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

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 Abdl-Gawad Nassif	ECE
5	2001457	Youssef Foda Mohamed	EPM
6	2001023	Abdullah Yasser Ahmed	EPM
7	2000037	Ahmed Mohammed Baker 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 the 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.

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.

4 PART I TRANSMITTER

4.1 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

    % 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

% 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
```

4.2 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 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
```

4.3 THE MAIN FUNCTION

```
% 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]);
```

4.4 HERE IS A SUMMARY OF THE CODE

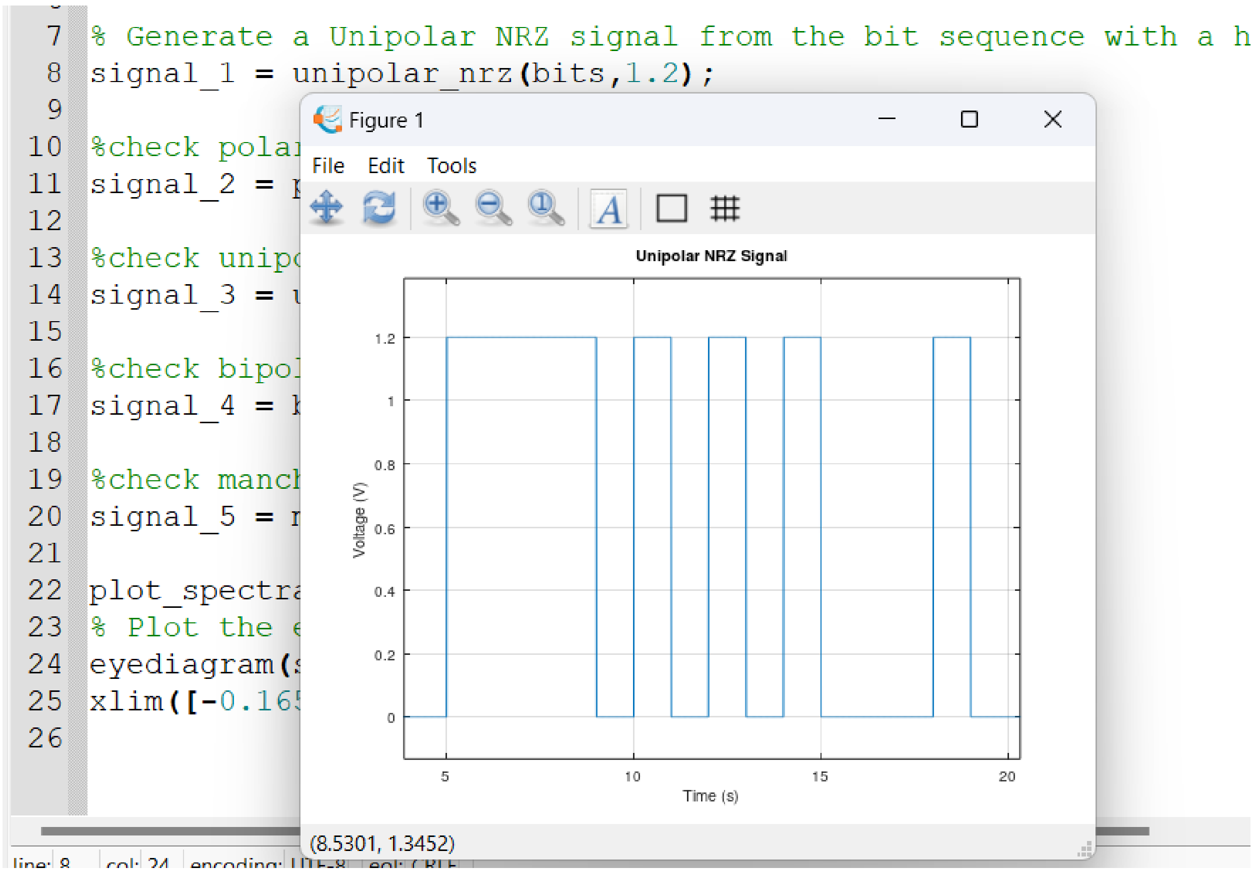
- The code loads the Communications Toolbox package in Octave, a numerical computing software. It then generates a sequence of 10000 random bits using the `generate_bits` function, which is not shown in the code snippet.
- Next, the code generates several different types of baseband digital signals from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

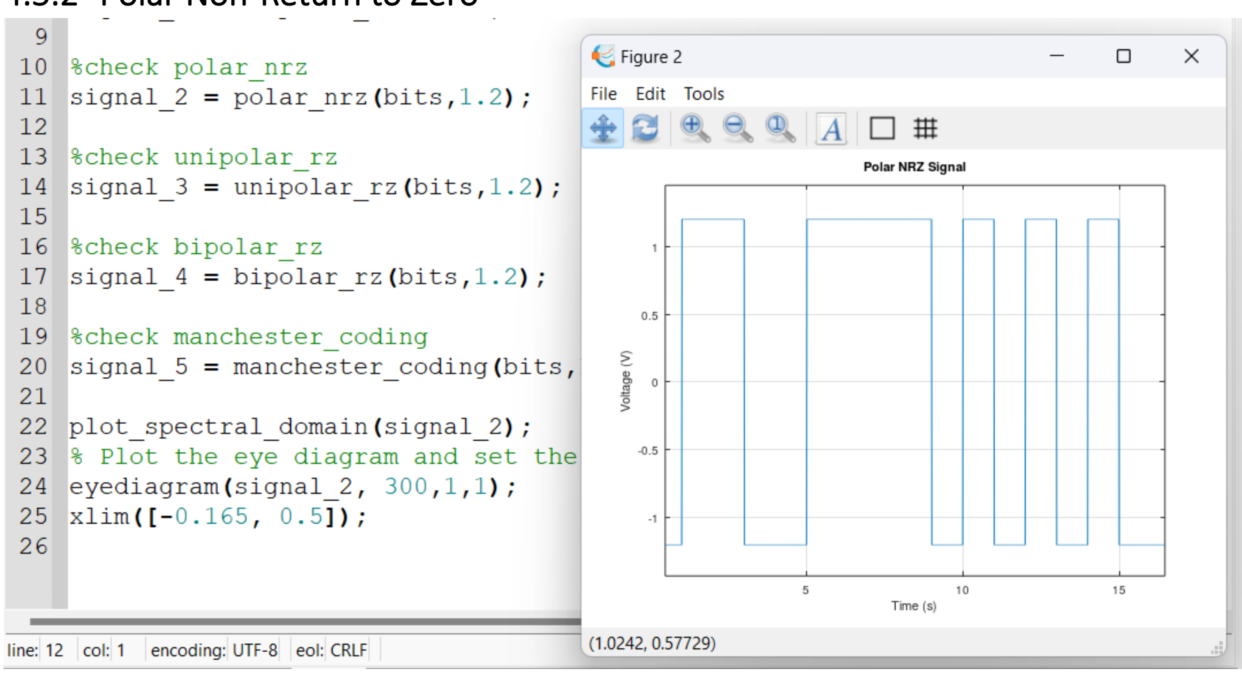
- Overall, the code demonstrates how to generate and analyze different types of digital baseband signals from a bit sequence, using different encoding techniques.

4.5 SNAPSHOTS

4.5.1 Unipolar Non-Return to Zero



4.5.2 Polar Non-Return to Zero

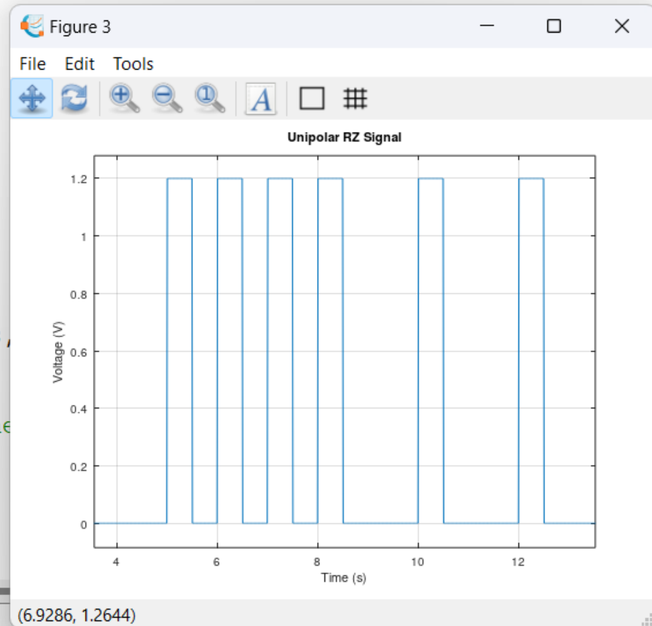


4.5.3 Unipolar Return to Zero

```
%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,

plot_spectral_domain(signal_2);
% Plot the eye diagram and set the
eyediagram(signal_2, 300,1,1);
xlim([-0.165, 0.5]);
```

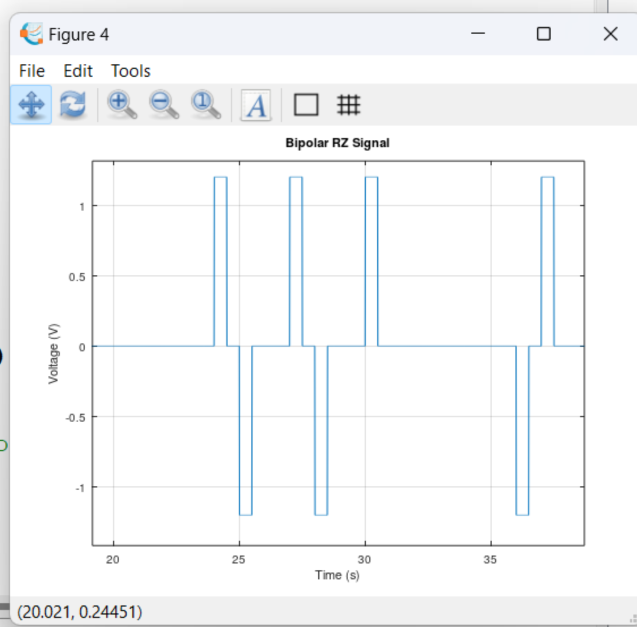
ine: 15 col: 1 encoding: UTF-8 eol: CRLF

Command Window Documentation Editor



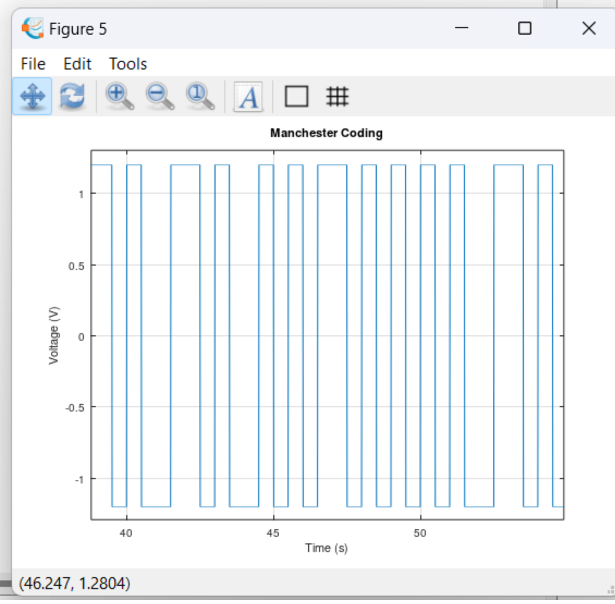
4.5.4 Bipolar Return To Zero

```
% Generate a Unipolar NRZ signal from the bit sequence with a high voltage lev
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 plo
eyediagram(signal_2, 300,1,1);
xlim([-0.165, 0.5]);
```



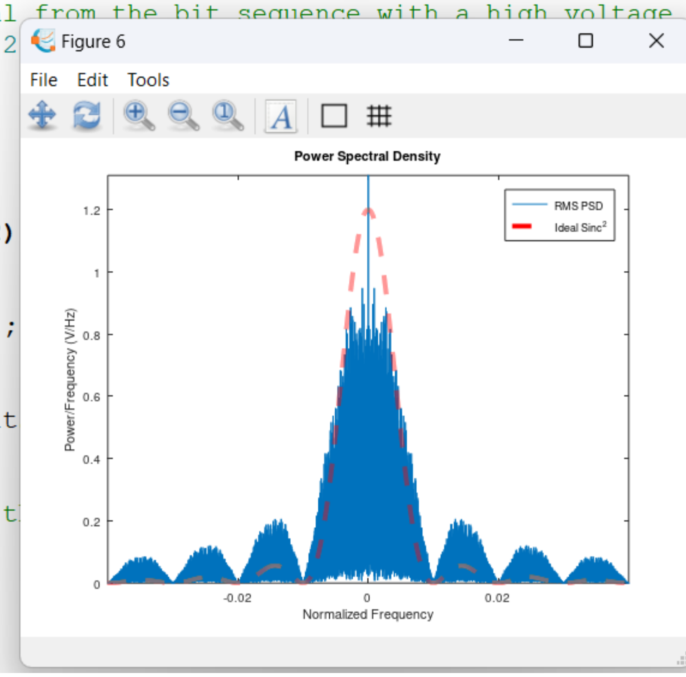
4.5.5 Manchester Coding

```
% Generate a Unipolar NRZ signal from the bit sequence with a high voltage lev
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
eyediagram(signal_2, 300,1,1);
xlim([-0.165, 0.5]);
```

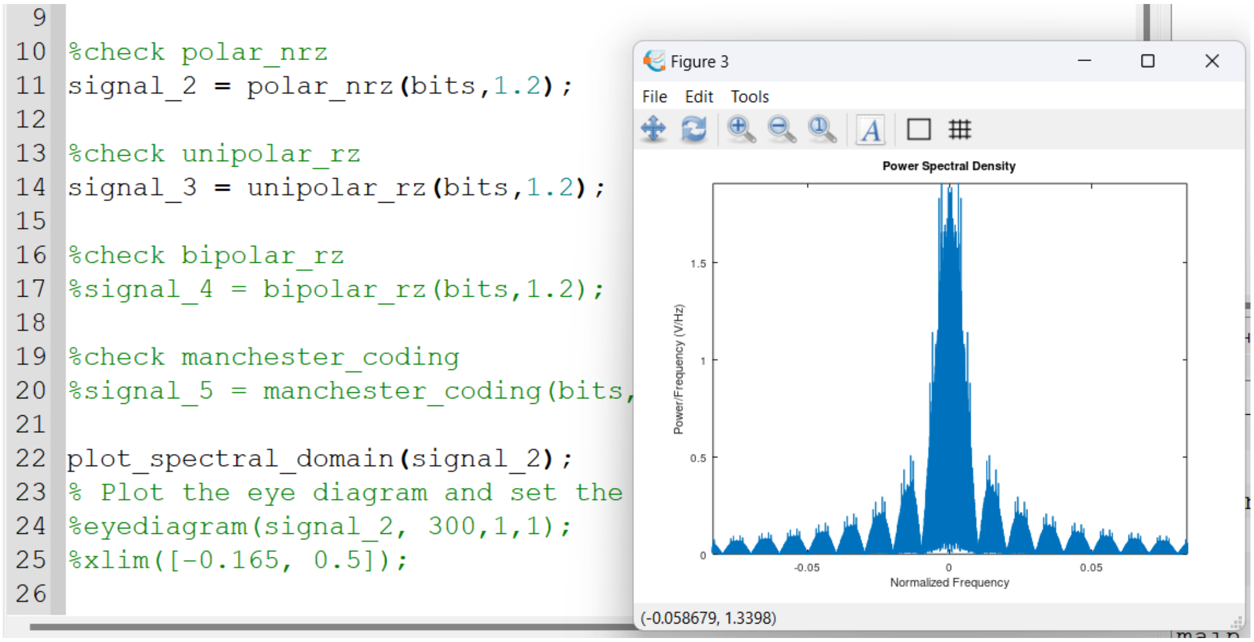


4.5.6 Spectral Domain of Unipolar Non-Return to Zero

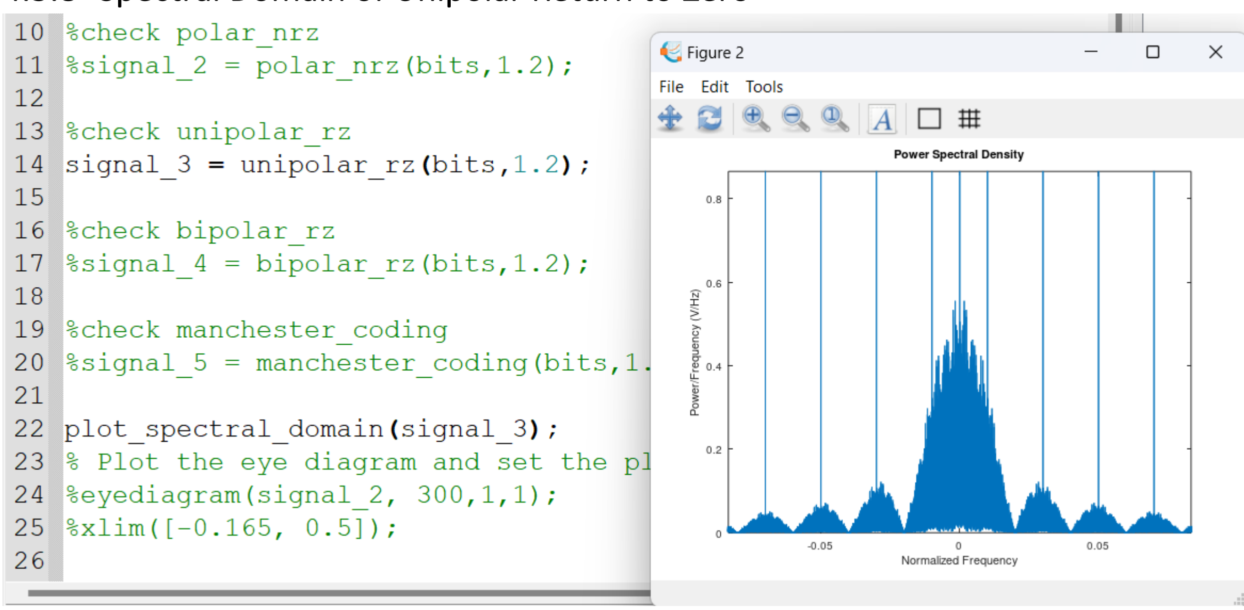
```
% Generate a Unipolar NRZ signal from the bit sequence with a high voltage l
signal_1 = unipolar_nrz(bits,1.2

%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(bit

plot_spectral_domain(signal_1);
% Plot the eye diagram and set t
eyediagram(signal_2, 300,1,1);
xlim([-0.165, 0.5]);
```



4.5.7 Spectral Domain of Polar Non-Return to Zero

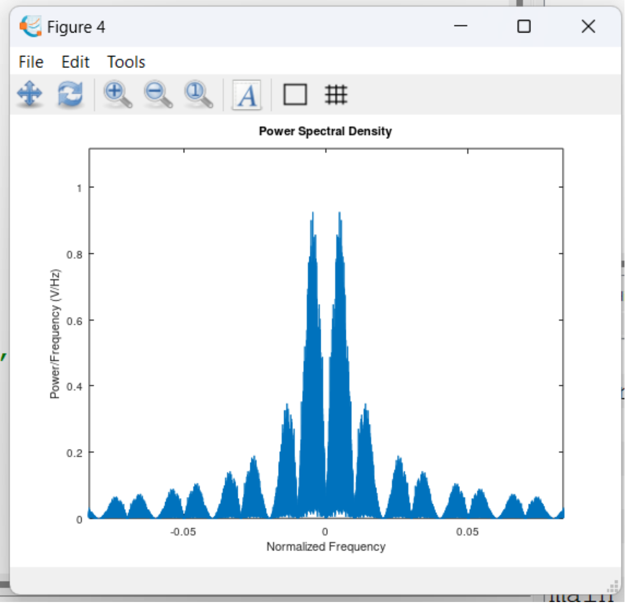


4.5.8 Spectral Domain of Unipolar Return to Zero



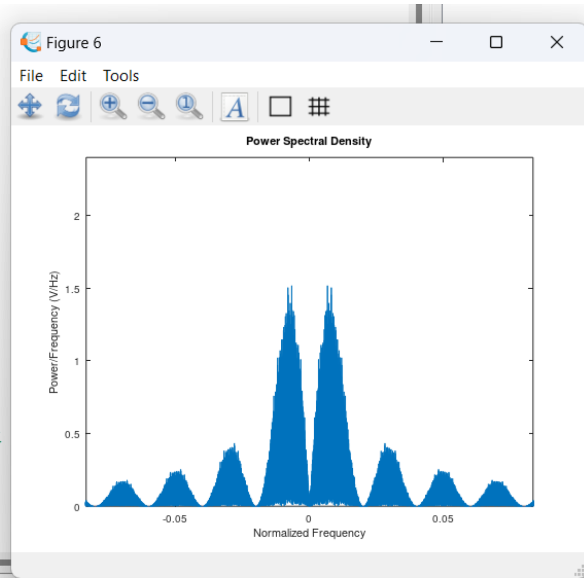
4.5.9 Spectral Domain of Bipolar Return to Zero

```
%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,

plot_spectral_domain(signal_4);
% Plot the eye diagram and set the
%eyediagram(signal_2, 300,1,1);
%xlim([-0.165, 0.5]);

```

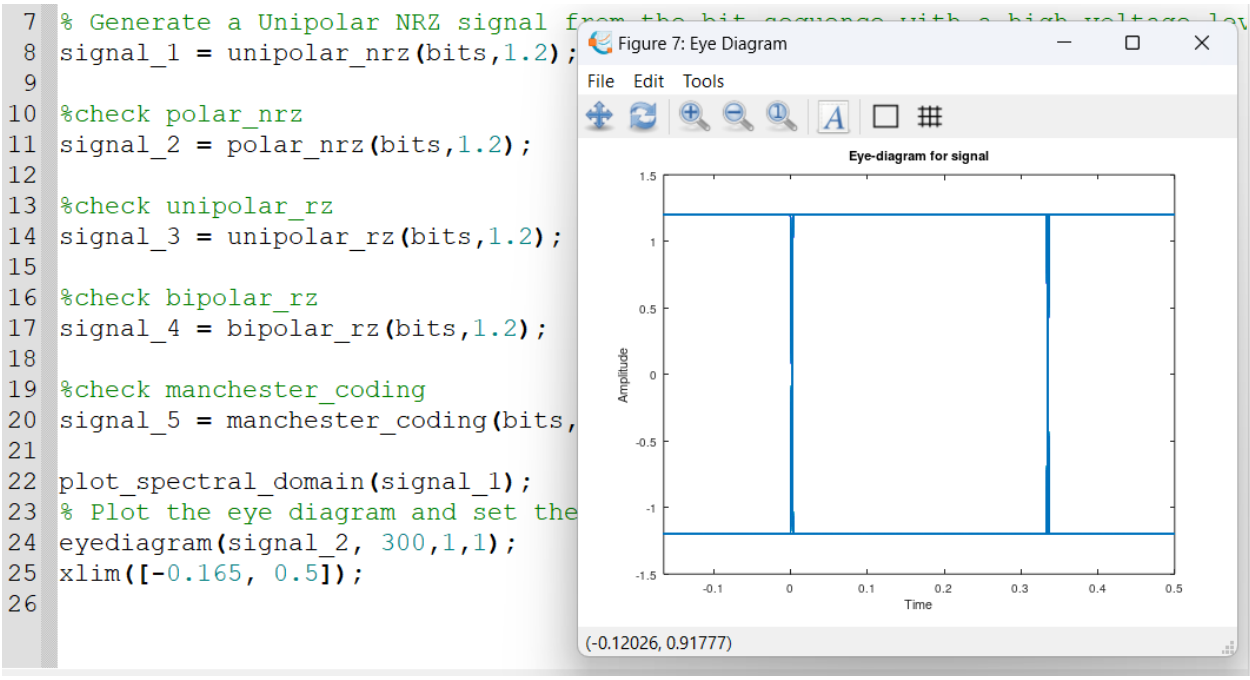


4.5.10 Spectral Domain of Manchester Coding

```
%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_5);
% Plot the eye diagram and set the plot
%eyediagram(signal_2, 300,1,1);
%xlim([-0.165, 0.5]);

```



4.5.11 Eye Diagram of Unipolar Non-Return to Zero



5 PART I RECEIVER

6 PART II TRANSIMITTER

6.1 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 transimitted & received bits, compares between them then calculates the bit error rate from the formula: $BER = \frac{no.\ of\ error\ bits}{Total\ no.\ of\ bits}$

```
% 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)
  NumOfErrors = 0;
  for index = 1:length(tx_bits);
    if tx_bits != rx_bits
      NumOfErrors = NumOfErrors +1;
```

```
        endif
      endfor
      BER = NumOfErrors/length(tx_bits);
    end
```

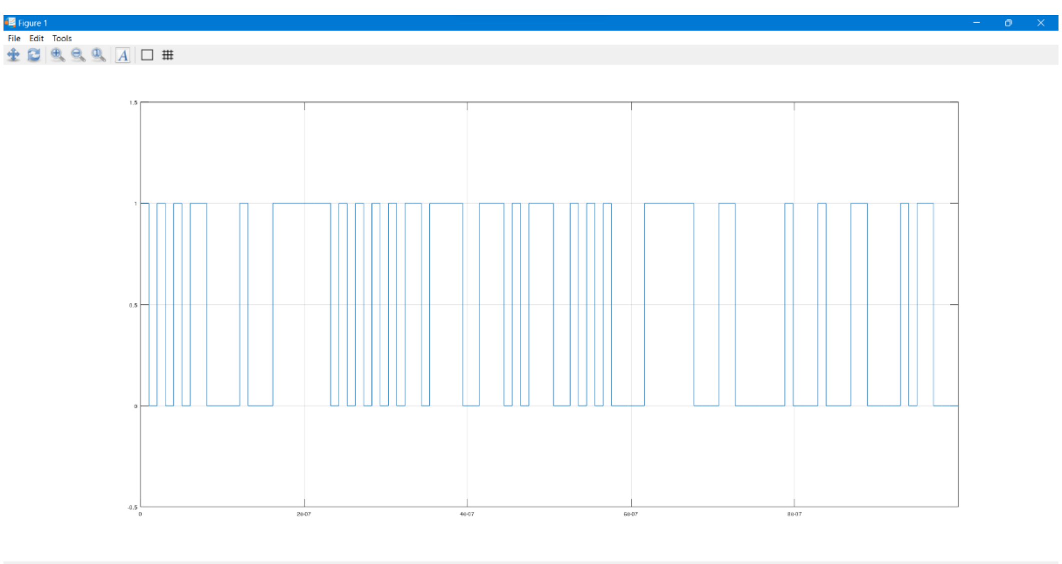
6.2 PART II TRANSIMITTER

6.2.1 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’.)

6.2.1.1 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]);
```

6.2.1.2 GRAPH

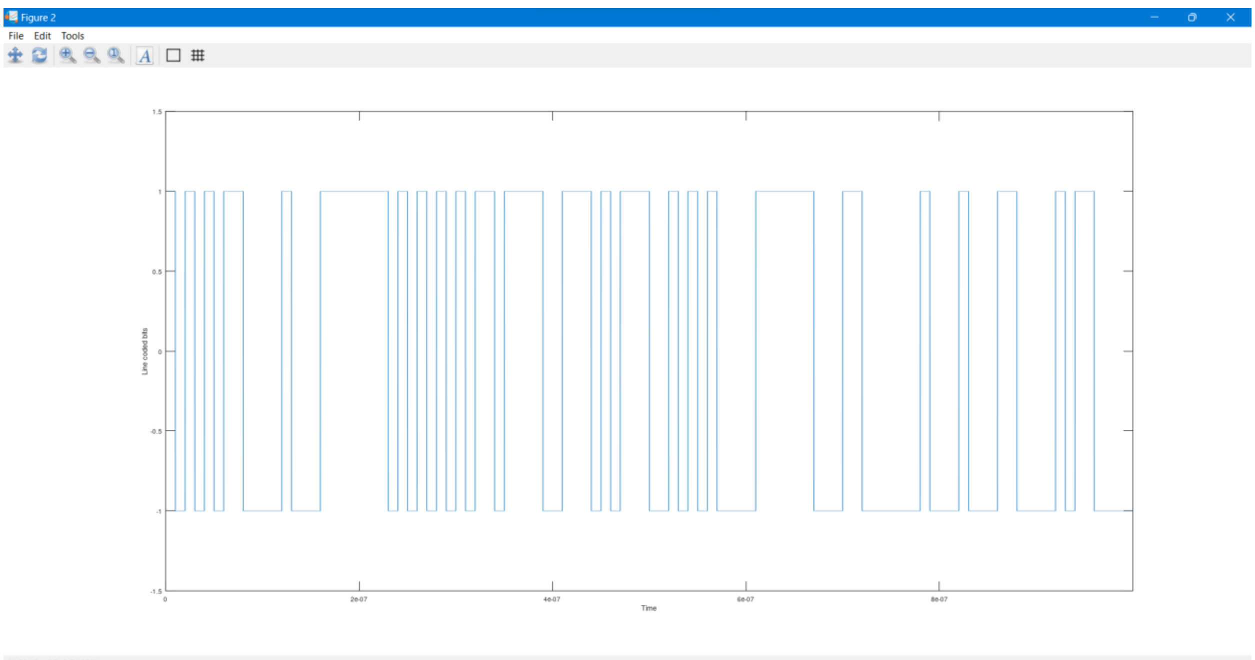


6.2.2 Line code the stream of bits (pulse shape) according to Polar non return to zero (Maximum voltage +1, Minimum voltage -1).

6.2.2.1 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]);
xlabel("Time");
ylabel("Line coded bits");
```

6.2.2.2 GRAPH



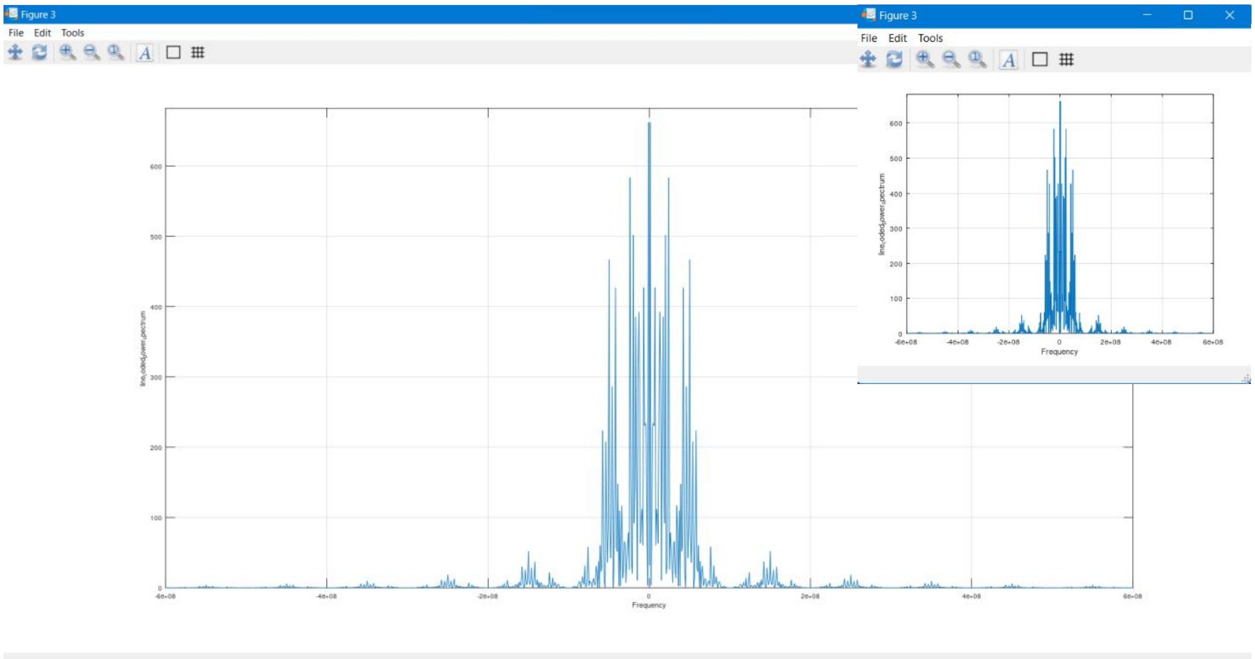
6.2.3 Plot the spectral domains.

6.2.3.1 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)).^2)/N);
```

```
figure
plot(f, line_coded_spectrum);
axis([-6e8 6e8 0 max(line_coded_spectrum)+20]);
xlabel("Frequency");
ylabel("line_coded_power_spectrum");
```

6.2.3.2 GRAPH



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

6.2.4.1 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");
```

6.2.4.2 GRAPH

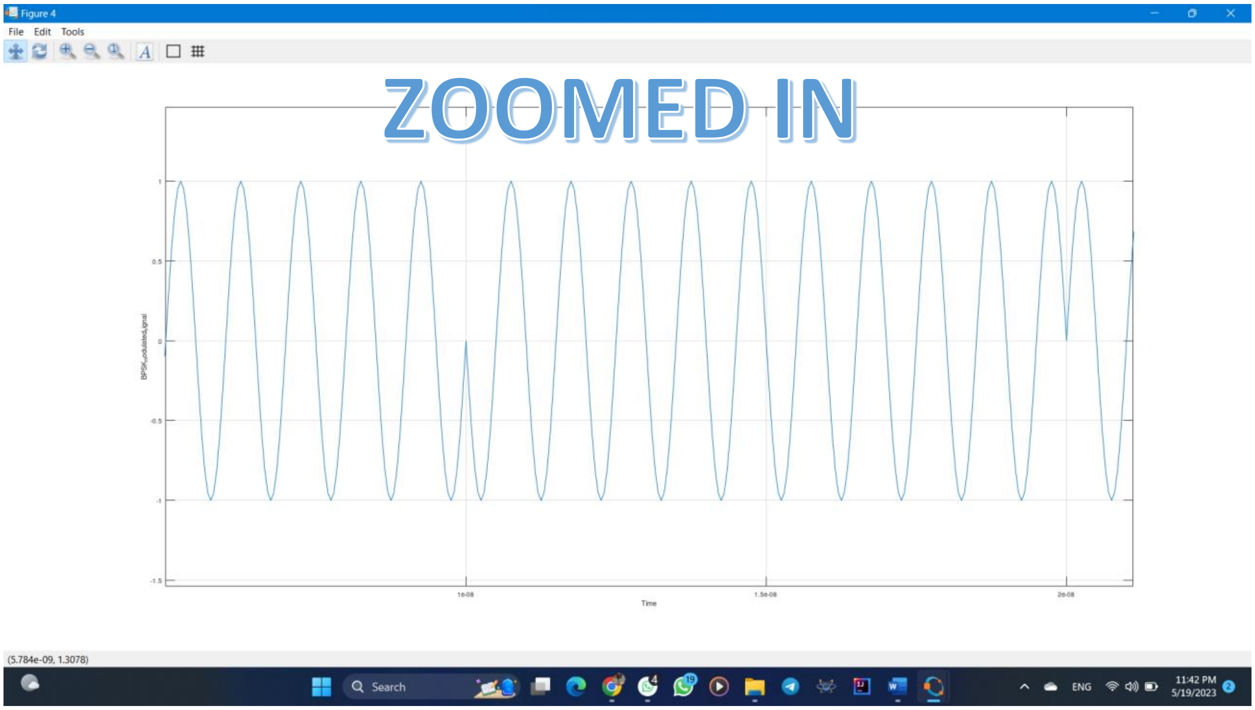


Figure 4
File Edit Tools
ZOOMED IN
BPSK_modulated_signal
Time
(5.784e-09, 1.3078)
Search
ENG
11:42 PM
5/19/2023

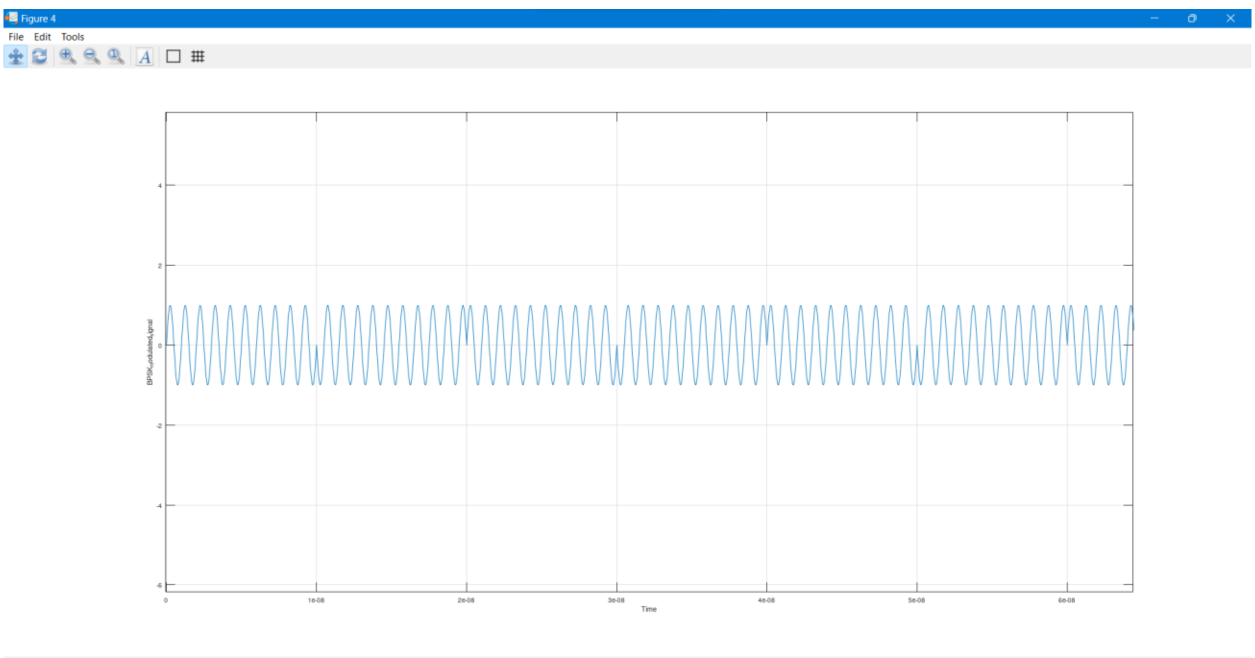


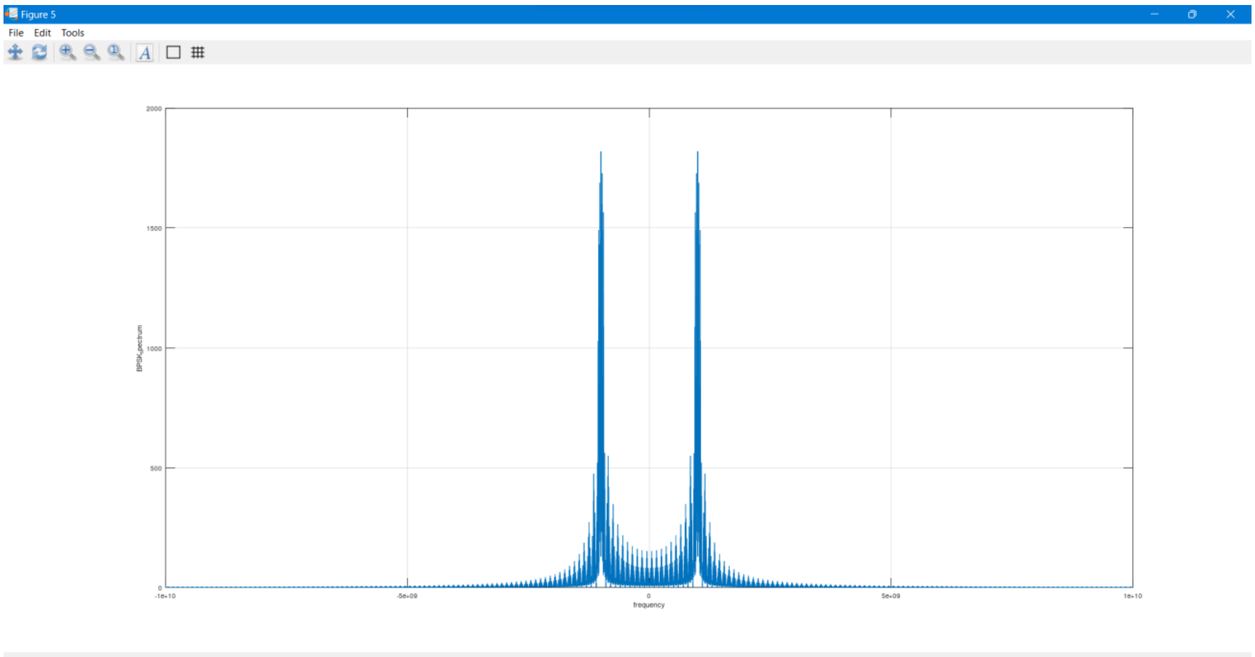
Figure 4
File Edit Tools
BPSK_modulated_signal
Time
(9.676e-10, 1.7028)

6.2.5 Plot the spectrum of the modulated BPSK signal.

6.2.5.1 CODE

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

6.2.5.2 GRAPH



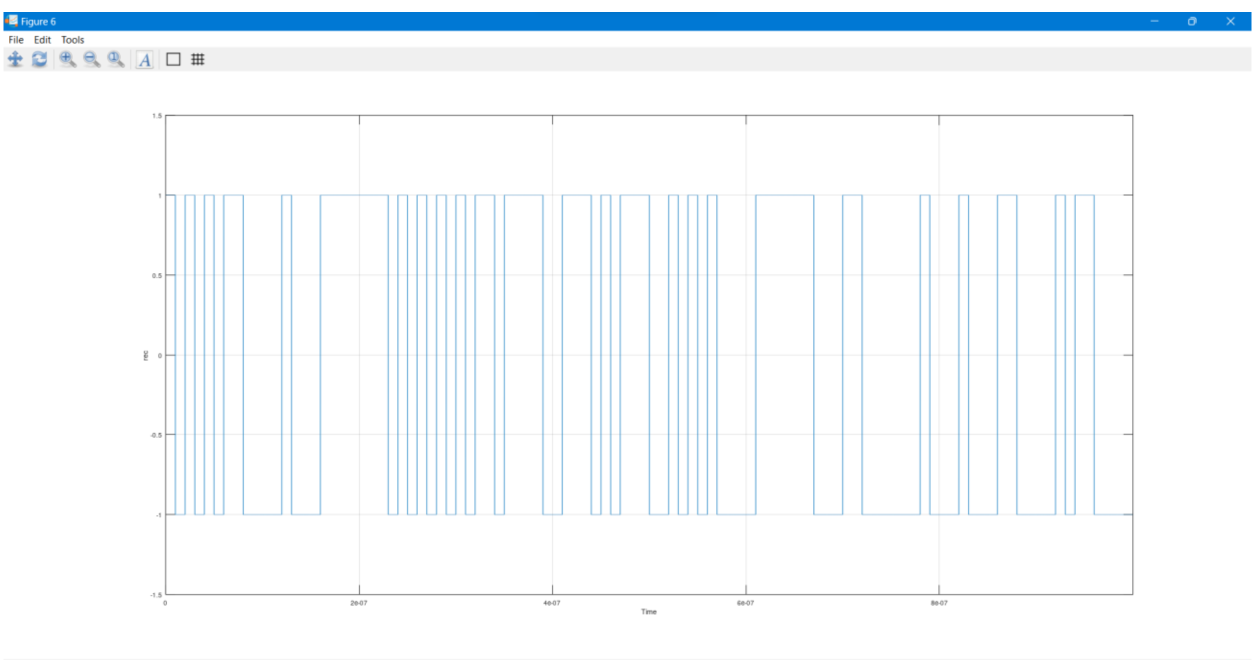
6.3 PART II RECEIVER

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

6.3.1.1 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");
```

6.3.1.2 GRAPH



6.3.2 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.

6.3.2.1 CODE

```
%%%%%%%%%%%%%%%%%%%%%% 7) Calculate BER %%%%%%%%%%%%%%%%%%%%%%%%%
BER = calculate_ber(line_coded_bits, reconstructed_bits)
```

6.3.2.2 Result

```
>> main

BER = 0
```

6.3.3 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 %%%%%%%%%%%%%%%%%%%%%%%%%%
```

6.3.4 WORKSPACE

Workspace

Filter

Name	Class	Dimension	Value
BER	double	1x1	0
BPSK_demodul...	double	1x20000	[0, 0.095492, 0.3...
BPSK_modulate...	double	1x20000	[0, 0.3090, 0.587...
BPSK_spectrum	double	1x20000	[1.4742e-13, 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, 0.3090, 0.587...
df	double	1x1	1000000
f	double	1x20000	-1e+10:1e+06:9....
fc	double	1x1	1.0000e+09
fs	double	1x1	2.0000e+10
index	double	1x1	19801
line_coded_bits	double	1x20000	[1, 1, 1, 1, 1, 1, 1,...
line_coded_spe...	double	1x20000	[0, 2.5725e-06, ...
numOfBits	double	1x1	100
rand_bits	double	1x100	[1, 1, 1, 0, 0, 0, 0,...
reconstructed_b...	double	1x20000	[1, 1, 1, 1, 1, 1, 1,...
t_bits	double	1x100	[0, 1.0101e-08, ...
time	double	1x20000	0:5e-11:9.9995e...
ts	double	1x1	5.0000e-11
y	double	1x100	[4.9976e-09, 4.9...