_spectrum = abs((fftshift(fft(line_coded_bits))).^2/N);
```

```

figure
plot(f, line_coded_spectrum);
title("Polar NRZ spectrum");
xlabel("Frequency");
ylabel("line_coded_spectrum");
figure
plot(f, line_coded_power_spectrum);
title("Polar NRZ power spectrum");
xlabel("Frequency");
ylabel("line_coded_power_spectrum");

```

### 7.2.3.2 GRAPH

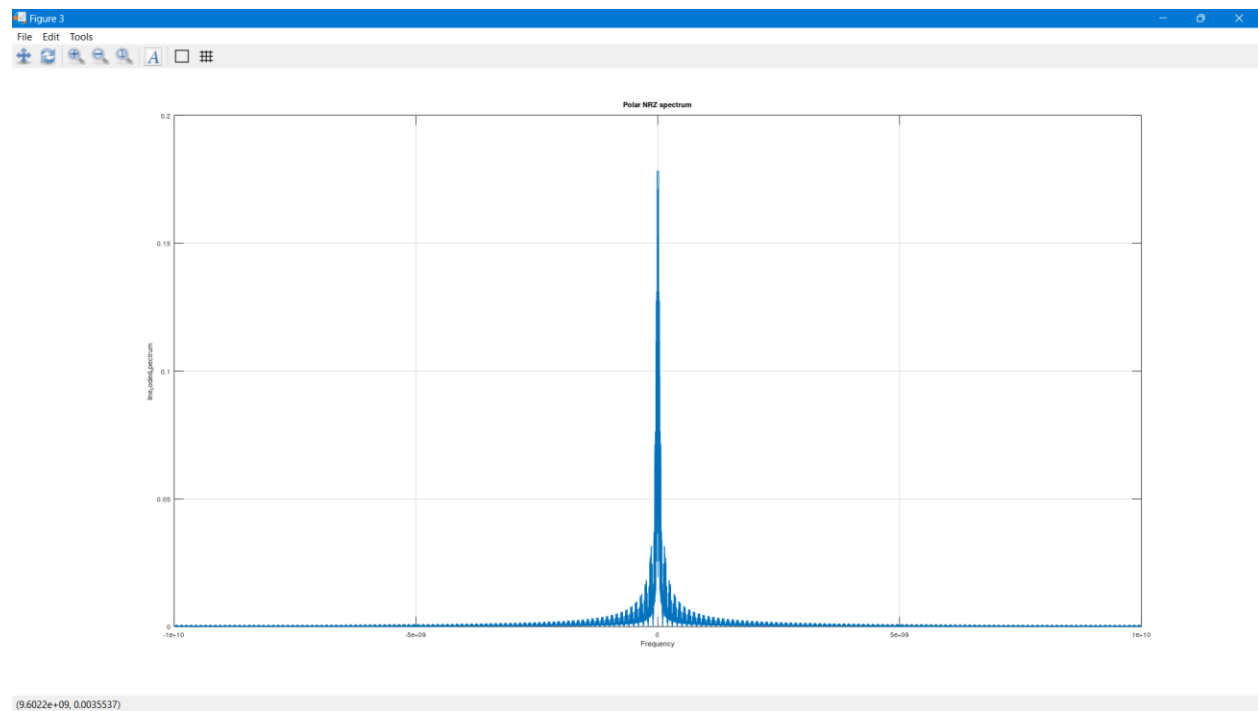


Figure: Polar NRZ spectrum

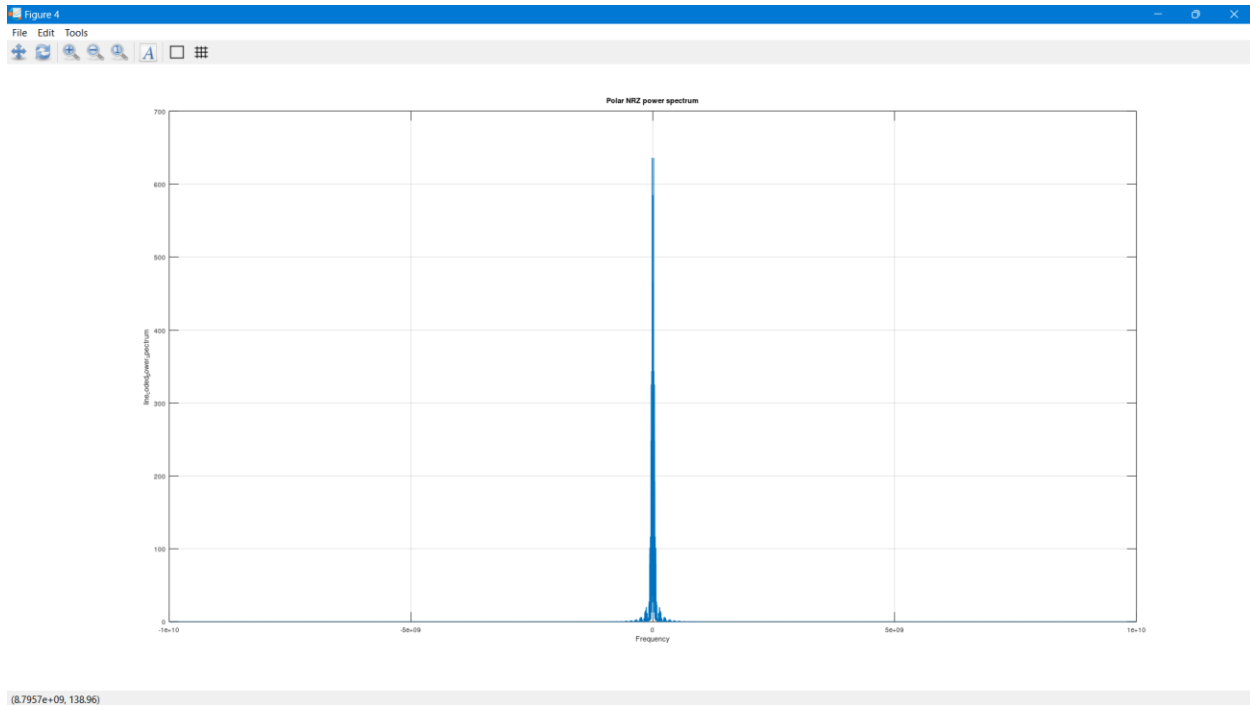


Figure: Polar NRZ PSD

## 7.2.4 Plot the time domain of the modulated BPSK signal ( $f_c = 1GHz$ )

### 7.2.4.1 CODE

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 4) BPSK modulation time domain %%%%%%%%%%
% carrier is a sine wave should be multiplied by sqrt(2/Tb) "basis"
carrier = sqrt(2/Tb)*sin(2*pi*fc*time);

BPSK_modulated_signal = line_coded_bits.*carrier; % modulating the signal
% plotting BPSK_modulated_signal
figure
plot(time, BPSK_modulated_signal);
axis([0 10/fc -15000 15000]);
title("BPSK modulated signal");
xlabel("Time");
ylabel("BPSK_modulated_signal");

```

### 7.2.4.2 GRAPH

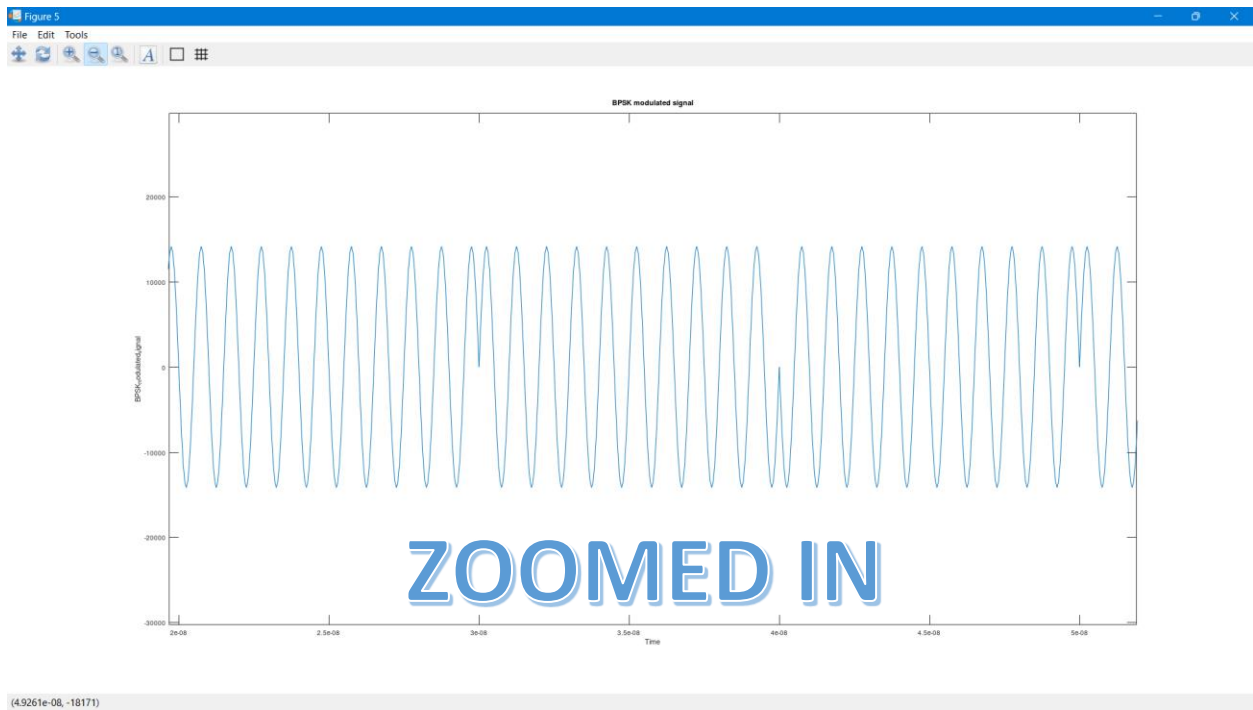


Figure: BPSK modulated signal

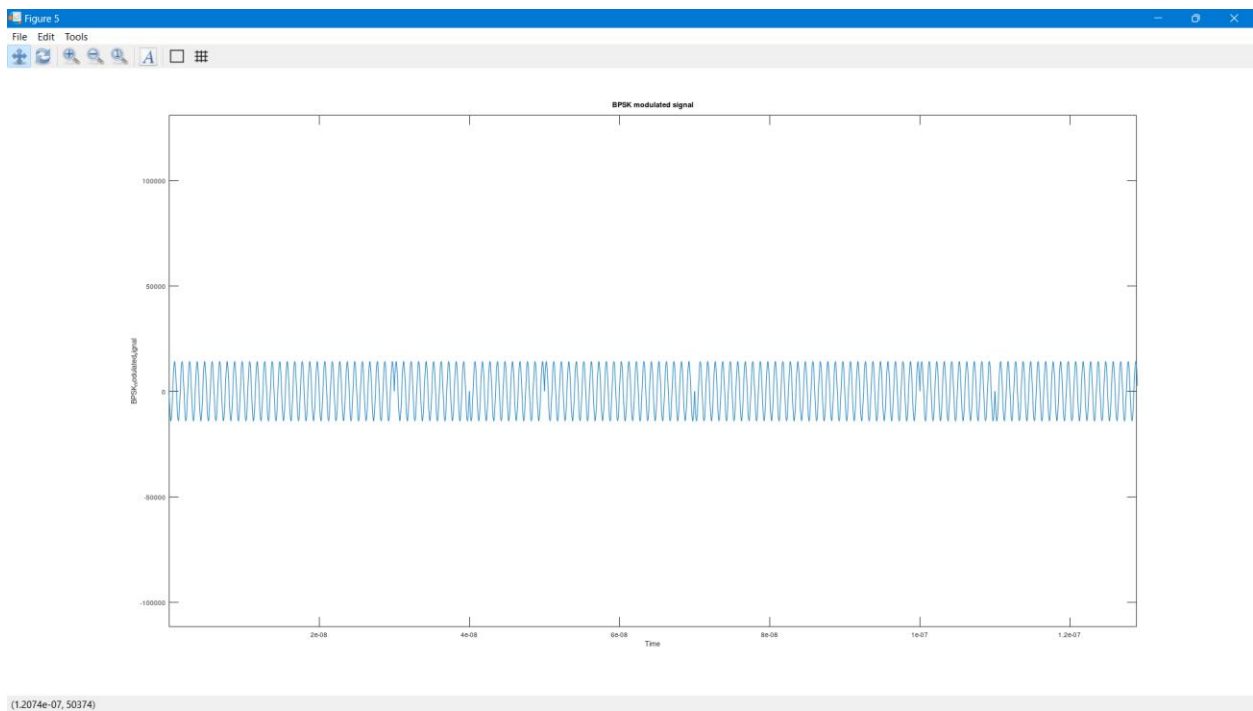


Figure: BPSK modulated signal



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

### 7.2.5.1 CODE

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 5) Plotting BPSK spectrum %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
BPSK_spectrum = abs(fftshift(fft(BPSK_modulated_signal)));
figure
plot(f, BPSK_spectrum);
title("BPSK Modulated signal spectrum");
xlabel("frequency");
ylabel("BPSK_spectrum");
```

### 7.2.5.2 GRAPH

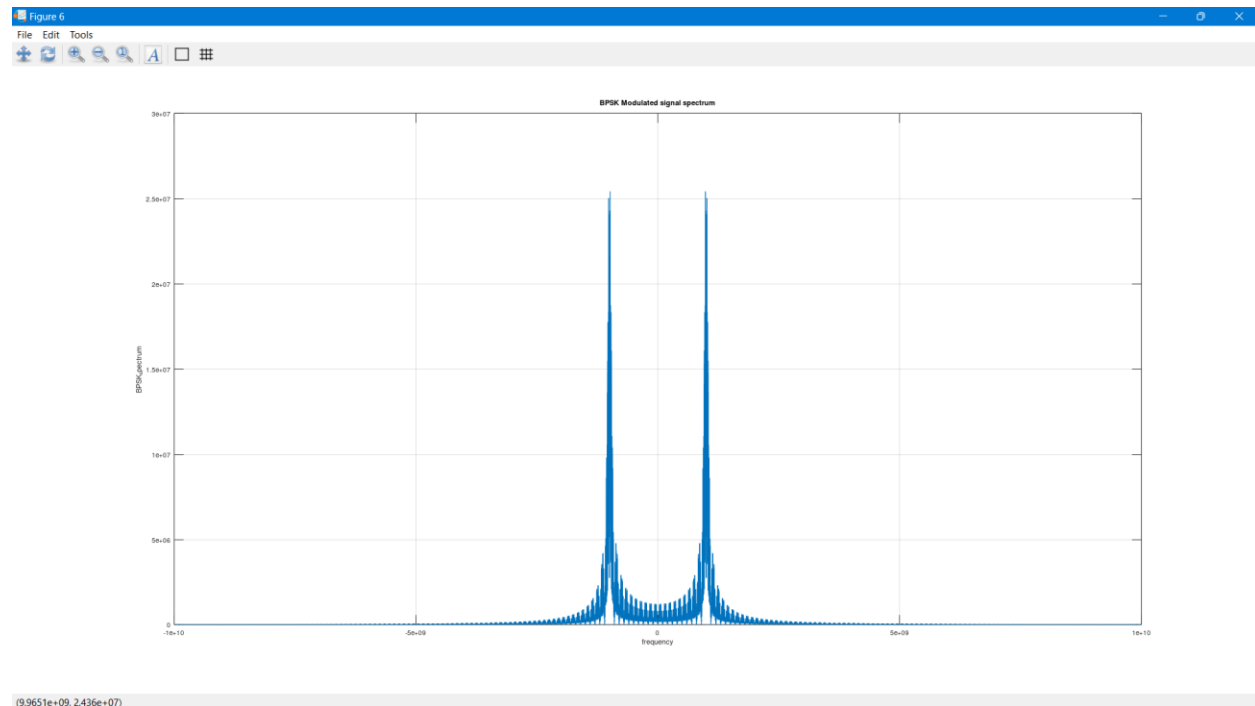


Figure: BPSK modulated signal spectrum

## 7.3 PART II RECEIVER

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

### 7.3.1.1 CODE

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Part II Receiver %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 6) BPSK demodulation %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
BPSK_demodulated_signal = BPSK_modulated_signal.*carrier;

BPSK_demodulated_spectrum = fftshift(fft(BPSK_demodulated_signal))/Ns;
figure
```

```

plot(f, abs(BPSK_demodulated_spectrum));
title("BPSK_demodulated_spectrum");
xlabel("frequency");
ylabel("BPSK_demodulated_spectrum");

BPSK_demodulated_spectrum_power =
abs(fftshift(fft(BPSK_demodulated_signal)).^2/Ns);
figure
plot(f, BPSK_demodulated_spectrum_power);
title("BPSK_demodulated_spectrum_power");
xlabel("frequency");
ylabel("BPSK_demodulated_spectrum_power");

LPF = abs(f)<Rb;
BPSK_LPF = BPSK_demodulated_spectrum .* LPF;
figure
plot(f, abs(BPSK_LPF));
title("BPSK after LPF spectral");
xlabel("frequency");
ylabel("BPSK after LPF spectral");

BPSK_LPF_time = real(ifft(ifftshift(BPSK_LPF))*Ns);
figure
plot(f, BPSK_LPF_time);
title("BPSK after LPF time domain");
xlabel("time");
ylabel("BPSK LPF signal");

reconstructed_bits = decision(BPSK_LPF_time);
figure
plot(time, reconstructed_bits);
title("reconstructed bits after LPF and demodulation");
ylabel("reconstructed_bits");
xlabel("time");

```

### 7.3.1.2 GRAPH

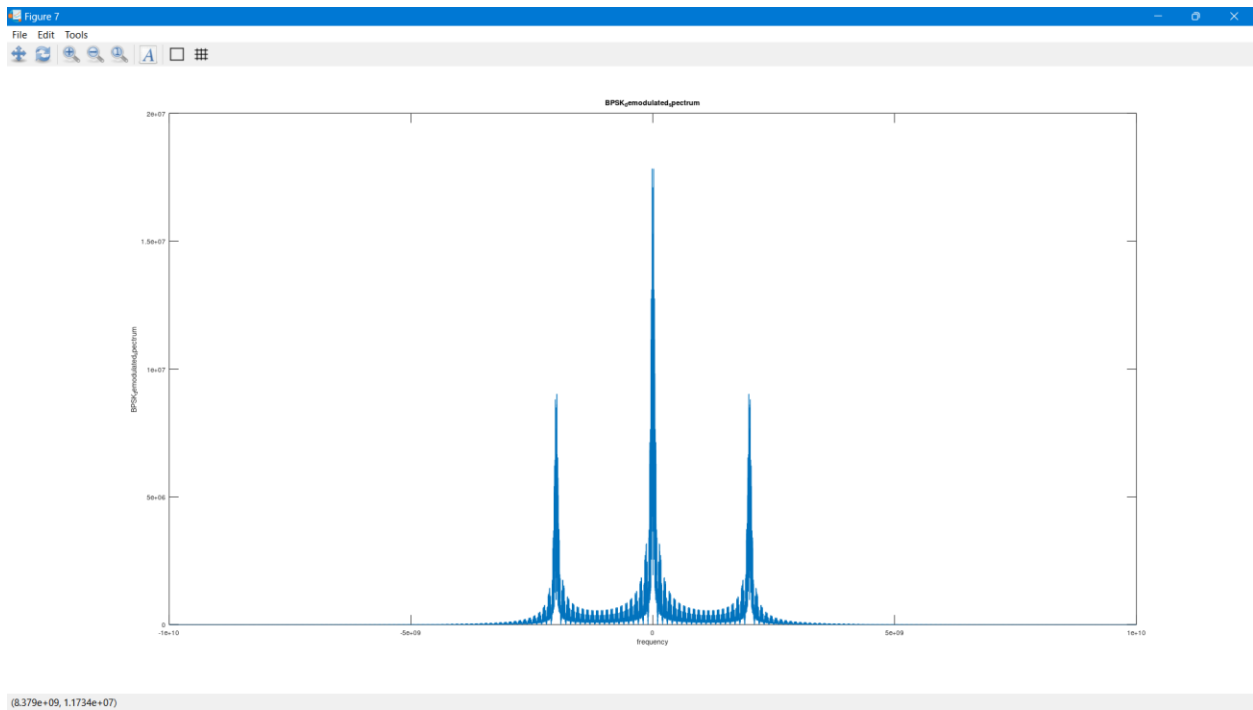


Figure: BPSK demodulated signal spectrum

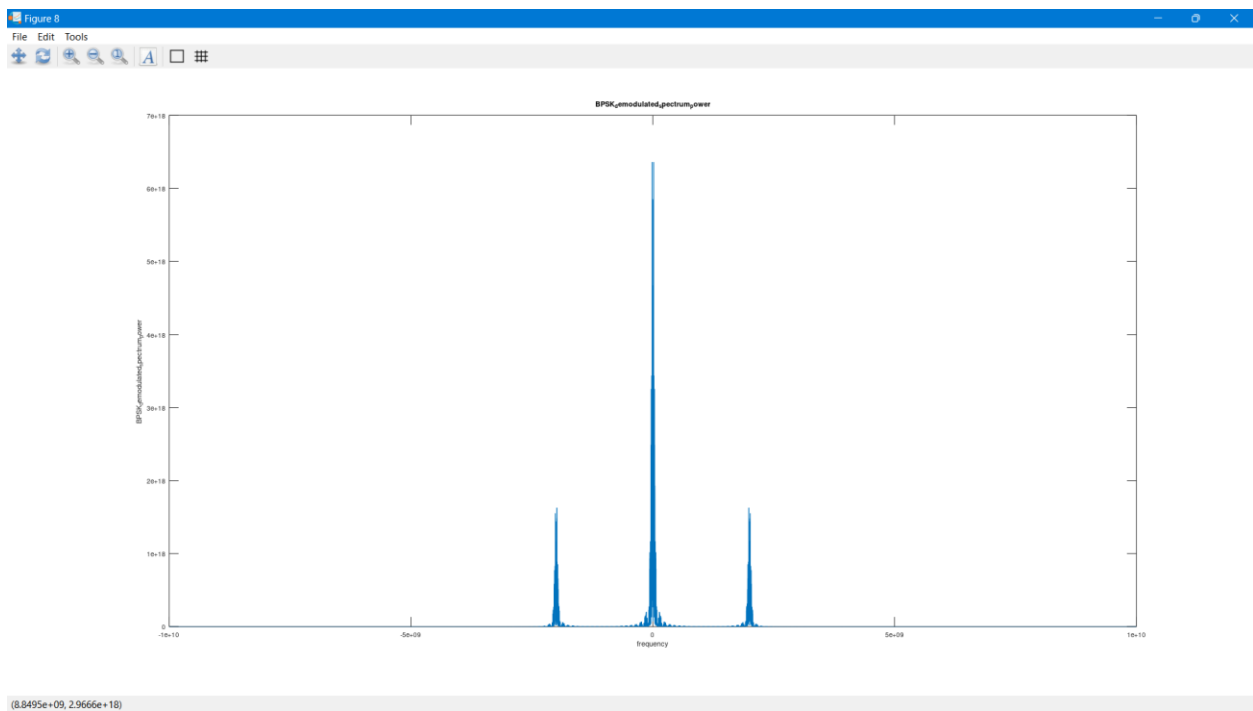


Figure: BPSK demodulated signal PSD

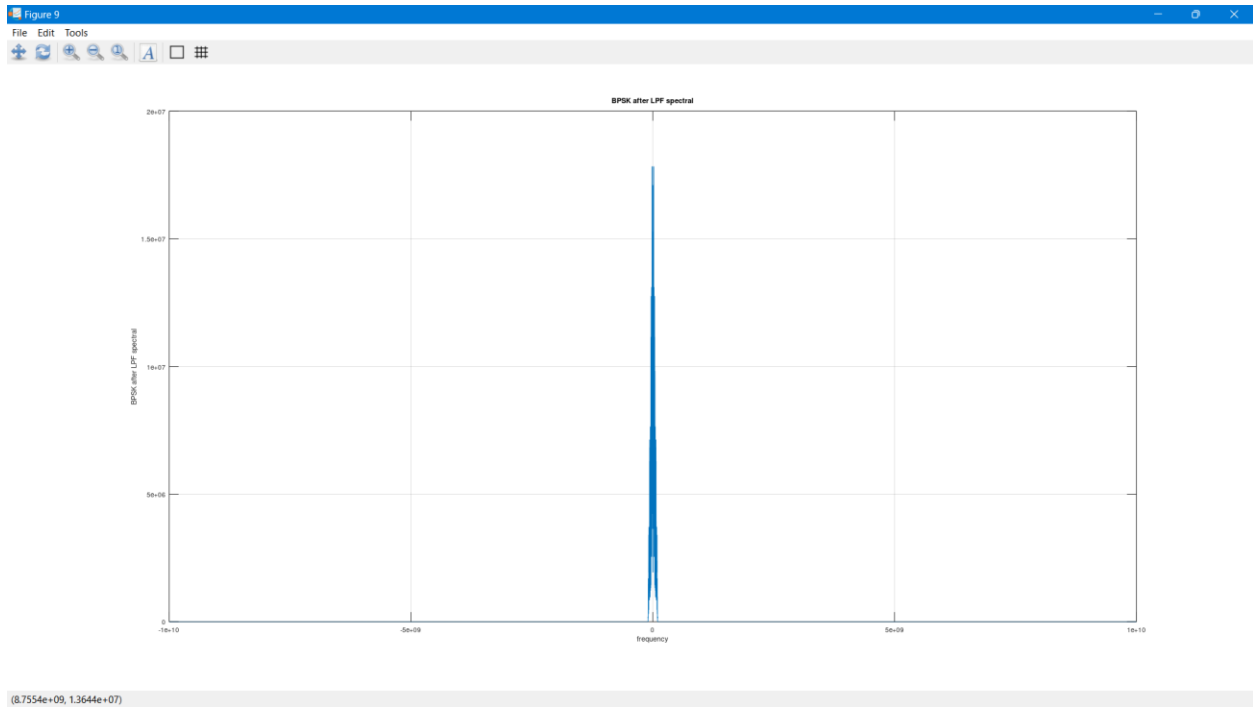


Figure: BPSK demodulated signal PSD after LPF

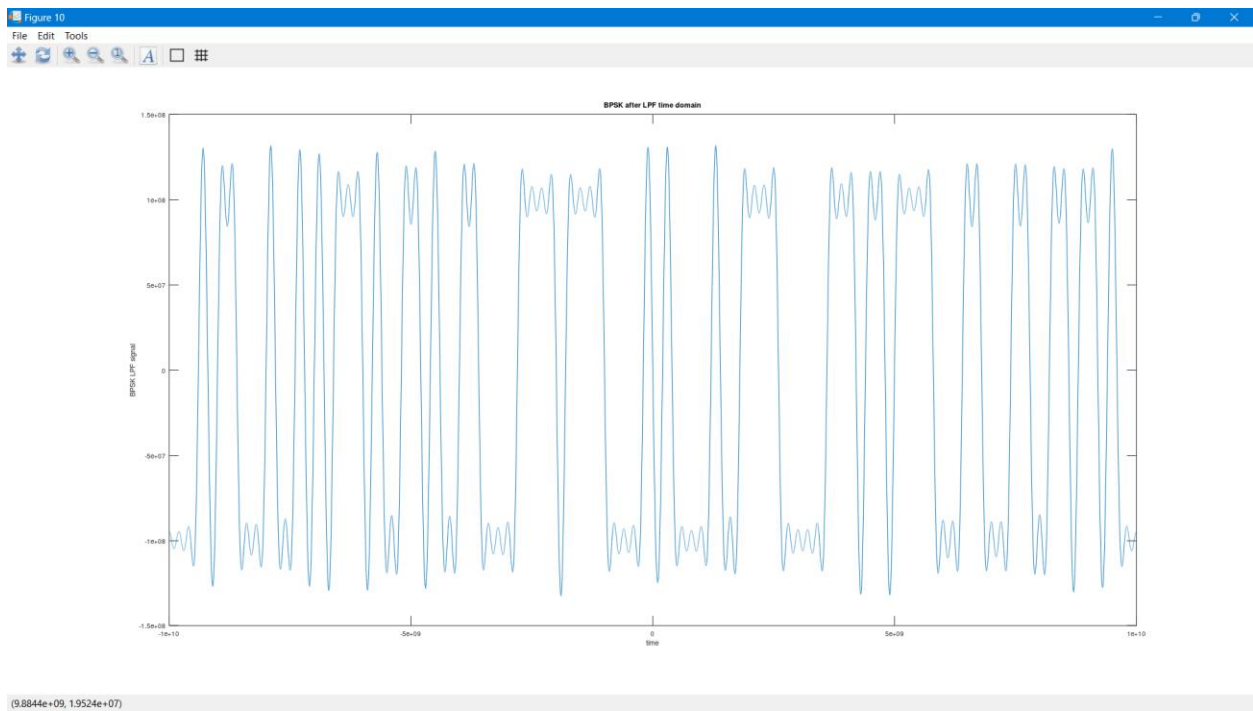


Figure: BPSK demodulated signal time domain after LPF

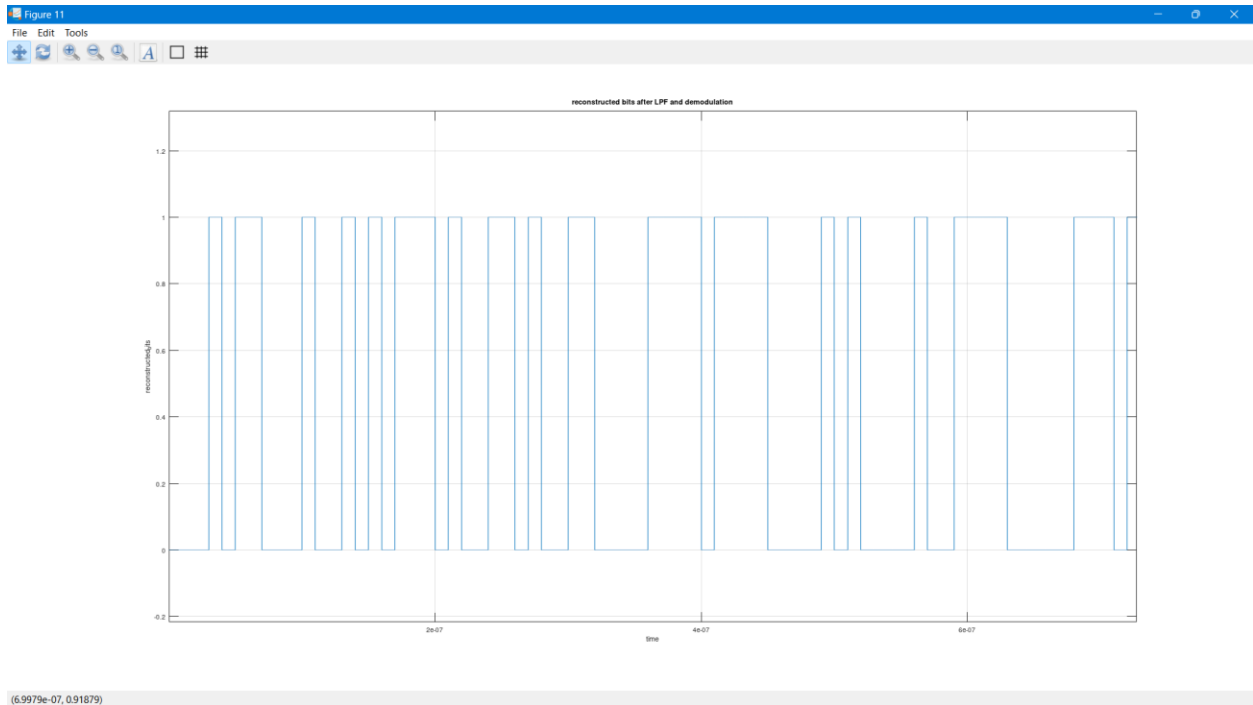


Figure: BPSK demodulated signal time domain after decision

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.

#### 7.3.2.1 CODE

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 7) Calculate BER %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
transmitted_bits = line_code(rand_bits, 1,0);
BER = calculate_ber(transmitted_bits, reconstructed_bits)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% End Of Main %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

#### 7.3.2.2 Result

```
>> main

BER = 0
```

### 7.4 PART II FULL CODE

```
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);
title("Generated bits");
xlabel("n_bits");
ylabel("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]);
title("Polar NRZ");
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)))/N);
line_coded_power_spectrum = abs((fftshift(fft(line_coded_bits))).^2/N);
figure
plot(f, line_coded_spectrum);
title("Polar NRZ spectrum");
xlabel("Frequency");
ylabel("line_coded_spectrum");
figure
plot(f, line_coded_power_spectrum);
title("Polar NRZ power spectrum");
xlabel("Frequency");
ylabel("line_coded_power_spectrum");
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 4) BPSK modulation time domain %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% carrier is a sine wave should be multiplied by sqrt(2/Tb)
carrier = sqrt(2/Tb)*sin(2*pi*fc*time);

BPSK_modulated_signal = line_coded_bits.*carrier; % modulating the signal
% plotting BPSK_modulated_signal
figure

```

```

plot(time, BPSK_modulated_signal);
axis([0 10/fc -15000 15000]);
title("BPSK modulated signal");
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;

BPSK_demodulated_spectrum = fftshift(fft(BPSK_demodulated_signal))/Ns;
figure
plot(f, abs(BPSK_demodulated_spectrum));
title("BPSK_demodulated_spectrum");
xlabel("frequency");
ylabel("BPSK_demodulated_spectrum");

BPSK_demodulated_spectrum_power =
abs(fftshift(fft(BPSK_demodulated_signal)).^2/Ns);
figure
plot(f, BPSK_demodulated_spectrum_power);
title("BPSK_demodulated_spectrum_power");
xlabel("frequency");
ylabel("BPSK_demodulated_spectrum_power");

LPF = abs(f)<Rb;
BPSK_LPF = BPSK_demodulated_spectrum .* LPF;
figure
plot(f, abs(BPSK_LPF));
title("BPSK after LPF spectral");
xlabel("frequency");
ylabel("BPSK after LPF spectral");

BPSK_LPF_time = real(ifft(ifftshift(BPSK_LPF))*Ns);
figure
plot(f, BPSK_LPF_time);
title("BPSK after LPF time domain");
xlabel("time");
ylabel("BPSK LPF signal");

reconstructed_bits = decision(BPSK_LPF_time);
figure
plot(time, reconstructed_bits);
title("reconstructed bits after LPF and demodulation");
ylabel("reconstructed_bits");
xlabel("time");
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 7) Calculate BER %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
transmitted_bits = line_code(rand_bits, 1,0);

```

```
BER = calculate_ber(transmitted_bits, reconstructed_bits)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% End Of Main %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

## 7.5 WORKSPACE

Workspace				
Filter <input type="checkbox"/>				
Name	Class	Dimension	Value	Attribute
BER	double	1x1	0	
BPSK_LPF	double	1x20000	[0 + 0i, 0 - 0i, 0 ...	complex
BPSK_LPF_time	double	1x20000	[-8.8565e+07, -...	
BPSK_demodul...	double	1x20000	[0, -1.9098e+07,...	
BPSK_demodul...	double	1x20000	[-3.5858e-08 + ...	complex
BPSK_demodul...	double	1x20000	[2.5716e-11, 6.2...	
BPSK_modulate...	double	1x20000	[0, -4370.2, -831...	
BPSK_spectrum	double	1x20000	[1.0539e-07, 47...	
LPF	logical	1x20000	[0, 0, 0, 0, 0, 0,...	
N	double	1x1	20000	
Ns	double	1x1	20000	
Rb	double	1x1	1.0000e+08	
Tb	double	1x1	1.0000e-08	
carrier	double	1x20000	[0, 4370.2, 8312....	
df	double	1x1	1000000	
f	double	1x20000	-1e+10:1e+06:9....	
fc	double	1x1	1.0000e+09	
fs	double	1x1	2.0000e+10	
line_coded_bits	double	1x20000	[-1, -1, -1, -1, -1,...	
line_coded_pow...	double	1x20000	[0, 2.2565e-06, ...	
line_coded_spe...	double	1x20000	[0, 1.0622e-05, ...	
numOfBits	double	1x1	100	
rand_bits	double	1x100	[0, 1, 1, 1, 1, 1, 0,...	
reconstructed_b...	double	1x20000	[0, 0, 0, 0, 0, 0, 0,...	
t_bits	double	1x100	[0, 1.0101e-08, ...	
time	double	1x20000	0:5e-11:9.9995e...	
transmitted_bits	double	1x20000	[0, 0, 0, 0, 0, 0, 0,...	
ts	double	1x1	5.0000e-11	



## Note:

We could also use a full signal integrator instead of LPF.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 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("reconstructed bits");
```

## Result:

