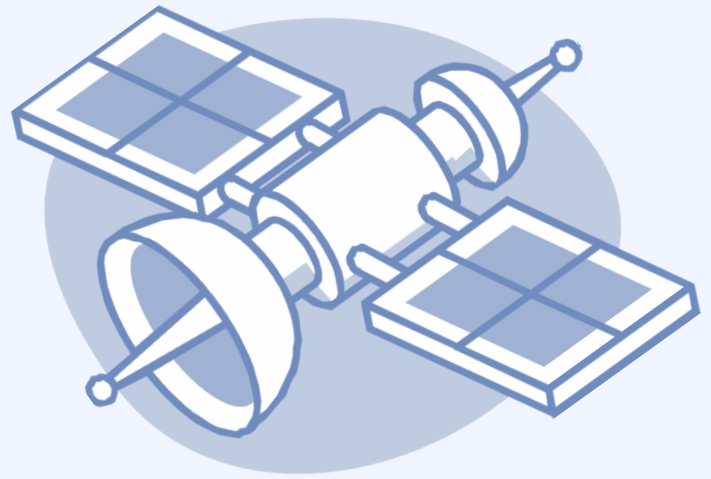


ECE252s



Fundamentals of communication systems project.

Line coding system & BPSK transmitter and Receiver using octave simulator.



Spring 2023

Prepared by: Team 08.

CSE & ECE & EPM students.

Prepared to: Dr. Alaa Eldin Fathy.

The team's details:

Team Members	ID	Program
Mohamed Ahmed Sayed	2001171	ECE
Eslam Mohamed Marzouk	2000252	CSE
Abdelrahman Ahmed Sayed	2001722	ECE
Youssef Hassanin Mahmoud	2001398	EPM
Ahmed Nader Ahmed	2000056	ECE
Karim Ibrahim Saad	2001118	CSE
Abdelrahman Elsayed Ahmed	2002139	CSE
Muhammad AbdelKhaleq Muhammad	2001052	ECE
Yasmeen Mahmoud Hassan	2001304	EPM
Ahmed Ali Abd Elhakeem	2000047	ECE

The team leader:

Mohamed Ahmed Sayed. ID: 2001171 ECE

Table of contents:

Part1: Line coding system:

Transmitter:

Step 1 : Generate stream of random bits. *Page:4*

Step 2: Line code the stream of bits. *Page:5*

Step 3: Plot the Eye diagram. *Page:13*

Step 4: Plot the spectral domains of the pulses . *Page:16*

Receiver:

Step 5-6-8-9-10: Step 5-6-8-9-10: design the receiver and add noise to the received signal then calculate BER and error and plot BER verses sigma for each line code. *Page:22*

Step7: Repeat the previous steps for different line coding (Polar non return to zero, Uni-polar return to zero, Bipolar return to zero and Manchester coding).

Step 11: (Bonus) For the case of Bipolar return to zero , design an error detection circuit. *Page:29*

Part2: Binary phase shift-keying:

Transmitter:

Step 1 : Generate stream of random bits. *Page:34*

Step 2: Line code the stream of bits. *Page:35*

Step 3 : spectral domain before Modulation: *Page:38*

Step 4: time domain after Modulation. *Page:41*

Step 5 : spectrum of the modulated BPSK signal: *Page:43*

Receiver:

Step 6: Design a receiver. *Page:46*

Step 7: calculate bit error rate (BER). *Page:53*

Code as text: *Page:54*

Script Part 1:

part1.m

Script Part 2:

part2.m

Functions codes:

generate_random_bits.m

line_coding.m

plot_eye_diagram.m

spectral_domain.m

Sweep_on_value_of_sigma.m

add_noise_to_linecoding.m

decision_device.m

convert_into_Binary_data.m

BER_device.m

Regenerative_Repeater.m

Error_Detection_Circuit.m

Bonus_Sweep_on_value_of_sigma.m

Part1: Line coding system:

Transmitter:

Step 1 : Generate stream of random bits:

Code:

Used functions:

File: generate_random_bits.m

```
1: function bit_stream = generate_random_bits( noOfBits )
2:     % generate random sequence of binary data zeros and ones
3:     bit_stream = randi( [ 0 , 1 ] , 1 , noOfBits );
4: end
```

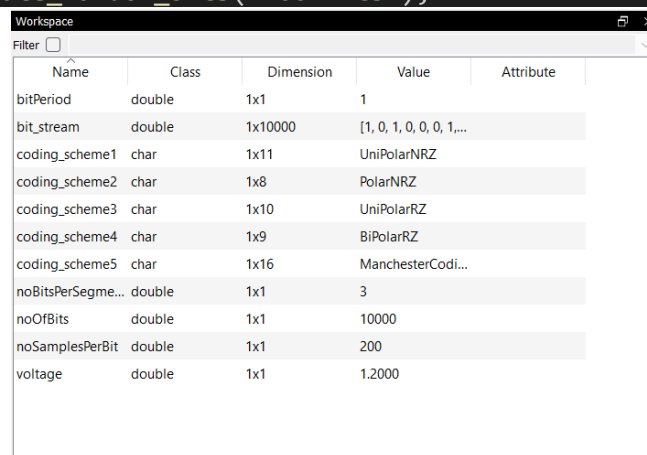
Script:

File: part1.m

```
1: % Clear all variables and close all figures
2: clear all;
3: close all;
4:
5: %set values
6: voltage=1.2;
7:
8: noOfBits=10000;
9: bitPeriod=1;
10: noSamplesPerBit=200;
11: noBitsPerSegments=3;
12:
13: coding_scheme1='UniPolarNRZ';
14: coding_scheme2='PolarNRZ';
15: coding_scheme3='UniPolarRZ';
16: coding_scheme4='BiPolarRZ';
17: coding_scheme5='ManchesterCoding';
18:
19:
20: % generate bit_stream
21: bit_stream = generate_random_bits( noOfBits );
```

Snapshots:

Workspace:



Name	Class	Dimension	Value	Attribute
bitPeriod	double	1x1	1	
bit_stream	double	1x10000	[1, 0, 1, 0, 0, 0, 1, ...	
coding_scheme1	char	1x11	UniPolarNRZ	
coding_scheme2	char	1x8	PolarNRZ	
coding_scheme3	char	1x10	UniPolarRZ	
coding_scheme4	char	1x9	BiPolarRZ	
coding_scheme5	char	1x16	ManchesterCodi...	
noBitsPerSegme...	double	1x1	3	
noOfBits	double	1x1	10000	
noSamplesPerBit	double	1x1	200	
voltage	double	1x1	1.2000	

Step 2: Line code the stream of bits:

Note: this part is containing all the line codes required in step 7 (Uni-polar non return to zero, Polar non return to zero, Uni-polar return to zero, Bipolar return to zero and Manchester coding).

Code:

Function:

File: line_coding.m

```
1: function [lineCodeVec,timeVec]=line_coding(bit_stream,coding_scheme,voltage
,bitPeriod,noSamplesPerBit,fc )
2:     switch nargin
3:         %choose according to number of input arguments
4:         case 5
5:             %belong to part 1
6:             [lineCodeVec,timeVec]=Baseband_communication(bit_stream,coding_scheme,voltage,bitPeriod,noSamplesPerBit);
7:         case 6
8:             %belong to part 2
9:             [lineCodeVec,timeVec]=Passband_communication(bit_stream,coding_scheme,voltage,bitPeriod,noSamplesPerBit,fc);
10:        end
11: end
12:

%% skip to the needed function in part1
34: function lineCodeVec = polarNRZ(bit_stream , voltage , timeVec , noSamplesPerBit)
35:     lineCodeVec = unipolarNRZ(bit_stream , voltage , timeVec , noSamplesPerBit);
36:     %same as unipolarNRZ but change all the zeros into -ve voltage
37:     lineCodeVec(lineCodeVec == 0) = -1 * voltage;
38: end
40: function lineCodeVec = unipolarRZ(bit_stream , voltage , timeVec , noSamplesPerBit)
41:     lineCodeVec = zeros(1 , length(timeVec));
42:     for i = 1 : length(bit_stream)
43:         if bit_stream(i) == 1
44:             %+ve voltage for the first half cycle of the bits
45:             lineCodeVec( ((i - 1) * noSamplesPerBit) + 1 : (i * noSamplesPerBit) -
(noSamplesPerBit / 2)) = voltage;
46:             % 0 voltage for the other half cycle of the bit
47:             lineCodeVec( (i * noSamplesPerBit) - (noSamplesPerBit / 2) + 1 : i *
noSamplesPerBit) = 0;
48:         end
49:     end
50: end
52: function lineCodeVec = bipolarRZ(bit_stream , voltage , timeVec , noSamplesPerBit)
53:     lineCodeVec = zeros(1 , length(timeVec));
54:     flag = 0; % to indicate whether the voltage to be +ve or -ve
55:     for i = 1 : length(bit_stream)
56:         if bit_stream(i) == 1
```

```

57:         if (flag == 0)
58:             %+ve voltage for the first half cycle of the bits
59:             lineCodeVec( ((i - 1) * noSamplesPerBit) + 1 : (i * noSamplesPerBit) -
(noSamplesPerBit / 2)) = voltage;
60:             % 0 voltage for the other half cycle of the bit
61:             lineCodeVec( (i * noSamplesPerBit) - (noSamplesPerBit / 2) + 1 : i *
noSamplesPerBit) = 0;
62:             flag = 1; %update the flag
63:         elseif(flag == 1)
64:             %+ve voltage for the first half cycle of the bits
65:             lineCodeVec( ((i - 1) * noSamplesPerBit) + 1 : (i * noSamplesPerBit) -
(noSamplesPerBit / 2)) = -voltage;
66:             % 0 voltage for the other half cycle of the bit
67:             lineCodeVec( (i * noSamplesPerBit) - (noSamplesPerBit / 2) + 1 : i *
noSamplesPerBit) = 0;
68:             flag = 0; %update the flag
69:         end
70:     end
71: end
72: end
74: function lineCodeVec = manchesterCoding(bit_stream , voltage , timeVec ,
noSamplesPerBit)
75:     lineCodeVec = zeros(1 , length(timeVec));
76:     for i = 1 : length(bit_stream)
77:         if bit_stream(i) == 1
78:             %+ve voltage for the first half cycle of the bits
79:             lineCodeVec( ((i - 1) * noSamplesPerBit) + 1 : (i * noSamplesPerBit) -
(noSamplesPerBit / 2)) = voltage;
80:             %+ve voltage for the other half cycle of the bit
81:             lineCodeVec( (i * noSamplesPerBit) - (noSamplesPerBit / 2) + 1 : i *
noSamplesPerBit) = -1* voltage;
82:         else
83:             %+ve voltage for the first half cycle of the bits
84:             lineCodeVec( ((i - 1) * noSamplesPerBit) + 1 : (i * noSamplesPerBit) -
(noSamplesPerBit / 2)) = -1 * voltage;
85:             %+ve voltage for the other half cycle of the bit
86:             lineCodeVec( (i * noSamplesPerBit) - (noSamplesPerBit / 2) + 1 : i *
noSamplesPerBit) = voltage;
87:         end
88:     end
89: end

```

Script:

File: part1.m

```
23: [lineCodeVec1, timeVec1] = line_coding(bit_stream, coding_scheme1, voltage, bitPeriod,
noSamplesPerBit);
24: [lineCodeVec2, timeVec2] = line_coding(bit_stream, coding_scheme2, voltage, bitPeriod,
noSamplesPerBit);
25: [lineCodeVec3, timeVec3] = line_coding(bit_stream, coding_scheme3, voltage, bitPeriod,
noSamplesPerBit);
26: [lineCodeVec4, timeVec4] = line_coding(bit_stream, coding_scheme4, voltage, bitPeriod,
noSamplesPerBit);
27: [lineCodeVec5, timeVec5] = line_coding(bit_stream, coding_scheme5, voltage, bitPeriod,
noSamplesPerBit);
29:
30: %% plot 5 line coding in time domain
31: % UniPolarNRZ line coding
32: figure(1);
33: plot(timeVec1, lineCodeVec1, 'color', [0.4940 0.1840 0.5560], 'LineWidth', 2);
34: axis([0 timeVec1(end) 0 - voltage/2 voltage*3/2]);
35: title("UniPolarNRZ line coding");
36: legend('UniPolarNRZ');
37:
38: % Zoomed
39: figure(2);
40: plot(timeVec1, lineCodeVec1, 'color', [0.4940 0.1840 0.5560], 'LineWidth', 2);
41: axis([0 100*bitPeriod 0 - voltage/2 voltage*3/2]);
42: title("UniPolarNRZ line coding (zoomed to first 100 bits)");
43: legend('UniPolarNRZ');
44: grid on;
46:
47: % PolarNRZ line coding
48: figure(3);
49: plot(timeVec2, lineCodeVec2, 'color', [1 0 0], 'LineWidth', 2);
50: axis([0 timeVec2(end) -3*voltage/2 3*voltage/2]);
51: title("PolarNRZ line coding");
52: legend('PolarNRZ');
53:
54: % Zoomed
55: figure(4);
56: plot(timeVec2, lineCodeVec2, 'color', [1 0 0], 'LineWidth', 2);
57: axis([0 100*bitPeriod -3*voltage/2 3*voltage/2]);
58: title("PolarNRZ line coding (zoomed to first 100 bits)");
59: legend('PolarNRZ');
60: grid on;
62:
63: % UniPolarRZ line coding
64: figure(5);
65: plot(timeVec3, lineCodeVec3, 'color', [0.8500 0.3250 0.0980], 'LineWidth', 2);
66: axis([0 timeVec3(end) 0 - voltage/2 voltage*3/2]);
67: title("UniPolarRZ line coding");
68: legend('UniPolarRZ');
```



```

69:
70: % Zoomed
71: figure(6);
72: plot(timeVec3, lineCodeVec3, 'color', [0.8500 0.3250 0.0980], 'LineWidth', 2);
73: axis([0 100*bitPeriod 0 - voltage/2 voltage*3/2]);
74: title("UniPolarRZ line coding (zoomed to first 100 bits)");
75: legend('UniPolarRZ');
76:
77: % BiPolarRZ line coding
78: figure(7);
79: plot(timeVec4, lineCodeVec4, 'color', [0 0.4470 0.7410], 'LineWidth', 2);
80: axis([0 timeVec4(end) -3*voltage/2 3*voltage/2]);
81: title("BiPolarRZ line coding");
82: legend('BiPolarRZ');
83:
84: % Zoomed
85: figure(8);
86: plot(timeVec4, lineCodeVec4, 'color', [0 0.4470 0.7410], 'LineWidth', 2);
87: axis([0 100*bitPeriod -3*voltage/2 3*voltage/2]);
88: title("BiPolarRZ line coding (zoomed to first 100 bits)");
89: legend('BiPolarRZ');
90: grid on;
92:
93: % ManchesterCoding line coding
94: figure(9);
95: plot(timeVec5, lineCodeVec5, 'color', [0.6350 0.0780 0.1840], 'LineWidth', 2);
96: axis([0 timeVec5(end) -3*voltage/2 3*voltage/2]);
97: title("ManchesterCoding line coding");
98: legend('ManchesterCoding');
99:
100: % Zoomed
101: figure(10);
102: plot(timeVec5, lineCodeVec5, 'color', [0.6350 0.0780 0.1840], 'LineWidth', 2);
103: axis([0 100*bitPeriod -3*voltage/2 3*voltage/2]);
104: title("ManchesterCoding line coding (zoomed to first 100 bits)");
105: legend('ManchesterCoding');
106: grid on;

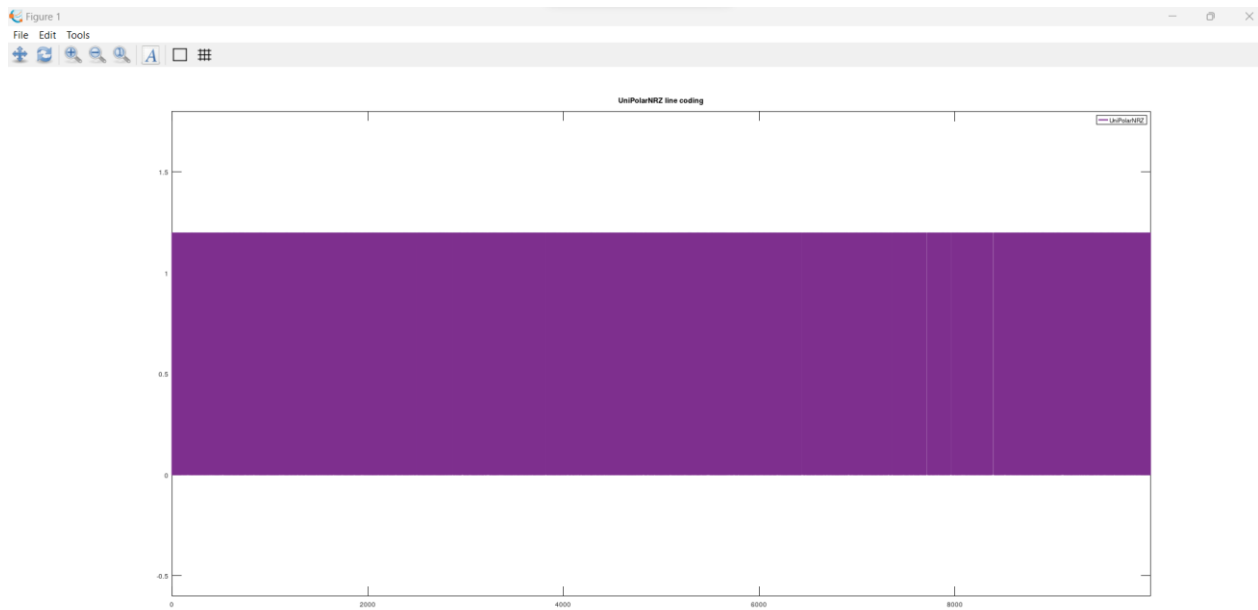
```

Snapshots: Workspace:

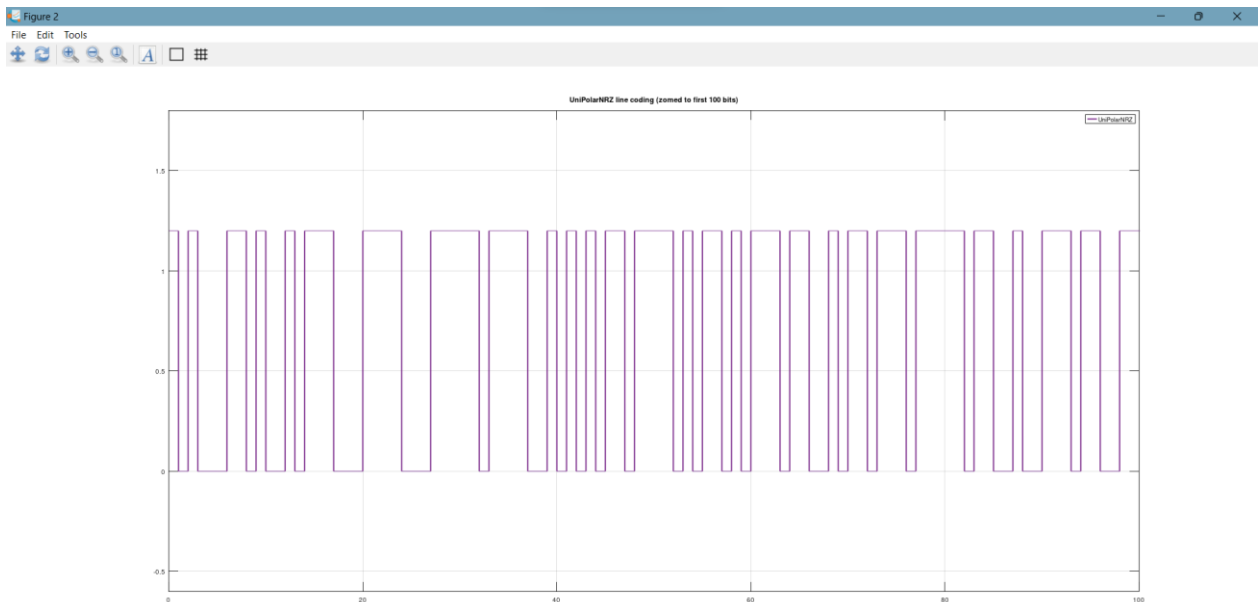
Name	Class	Dimension	Value	Attribute
bitPeriod	double	1x1	1	
bit_stream	double	1x10000	[1, 0, 1, 0, 0, 1, ...]	
coding_scheme1	char	1x11	UniPolarNRZ	
coding_scheme2	char	1x8	PolarNRZ	
coding_scheme3	char	1x10	UniPolarRZ	
coding_scheme4	char	1x9	BiPolarRZ	
coding_scheme5	char	1x16	ManchesterCodi...	
lineCodeVec1	double	1x2000000	[1.2000, 1.2000, ...]	
lineCodeVec2	double	1x2000000	[1.2000, 1.2000, ...]	
lineCodeVec3	double	1x2000000	[1.2000, 1.2000, ...]	
lineCodeVec4	double	1x2000000	[1.2000, 1.2000, ...]	
lineCodeVec5	double	1x2000000	[1.2000, 1.2000, ...]	
noBitsPerSegme...	double	1x1	3	
noOfBits	double	1x1	10000	
noSamplesPerBit	double	1x1	200	
timeVec1	double	1x2000000	0:0.005:10000	
timeVec2	double	1x2000000	0:0.005:10000	
timeVec3	double	1x2000000	0:0.005:10000	
timeVec4	double	1x2000000	0:0.005:10000	
timeVec5	double	1x2000000	0:0.005:10000	
voltage	double	1x1	1.2000	

Time domain plot of UniPolar NRZ signal:

Before zoom:



After zoom:

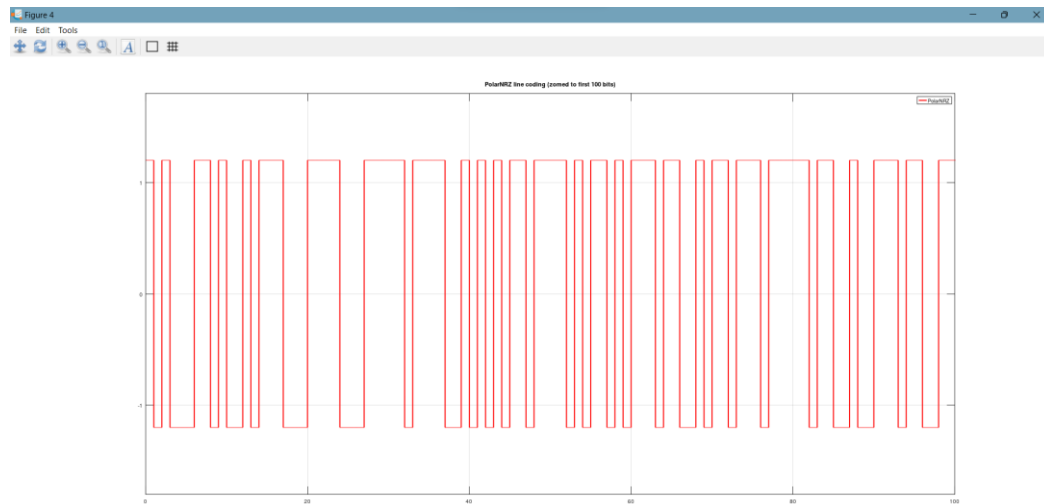


Time domain plot of Polar NRZ signal:

Before zoom:

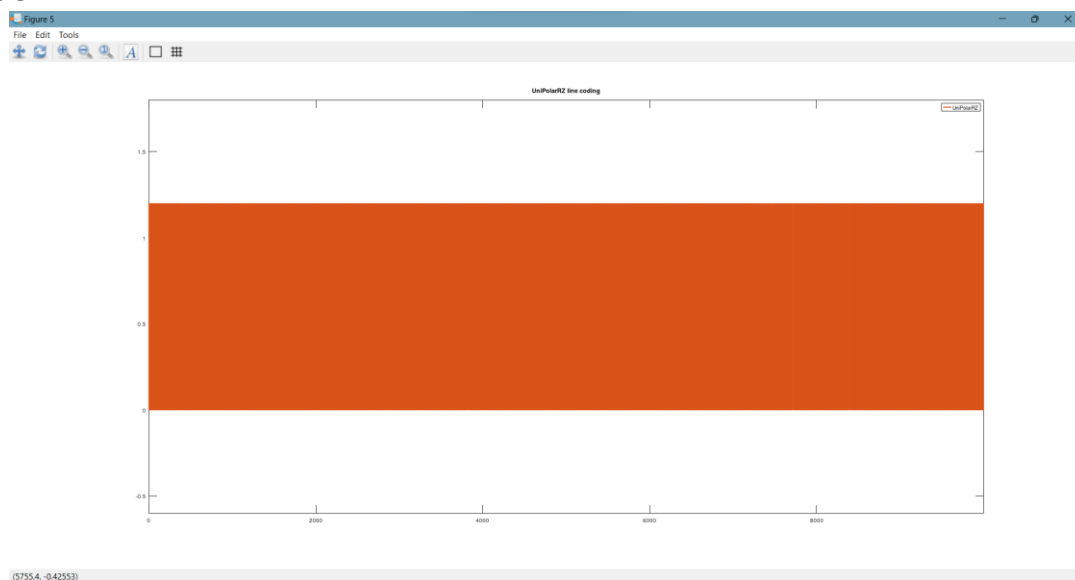


After zoom:

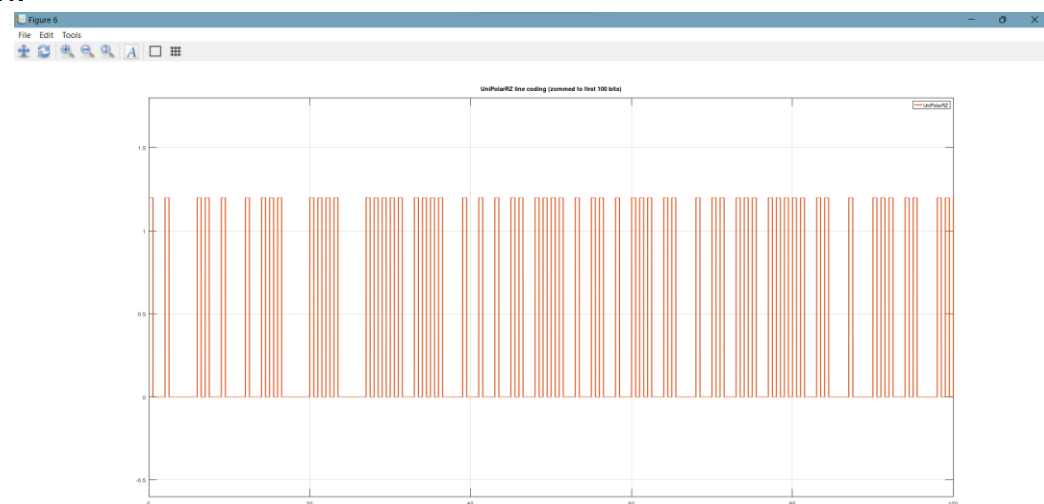


Time domain plot of UniPolar RZ signal:

Before zoom:

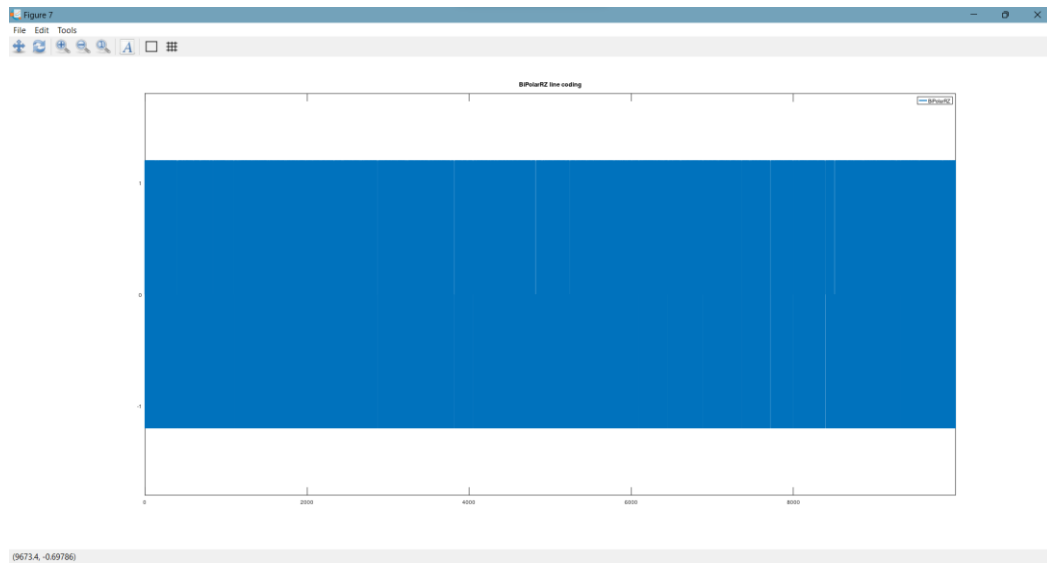


After zoom:

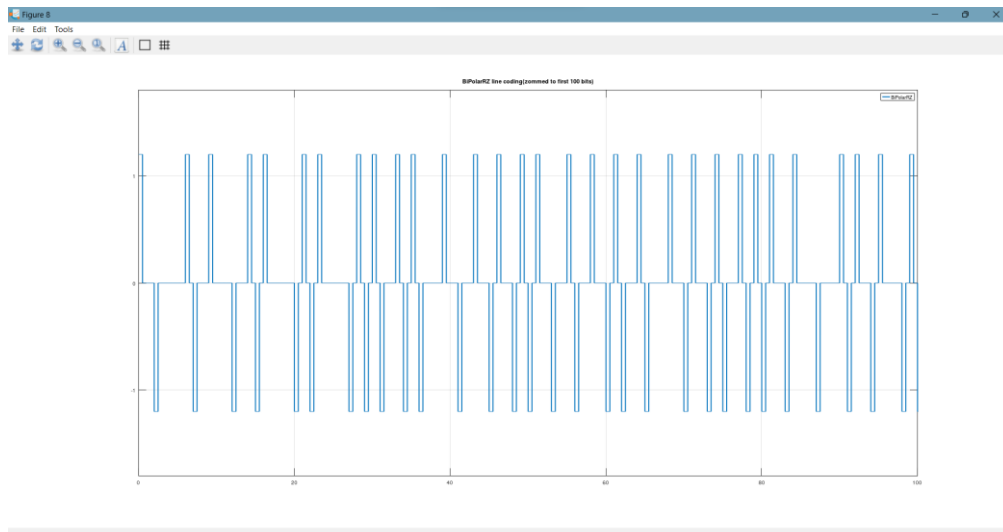


Time domain plot of BiPolar RZ signal:

Before zoom:



After zoom:

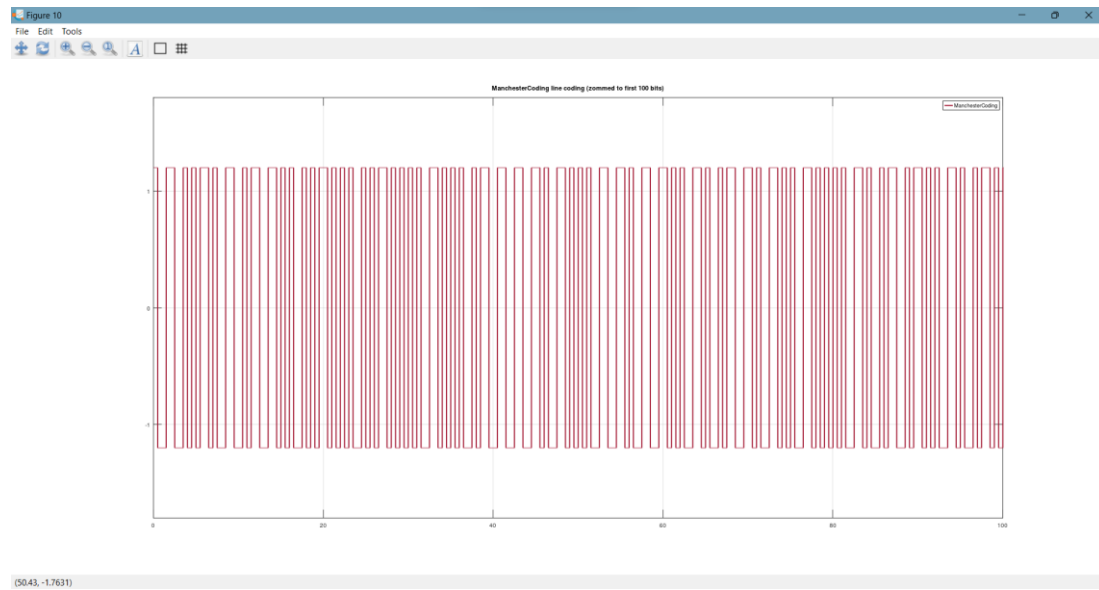


Time domain plot of Manchester signal:

Before zoom:



After zoom:



Step 3: Plot the Eye diagram.:

Note: this part is containing all the line codes required in step 7 (Uni-polar non return to zero, Polar non return to zero, Uni-polar return to zero, Bipolar return to zero and Manchester coding).

Code:

Functions:

File: plot_eye_diagram.m

```
1: function plot_eye_diagram(noBitsPerSegments , noSamplePerBit , lineCodeVec , bitPeriod)
2:     % segment is a collection of bits we need to shift them to create the eye diagram
3:     % segmentLength: total samples in a segment
4:     segmentLength = noSamplePerBit * noBitsPerSegments;
5:     % periodic time of the samples
6:     samplePeriod = bitPeriod / noSamplePerBit;
7:     % time vector is created with a step of samplePeriod, and the end is the total time
   a segment
8:     % could be calculated using noBitsPerSegments and bitPeriod, with a step =
   samplePeriod
9:     timeVec = (0 : segmentLength - 1) * samplePeriod;
10:    % to center the zero in the mid
11:    timeVec = timeVec - (noBitsPerSegments * bitPeriod / 2);
12:    % total number of segment layers to place on top of each other
13:    % segmentLayers = total number of samples in line code / total number of samples in
   the segment
14:    % it's rounded down to the nearest integer, as we need to keep the size of the
   segment
15:    % equal to the size of the time vector in terms of the samples
16:    segmentLayers = floor(length(lineCodeVec) / segmentLength);
17:    for i = 1 : segmentLayers
18:        % find the start and end points of the segment in the line code
19:        % it's multiplied by the number of bits per segment
20:        segmentStart = (i - 1) * segmentLength + 1;
21:        segmentEnd = i * segmentLength;
22:        % x-axis is the time vector, and the y-axis is the amplitude of each bit
23:        plot(timeVec , lineCodeVec(segmentStart : segmentEnd),'LineWidth',2);
24:        % hold to plot all the segments on top of each other
25:        hold on;
26:    end
27:    axis([-1.5 1.5 -bitPeriod bitPeriod]);
28: end
```

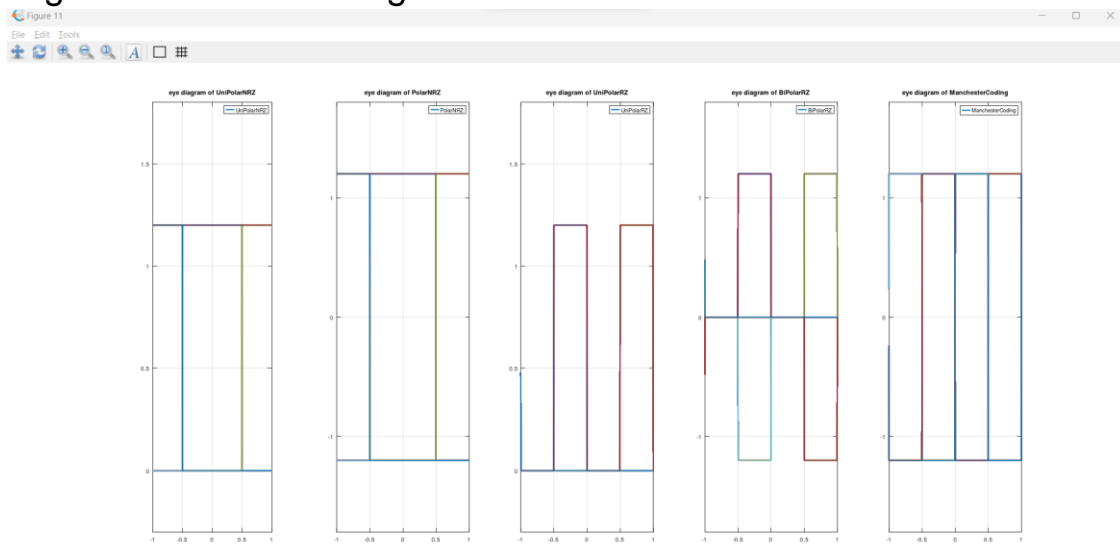
Script:

File: part1.m

```
101: %% plot eye diagram of line coding(transmitted signal)
102: % UniPolarNRZ
103: figure(11);
104: subplot(1,5,1);
105: plot_eye_diagram(noBitsPerSegments , noSamplesPerBit , lineCodeVec1 , bitPeriod);
106: title("eye diagram of UniPolarNRZ");
107: legend('UniPolarNRZ');
108: ylim([0-voltage/4 3*voltage/2]);
109: grid on;
110:
111: % PolarNRZ
112: subplot(1,5,2);
113: plot_eye_diagram(noBitsPerSegments , noSamplesPerBit , lineCodeVec2 , bitPeriod);
114: title("eye diagram of PolarNRZ");
115: legend('PolarNRZ');
116: ylim([-3*voltage/2 3*voltage/2]);
117: grid on;
118:
119: % UniPolarRZ
120: subplot(1,5,3);
121: plot_eye_diagram(noBitsPerSegments , noSamplesPerBit , lineCodeVec3 , bitPeriod);
122: title("eye diagram of UniPolarRZ");
123: legend('UniPolarRZ');
124: ylim([0-voltage/4 3*voltage/2]);
125: grid on;
126:
127: % BiPolarRZ
128: subplot(1,5,4);
129: plot_eye_diagram(noBitsPerSegments , noSamplesPerBit , lineCodeVec4 , bitPeriod);
130: title("eye diagram of BiPolarRZ");
131: legend('BiPolarRZ');
132: ylim([-3*voltage/2 3*voltage/2]);
133: grid on;
134:
135: % ManchesterCoding
136: subplot(1,5,5);
137: plot_eye_diagram(noBitsPerSegments , noSamplesPerBit , lineCodeVec5 , bitPeriod);
138: title("eye diagram of ManchesterCoding");
139: legend('ManchesterCoding');
140: ylim([-3*voltage/2 3*voltage/2]);
141: grid on;
```

Snapshots:

Eye-diagram of all line coding:



Step 4: Plot the spectral domains of the pulses :

Note: this part is containing all the line codes required in step 7 (Uni-polar non return to zero, Polar non return to zero, Uni-polar return to zero, Bipolar return to zero and Manchester coding).

Code:

Functions:

File: spectral_domain.m

```
1: function
[spectral,f,BW,Spec_Original_line_coding]=spectral_domain(lineCodeVec,bit_stream,noSamplesPerBit,bitPeriod,fc)
2:     switch nargin
3:         %choose according to number of input arguments
4:         case 4
5:             [spectral,f]=Baseband_communication(lineCodeVec,bit_stream,noSamplesPerBit,bitPeriod);
6:         case 5
7:             [spectral,f,BW,Spec_Original_line_coding]=Passband_communication(lineCodeVec,bit_stream,noSamplesPerBit,bitPeriod,fc);
8:         end
9:     end
10:

%% skip into the needed function for part1
31:
32: function
[spectral,f]=Baseband_communication(lineCodeVec,noSamplesPerBit,bitPeriod,bit_stream)
33:     %calculation to generate frequency domain
34:     T = length(bit_stream)*bitPeriod; %simulation time
35:     df=1/T; %frequency step
36:     fs=noSamplesPerBit/bitPeriod; % sampling frequency
37:     N=noSamplesPerBit*length(bit_stream);
38:     %spectrum of Original Digital Signal
39:     Spec_Original_line_coding = (fftshift(fft(lineCodeVec)))/N;
40:     % frequency doomain
41:     if(rem(N,2)==0) %% even
42:         f = ((-(0.5*fs)): df : ((0.5*fs)-df)); %% frequency vector if x/f even
43:     else %% odd
44:         f = (-(0.5*fs-0.5*df)) : df : (((0.5*fs)+1)-0.5*df); %% frequency vector if X/f is odd
45:     end
46:     %power spectral of Original Digital Signal
47:     spectral =abs(Spec_Original_line_coding).^2;
48: end
```

Script:

File: part1.m

```
143: %% get power spectrum of line coding(transmitted signal)
144:
[spectral_of_lineCodeVec1,f1]=spectral_domain(lineCodeVec1,noSamplesPerBit,bitPeriod,bit_s
tream);
145:
[spectral_of_lineCodeVec2,f2]=spectral_domain(lineCodeVec2,noSamplesPerBit,bitPeriod,bit_s
tream);
146:
[spectral_of_lineCodeVec3,f3]=spectral_domain(lineCodeVec3,noSamplesPerBit,bitPeriod,bit_s
tream);
147:
[spectral_of_lineCodeVec4,f4]=spectral_domain(lineCodeVec4,noSamplesPerBit,bitPeriod,bit_s
tream);
148:
[spectral_of_lineCodeVec5,f5]=spectral_domain(lineCodeVec5,noSamplesPerBit,bitPeriod,bit_s
tream);
149:
150: %% plot power spectrum of any type of line coding
151: % UniPolarNRZ
152: figure(12);
153: plot(f1,spectral_of_lineCodeVec1,'color',[0.4940 0.1840 0.5560],'LineWidth',2);
154: title("power spectral of UniPolarNRZ");
155: xlabel("frequency (Hz)");
156: ylabel('power spectral density');
157: legend('UniPolarNRZ');
158: grid on;
159:
160: % Zoomed UniPolarNRZ
161: figure(13);
162: plot(f1,spectral_of_lineCodeVec1,'color',[0.4940 0.1840 0.5560],'LineWidth',2);
163: title("power spectral of UniPolarNRZ (zoomed)");
164: xlabel("frequency (Hz)");
165: ylabel('power spectral density');
166: legend('UniPolarNRZ');
167: axis([-3 3 0 0.0003]);
168: grid on;
169:
170: % PolarNRZ
171: figure(14);
172: plot(f2,spectral_of_lineCodeVec2,'color',[1 0 0],'LineWidth',2);
173: title("power spectral of PolarNRZ");
174: xlabel("frequency (Hz)");
175: ylabel('power spectral density');
176: legend('PolarNRZ');
177: grid on;
178: axis([-5 5 0 0.0012]);
179:
180: % UniPolarRZ
181: figure(15);
```

```

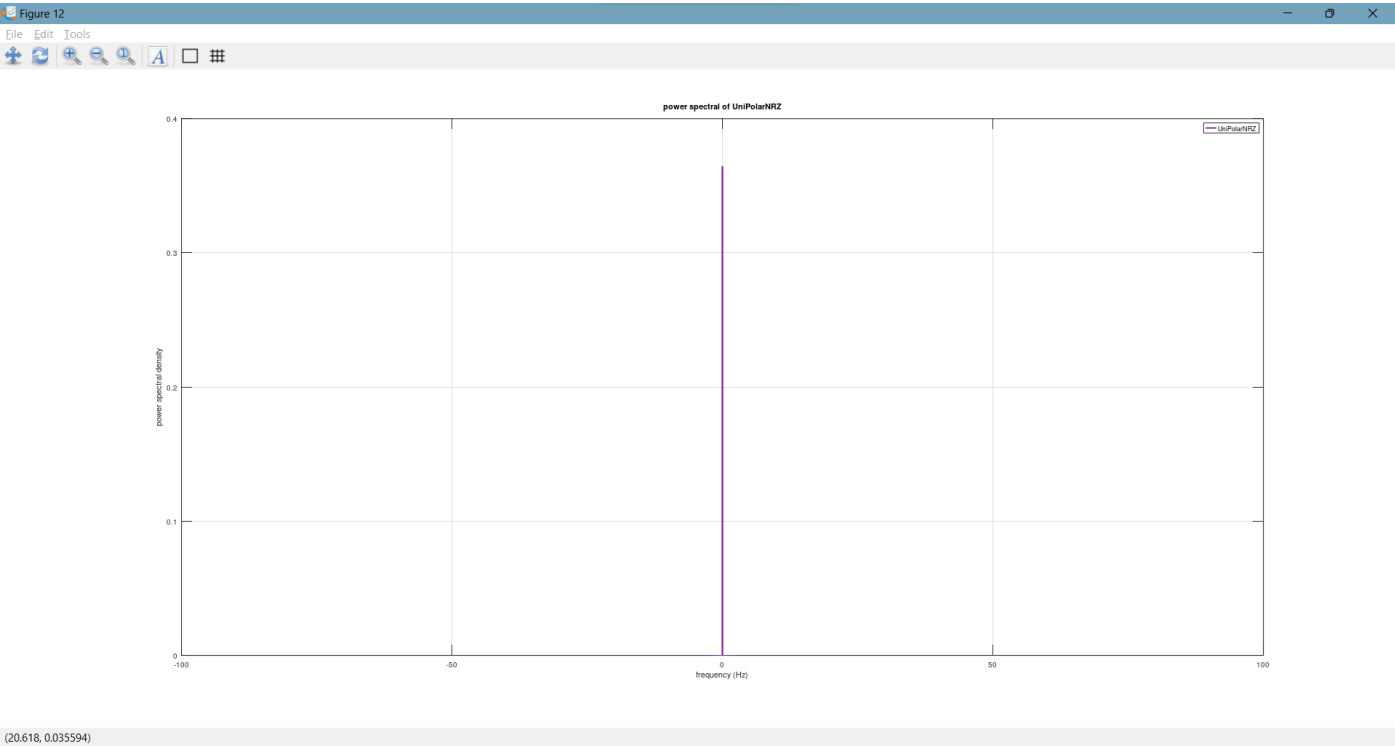
182: plot(f3,spectral_of_lineCodeVec3,'color',[0.8500 0.3250 0.0980],'LineWidth',2);
183: title("power spectral of UniPolarRZ");
184: xlabel("frequency (Hz)");
185: ylabel('power spectral density');
186: legend('UniPolarRZ');
187: grid on;
188:
189: % Zoomed UniPolarRZ
190: figure(16);
191: plot(f3,spectral_of_lineCodeVec3,'color',[0.8500 0.3250 0.0980],'LineWidth',2);
192: title("power spectral of UniPolarRZ (zoomed)");
193: xlabel("frequency (Hz)");
194: ylabel('power spectral density');
195: legend('UniPolarRZ');
196: grid on;
197: axis([-6 6 0 0.0001]);
198:
199: % BiPolarRZ
200: figure(17);
201: plot(f4,spectral_of_lineCodeVec4,'color',[0 0.4470 0.7410],'LineWidth',2);
202: title("power spectral of BiPolarRZ");
203: xlabel("frequency (Hz)");
204: ylabel('power spectral density');
205: legend('BiPolarRZ');
206: axis([-15 15 0 0.00025]);
207: grid on;
208:
209: % ManchesterCoding
210: figure(18);
211: plot(f5,spectral_of_lineCodeVec5,'color',[0.6350 0.0780 0.1840],'LineWidth',2);
212: title("power spectral of ManchesterCoding");
213: xlabel("frequency (Hz)");
214: ylabel('power spectral density');
215: legend('ManchesterCoding');
216: axis([-10 10 0 0.0008]);

```

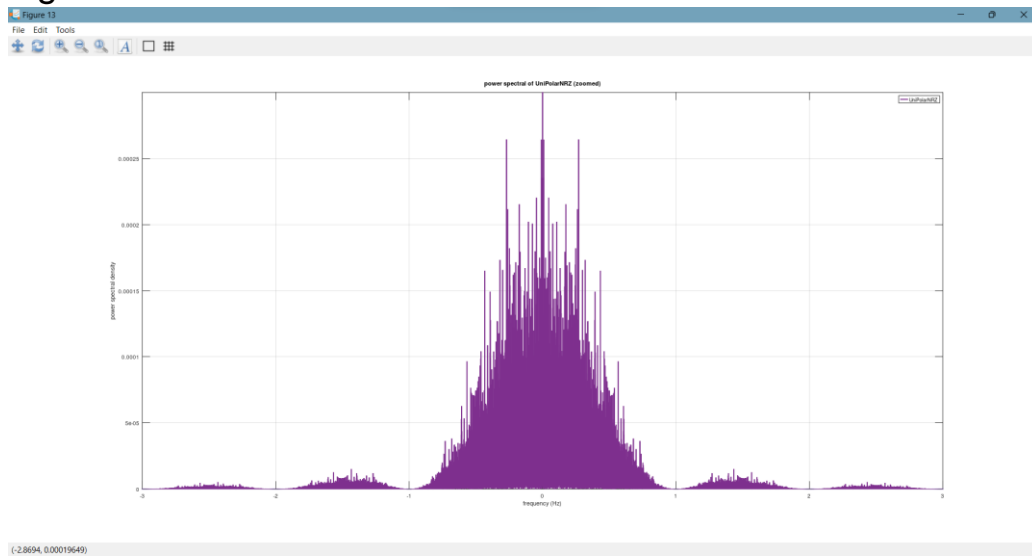
Snapshots:
Workspace:

Workspace				
Filter <input type="text"/>				
Name	Class	Dimension	Value	Attribute
bitPeriod	double	1x1	1	
bit_stream	double	1x10000	[1, 0, 1, 0, 0, 0, 1, ...	
coding_scheme1	char	1x11	UniPolarNRZ	
coding_scheme2	char	1x8	PolarNRZ	
coding_scheme3	char	1x10	UniPolarRZ	
coding_scheme4	char	1x9	BiPolarRZ	
coding_scheme5	char	1x16	ManchesterCodi...	
f1	double	1x2000000	-100:0.0001-99.9...	
f2	double	1x2000000	-100:0.0001-99.9...	
f3	double	1x2000000	-100:0.0001-99.9...	
f4	double	1x2000000	-100:0.0001-99.9...	
f5	double	1x2000000	-100:0.0001-99.9...	
lineCodeVec1	double	1x2000000	[1.2000, 1.2000, ...	
lineCodeVec2	double	1x2000000	[1.2000, 1.2000, ...	
lineCodeVec3	double	1x2000000	[1.2000, 1.2000, ...	
lineCodeVec4	double	1x2000000	[1.2000, 1.2000, ...	
lineCodeVec5	double	1x2000000	[1.2000, 1.2000, ...	
noBitsPerSegme...	double	1x1	3	
noOfBits	double	1x1	10000	
noSamplesPerBit	double	1x1	200	
spectral_of_line...	double	1x2000000	[0, 4.6797e-17, ...	
spectral_of_line...	double	1x2000000	[0, 1.8719e-16, ...	
spectral_of_line...	double	1x2000000	[0, 1.1699e-17, ...	
spectral_of_line...	double	1x2000000	[0, 9.3060e-21, ...	
spectral_of_line...	double	1x2000000	[0, 4.6187e-24, ...	
timeVec1	double	1x2000000	0:0.005:10000	
timeVec2	double	1x2000000	0:0.005:10000	
timeVec3	double	1x2000000	0:0.005:10000	
timeVec4	double	1x2000000	0:0.005:10000	
timeVec5	double	1x2000000	0:0.005:10000	

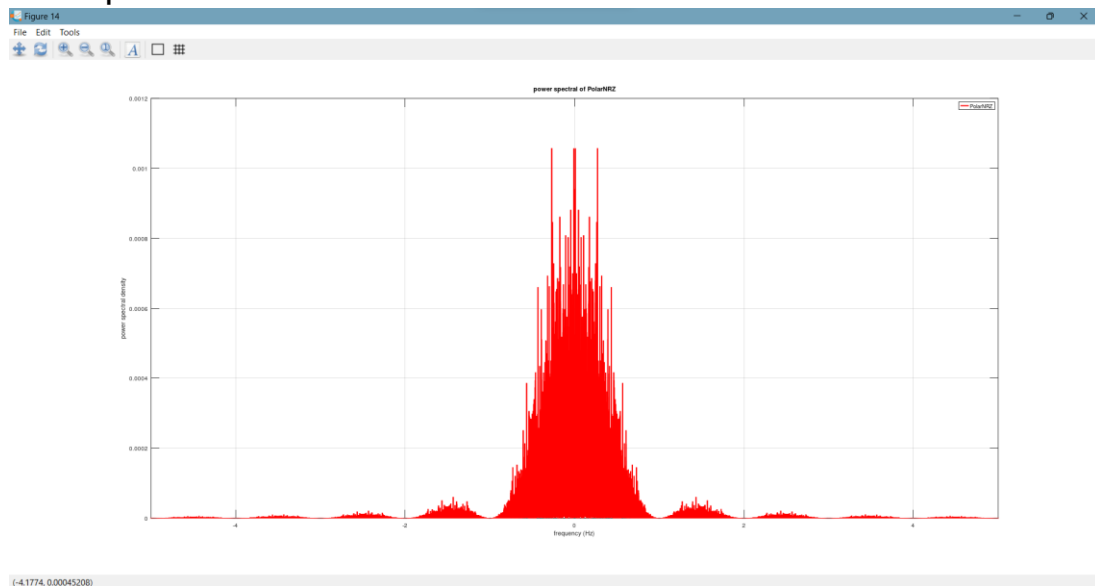
Plot of power spectral of UniPolar NRZ:



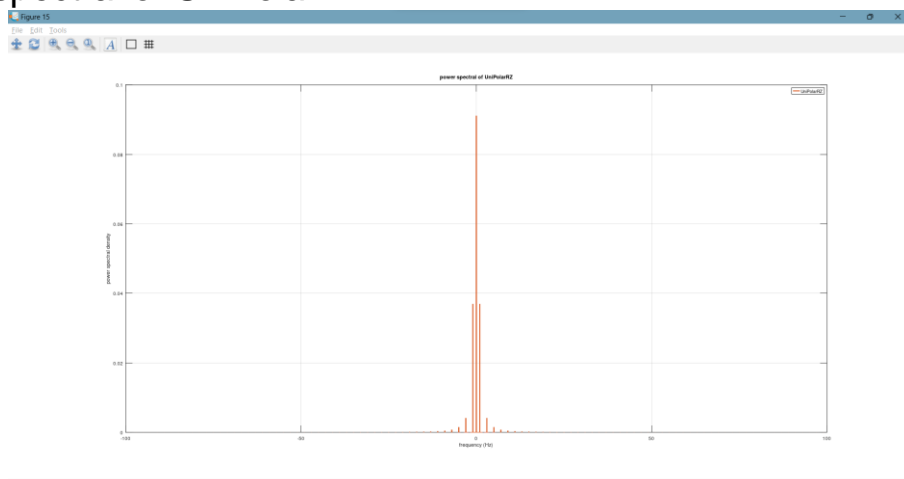
After zooming:



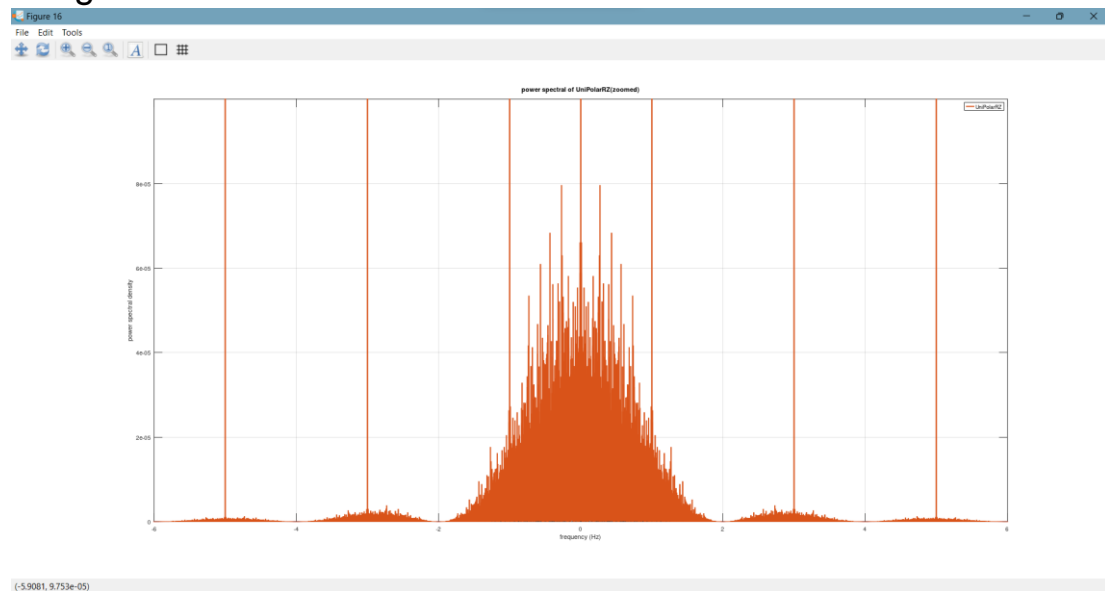
Plot of power spectral of Polar NRZ:



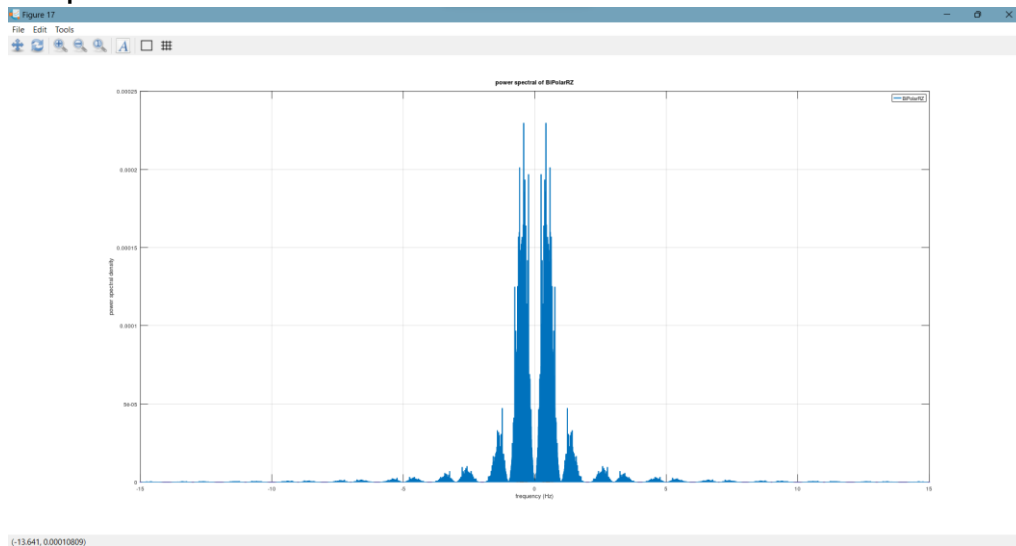
Plot of power spectral of UniPolar RZ:



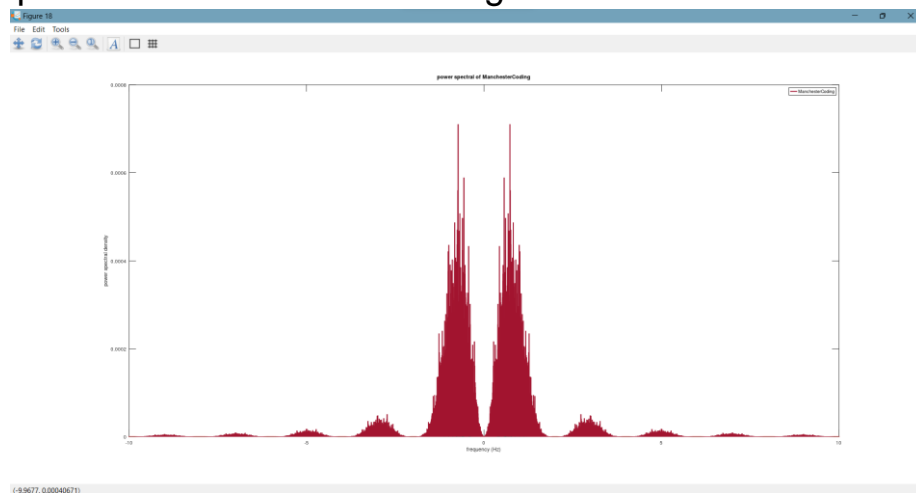
After zooming:



Plot of power spectral of BiPolar RZ:



Plot of power spectral of Manchester Coding:



Receiver:

Step 5-6-8-9-10: design the receiver and add noise to the received signal then calculate BER and error and plot BER verses sigma for each line code:

Code:

Used functions:

File: Sweep_on_value_of_sigma.m

```
1: function [BER_values, num_errors] = Sweep_on_value_of_sigma(lineCodeVec, voltage,
timeVec, coding_scheme, noSamplesPerBit, noOfBits, bit_stream)
2:     % generate 10 ranges for sigma that range from 0 to the maximum supply voltage
3:     sigma_ranges = linspace(0, voltage, 10);
4:
5:     % pre-allocate a vector to store the BER values & num_errors
6:     BER_values = zeros(1, 10);
7:     num_errors = zeros(1, 10);
8:
9:     % loop choose each value of sigma by index i
10:    for i = 1:10
11:        % add noise to signal
12:        received_signal_with_noise = add_noise_to_linecoding(lineCodeVec,
sigma_ranges(i), timeVec);
13:
14:        % path signal through the decision device
15:        Reciever_output = decision_device(received_signal_with_noise, coding_scheme,
voltage, timeVec, noSamplesPerBit, noOfBits);
16:
17:        % this loop is used to convert Reciever output into binary data
18:        binary_data = convert_into_Binary_data(Reciever_output, bit_stream,
noSamplesPerBit);
19:
20:        % calculate the BER & num_errors for this value of sigma
21:        [BER_values(i), num_errors(i)] = BER_device(binary_data, bit_stream); % insert
your code for calculating BER & num_errors here
22:    end
23: end
```

File: add_noise_to_linecoding.m

```
1: function received_signal_with_noise = add_noise_to_linecoding(lineCode, sigma, Vectime)
2:     % this function simulates external noise from the communication channel (telephone
line) or noise from the receiver added to the transmitted line coding
3:
4:     % define your time vector
5:     t = Vectime;
6:
7:     % noise
8:     n = sigma * randn(1, length(t));
9:
10:    % add the noise to your received signal
11:    received_signal_with_noise = lineCode + n;
12: end
```

File: decision_device.m

```
1: function [Reciever_output] = decision_device(received_signal_with_noise, coding_scheme,
voltage, timeVec, noSamplesPerBit, noOfBits)
2:     % to select the type of line coding by coding scheme
3:     switch (coding_scheme)
4:         case 'UniPolarNRZ'
5:             Reciever_output = r_unipolarNRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits);
6:         case 'PolarNRZ'
7:             Reciever_output = r_polarNRZ(received_signal_with_noise, voltage, timeVec,
noSamplesPerBit, noOfBits);
8:         case 'UniPolarRZ'
9:             Reciever_output = r_unipolarRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits);
10:        case 'BiPolarRZ'
11:            Reciever_output = r_bipolarRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits);
12:        case 'ManchesterCoding'
13:            Reciever_output = r_manchesterCoding(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits);
14:    end
15: end

17: function [Reciever_output] = Master_source_and_comparator (received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold_1, threshold_2)
18:     % pre-allocate a vector to store the Reciever_output
19:     Reciever_output = zeros(1, length(timeVec));
20:     switch nargin
21:         case 8
22:             for i = 1:1:noOfBits
23:                 if (received_signal_with_noise(L) < threshold_1)
24:                     for k = M:1:P
25:                         Reciever_output(k) = 0;
26:                     end
27:                 elseif (received_signal_with_noise(L) > threshold_1)
28:                     for k = M:1:P
```



```

29:         Reciever_output(k) = 1;
30:     end
31: end
32:     M = M + noSamplesPerBit;
33:     P = P + noSamplesPerBit;
34:     L = L + noSamplesPerBit;
35: end
36: case 9
37:     for i = 1:1:noOfBits
38:         if (received_signal_with_noise(L) < threshold_2)
39:             for k = M:1:P
40:                 Reciever_output(k) = 1;
41:             end
42:         elseif (received_signal_with_noise(L) > threshold_1)
43:             for k = M:1:P
44:                 Reciever_output(k) = 1;
45:             end
46:         elseif (received_signal_with_noise(L) < threshold_1)
47:             for k = M:1:P
48:                 Reciever_output(k) = 0;
49:             end
50:         end
51:         M = M + noSamplesPerBit;
52:         P = P + noSamplesPerBit;
53:         L = L + noSamplesPerBit;
54:     end
55: end
56: end

58: function [Reciever_output] = r_unipolarNRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits)
59:     % L decision level of timing circuit
60:     % M, P time of bit in our time vector
61:     threshold = voltage / 2;
62:     L = noSamplesPerBit / 2;
63:     M = 1;
64:     P = noSamplesPerBit;
65:     [Reciever_output] = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
66: end

68: function [Reciever_output] = r_polarNRZ(received_signal_with_noise, voltage, timeVec,
noSamplesPerBit, noOfBits)
69:     % L decision level of timing circuit
70:     % M, P time of bit in our time vector
71:     threshold = (voltage + (-1 * (voltage))) / 2;
72:     L = noSamplesPerBit / 2;
73:     M = 1;
74:     P = noSamplesPerBit;
75:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);

```

```

76: end

78: function [Reciever_output] = r_unipolarRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits)
79:     % L decision level of timing circuit
80:     % M, P time of bit in our time vector
81:     threshold = voltage / 2;
82:     L = noSamplesPerBit / 4;
83:     M = 1;
84:     P = noSamplesPerBit;
85:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
86: end

88: function [Reciever_output] = r_bipolarRZ(received_signal_with_noise, voltage, timeVec,
noSamplesPerBit, noOfBits)
89:     % L decision level of timing circuit
90:     % M, P time of bit in our time vector
91:     threshold_1 = voltage / 2;
92:     threshold_2 = (-1 * (voltage)) / 2;
93:     L = noSamplesPerBit / 4;
94:     M = 1;
95:     P = noSamplesPerBit;
96:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold_1, threshold_2);
97: end

99: function [Reciever_output] = r_manchesterCoding(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits)
100:     % L decision level of timing circuit
101:     % M, P time of bit in our time vector
102:     threshold = (voltage + (-1 * (voltage))) / 2;
103:     L = noSamplesPerBit / 4;
104:     M = 1;
105:     P = noSamplesPerBit;
106:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
107: end

```

File: convert_into_Binary_data.m

```
1: function [binary_data] = convert_into_Binary_data(Reciever_output, bit_stream,
noSamplesPerBit)
2:     % pre-allocate a vector to store the binary_data
3:     binary_data = zeros(1, length(bit_stream));
4:
5:     L = noSamplesPerBit / 2;
6:
7:     for P = 1:1:length(bit_stream)
8:         if (Reciever_output(L) == 0)
9:             binary_data(P) = 0;
10:        elseif (Reciever_output(L) == 1)
11:            binary_data(P) = 1;
12:        end
13:
14:        L = L + noSamplesPerBit;
15:    end
16: end
```

File: BER_device.m

```
1: function [BER, num_errors] = BER_device(Reciever_output, bit_stream)
2:     % count the number of errors & calculate BER
3:     num_errors = sum(Reciever_output ~= bit_stream);
4:     BER = num_errors / length(bit_stream);
5: end
```

Script:

File: part1.m

```
218: figure(19);
219: sigma_ranges = linspace(0, voltage, 10);
220: [BER_values1, num_errors1] = Sweep_on_value_of_sigma(lineCodeVec1, voltage, timeVec1,
coding_scheme1, noSamplesPerBit, noOfBits, bit_stream);
221: [BER_values2, num_errors2] = Sweep_on_value_of_sigma(lineCodeVec2, voltage, timeVec2,
coding_scheme2, noSamplesPerBit, noOfBits, bit_stream);
222: [BER_values3, num_errors3] = Sweep_on_value_of_sigma(lineCodeVec3, voltage, timeVec3,
coding_scheme3, noSamplesPerBit, noOfBits, bit_stream);
223: [BER_values4, num_errors4] = Sweep_on_value_of_sigma(lineCodeVec4, voltage, timeVec4,
coding_scheme4, noSamplesPerBit, noOfBits, bit_stream);
224: [BER_values5, num_errors5] = Sweep_on_value_of_sigma(lineCodeVec5, voltage, timeVec5,
coding_scheme5, noSamplesPerBit, noOfBits, bit_stream);
225:
226: semilogy(sigma_ranges, BER_values1, sigma_ranges, BER_values2, sigma_ranges,
BER_values3, sigma_ranges, BER_values4, sigma_ranges, BER_values5, 'LineWidth', 2);
227:
228: grid on;
229: xlabel('Sigma');
230: ylabel('BER');
231: legend({'UniPolarNRZ', 'PolarNRZ', 'UniPolarRZ', 'BiPolarRZ', 'ManchesterCoding'});
```

Snapshots:

Workspace:

Workspace				
Workspace				
Filter <input type="text"/>				
Name	Class	Dimension	Value	Attribute
BER_values1	double	1x10	[0, 0, 0.010900, ...	
BER_values2	double	1x10	[0, 0, 0, 1.3000e-...	
BER_values3	double	1x10	[0, 0, 0.012900, ...	
BER_values4	double	1x10	[0, 0, 0.018600, ...	
BER_values5	double	1x10	[0, 0, 0, 1.0000e-...	
bitPeriod	double	1x1	1	
bit_stream	double	1x10000	[1, 0, 1, 0, 0, 0, 1,...	
coding_scheme1	char	1x11	UniPolarNRZ	
coding_scheme2	char	1x8	PolarNRZ	
coding_scheme3	char	1x10	UniPolarRZ	
coding_scheme4	char	1x9	BiPolarRZ	
coding_scheme5	char	1x16	ManchesterCodi...	
f1	double	1x2000000	-100:0.0001:99.9...	
f2	double	1x2000000	-100:0.0001:99.9...	
f3	double	1x2000000	-100:0.0001:99.9...	
f4	double	1x2000000	-100:0.0001:99.9...	
f5	double	1x2000000	-100:0.0001:99.9...	
lineCodeVec1	double	1x2000000	[1.2000, 1.2000, ...	
lineCodeVec2	double	1x2000000	[1.2000, 1.2000, ...	
lineCodeVec3	double	1x2000000	[1.2000, 1.2000, ...	
lineCodeVec4	double	1x2000000	[1.2000, 1.2000, ...	
lineCodeVec5	double	1x2000000	[1.2000, 1.2000, ...	
noBitsPerSegme...	double	1x1	3	
noOfBits	double	1x1	10000	
noSamplesPerBit	double	1x1	200	
num_errors1	double	1x10	[0, 0, 109, 657, 1...	
num_errors2	double	1x10	[0, 0, 0, 13, 113, ...	
num_errors3	double	1x10	[0, 0, 129, 646, 1...	
num_errors4	double	1x10	[0, 0, 186, 999, 1...	
num_errors5	double	1x10	[0, 0, 0, 10, 121, ...	
sigma_ranges	double	1x10	[0, 0.1333, 0.266...	
spectral_of_line...	double	1x2000000	[0, 4.6797e-17, ...	
spectral_of_line...	double	1x2000000	[0, 1.8719e-16, ...	
spectral_of_line...	double	1x2000000	[0, 1.1699e-17, ...	
spectral_of_line...	double	1x2000000	[0, 9.3060e-21, ...	
spectral_of_line...	double	1x2000000	[0, 4.6187e-24, ...	
timeVec1	double	1x2000000	0:0.005:10000	
timeVec2	double	1x2000000	0:0.005:10000	
timeVec3	double	1x2000000	0:0.005:10000	
timeVec4	double	1x2000000	0:0.005:10000	
timeVec5	double	1x2000000	0:0.005:10000	
voltage	double	1x1	1.2000	

num_errors vectors:

num_errors1 [1x10 double]											
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	109	657	1331	1870	2268	2546	2891	3095	
2											

num_errors2 [1x10 double]											
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	13	113	357	652	997	1324	1544	
2											

num_errors3 [1x10 double]											
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	129	646	1266	1850	2258	2647	2833	3099	
2											

num_errors4 [1x10 double]											
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	186	999	1918	2734	3320	3794	4121	4355	
2											

num_errors5 [1x10 double]											
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	10	121	383	659	979	1308	1574	
2											

Editor Command Window Documentation Variable Editor

sigame_ranges and BER_values vectors:

sigma_ranges [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

BER_values1 [1x10 double]										
	1	2	3	4	5	6	7	8	9	10
1	0	0	0.0109	0.0657	0.1331	0.187	0.2268	0.2546	0.2891	0.3095

BER_values2 [1x10 double]											
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0.0013	0.0113	0.0357	0.0652	0.0997	0.1324	0.1544	

BER_values3 [1x10 double]										
	1	2	3	4	5	6	7	8	9	10
1	0	0	0.0129	0.0646	0.1266	0.185	0.2258	0.2647	0.2833	0.3099

BER_values4 [1x10 double]											
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0.0186	0.0999	0.1918	0.2734	0.332	0.3794	0.4121	0.4355	

BER_values5 [1x10 double]											
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0.001	0.0121	0.0383	0.0659	0.0979	0.1308	0.1574	

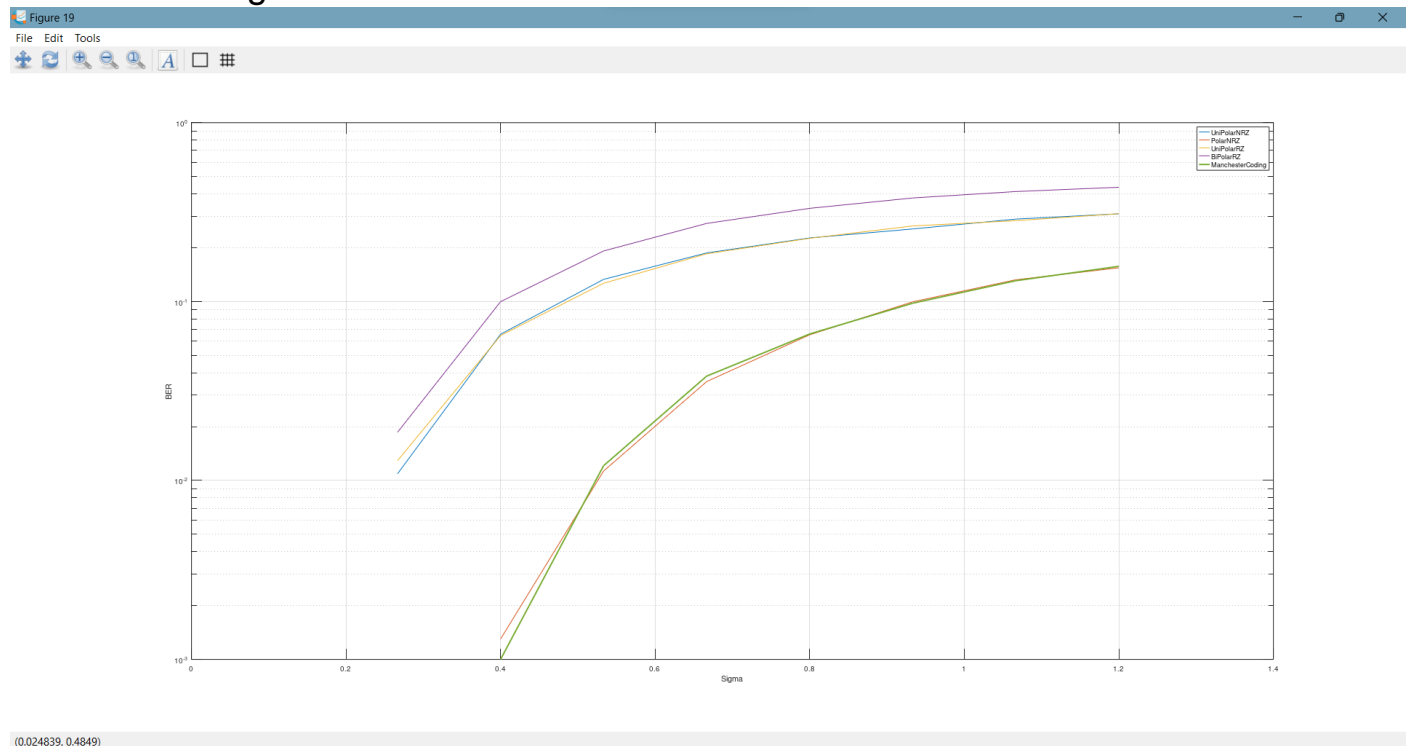
Editor

Command Window

Documentation

Variable Editor

Plot BER vs sigma:



(0.024839, 0.4849)

Step 11: **(Bonus)** For the case of Bipolar return to zero , design an error detection circuit:

Code:

Functions:

File: Bonus_Sweep_on_value_of_sigma.m

```
1: function [number_of_detected_errors] = Bonus_Sweep_on_value_of_sigma(lineCodeVec,  
voltage, timeVec, coding_scheme, noSamplesPerBit, noOfBits)  
2:     % Generate 10 ranges for sigma that range from 0 to the maximum supply voltage  
3:     sigma_ranges = linspace(0, voltage, 10);  
4:  
5:     % Pre-allocate a vector to store the BER values & num_errors  
6:     number_of_detected_errors = zeros(1, 10);  
7:  
8:     % Loop chooses each value of sigma by index i  
9:     for i = 1:10  
10:         % Add noise to signal  
11:         received_signal_with_noise = add_noise_to_linecoding(lineCodeVec,  
sigma_ranges(i), timeVec);  
12:  
13:         % Path signal through Regenerative_Repeater  
14:         [Repeater_output] = Regenerative_Repeater(received_signal_with_noise,  
coding_scheme, voltage, timeVec, noSamplesPerBit, noOfBits);  
15:  
16:         % Calculate number_of_detected_errors for this value of sigma  
17:         number_of_detected_errors(i) = Error_Detection_Circuit(Repeater_output,  
voltage, noSamplesPerBit, noOfBits);  
18:     end  
19: end
```

File: Regenerative_Repeater.m

```
1: function [Repeater_output] = Regenerative_Repeater(received_signal_with_noise,  
coding_scheme, voltage, timeVec, noSamplesPerBit, noOfBits)  
2:     % This function represents the regenerative repeater block used in telephone lines.  
3:     % When entering the receiver to select the type of line coding by coding scheme.  
4:     switch (coding_scheme)  
5:         case 'BiPolarRZ'  
6:             Repeater_output = r_bipolarRZ(received_signal_with_noise, voltage, timeVec,  
noSamplesPerBit, noOfBits);  
7:         end  
8: end  
  
10: function [Repeater_output] = r_bipolarRZ(received_signal_with_noise, voltage, timeVec,  
noSamplesPerBit, noOfBits)  
11:     % Pre-allocate a vector to store the Receiver_output  
12:     Repeater_output = zeros(1, length(timeVec));  
13:     % L decision level of timing circuit  
14:     % M, P time of bit in our time vector  
15:     threshold_1 = voltage / 2;  
16:     threshold_2 = (-1 * (voltage)) / 2;  
17:     L = noSamplesPerBit / 4;  
18:     M = 1;  
19:     P = noSamplesPerBit / 2;  
20:     for i = 1:1:noOfBits  
21:         if (received_signal_with_noise(L) < threshold_2)  
22:             for k = M:1:P  
23:                 Repeater_output(k) = -1 * voltage;  
24:             end  
25:         elseif (received_signal_with_noise(L) > threshold_1)  
26:             for k = M:1:P  
27:                 Repeater_output(k) = voltage;  
28:             end  
29:         elseif (received_signal_with_noise(L) < threshold_1)  
30:             for k = M:1:P  
31:                 Repeater_output(k) = 0;  
32:             end  
33:         end  
34:         M = M + noSamplesPerBit;  
35:         P = P + noSamplesPerBit;  
36:         L = L + noSamplesPerBit;  
37:     end  
38: end
```

File: Error_Detection_Circuit.m

```
1: function [number_of_detected_errors] = Error_Detection_Circuit(Repeater_output,
voltage, noSamplesPerBit, noOfBits)
2:     % Error detection circuit for Bipolar return to zero
3:     number_of_detected_errors = 0;
4:     L = noSamplesPerBit / 4;
5:
6:     for i = 1:1:noOfBits
7:         if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
8:             break;
9:         end
10:
11:         % 1000000000...01
12:         if (Repeater_output(L) == voltage && Repeater_output(L + noSamplesPerBit) ==
0)
13:             if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
14:                 break;
15:             else
16:                 L = L + noSamplesPerBit;
17:             end
18:
19:             while (Repeater_output(L) == 0)
20:                 if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
21:                     break;
22:                 else
23:                     L = L + noSamplesPerBit;
24:                 end
25:             end
26:
27:             if (Repeater_output(L) == voltage)
28:                 number_of_detected_errors = number_of_detected_errors + 1;
29:             end
30:             % -100000000...0-1
31:             elseif (Repeater_output(L) == -1 * voltage && Repeater_output(L +
noSamplesPerBit) == 0)
32:                 if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
33:                     break;
34:                 else
35:                     L = L + noSamplesPerBit;
36:                 end
37:
38:                 while (Repeater_output(L) == 0)
39:                     if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >=
noOfBits * noSamplesPerBit)
40:                         break;
41:                     else
42:                         L = L + noSamplesPerBit;
```



```

43:         end
44:     end
45:
46:     if (Repeater_output(L) == -1 * voltage)
47:         number_of_detected_errors = number_of_detected_errors + 1;
48:     end
49:
50:     % -1-1
51:     elseif (Repeater_output(L) == -1 * voltage && Repeater_output(L +
noSamplesPerBit) == -1 * voltage)
52:         number_of_detected_errors = number_of_detected_errors + 1;
53:         if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
54:             break;
55:         else
56:             L = L + noSamplesPerBit;
57:         end
58:
59:     % 11
60:     elseif (Repeater_output(L) == voltage && Repeater_output(L + noSamplesPerBit)
== voltage)
61:         number_of_detected_errors = number_of_detected_errors + 1;
62:         if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
63:             break;
64:         else
65:             L = L + noSamplesPerBit;
66:         end
67:     end
68:
69:     if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
70:         break;
71:     else
72:         L = L + noSamplesPerBit;
73:     end
74: end
75: end

```

Snapshots: Workspace:

Workspace				
Filter <input type="checkbox"/>				
Name	Class	Dimension	Value	Attribute
bitPeriod	double	1x1	1	
noBitsPerSegments	double	1x1	3	
noOfBits	double	1x1	10000	
noSamplesPerBit	double	1x1	200	
voltage	double	1x1	1.2000	
BER_values1	double	1x10	[0, 0, 0.010900, ...	
BER_values2	double	1x10	[0, 0, 0, 1.3000e-...	
BER_values3	double	1x10	[0, 0, 0.012900, ...	
BER_values4	double	1x10	[0, 0, 0.018600, ...	
BER_values5	double	1x10	[0, 0, 0, 1.0000e-...	
coding_scheme3	char	1x10	UniPolarRZ	
num_errors1	double	1x10	[0, 0, 109, 657, 1...	
num_errors2	double	1x10	[0, 0, 0, 13, 113, ...	
num_errors3	double	1x10	[0, 0, 129, 646, 1...	
num_errors4	double	1x10	[0, 0, 186, 999, 1...	
num_errors5	double	1x10	[0, 0, 0, 10, 121, ...	
number_of_detected_errors	double	1x10	[0, 0, 123, 551, 9...	
sigma_ranges	double	1x10	[0, 0.1333, 0.266...	
bit_stream	double	1x10000	[1, 0, 1, 0, 0, 0, 1,...	
coding_scheme1	char	1x11	UniPolarNRZ	
coding_scheme5	char	1x16	ManchesterCodi...	
f1	double	1x2000000	-100:0.0001:99.9...	lineCodeVec5 double 1x2000000 [1.2000, 1.2000, ...
f2	double	1x2000000	-100:0.0001:99.9...	spectral_of_lineCodeVec1 double 1x2000000 [0, 4.6797e-17, ...
f3	double	1x2000000	-100:0.0001:99.9...	spectral_of_lineCodeVec2 double 1x2000000 [0, 1.8719e-16, ...
f4	double	1x2000000	-100:0.0001:99.9...	spectral_of_lineCodeVec3 double 1x2000000 [0, 1.1699e-17, ...
f5	double	1x2000000	-100:0.0001:99.9...	spectral_of_lineCodeVec4 double 1x2000000 [0, 9.3060e-21, ...
lineCodeVec1	double	1x2000000	[1.2000, 1.2000, ...	spectral_of_lineCodeVec5 double 1x2000000 [0, 4.6187e-24, ...
lineCodeVec2	double	1x2000000	[1.2000, 1.2000, ...	timeVec1 double 1x2000000 0:0.005:10000
lineCodeVec3	double	1x2000000	[1.2000, 1.2000, ...	timeVec2 double 1x2000000 0:0.005:10000
lineCodeVec4	double	1x2000000	[1.2000, 1.2000, ...	timeVec3 double 1x2000000 0:0.005:10000
				timeVec4 double 1x2000000 0:0.005:10000
				timeVec5 double 1x2000000 0:0.005:10000
				coding_scheme2 char 1x8 PolarNRZ
				coding_scheme4 char 1x9 BiPolarRZ

Number_of_detected_errors Vector:

number_of_detected_errors [1x10 double]											
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	123	551	948	1187	1465	1562	1716	1853	
2											
3											
4											

Part2: binary phase shift-keying:

Transmitter:

Step 1 : Generate stream of random bits:

Code:

Used functions:

File: generate_random_bits.m

```
1: function bit_stream = generate_random_bits(noOfBits)
2:     % Generate random sequence of binary data zeros and ones
3:
4:     bit_stream = randi([0, 1], 1, noOfBits);
5: end
```

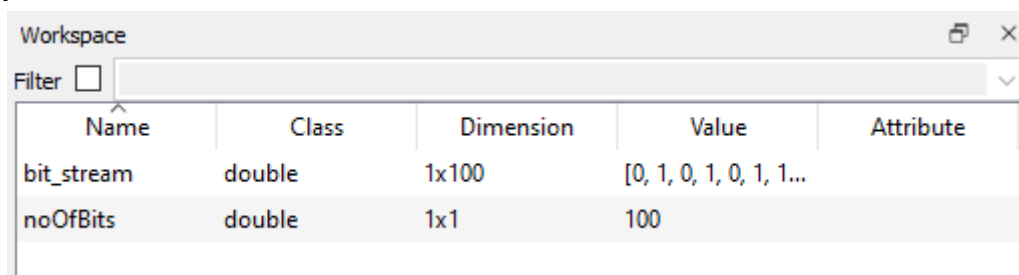
Script:

File: part2.m

```
1: % MATLAB Script for a Binary PSK (passband communication)
2:
3: % Clear all variables and close all figures
4: clear all;
5: close all;
6:
7: noOfBits = 100; % Number of time samples
8:
9: % Generate bit stream
10: bit_stream = generate_random_bits(noOfBits);
```

Snapshots:

workspace:



The image shows a screenshot of the MATLAB Workspace window. It contains a table with the following data:

Name	Class	Dimension	Value	Attribute
bit_stream	double	1x100	[0, 1, 0, 1, 0, 1, 1...	
noOfBits	double	1x1	100	

Step 2 : Line coding:

Code:

Used functions:

File: line_coding.m

```
1: function [lineCodeVec, timeVec] = line_coding(bit_stream, coding_scheme, voltage,
bitPeriod, noSamplesPerBit, fc)
2:     switch nargin
3:         % Choose according to number of input arguments
4:         case 5
5:             [lineCodeVec, timeVec] = Baseband_communication(bit_stream, coding_scheme,
voltage, bitPeriod, noSamplesPerBit);
6:         case 6
7:             % Belongs to Part 2
8:             [lineCodeVec, timeVec] = Passband_communication(bit_stream, coding_scheme,
voltage, bitPeriod, noSamplesPerBit, fc);
9:         end
10:    end
11:
12: function [lineCodeVec, timeVec] = Passband_communication(bit_stream, coding_scheme,
voltage, bitPeriod, noSamplesPerBit, fc)
13:     % This function implements the function of polarNRZ block in transmitter
14:
15:     % Calculations
16:     noOfBits = length(bit_stream);
17:     fs = 10 * fc; % Sampling frequency - Sampling rate - This will define the
resolution
18:     ts = 1 / fs; % Time step
19:     t = 0:ts:(noOfBits - 1) * ts;
20: % Generate time domain
21: timeVec = [];
22: for ii = 1:length(bit_stream)
23: timeVec = [timeVec t];
24: t = t + (noOfBits - 1) * ts;
25: end
26:
27: noSamplesPerBit = length(t);
28:
29: % Generate polarNRZ line coding
30: switch (coding_scheme)
31: case 'PolarNRZ'
32: lineCodeVec = polarNRZ(bit_stream, voltage, timeVec, noSamplesPerBit);
33: end
34: end
```

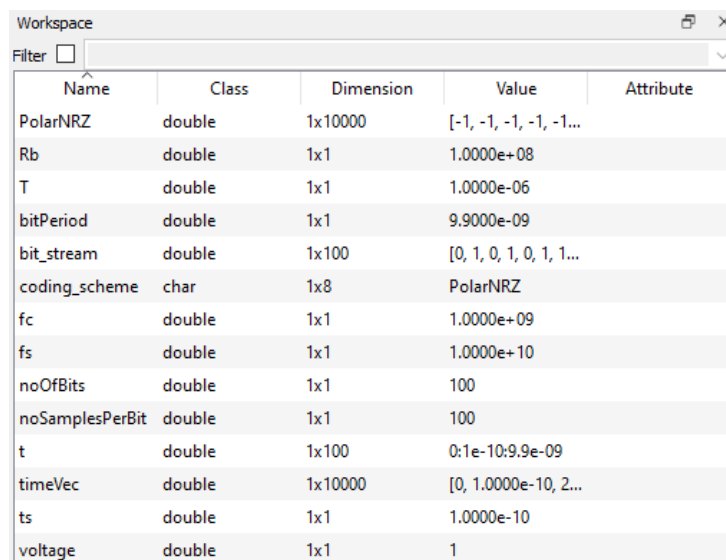
Script:

File: part2.m

```
11: %% Calculations
12: fc = 10^9; % Frequency of Modulating Signal
13: fs = 10 * fc; % Sampling frequency - Sampling rate - This will define the resolution
14: ts = 1 / fs; % Time step
15: t = 0:ts:(noOfBits - 1) * ts; % Time for one bit - Time of one bit = 1 second per one bit
16: T = (noOfBits) * (noOfBits * ts); % Simulation time
17: Rb = noOfBits / T; % Bit rate = N / simulation time
18:
19: voltage = 1;
20: bitPeriod = t(end);
21: noSamplesPerBit = length(t);
22: coding_scheme = 'PolarNRZ';
23:
24: %% The PolarNRZ Signal
25: [PolarNRZ, timeVec] = line_coding(bit_stream, coding_scheme, voltage, bitPeriod, noSamplesPerBit, fc);
26:
27: % Plot the PolarNRZ Signal with time
28: figure;
29: plot(timeVec, PolarNRZ, 'r', 'LineWidth', 2);
30: xlabel('Time (s)');
31: ylabel('Amplitude (V)');
32: title('PolarNRZ Signal');
33: axis([0 timeVec(end) -1.5 1.5]);
34: grid on;
```

Snapshots:

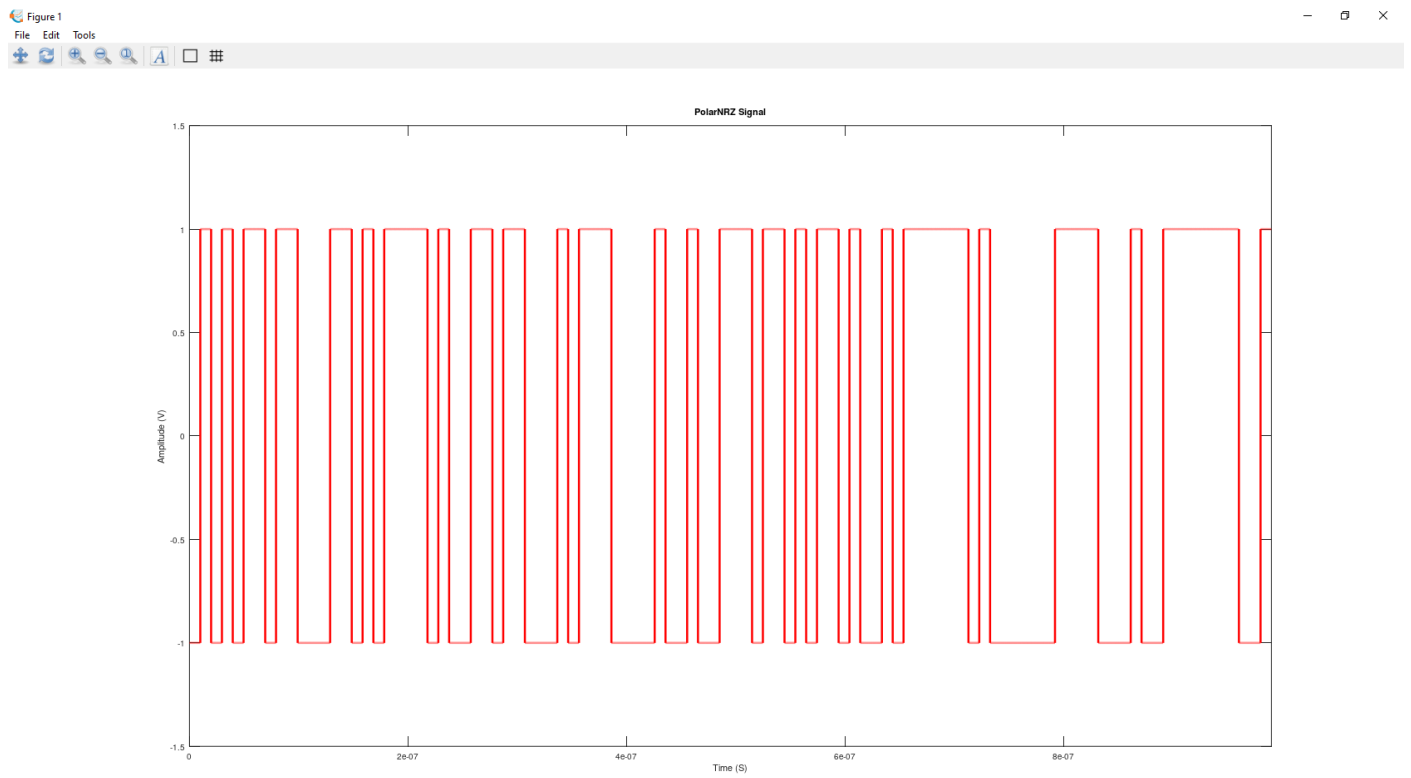
workspace:



The screenshot shows the MATLAB Workspace window with a table of variables. The table has five columns: Name, Class, Dimension, Value, and Attribute. The variables listed are PolarNRZ, Rb, T, bitPeriod, bit_stream, coding_scheme, fc, fs, noOfBits, noSamplesPerBit, t, timeVec, ts, and voltage.

Name	Class	Dimension	Value	Attribute
PolarNRZ	double	1x10000	[-1, -1, -1, -1...	
Rb	double	1x1	1.0000e+08	
T	double	1x1	1.0000e-06	
bitPeriod	double	1x1	9.9000e-09	
bit_stream	double	1x100	[0, 1, 0, 1, 0, 1, 1...	
coding_scheme	char	1x8	PolarNRZ	
fc	double	1x1	1.0000e+09	
fs	double	1x1	1.0000e+10	
noOfBits	double	1x1	100	
noSamplesPerBit	double	1x1	100	
t	double	1x100	0:1e-10:9.9e-09	
timeVec	double	1x10000	[0, 1.0000e-10, 2...	
ts	double	1x1	1.0000e-10	
voltage	double	1x1	1	

plot of time domain with PolarNRZ signal:



Step 3 : spectral domain before Modulation:

Code:

Used functions:

File: spectral_domain.m

```
1: function [spectral, f, BW, Spec_Original_line_coding] = spectral_domain(lineCodeVec,
bit_stream, noSamplesPerBit, bitPeriod, fc)
2:     switch nargin
3:         % Choose according to number of input arguments
4:         case 4
5:             % Belongs to Part 1
6:             [spectral, f] = Baseband_communication(lineCodeVec, bit_stream,
noSamplesPerBit, bitPeriod);
7:         case 5
8:             % Belongs to Part 2
9:             [spectral, f, BW, Spec_Original_line_coding] =
Passband_communication(lineCodeVec, bit_stream, noSamplesPerBit, bitPeriod, fc);
10:     end
11: end
12:
13: function [spectral, f, BW, Spec_Original_line_coding] =
Passband_communication(lineCodeVec, bit_stream, noSamplesPerBit, bitPeriod, fc)
14:     % Calculation to generate frequency domain
15:     noOfBits = length(bit_stream);
16:     fs = 10 * fc; % Sampling frequency - Sampling rate - This will define the
resolution
17:     ts = 1 / fs; % Time step
18:     T = (noOfBits) * (I apologize for the incomplete response in my previous message.
Here's the remainder of the reformatted code starting from line 18 with line
numbers:noOfBits * ts); % Simulation time
19:     Rb = noOfBits / T; % Bit rate = N / simulation time
20:     BW = Rb; % Polar & NRZ
21:     df = 1 / T; % Frequency step
22:
23: % Spectrum of Original Digital Signal
24: Spec_Original_line_coding = (fftshift(fft(lineCodeVec))) / noOfBits; % We put message
in frequency domain because it only shifts function in FFT
25:
26: % Frequency domain
27: if (rem(noOfBits,2) == 0) % Even
28:     f = ((- (0.5 * fs)):df:((0.5 * fs) - df)); % Frequency vector if x/f even
29: else % Odd
30:     f = (- (0.5 * fs - 0.5 * df)) : df : (((0.5 * fs) + 1) - 0.5 * df); % Frequency vector
if X/f is odd
31: end
32:
33: % Power spectral of Original Digital Signal
34: spectral = abs(Spec_Original_line_coding).^2;
35: end
```

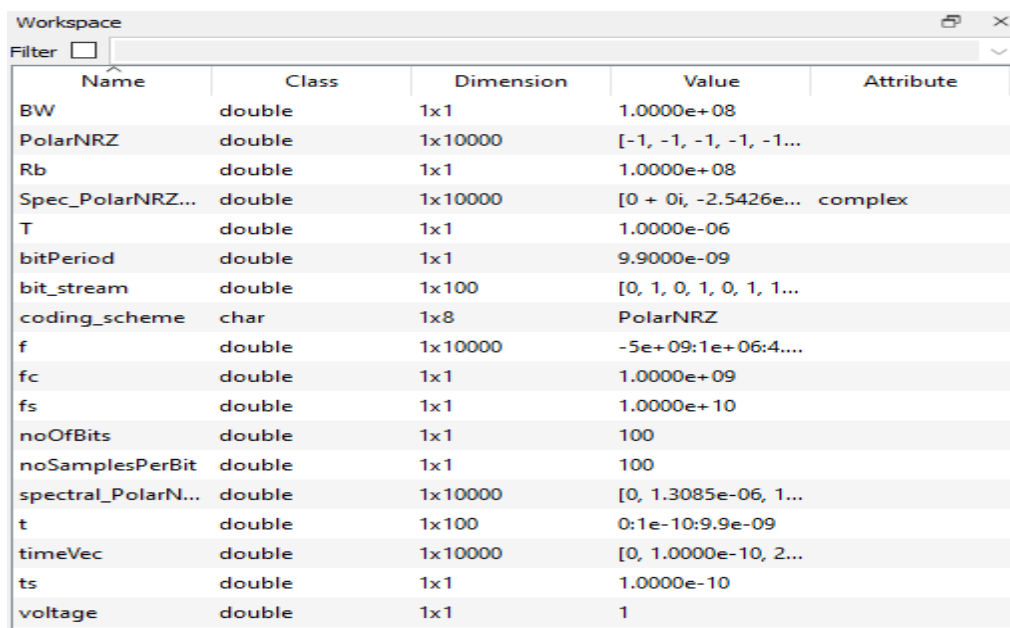
Script:

File: part2.m

```
36: % Spectrum & Spectral of PolarNRZ Signal
37:
38: [spectral_PolarNRZ_Signal, f, BW, Spec_PolarNRZ_Signal] = spectral_domain(PolarNRZ,
bit_stream, noSamplesPerBit, bitPeriod, fc);
39:
40: figure;
41: grid on;
42:
43: plot(f, abs(Spec_PolarNRZ_Signal));
44: title('Spectrum of Original PolarNRZ Signal');
45: xlabel("Frequency (Hz)");
46: ylabel('Amplitude (V)');
47:
48: figure;
49: plot(f, spectral_PolarNRZ_Signal);
50: title('Power Spectral Density of PolarNRZ Signal');
51: xlabel("Frequency (Hz)");
52: ylabel('Power Spectral Density');
```

Snapshots:

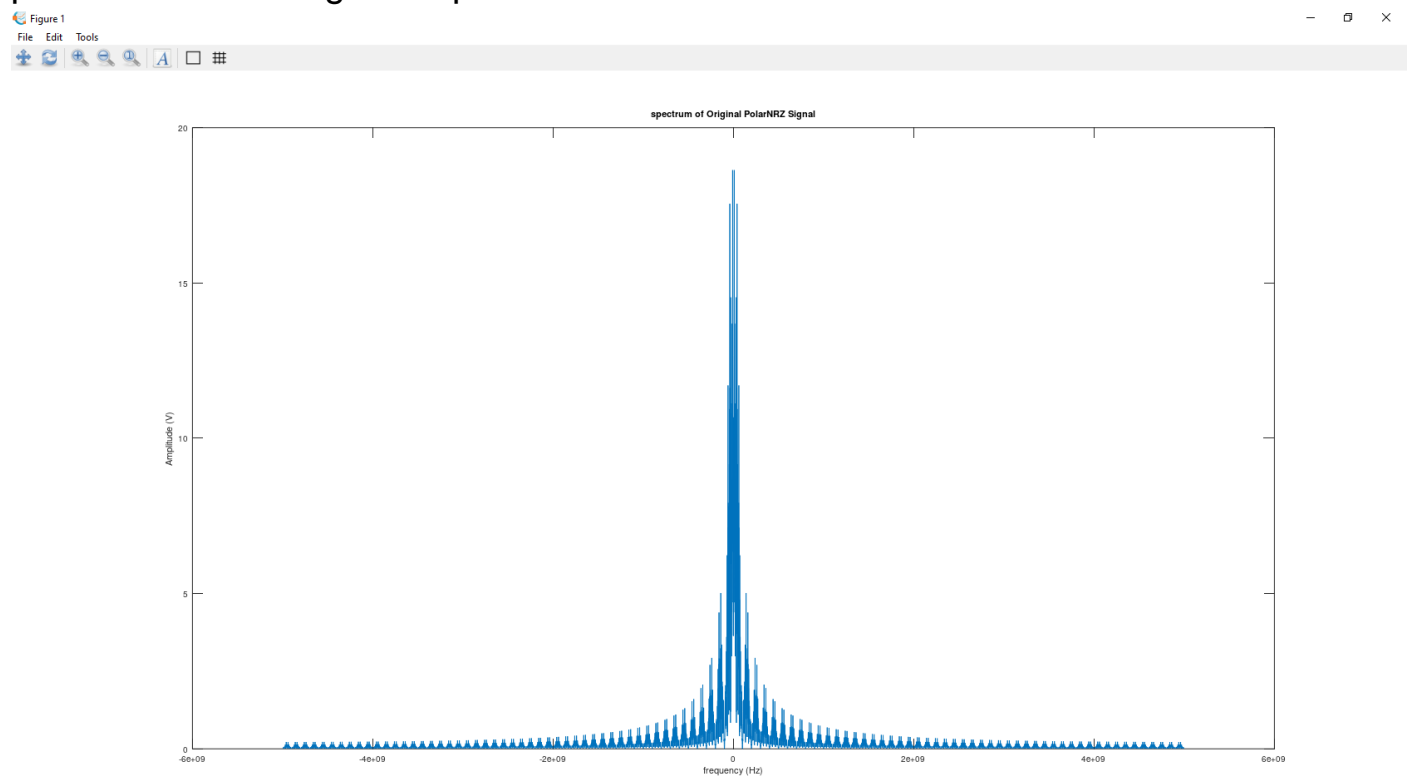
workspace:



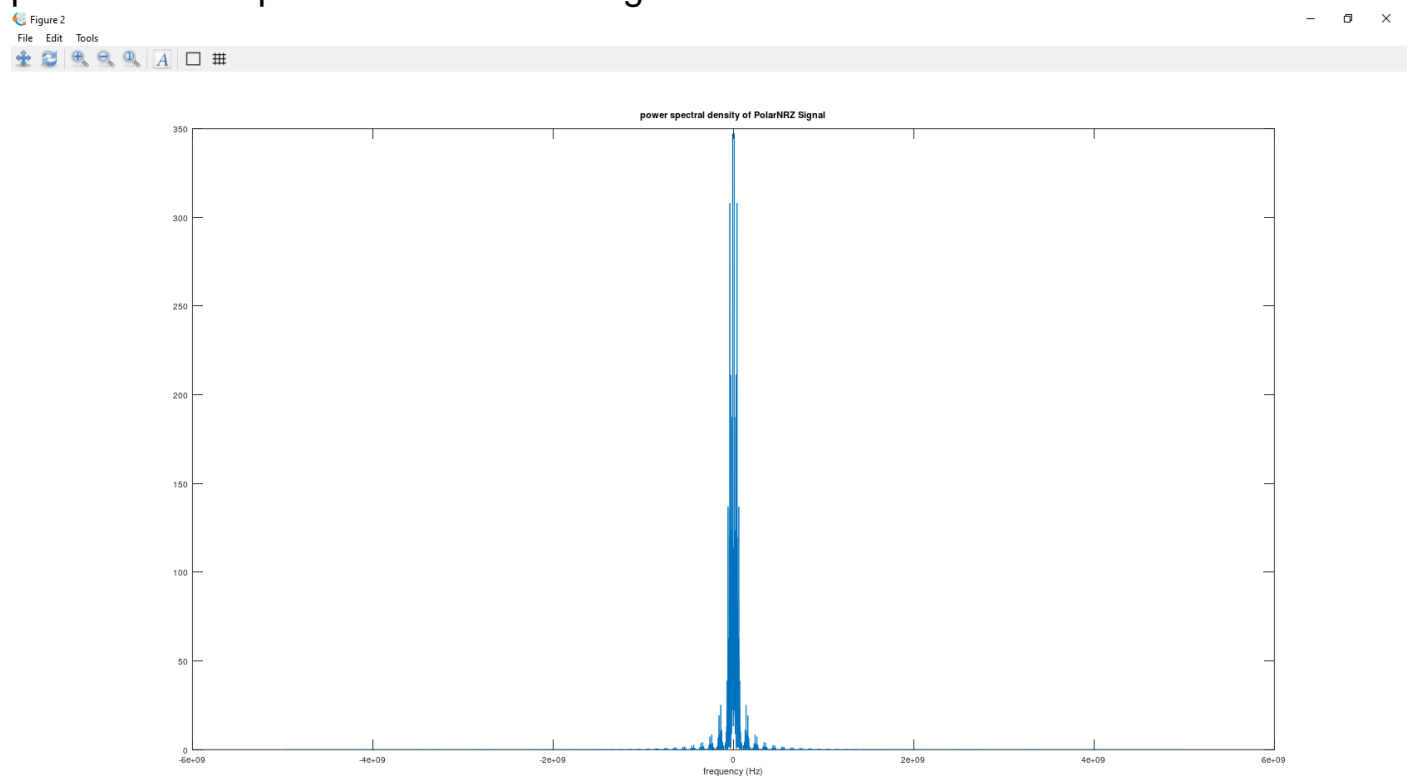
The screenshot shows the MATLAB Workspace window with a table of variables. The table has five columns: Name, Class, Dimension, Value, and Attribute. The variables listed are BW, PolarNRZ, Rb, Spec_PolarNRZ..., T, bitPeriod, bit_stream, coding_scheme, f, fc, fs, noOfBits, noSamplesPerBit, spectral_PolarN..., t, timeVec, ts, and voltage.

Name	Class	Dimension	Value	Attribute
BW	double	1x1	1.0000e+08	
PolarNRZ	double	1x10000	[-1, -1, -1, -1, -1...	
Rb	double	1x1	1.0000e+08	
Spec_PolarNRZ...	double	1x10000	[0 + 0i, -2.5426e...	complex
T	double	1x1	1.0000e-06	
bitPeriod	double	1x1	9.9000e-09	
bit_stream	double	1x100	[0, 1, 0, 1, 0, 1, 1...	
coding_scheme	char	1x8	PolarNRZ	
f	double	1x10000	-5e+09:1e+06:4....	
fc	double	1x1	1.0000e+09	
fs	double	1x1	1.0000e+10	
noOfBits	double	1x1	100	
noSamplesPerBit	double	1x1	100	
spectral_PolarN...	double	1x10000	[0, 1.3085e-06, 1...	
t	double	1x100	0:1e-10:9.9e-09	
timeVec	double	1x10000	[0, 1.0000e-10, 2...	
ts	double	1x1	1.0000e-10	
voltage	double	1x1	1	

plot of PolarNRZ signal's spectrum:



plot of Power spectral of PolarNRZ signal:



Step 4 : time domain after Modulation:

Code:

Script:

File: part2.m

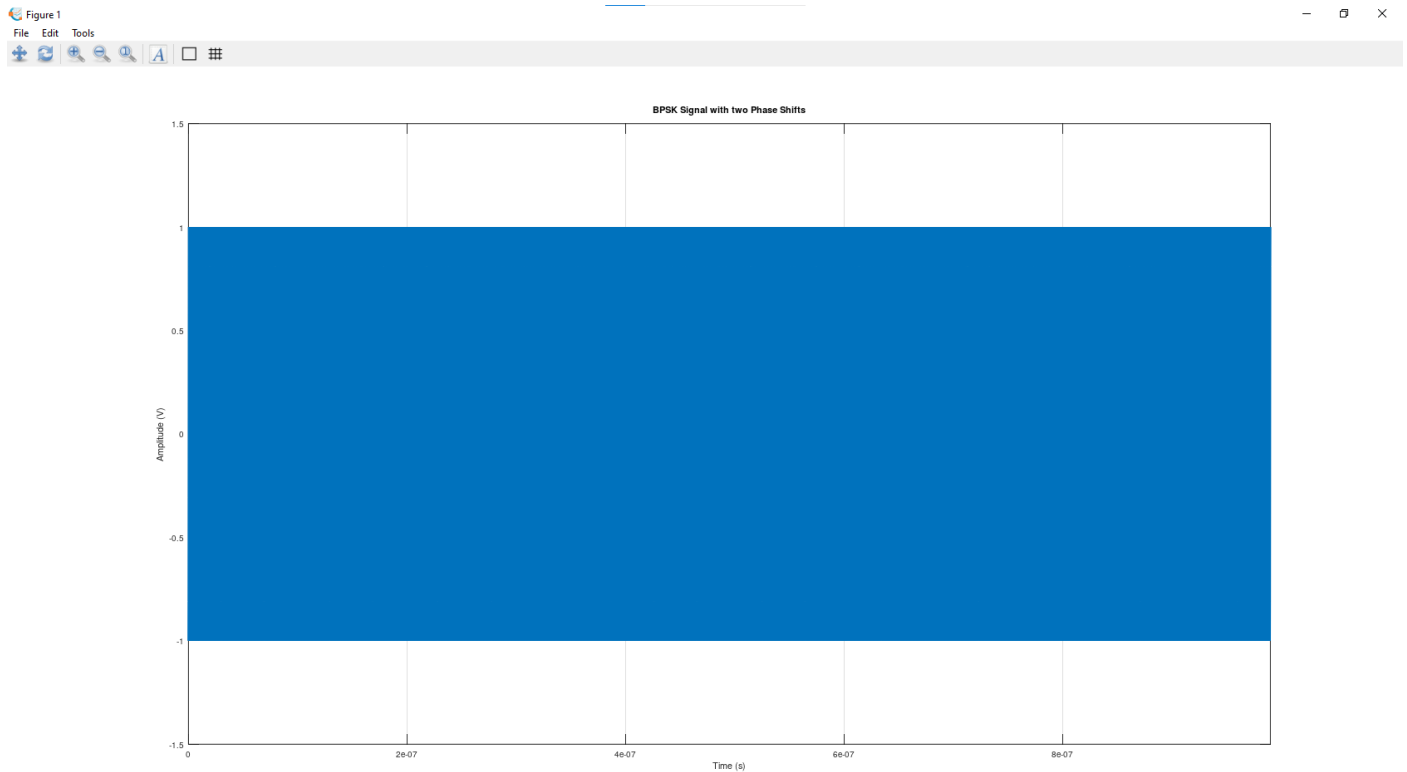
```
54: % The BPSK Signal (modulated signal of PolarNRZ Signal after product modulator in
transmitter circuit)
55: BPSK_signal = PolarNRZ .* cos(2 * pi * fc * timeVec);
56:
57: % Plot the BPSK Signal (line coding) with time
58: figure;
59: plot(timeVec, BPSK_signal, 'LineWidth', 2);
60: xlabel('Time (s)');
61: ylabel('Amplitude (V)');
62: title('BPSK Signal with Two Phase Shifts');
63: axis([0 timeVec(end) -1.5 1.5]);
64: grid on;
```

Snapshots:

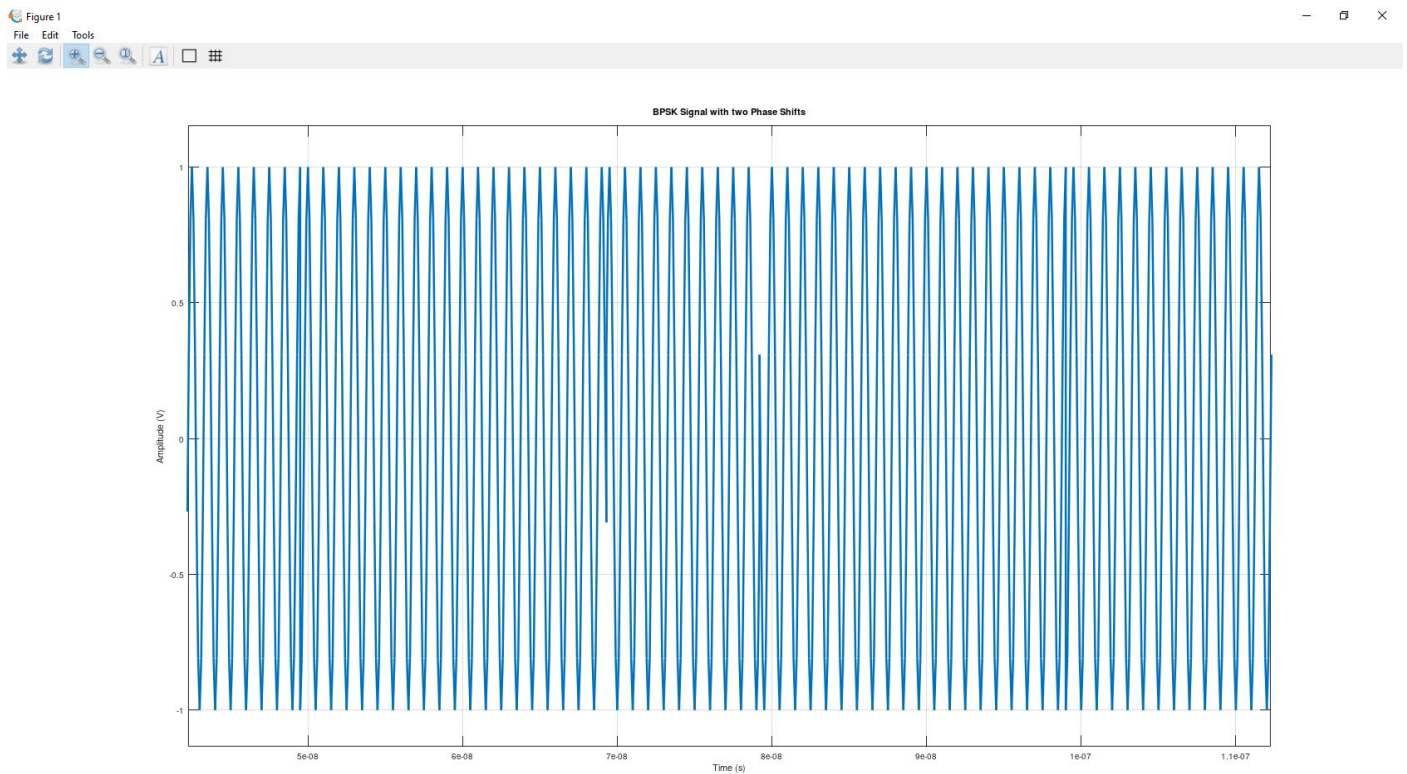
workspace:

Workspace				
Filter <input type="checkbox"/>				
Name	Class	Dimension	Value	Attribute
BPSK_signal	double	1x10000	[-1, -0.8090, -0....	
BW	double	1x1	1.0000e+08	
PolarNRZ	double	1x10000	[-1, -1, -1, -1, -1...	
Rb	double	1x1	1.0000e+08	
Spec_PolarNRZ...	double	1x10000	[0 + 0i, -2.5426e... complex	
T	double	1x1	1.0000e-06	
bitPeriod	double	1x1	9.9000e-09	
bit_stream	double	1x100	[0, 1, 0, 1, 0, 1, 1...	
coding_scheme	char	1x8	PolarNRZ	
f	double	1x10000	-5e+09:1e+06:4....	
fc	double	1x1	1.0000e+09	
fs	double	1x1	1.0000e+10	
noOfBits	double	1x1	100	
noSamplesPerBit	double	1x1	100	
spectral_PolarN...	double	1x10000	[0, 1.3085e-06, 1...	
t	double	1x100	0:1e-10:9.9e-09	
timeVec	double	1x10000	[0, 1.0000e-10, 2...	
ts	double	1x1	1.0000e-10	
voltage	double	1x1	1	

Plot of BPSK signal with two phase shifts:
Before zoom:



After zoom:



Step 5 : spectrum of the modulated BPSK signal:

Code:

Used functions:

File: spectral_domain.m

```
1: function [spectral, f, BW, Spec_Original_line_coding] = spectral_domain(lineCodeVec,
bit_stream, noSamplesPerBit, bitPeriod, fc)
2:     switch nargin
3:         % Choose according to number of input arguments
4:         case 4
5:             % Belongs to Part 1
6:             [spectral, f] = Baseband_communication(lineCodeVec, bit_stream,
noSamplesPerBit, bitPeriod);
7:         case 5
8:             % Belongs to Part 2
9:             [spectral, f, BW, Spec_Original_line_coding] =
Passband_communication(lineCodeVec, bit_stream, noSamplesPerBit, bitPeriod, fc);
10:     end
11: end
12:
13: function [spectral, f, BW, Spec_Original_line_coding] =
Passband_communication(lineCodeVec, bit_stream, noSamplesPerBit, bitPeriod, fc)
14:     % Calculation to generate frequency domain
15:     noOfBits = length(bit_stream);
16:     fs = 10 * fc; % Sampling frequency - Sampling rate - This will define the
resolution
17:     ts = 1 / fs; % Time step
18:     T = (noOfBits) * (I apologize for the incomplete response in my previous message.
Here's the remainder of the reformatted code starting from line 18 with line
numbers:noOfBits * ts); % Simulation time
19:     Rb = noOfBits / T; % Bit rate = N / simulation time
20:     BW = Rb; % Polar & NRZ
21:     df = 1 / T; % Frequency step
22:
23: % Spectrum of Original Digital Signal
24: Spec_Original_line_coding = (fftshift(fft(lineCodeVec))) / noOfBits; % We put message
in frequency domain because it only shifts function in FFT
25:
26: % Frequency domain
27: if (rem(noOfBits,2) == 0) % Even
28:     f = ((- (0.5 * fs)):df:((0.5 * fs) - df)); % Frequency vector if x/f even
29: else % Odd
30:     f = (- (0.5 * fs - 0.5 * df)) : df : (((0.5 * fs) + 1) - 0.5 * df); % Frequency vector
if X/f is odd
31: end
32:
33: % Power spectral of Original Digital Signal
34: spectral = abs(Spec_Original_line_coding).^2;
35: end
```

Script:

File: part2.m

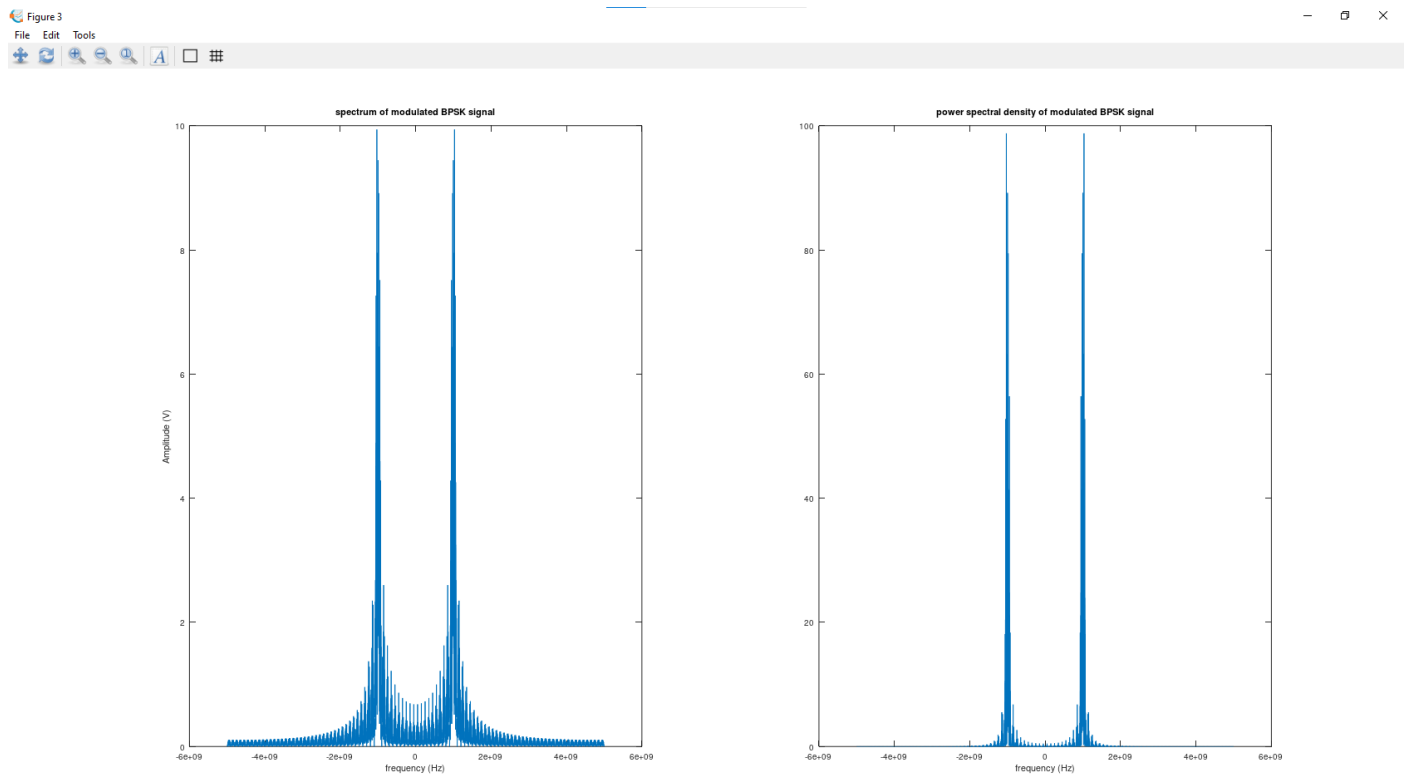
```
66: % Spectrum & Spectral of BPSK signal
67:
68: [spectral_BPSK_signal, f, BW, spec_BPSK_signal] = spectral_domain(BPSK_signal,
bit_stream, noSamplesPerBit, bitPeriod, fc);
69:
70: figure(3);
71: grid on;
72:
73: subplot(1, 2, 1);
74: plot(f, abs(spec_BPSK_signal));
75: title("Spectrum of Modulated BPSK Signal");
76: xlabel("Frequency (Hz)");
77: ylabel('Amplitude (V)');
78:
79: subplot(1, 2, 2);
80: plot(f, spectral_BPSK_signal);
81: title("Power Spectral Density of Modulated BPSK Signal");
82: xlabel("Frequency (Hz)");
83: ylabel('Power Spectral Density');
```

Snapshots:

workspace:

Workspace				
Filter <input type="text"/>				
Name	Class	Dimension	Value	Attribute
BPSK_signal	double	1x10000	[-1, -0.8090, -0....	
BW	double	1x1	1.0000e+08	
PolarNRZ	double	1x10000	[-1, -1, -1, -1, -1...	
Rb	double	1x1	1.0000e+08	
Spec_PolarNRZ...	double	1x10000	[0 + 0i, -2.5426e...	complex
T	double	1x1	1.0000e-06	
bitPeriod	double	1x1	9.9000e-09	
bit_stream	double	1x100	[0, 1, 0, 1, 0, 1, 1...	
coding_scheme	char	1x8	PolarNRZ	
f	double	1x10000	-5e+09:1e+06:4....	
fc	double	1x1	1.0000e+09	
fs	double	1x1	1.0000e+10	
noOfBits	double	1x1	100	
noSamplesPerBit	double	1x1	100	
spec_BPSK_signal	double	1x10000	[-1.0083e-13 + ...	complex
spectral_BPSK_s...	double	1x10000	[1.0166e-26, 3.1...	
spectral_PolarN...	double	1x10000	[0, 1.3085e-06, 1...	
t	double	1x100	0:1e-10:9.9e-09	
timeVec	double	1x10000	[0, 1.0000e-10, 2...	
ts	double	1x1	1.0000e-10	
voltage	double	1x1	1	

Plot of spectrum of modulated BPSK signal & Power of spectral modulated BPSK signal:



Receiver:

Step 6 : Design a receiver:

Code:

Used functions:

File: decision_device.m

```
1: function [Reciever_output] = decision_device(received_signal_with_noise, coding_scheme, voltage, timeVec, noSamplesPerBit, noOfBits)
2:     % to select the type of line coding by coding scheme
3:     switch (coding_scheme)
4:         case 'UniPolarNRZ'
5:             Reciever_output = r_unipolarNRZ(received_signal_with_noise, voltage, timeVec, noSamplesPerBit, noOfBits);
6:         case 'PolarNRZ'
7:             Reciever_output = r_polarNRZ(received_signal_with_noise, voltage, timeVec, noSamplesPerBit, noOfBits);
8:         case 'UniPolarRZ'
9:             Reciever_output = r_unipolarRZ(received_signal_with_noise, voltage, timeVec, noSamplesPerBit, noOfBits);
10:        case 'BiPolarRZ'
11:            Reciever_output = r_bipolarRZ(received_signal_with_noise, voltage, timeVec, noSamplesPerBit, noOfBits);
12:        case 'ManchesterCoding'
13:            Reciever_output = r_manchesterCoding(received_signal_with_noise, voltage, timeVec, noSamplesPerBit, noOfBits);
14:        end
15: end

17: function [Reciever_output] = Master_source_and_comparator(received_signal_with_noise, noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold_1, threshold_2)
18:     % pre-allocate a vector to store the Reciever_output
19:     Reciever_output = zeros(1, length(timeVec));
20:     switch nargin
21:         case 8
22:             for i = 1:1:noOfBits
23:                 if (received_signal_with_noise(L) < threshold_1)
24:                     for k = M:1:P
25:                         Reciever_output(k) = 0;
26:                     end
27:                 elseif (received_signal_with_noise(L) > threshold_1)
28:                     for k = M:1:P
29:                         Reciever_output(k) = 1;
30:                     end
31:                 end
32:                 M = M + noSamplesPerBit;
33:                 P = P + noSamplesPerBit;
34:                 L = L + noSamplesPerBit;
35:             end
36:         case 9
```

```

37:         for i = 1:1:noOfBits
38:             if (received_signal_with_noise(L) < threshold_2)
39:                 for k = M:1:P
40:                     Reciever_output(k) = 1;
41:                 end
42:             elseif (received_signal_with_noise(L) > threshold_1)
43:                 for k = M:1:P
44:                     Reciever_output(k) = 1;
45:                 end
46:             elseif (received_signal_with_noise(L) < threshold_1)
47:                 for k = M:1:P
48:                     Reciever_output(k) = 0;
49:                 end
50:             end
51:             M = M + noSamplesPerBit;
52:             P = P + noSamplesPerBit;
53:             L = L + noSamplesPerBit;
54:         end
55:     end
56: end

58: function [Reciever_output] = r_unipolarNRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits)
59:     % L decision level of timing circuit
60:     % M, P time of bit in our time vector
61:     threshold = voltage / 2;
62:     L = noSamplesPerBit / 2;
63:     M = 1;
64:     P = noSamplesPerBit;
65:     [Reciever_output] = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
66: end

68: function [Reciever_output] = r_polarNRZ(received_signal_with_noise, voltage, timeVec,
noSamplesPerBit, noOfBits)
69:     % L decision level of timing circuit
70:     % M, P time of bit in our time vector
71:     threshold = (voltage + (-1 * (voltage))) / 2;
72:     L = noSamplesPerBit / 2;
73:     M = 1;
74:     P = noSamplesPerBit;
75:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
76: end

78: function [Reciever_output] = r_unipolarRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits)
79:     % L decision level of timing circuit
80:     % M, P time of bit in our time vector
81:     threshold = voltage / 2;
82:     L = noSamplesPerBit / 4;

```



```

83:     M = 1;
84:     P = noSamplesPerBit;
85:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
86: end

88: function [Reciever_output] = r_bipolarRZ(received_signal_with_noise, voltage, timeVec,
noSamplesPerBit, noOfBits)
89:     % L decision level of timing circuit
90:     % M, P time of bit in our time vector
91:     threshold_1 = voltage / 2;
92:     threshold_2 = (-1 * (voltage)) / 2;
93:     L = noSamplesPerBit / 4;
94:     M = 1;
95:     P = noSamplesPerBit;
96:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold_1, threshold_2);
97: end

99: function [Reciever_output] = r_manchesterCoding(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits)
100:     % L decision level of timing circuit
101:     % M, P time of bit in our time vector
102:     threshold = (voltage + (-1 * (voltage))) / 2;
103:     L = noSamplesPerBit / 4;
104:     M = 1;
105:     P = noSamplesPerBit;
106:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
107: end

```

File: convert_into_Binary_data.m

```

1: function
[binary_data]=convert_into_Binary_data(Reciever_output,bit_stream,noSamplesPerBit)
2:     % pre-allocate an vector to store the binary_data
3:     binary_data = zeros(1, length(bit_stream));
4:     L = noSamplesPerBit / 2;
5:     for P = 1:1:length(bit_stream)
6:         if (Reciever_output(L) == 0)
7:             binary_data(P) = 0;
8:         elseif (Reciever_output(L) == 1)
9:             binary_data(P) = 1;
10:        end
11:        L = L + noSamplesPerBit;
12:    end
13: end

```

Script:

File: part2.m

```
85: % demodulated signal (demodulation for BPSK signal after product modulator in recieved circuit)
86: demodulated_BPSK_signal = BPSK_signal .* cos(2 * pi * fc * timeVec);

87: % specterum & spectral of recieved signal (demodulated BPSK signal) after product modulator in reciver circuit
88: [spectral_demodulated_BPSK_signal,f,BW,spec_demodulated_BPSK_signal] = spectral_domain(demodulated_BPSK_signal, bit_stream, noSamplesPerBit, bitPeriod, fc);
89: figure(4);
90: grid on;
91: subplot(1,2,1);
92: plot(f, abs(spec_demodulated_BPSK_signal));
93: title("specterum of recieved signal after product modulator in reciver circuit ");
94: xlabel("frequency (Hz)");
95: ylabel("Amplitude (V)");
96: subplot(1,2,2);
97: plot(f, spectral_demodulated_BPSK_signal);
98: title("power spectral density of demodulated BPSK signal");
99: xlabel("frequency (Hz)");
100: ylabel("power spectral density ");

101: % LPF (Ideal)
102: H = abs(f) < (BW);

103: % specterum & spectral of demodulated BPSK signal After LPF
104: demodulated_BPSK_signal_After_Lpf = abs(real(H .* spec_demodulated_BPSK_signal));
105: figure(5);
106: grid on;
107: subplot(1,2,1);
108: plot(f, demodulated_BPSK_signal_After_Lpf);
109: title("spectrum of demodulated BPSK signal After Lpf");
110: xlabel("frequency (Hz)");
111: ylabel("Amplitude (V)");
112: subplot(1,2,2);
113: plot(f, (demodulated_BPSK_signal_After_Lpf .^ 2));
114: title("power spectral density of demodulated BPSK signal After Lpf");
115: xlabel("frequency (Hz)");
116: ylabel("power spectral density");

117: % demodulated BPSK signal AFTER LPF in time domain
118: demodulated_BPSK_signal_After_Lpf_t = real(ifft(fftshift(H .* spec_demodulated_BPSK_signal) * noOfBits)); %1/N

119: figure(6);
120: grid on;
121: subplot(2,1,1);
122: plot(timeVec, demodulated_BPSK_signal_After_Lpf_t, 'LineWidth', 2);
123: axis([0 timeVec(end) -1*voltage voltage]);
124: title("demodulated BPSK signal After LPF in time domain");
```

```

125: xlabel("time(s)");
126: ylabel("Amplitude (V)");

127: % decision device circuit give wave form in ones and zeros and remove effect of LPF
128: [Reciever_output] = decision_device(demodulated_BPSK_signal_After_Lpf_t,
coding_scheme, voltage, timeVec, noSamplesPerBit, noOfBits);

129: subplot(2,1,2);
130: plot(timeVec, Reciever_output, 'color', [0.6350 0.0780 0.1840], 'LineWidth', 2);
131: axis([0 timeVec(end) 0 voltage]);
132: title("Reciever output");
133: xlabel("time(s)");
134: ylabel("Amplitude (V)");

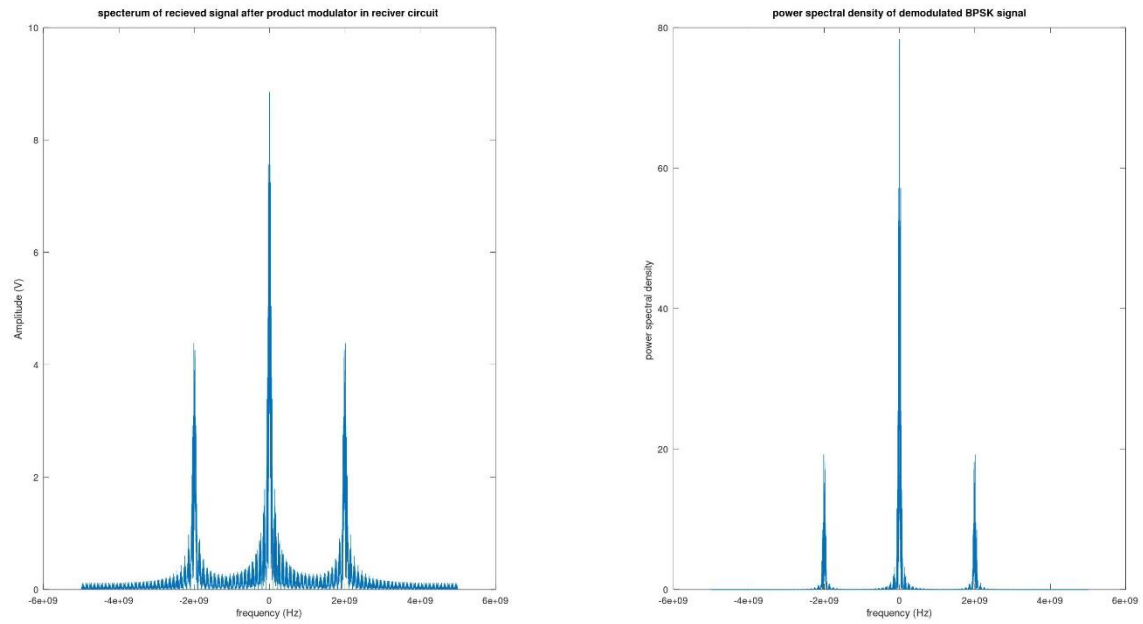
135: % return reciever output in form of binary data
136: binary_data = convert_into_Binary_data(Reciever_output, bit_stream, noSamplesPerBit);

```

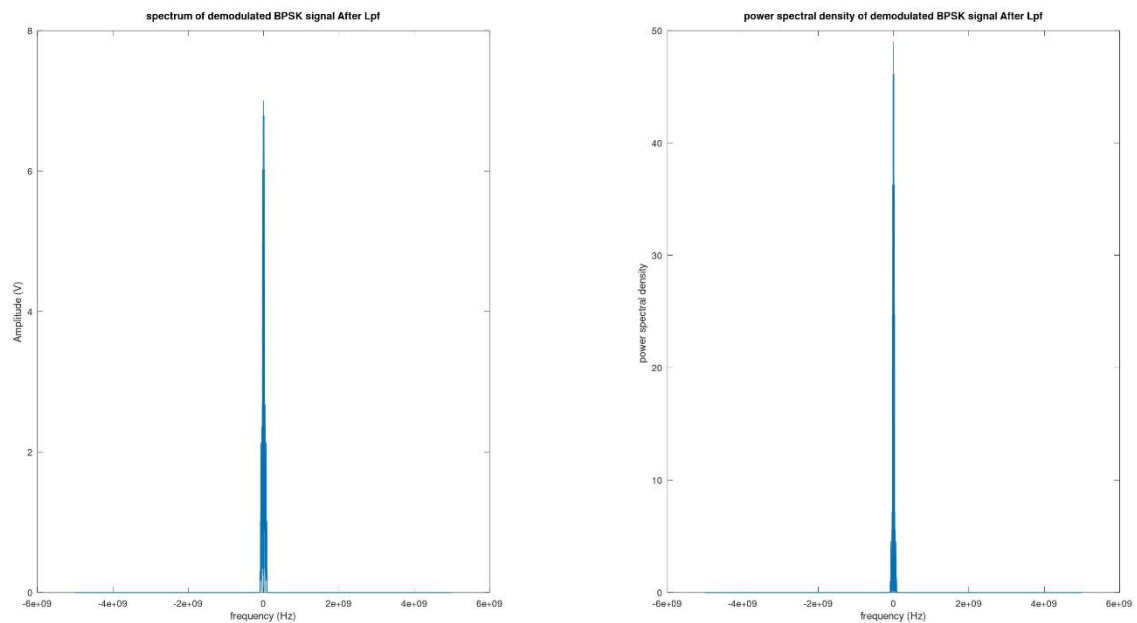
Snapshots: workspace:

Name	Class	Dimension	Value	Attribute
BPSK_signal	double	1x10000	[-1, -0.8090, -0.0...	
BW	double	1x1	1.0000e+08	
H	logical	1x10000	[0, 0, 0, 0, 0, 0, 0...	
PolarNRZ	double	1x10000	[-1, -1, -1, -1, -1...	
Rb	double	1x1	1.0000e+08	
Reciever_output	double	1x10000	[0, 0, 0, 0, 0, 0, 0...	
Spec_PolarNRZ...	double	1x10000	[0 + 0i, -2.5426e...	complex
T	double	1x1	1.0000e-06	
binary_data	double	1x100	[0, 1, 0, 1, 0, 1, 1...	
bitPeriod	double	1x1	9.9000e-09	
bit_stream	double	1x100	[0, 1, 0, 1, 0, 1, 1...	
coding_scheme	char	1x8	PolarNRZ	
demodulated_B...	double	1x10000	[-1, -0.6545, -0.0...	
demodulated_B...	double	1x10000	[0, 0, 0, 0, 0, 0, 0...	
demodulated_B...	double	1x10000	[-0.010618, -0.0...	
f	double	1x10000	-5e+09:1e+06:4....	
fc	double	1x1	1.0000e+09	
fs	double	1x1	1.0000e+10	
noOfBits	double	1x1	100	
noSamplesPerBit	double	1x1	100	
spec_BPSK_signal	double	1x10000	[-1.0083e-13 + ...	complex
spec_demodula...	double	1x10000	[-7.3612e-14 + ...	complex
spectral_BPSK_s...	double	1x10000	[1.0166e-26, 3.1...	
spectral_PolarN...	double	1x10000	[0, 1.3085e-06, 1...	
t	double	1x100	0:1e-10:9.9e-09	
timeVec	double	1x10000	[0, 1.0000e-10, 2...	
ts	double	1x1	1.0000e-10	
voltage	double	1x1	1	

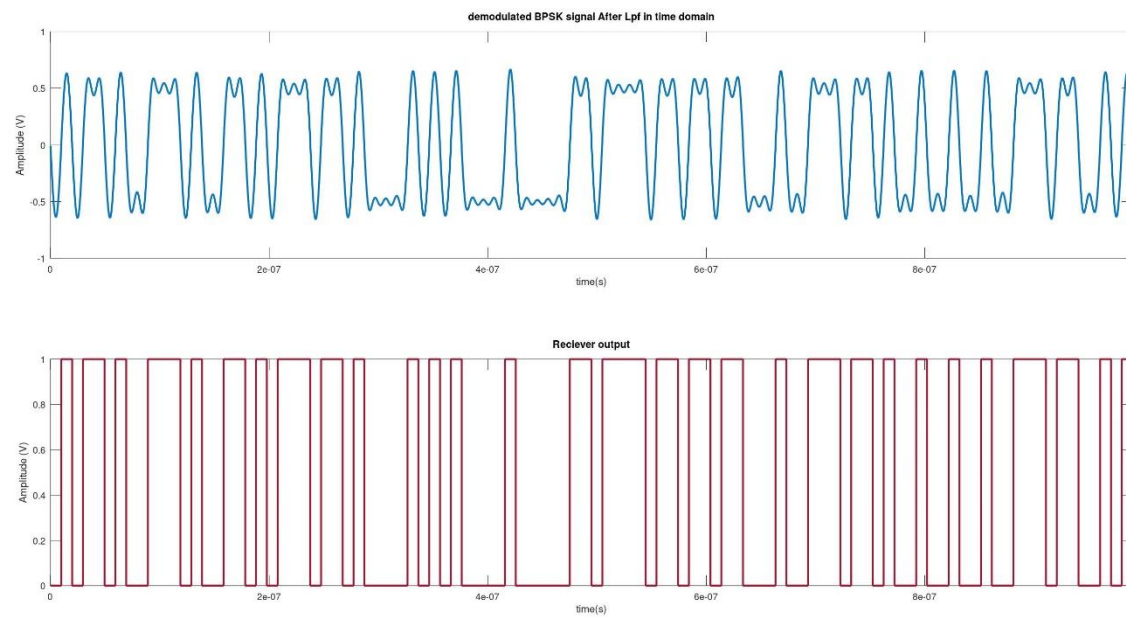
plot of spectrum of modulated signal & Power spectral demodulated BPSK signal:



plot of spectrum of BPSK signal after LPF& Power spectral BPSK signal after LPF:



plot of demodulated BPSK signal after LPF in time domain& Output BPSK signal:



Step 7: calculate bit error rate (BER):

Code:

Used functions:

File: BER_device.m

```
1: function [BER, num_errors] = BER_device(Reciever_output, bit_stream)
2:     % count the number of errors & calculate BER
3:     num_errors = sum(Reciever_output ~= bit_stream);
4:     BER = num_errors / length(bit_stream);
5: end
```

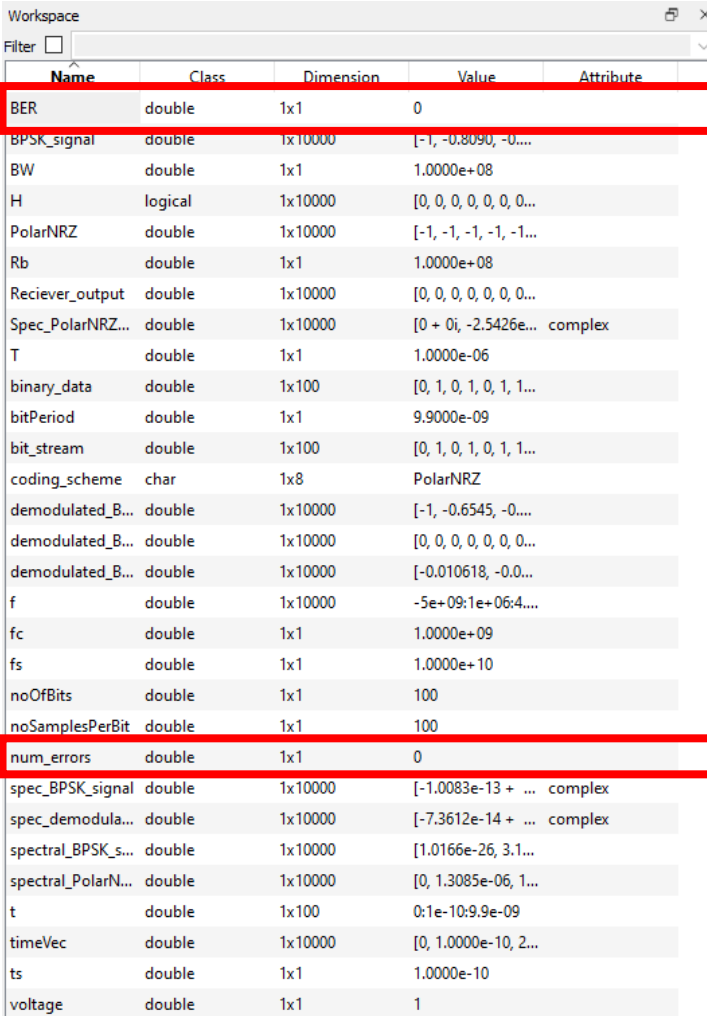
Script:

File: part2.m

```
138: % calculate BER & num_errors
139: [BER, num_errors] = BER_device(binary_data, bit_stream);
```

Snapshots:

workspace:



Name	Class	Dimension	Value	Attribute
BER	double	1x1	0	
BPSK_signal	double	1x10000	[-1, -0.8090, -0.0...	
BW	double	1x1	1.0000e+08	
H	logical	1x10000	[0, 0, 0, 0, 0, 0, 0...	
PolarNRZ	double	1x10000	[-1, -1, -1, -1, -1...	
Rb	double	1x1	1.0000e+08	
Reciever_output	double	1x10000	[0, 0, 0, 0, 0, 0, 0...	
Spec_PolarNRZ...	double	1x10000	[0 + 0i, -2.5426e...	complex
T	double	1x1	1.0000e-06	
binary_data	double	1x100	[0, 1, 0, 1, 0, 1, 1...	
bitPeriod	double	1x1	9.9000e-09	
bit_stream	double	1x100	[0, 1, 0, 1, 0, 1, 1...	
coding_scheme	char	1x8	PolarNRZ	
demodulated_B...	double	1x10000	[-1, -0.6545, -0.0...	
demodulated_B...	double	1x10000	[0, 0, 0, 0, 0, 0, 0...	
demodulated_B...	double	1x10000	[-0.010618, -0.0...	
f	double	1x10000	-5e+09:1e+06:4...	
fc	double	1x1	1.0000e+09	
fs	double	1x1	1.0000e+10	
noOfBits	double	1x1	100	
noSamplesPerBit	double	1x1	100	
num_errors	double	1x1	0	
spec_BPSK_signal	double	1x10000	[-1.0083e-13 + ...	complex
spec_demodula...	double	1x10000	[-7.3612e-14 + ...	complex
spectral_BPSK_s...	double	1x10000	[1.0166e-26, 3.1...	
spectral_PolarN...	double	1x10000	[0, 1.3085e-06, 1...	
t	double	1x100	0:1e-10:9.9e-09	
timeVec	double	1x10000	[0, 1.0000e-10, 2...	
ts	double	1x1	1.0000e-10	
voltage	double	1x1	1	

Code as text:

Script Part 1:

part1.m

```
1: % Clear all variables and close all figures
2: clear all;
3: close all;
4:
5: %set values
6: voltage=1.2;
7:
8: noOfBits=10000;
9: bitPeriod=1;
10: noSamplesPerBit=200;
11: noBitsPerSegments=3;
12:
13: coding_scheme1='UniPolarNRZ';
14: coding_scheme2='PolarNRZ';
15: coding_scheme3='UniPolarRZ';
16: coding_scheme4='BiPolarRZ';
17: coding_scheme5='ManchesterCoding';
18:
19:
20: % generate bit_stream
21: bit_stream = generate_random_bits( noOfBits );
22:
23: [lineCodeVec1, timeVec1] = line_coding(bit_stream, coding_scheme1, voltage, bitPeriod,
noSamplesPerBit);
24: [lineCodeVec2, timeVec2] = line_coding(bit_stream, coding_scheme2, voltage, bitPeriod,
noSamplesPerBit);
25: [lineCodeVec3, timeVec3] = line_coding(bit_stream, coding_scheme3, voltage, bitPeriod,
noSamplesPerBit);
26: [lineCodeVec4, timeVec4] = line_coding(bit_stream, coding_scheme4, voltage, bitPeriod,
noSamplesPerBit);
27: [lineCodeVec5, timeVec5] = line_coding(bit_stream, coding_scheme5, voltage, bitPeriod,
noSamplesPerBit);
28:
29:
30: %% plot 5 line coding in time domain
31: % UniPolarNRZ line coding
32: figure(1);
33: plot(timeVec1, lineCodeVec1, 'color', [0.4940 0.1840 0.5560], 'LineWidth', 2);
34: axis([0 timeVec1(end) 0 - voltage/2 voltage*3/2]);
35: title("UniPolarNRZ line coding");
36: legend('UniPolarNRZ');
37:
38: % Zoomed
39: figure(2);
```

```

40: plot(timeVec1, lineCodeVec1, 'color', [0.4940 0.1840 0.5560], 'LineWidth', 2);
41: axis([0 100*bitPeriod 0 - voltage/2 voltage*3/2]);
42: title("UniPolarNRZ line coding (zoomed to first 100 bits)");
43: legend('UniPolarNRZ');
44: grid on;
45:
46:
47: % PolarNRZ line coding
48: figure(3);
49: plot(timeVec2, lineCodeVec2, 'color', [1 0 0], 'LineWidth', 2);
50: axis([0 timeVec2(end) -3*voltage/2 3*voltage/2]);
51: title("PolarNRZ line coding");
52: legend('PolarNRZ');
53:
54: % Zoomed
55: figure(4);
56: plot(timeVec2, lineCodeVec2, 'color', [1 0 0], 'LineWidth', 2);
57: axis([0 100*bitPeriod -3*voltage/2 3*voltage/2]);
58: title("PolarNRZ line coding (zoomed to first 100 bits)");
59: legend('PolarNRZ');
60: grid on;
61:
62:
63: % UniPolarRZ line coding
64: figure(5);
65: plot(timeVec3, lineCodeVec3, 'color', [0.8500 0.3250 0.0980], 'LineWidth', 2);
66: axis([0 timeVec3(end) 0 - voltage/2 voltage*3/2]);
67: title("UniPolarRZ line coding");
68: legend('UniPolarRZ');
69:
70: % Zoomed
71: figure(6);
72: plot(timeVec3, lineCodeVec3, 'color', [0.8500 0.3250 0.0980], 'LineWidth', 2);
73: axis([0 100*bitPeriod 0 - voltage/2 voltage*3/2]);
74: title("UniPolarRZ line coding (zoomed to first 100 bits)");
75: legend('UniPolarRZ');
76:
77: % BiPolarRZ line coding
78: figure(7);
79: plot(timeVec4, lineCodeVec4, 'color', [0 0.4470 0.7410], 'LineWidth', 2);
80: axis([0 timeVec4(end) -3*voltage/2 3*voltage/2]);
81: title("BiPolarRZ line coding");
82: legend('BiPolarRZ');
83:
84: % Zoomed
85: figure(8);
86: plot(timeVec4, lineCodeVec4, 'color', [0 0.4470 0.7410], 'LineWidth', 2);
87: axis([0 100*bitPeriod -3*voltage/2 3*voltage/2]);
88: title("BiPolarRZ line coding (zoomed to first 100 bits)");
89: legend('BiPolarRZ');
90: grid on;

```



```

91:
92:
93: % ManchesterCoding line coding
94: figure(9);
95: plot(timeVec5, lineCodeVec5, 'color', [0.6350 0.0780 0.1840], 'LineWidth', 2);
96: axis([0 timeVec5(end) -3*voltage/2 3*voltage/2]);
97: title("ManchesterCoding line coding");
98: legend('ManchesterCoding');
99:
100: % Zoomed
101: figure(10);
102: plot(timeVec5, lineCodeVec5, 'color', [0.6350 0.0780 0.1840], 'LineWidth', 2);
103: axis([0 100*bitPeriod -3*voltage/2 3*voltage/2]);
104: title("ManchesterCoding line coding (zoomed to first 100 bits)");
105: legend('ManchesterCoding');
106: grid on;
107: figure(11);
108: subplot(1,5,1);
109: plot_eye_diagram(noBitsPerSegments , noSamplesPerBit , lineCodeVec1 , bitPeriod);
110: title("eye diagram of UniPolarNRZ");
111: legend('UniPolarNRZ');
112: ylim([0-voltage/4 3*voltage/2]);
113: grid on;
114: % PolarNRZ
115: subplot(1,5,2);
116: plot_eye_diagram(noBitsPerSegments , noSamplesPerBit , lineCodeVec2 , bitPeriod);
117: title("eye diagram of PolarNRZ");
118: legend('PolarNRZ');
119: ylim([-3*voltage/2 3*voltage/2]);
120: grid on;
121: % UniPolarRZ
122: subplot(1,5,3);
123: plot_eye_diagram(noBitsPerSegments , noSamplesPerBit , lineCodeVec3 , bitPeriod);
124: title("eye diagram of UniPolarRZ");
125: legend('UniPolarRZ');
126: ylim([0-voltage/4 3*voltage/2]);
127: grid on;
128: % BiPolarRZ
129: subplot(1,5,4);
130: plot_eye_diagram(noBitsPerSegments , noSamplesPerBit , lineCodeVec4 , bitPeriod);
131: title("eye diagram of BiPolarRZ");
132: legend('BiPolarRZ');
133: ylim([-3*voltage/2 3*voltage/2]);
134: grid on;
135: % ManchesterCoding
136: subplot(1,5,5);
137: plot_eye_diagram(noBitsPerSegments , noSamplesPerBit , lineCodeVec5 , bitPeriod);
138: title("eye diagram of ManchesterCoding");
139: legend('ManchesterCoding');
140: ylim([-3*voltage/2 3*voltage/2]);
141: grid on;

```

```

142:
143: %% get power spectrum of line coding (transmitted signal)
144: [spectral_of_lineCodeVec1,f1] =
spectral_domain(lineCodeVec1,noSamplesPerBit,bitPeriod,bit_stream);
145: [spectral_of_lineCodeVec2,f2] =
spectral_domain(lineCodeVec2,noSamplesPerBit,bitPeriod,bit_stream);
146: [spectral_of_lineCodeVec3,f3] =
spectral_domain(lineCodeVec3,noSamplesPerBit,bitPeriod,bit_stream);
147: [spectral_of_lineCodeVec4,f4] =
spectral_domain(lineCodeVec4,noSamplesPerBit,bitPeriod,bit_stream);
148: [spectral_of_lineCodeVec5,f5] =
spectral_domain(lineCodeVec5,noSamplesPerBit,bitPeriod,bit_stream);
149:
150: %% plot power spectrum of any type of line coding
151: % UniPolarNRZ
152: figure(12);
153: plot(f1,spectral_of_lineCodeVec1,'color',[0.4940 0.1840 0.5560],'LineWidth',2);
154: title("power spectral of UniPolarNRZ");
155: xlabel("frequency (Hz)");
156: ylabel('power spectral density');
157: legend('UniPolarNRZ');
158: grid on;
159:
160: % Zoomed UniPolarNRZ
161: figure(13);
162: plot(f1,spectral_of_lineCodeVec1,'color',[0.4940 0.1840 0.5560],'LineWidth',2);
163: title("power spectral of UniPolarNRZ (zoomed)");
164: xlabel("frequency (Hz)");
165: ylabel('power spectral density');
166: legend('UniPolarNRZ');
167: axis([-3 3 0 0.0003]);
168: grid on;
169:
170: % PolarNRZ
171: figure(14);
172: plot(f2,spectral_of_lineCodeVec2,'color',[1 0 0],'LineWidth',2);
173: title("power spectral of PolarNRZ");
174: xlabel("frequency (Hz)");
175: ylabel('power spectral density');
176: legend('PolarNRZ');
177: grid on;
178: axis([-5 5 0 0.0012]);
179:
180: % UniPolarRZ
181: figure(15);
182: plot(f3,spectral_of_lineCodeVec3,'color',[0.8500 0.3250 0.0980],'LineWidth',2);
183: title("power spectral of UniPolarRZ");
184: xlabel("frequency (Hz)");
185: ylabel('power spectral density');
186: legend('UniPolarRZ');
187: grid on;

```

```

188:
189: % Zoomed UniPolarRZ
190: figure(16);
191: plot(f3,spectral_of_lineCodeVec3,'color',[0.8500 0.3250 0.0980],'LineWidth',2);
192: title("power spectral of UniPolarRZ (zoomed)");
193: xlabel("frequency (Hz)");
194: ylabel('power spectral density');
195: legend('UniPolarRZ');
196: grid on;
197: axis([-6 6 0 0.0001]);
198:
199: % BiPolarRZ
200: figure(17);
201: plot(f4,spectral_of_lineCodeVec4,'color',[0 0.4470 0.7410],'LineWidth',2);
202: title("power spectral of BiPolarRZ");
203: xlabel("frequency (Hz)");
204: ylabel('power spectral density');
205: legend('BiPolarRZ');
206: axis([-15 15 0 0.00025]);
207: grid on;
208:
209: % ManchesterCoding
210: figure(18);
211: plot(f5,spectral_of_lineCodeVec5,'color',[0.6350 0.0780 0.1840],'LineWidth',2);
212: title("power spectral of ManchesterCoding");
213: xlabel("frequency (Hz)");
214: ylabel('power spectral density');
215: legend('ManchesterCoding');
216: axis([-10 10 0 0.0008]);
217:
218: figure(19);
219: sigma_ranges = linspace(0, voltage, 10);
220: [BER_values1, num_errors1] = Sweep_on_value_of_sigma(lineCodeVec1, voltage, timeVec1,
coding_scheme1, noSamplesPerBit, noOfBits, bit_stream);
221: [BER_values2, num_errors2] = Sweep_on_value_of_sigma(lineCodeVec2, voltage, timeVec2,
coding_scheme2, noSamplesPerBit, noOfBits, bit_stream);
222: [BER_values3, num_errors3] = Sweep_on_value_of_sigma(lineCodeVec3, voltage, timeVec3,
coding_scheme3, noSamplesPerBit, noOfBits, bit_stream);
223: [BER_values4, num_errors4] = Sweep_on_value_of_sigma(lineCodeVec4, voltage, timeVec4,
coding_scheme4, noSamplesPerBit, noOfBits, bit_stream);
224: [BER_values5, num_errors5] = Sweep_on_value_of_sigma(lineCodeVec5, voltage, timeVec5,
coding_scheme5, noSamplesPerBit, noOfBits, bit_stream);
225:
226: semilogy(sigma_ranges, BER_values1, sigma_ranges, BER_values2, sigma_ranges,
BER_values3, sigma_ranges, BER_values4, sigma_ranges, BER_values5, 'LineWidth', 2);
227:
228: grid on;
229: xlabel('Sigma');
230: ylabel('BER');
231: legend({'UniPolarNRZ', 'PolarNRZ', 'UniPolarRZ', 'BiPolarRZ', 'ManchesterCoding'});
232: %% BONUS

```

```
233:[number_of_detected_errors]=Bonus_Sweep_on_value_of_sigma(lineCodeVec4,voltage,timeVec
4,coding_scheme4,noSamplesPerBit,noOfBits);
```

Script Part 2 :

part2.m:

```
1: % octave Script for a Binary PSK (passband communication)
2:
3: % Clear all variables and close all figures
4: clear all;
5: close all;
6:
7: noOfBits = 100; % Number of time samples
8:
9: % Generate bit stream
10: bit_stream = generate_random_bits(noOfBits);
11: %% Calculations
12: fc = 10^9; % Frequency of Modulating Signal
13: fs = 10 * fc; % Sampling frequency - Sampling rate - This will define the resolution
14: ts = 1 / fs; % Time step
15: t = 0:ts:(noOfBits - 1) * ts; % Time for one bit - Time of one bit = 1 second per
one bit
16: T = (noOfBits) * (noOfBits * ts); % Simulation time
17: Rb = noOfBits / T; % Bit rate = N / simulation time
18:
19: voltage = 1;
20: bitPeriod = t(end);
21: noSamplesPerBit = length(t);
22: coding_scheme = 'PolarNRZ';
23:
24: %% The PolarNRZ Signal
25: [PolarNRZ, timeVec] = line_coding(bit_stream, coding_scheme, voltage, bitPeriod,
noSamplesPerBit, fc);
26:
27: % Plot the PolarNRZ Signal with time
28: figure;
29: plot(timeVec, PolarNRZ, 'r', 'LineWidth', 2);
30: xlabel('Time (s)');
31: ylabel('Amplitude (V)');
32: title('PolarNRZ Signal');
33: axis([0 timeVec(end) -1.5 1.5]);
34: grid on;
35:
36: % Spectrum & Spectral of PolarNRZ Signal
37:
38: [spectral_PolarNRZ_Signal, f, BW, Spec_PolarNRZ_Signal] = spectral_domain(PolarNRZ,
bit_stream, noSamplesPerBit, bitPeriod, fc);
39:
40: figure;
41: grid on;
42:
43: plot(f, abs(Spec_PolarNRZ_Signal));
44: title('Spectrum of Original PolarNRZ Signal');
45: xlabel('Frequency (Hz)');
```

```

46: ylabel('Amplitude (V)');
47:
48: figure;
49: plot(f, spectral_PolarNRZ_Signal);
50: title('Power Spectral Density of PolarNRZ Signal');
51: xlabel("Frequency (Hz)");
52: ylabel('Power Spectral Density');
53:
54: % The BPSK Signal (modulated signal of PolarNRZ Signal after product modulator in
transmitter circuit)
55: BPSK_signal = PolarNRZ .* cos(2 * pi * fc * timeVec);
56:
57: % Plot the BPSK Signal (line coding) with time
58: figure;
59: plot(timeVec, BPSK_signal, 'LineWidth', 2);
60: xlabel('Time (s)');
61: ylabel('Amplitude (V)');
62: title('BPSK Signal with Two Phase Shifts');
63: axis([0 timeVec(end) -1.5 1.5]);
64: grid on;
65:
66: % Spectrum & Spectral of BPSK signal
67:
68: [spectral_BPSK_signal, f, BW, spec_BPSK_signal] = spectral_domain(BPSK_signal,
bit_stream, noSamplesPerBit, bitPeriod, fc);
69:
70: figure(3);
71: grid on;
72:
73: subplot(1, 2, 1);
74: plot(f, abs(spec_BPSK_signal));
75: title("Spectrum of Modulated BPSK Signal");
76: xlabel("Frequency (Hz)");
77: ylabel('Amplitude (V)');
78:
79: subplot(1, 2, 2);
80: plot(f, spectral_BPSK_signal);
81: title("Power Spectral Density of Modulated BPSK Signal");
82: xlabel("Frequency (Hz)");
83: ylabel('Power Spectral Density');
84:
85: % demodulated signal (demodulation for BPSK signal after product modulator in recieved
circuit)
86: demodulated_BPSK_signal = BPSK_signal .* cos(2 * pi * fc * timeVec);
87: % specterum & spectral of recieved signal (demodulated BPSK signal) after product
modulator in reciver circuit
88: [spectral_demodulated_BPSK_signal,f,BW,spec_demodulated_BPSK_signal] =
spectral_domain(demodulated_BPSK_signal, bit_stream, noSamplesPerBit, bitPeriod, fc);
89: figure(4);
90: grid on;

```

```

91: subplot(1,2,1);
92: plot(f, abs(spec_demodulated_BPSK_signal));
93: title("spectrum of recieved signal after product modulator in reciver circuit ");
94: xlabel("frequency (Hz)");
95: ylabel("Amplitude (V)");
96: subplot(1,2,2);
97: plot(f, spectral_demodulated_BPSK_signal);
98: title("power spectral density of demodulated BPSK signal");
99: xlabel("frequency (Hz)");
100: ylabel("power spectral density ");

101: % LPF (Ideal)
102: H = abs(f) < (BW);

103: % spectrum & spectral of demodulated BPSK signal After LPF
104: demodulated_BPSK_signal_After_Lpf = abs(real(H .* spec_demodulated_BPSK_signal));
105: figure(5);
106: grid on;
107: subplot(1,2,1);
108: plot(f, demodulated_BPSK_signal_After_Lpf);
109: title("spectrum of demodulated BPSK signal After Lpf");
110: xlabel("frequency (Hz)");
111: ylabel("Amplitude (V)");
112: subplot(1,2,2);
113: plot(f, (demodulated_BPSK_signal_After_Lpf .^ 2));
114: title("power spectral density of demodulated BPSK signal After Lpf");
115: xlabel("frequency (Hz)");
116: ylabel("power spectral density ");

117: % demodulated BPSK signal AFTER LPF in time domain
118: demodulated_BPSK_signal_After_Lpf_t = real(iffshift(fftshift(H .*
spec_demodulated_BPSK_signal) * noOfBits)); %1/N

119: figure(6);
120: grid on;
121: subplot(2,1,1);
122: plot(timeVec, demodulated_BPSK_signal_After_Lpf_t, 'LineWidth', 2);
123: axis([0 timeVec(end) -1*voltage voltage]);
124: title("demodulated BPSK signal After LPF in time domain");
125: xlabel("time(s)");
126: ylabel("Amplitude (V)");

127: % decision device circuit give wave form in ones and zeros and remove effect of LPF
128: [Reciever_output] = decision_device(demodulated_BPSK_signal_After_Lpf_t,
coding_scheme, voltage, timeVec, noSamplesPerBit, noOfBits);

129: subplot(2,1,2);
130: plot(timeVec, Reciever_output, 'color', [0.6350 0.0780 0.1840], 'LineWidth', 2);
131: axis([0 timeVec(end) 0 voltage]);
132: title("output demodulated BPSK signal from recieved circuit");
133: xlabel("time(s)");

```

```

134: ylabel("Amplitude (V)");

135: % return reciever output in form of binary data
136: binary_data = convert_into_Binary_data(Reciever_output, bit_stream, noSamplesPerBit);
137:
138: % calculate BER & num_errors
139: [BER, num_errors] = BER_device(binary_data, bit_stream);

```

Functions codes:

File: generate_random_bits.m

```

1: function bit_stream = generate_random_bits( noOfBits )
2:     % generate random sequence of binary data zeros and ones
3:     bit_stream = randi( [ 0 , 1 ] , 1 , noOfBits );
4: end

```

File: line_coding.m

```

1: function [lineCodeVec,timeVec]=line_coding(bit_stream,coding_scheme,voltage
,bitPeriod,noSamplesPerBit,fc )
2:     switch nargin
3:         %choose according to number of input arguments
4:         case 5
5:             %belong to part 1
6:             [lineCodeVec,timeVec]=Baseband_communication(bit_stream,coding_scheme,voltage,bitPeriod,noSamplesPerBit);
7:         case 6
8:             %belong to part 2
9:             [lineCodeVec,timeVec]=Passband_communication(bit_stream,coding_scheme,voltage,bitPeriod,noSamplesPerBit,fc);
10:        end
11: end
12: function [lineCodeVec, timeVec] = Passband_communication(bit_stream, coding_scheme,
voltage, bitPeriod, noSamplesPerBit, fc)
13:     % This function implements the function of polarNRZ block in transmitter
14:
15:     % Calculations
16:     noOfBits = length(bit_stream);
17:     fs = 10 * fc; % Sampling frequency - Sampling rate - This will define the
resolution
18:     ts = 1 / fs; % Time step
19:     t = 0:ts:(noOfBits - 1) * ts;
20: % Generate time domain
21: timeVec = [];
22: for ii = 1:1:length(bit_stream)
23: timeVec = [timeVec t];
24: t = t + (noOfBits - 1) * ts;
25: end
26:
27: noSamplesPerBit = length(t);

```



```

28:
29: % Generate polarNRZ line coding
30: switch (coding_scheme)
31: case 'PolarNRZ'
32: lineCodeVec = polarNRZ(bit_stream, voltage, timeVec, noSamplesPerBit);
33: end
34: end
34: function lineCodeVec = polarNRZ(bit_stream , voltage , timeVec , noSamplesPerBit)
35:     lineCodeVec = unipolarNRZ(bit_stream , voltage , timeVec , noSamplesPerBit);
36:     %same as unipolarNRZ but change all the zeros into -ve voltage
37:     lineCodeVec(lineCodeVec == 0) = -1 * voltage;
38: end
40: function lineCodeVec = unipolarRZ(bit_stream , voltage , timeVec , noSamplesPerBit)
41:     lineCodeVec = zeros(1 , length(timeVec));
42:     for i = 1 : length(bit_stream)
43:         if bit_stream(i) == 1
44:             %+ve voltage for the first half cycle of the bits
45:             lineCodeVec( ((i - 1) * noSamplesPerBit) + 1 : (i * noSamplesPerBit) -
(noSamplesPerBit / 2)) = voltage;
46:             % 0 voltage for the other half cycle of the bit
47:             lineCodeVec( (i * noSamplesPerBit) - (noSamplesPerBit / 2) + 1 : i *
noSamplesPerBit) = 0;
48:         end
49:     end
50: end
52: function lineCodeVec = bipolarRZ(bit_stream , voltage , timeVec , noSamplesPerBit)
53:     lineCodeVec = zeros(1 , length(timeVec));
54:     flag = 0; % to indicate whether the voltage to be +ve or -ve
55:     for i = 1 : length(bit_stream)
56:         if bit_stream(i) == 1
57:             if (flag == 0)
58:                 %+ve voltage for the first half cycle of the bits
59:                 lineCodeVec( ((i - 1) * noSamplesPerBit) + 1 : (i * noSamplesPerBit) -
(noSamplesPerBit / 2)) = voltage;
60:                 % 0 voltage for the other half cycle of the bit
61:                 lineCodeVec( (i * noSamplesPerBit) - (noSamplesPerBit / 2) + 1 : i *
noSamplesPerBit) = 0;
62:                 flag = 1; %update the flag
63:             elseif(flag == 1)
64:                 %-ve voltage for the first half cycle of the bits
65:                 lineCodeVec( ((i - 1) * noSamplesPerBit) + 1 : (i * noSamplesPerBit) -
(noSamplesPerBit / 2)) = -voltage;
66:                 % 0 voltage for the other half cycle of the bit
67:                 lineCodeVec( (i * noSamplesPerBit) - (noSamplesPerBit / 2) + 1 : i *
noSamplesPerBit) = 0;
68:                 flag = 0; %update the flag
69:             end
70:         end
71:     end
72: end

```

```

74: function lineCodeVec = manchesterCoding(bit_stream , voltage , timeVec ,
noSamplesPerBit)
75:     lineCodeVec = zeros(1 , length(timeVec));
76:     for i = 1 : length(bit_stream)
77:         if bit_stream(i) == 1
78:             %+ve voltage for the first half cycle of the bits
79:             lineCodeVec( ((i - 1) * noSamplesPerBit) + 1 : (i * noSamplesPerBit) -
(noSamplesPerBit / 2)) = voltage;
80:             %-ve voltage for the other half cycle of the bit
81:             lineCodeVec( (i * noSamplesPerBit) - (noSamplesPerBit / 2) + 1 : i *
noSamplesPerBit) = -1* voltage;
82:         else
83:             %-ve voltage for the first half cycle of the bits
84:             lineCodeVec( ((i - 1) * noSamplesPerBit) + 1 : (i * noSamplesPerBit) -
(noSamplesPerBit / 2)) = -1 * voltage;
85:             %+ve voltage for the other half cycle of the bit
86:             lineCodeVec( (i * noSamplesPerBit) - (noSamplesPerBit / 2) + 1 : i *
noSamplesPerBit) = voltage;
87:         end
88:     end
89: end

```

File: plot_eye_diagram.m

```

1: function plot_eye_diagram(noBitsPerSegments , noSamplePerBit , lineCodeVec , bitPeriod)
2:     % segment is a collection of bits we need to shift them to create the eye diagram
3:     % segmentLength: total samples in a segment
4:     segmentLength = noSamplePerBit * noBitsPerSegments;
5:     % periodic time of the samples
6:     samplePeriod = bitPeriod / noSamplePerBit;
7:     % time vector is created with a step of samplePeriod, and the end is the total time
a segment
8:     % could be calculated using noBitsPerSegments and bitPeriod, with a step =
samplePeriod
9:     timeVec = (0 : segmentLength - 1) * samplePeriod;
10:    % to center the zero in the mid
11:    timeVec = timeVec - (noBitsPerSegments * bitPeriod / 2);
12:    % total number of segment layers to place on top of each other
13:    % segmentLayers = total number of samples in line code / total number of samples in
the segment
14:    % it's rounded down to the nearest integer, as we need to keep the size of the
segment
15:    % equal to the size of the time vector in terms of the samples
16:    segmentLayers = floor(length(lineCodeVec) / segmentLength);
17:    for i = 1 : segmentLayers
18:        % find the start and end points of the segment in the line code
19:        % it's multiplied by the number of bits per segment
20:        segmentStart = (i - 1) * segmentLength + 1;

```

```

21:     segmentEnd = i * segmentLength;
22:     % x-axis is the time vector, and the y-axis is the amplitude of each bit
23:     plot(timeVec , lineCodeVec(segmentStart : segmentEnd),'LineWidth',2);
24:     % hold to plot all the segments on top of each other
25:     hold on;
26: end
27: axis([-1.5 1.5 -bitPeriod bitPeriod]);
28: end

```

File: spectral_domain.m

```

1: function [spectral, f, BW, Spec_Original_line_coding] = spectral_domain(lineCodeVec,
bit_stream, noSamplesPerBit, bitPeriod, fc)
2:     switch nargin
3:         % choose according to number of input arguments
4:         case 4
5:             [spectral, f] = Baseband_communication(lineCodeVec, noSamplesPerBit,
bitPeriod, bit_stream);
6:         case 5
7:             [spectral, f, BW, Spec_Original_line_coding] =
Passband_communication(lineCodeVec, bit_stream, noSamplesPerBit, bitPeriod, fc);
8:         end
9: end
10:
11: function [spectral, f, BW, Spec_Original_line_coding] =
Passband_communication(lineCodeVec, bit_stream, noSamplesPerBit, bitPeriod, fc)
12:     % Calculation to generate frequency domain
13:     noOfBits = length(bit_stream);
14:     fs = 10 * fc; % Sampling frequency - Sampling rate - This will define the
resolution
15:     ts = 1 / fs; % Time step
16:     T = noOfBits * ts; % Simulation time
17:     Rb = noOfBits / T; % Bit rate = N / simulationtime
18:     BW = Rb; % Polar & NRZ
19:     df = 1 / T; % Frequency step
20:
21:     % Spectrum of Original Digital Signal
22:     Spec_Original_line_coding = (fftshift(fft(lineCodeVec))) / noOfBits; % We put
message in frequency domain because it only shifts function in FFT
23:
24:     % Frequency domain
25:     if (rem(noOfBits,2) == 0) % Even
26:         f = ((-0.5 * fs)):df:((0.5 * fs) - df)); % Frequency vector if x/f even
27:     else % Odd
28:         f = ((-0.5 * fs - 0.5 * df)) : df :(((0.5 * fs) + 1) - 0.5 * df); % Frequency
vector if X/f is odd
29:     end
30:
31:     % Power spectral of Original Digital Signal
32:     spectral = abs(Spec_Original_line_coding).^2;

```

```

33: end
34:
35: function [spectral, f] = Baseband_communication(lineCodeVec, noSamplesPerBit,
bitPeriod, bit_stream)
36:     % Calculation to generate frequency domain
37:     T = length(bit_stream) * bitPeriod; % Simulation time
38:     df = 1/T %frequency step
39:     fs=noSamplesPerBit/bitPeriod; % sampling frequency
40:     N=noSamplesPerBit*length(bit_stream);
41:     %spectrum of Original Digital Signal
42:     Spec_Original_line_coding = (fftshift(fft(lineCodeVec)))/N;
43:     % frequency doomain
44:     if(rem(N,2)==0) %% even
45:         f = ((-(0.5*fs)): df : ((0.5*fs)-df)); %% frequency vector if x/f even
46:     else %% odd
47:         f = (-(0.5*fs-0.5*df)) : df : (((0.5*fs)+1)-0.5*df); %% frequency vector if X/f is odd
48:     end
49:     %power spectral of Original Digital Signal
50:     spectral =abs(Spec_Original_line_coding).^2;
51:     end

```

File: Sweep_on_value_of_sigma.m

```

1: function [BER_values, num_errors] = Sweep_on_value_of_sigma(lineCodeVec, voltage,
timeVec, coding_scheme, noSamplesPerBit, noOfBits, bit_stream)
2:     % generate 10 ranges for sigma that range from 0 to the maximum supply voltage
3:     sigma_ranges = linspace(0, voltage, 10);
4:
5:     % pre-allocate a vector to store the BER values & num_errors
6:     BER_values = zeros(1, 10);
7:     num_errors = zeros(1, 10);
8:
9:     % loop choose each value of sigma by index i
10:    for i = 1:10
11:        % add noise to signal
12:        received_signal_with_noise = add_noise_to_linecoding(lineCodeVec,
sigma_ranges(i), timeVec);
13:
14:        % path signal through the decision device
15:        Reciever_output = decision_device(received_signal_with_noise, coding_scheme,
voltage, timeVec, noSamplesPerBit, noOfBits);
16:
17:        % this loop is used to convert Reciever output into binary data
18:        binary_data = convert_into_Binary_data(Reciever_output, bit_stream,
noSamplesPerBit);
19:
20:        % calculate the BER & num_errors for this value of sigma
21:        [BER_values(i), num_errors(i)] = BER_device(binary_data, bit_stream); % insert
your code for calculating BER & num_errors here

```

```
22:     end
23: end
```

File: add_noise_to_linecoding.m

```
1: function received_signal_with_noise = add_noise_to_linecoding(lineCode, sigma, Vectime)
2:     % this function simulates external noise from the communication channel (telephone
line) or noise from the receiver added to the transmitted line coding
3:
4:     % define your time vector
5:     t = Vectime;
6:
7:     % noise
8:     n = sigma * randn(1, length(t));
9:
10:    % add the noise to your received signal
11:    received_signal_with_noise = lineCode + n;
12: end
```

File: decision_device.m

```
1: function [Reciever_output] = decision_device(received_signal_with_noise, coding_scheme,
voltage, timeVec, noSamplesPerBit, noOfBits)
2:     % to select the type of line coding by coding scheme
3:     switch (coding_scheme)
4:         case 'UniPolarNRZ'
5:             Reciever_output = r_unipolarNRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits);
6:         case 'PolarNRZ'
7:             Reciever_output = r_polarNRZ(received_signal_with_noise, voltage, timeVec,
noSamplesPerBit, noOfBits);
8:         case 'UniPolarRZ'
9:             Reciever_output = r_unipolarRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits);
10:        case 'BiPolarRZ'
11:            Reciever_output = r_bipolarRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits);
12:        case 'ManchesterCoding'
13:            Reciever_output = r_manchesterCoding(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits);
14:        end
15: end
16:
17: function [Reciever_output] = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold_1, threshold_2)
18:     % pre-allocate a vector to store the Reciever_output
19:     Reciever_output = zeros(1, length(timeVec));
```

```

20:     switch nargin
21:         case 8
22:             for i = 1:1:noOfBits
23:                 if (received_signal_with_noise(L) < threshold_1)
24:                     for k = M:1:P
25:                         Reciever_output(k) = 0;
26:                     end
27:                 elseif (received_signal_with_noise(L) > threshold_1)
28:                     for k = M:1:P
29:                         Reciever_output(k) = 1;
30:                     end
31:                 end
32:                 M = M + noSamplesPerBit;
33:                 P = P + noSamplesPerBit;
34:                 L = L + noSamplesPerBit;
35:             end
36:         case 9
37:             for i = 1:1:noOfBits
38:                 if (received_signal_with_noise(L) < threshold_2)
39:                     for k = M:1:P
40:                         Reciever_output(k) = 1;
41:                     end
42:                 elseif (received_signal_with_noise(L) > threshold_1)
43:                     for k = M:1:P
44:                         Reciever_output(k) = 1;
45:                     end
46:                 elseif (received_signal_with_noise(L) < threshold_1)
47:                     for k = M:1:P
48:                         Reciever_output(k) = 0;
49:                     end
50:                 end
51:                 M = M + noSamplesPerBit;
52:                 P = P + noSamplesPerBit;
53:                 L = L + noSamplesPerBit;
54:             end
55:         end
56:     end

58: function [Reciever_output] = r_unipolarNRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits)
59:     % L decision level of timing circuit
60:     % M, P time of bit in our time vector
61:     threshold = voltage / 2;
62:     L = noSamplesPerBit / 2;
63:     M = 1;
64:     P = noSamplesPerBit;
65:     [Reciever_output] = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
66: end

```

```

68: function [Reciever_output] = r_polarNRZ(received_signal_with_noise, voltage, timeVec,
noSamplesPerBit, noOfBits)
69:     % L decision level of timing circuit
70:     % M, P time of bit in our time vector
71:     threshold = (voltage + (-1 * (voltage))) / 2;
72:     L = noSamplesPerBit / 2;
73:     M = 1;
74:     P = noSamplesPerBit;
75:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
76: end

78: function [Reciever_output] = r_unipolarRZ(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits)
79:     % L decision level of timing circuit
80:     % M, P time of bit in our time vector
81:     threshold = voltage / 2;
82:     L = noSamplesPerBit / 4;
83:     M = 1;
84:     P = noSamplesPerBit;
85:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
86: end

88: function [Reciever_output] = r_bipolarRZ(received_signal_with_noise, voltage, timeVec,
noSamplesPerBit, noOfBits)
89:     % L decision level of timing circuit
90:     % M, P time of bit in our time vector
91:     threshold_1 = voltage / 2;
92:     threshold_2 = (-1 * (voltage)) / 2;
93:     L = noSamplesPerBit / 4;
94:     M = 1;
95:     P = noSamplesPerBit;
96:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold_1, threshold_2);
97: end

99: function [Reciever_output] = r_manchesterCoding(received_signal_with_noise, voltage,
timeVec, noSamplesPerBit, noOfBits)
100:     % L decision level of timing circuit
101:     % M, P time of bit in our time vector
102:     threshold = (voltage + (-1 * (voltage))) / 2;
103:     L = noSamplesPerBit / 4;
104:     M = 1;
105:     P = noSamplesPerBit;
106:     Reciever_output = Master_source_and_comparator(received_signal_with_noise,
noOfBits, noSamplesPerBit, timeVec, L, M, P, threshold);
107: end

```

File: convert_into_Binary_data.m

```
1: function [binary_data] = convert_into_Binary_data(Reciever_output, bit_stream,
noSamplesPerBit)
2:     % pre-allocate a vector to store the binary_data
3:     binary_data = zeros(1, length(bit_stream));
4:
5:     L = noSamplesPerBit / 2;
6:
7:     for P = 1:1:length(bit_stream)
8:         if (Reciever_output(L) == 0)
9:             binary_data(P) = 0;
10:        elseif (Reciever_output(L) == 1)
11:            binary_data(P) = 1;
12:        end
13:
14:        L = L + noSamplesPerBit;
15:    end
16: end
```

File: BER_device.m

```
1: function [BER, num_errors] = BER_device(Reciever_output, bit_stream)
2:     % count the number of errors & calculate BER
3:     num_errors = sum(Reciever_output ~= bit_stream);
4:     BER = num_errors / length(bit_stream);
5: end
```

File: Regenerative_Repeater.m

```
1: function [Repeater_output] = Regenerative_Repeater(received_signal_with_noise,
coding_scheme, voltage, timeVec, noSamplesPerBit, noOfBits)
2:     % This function represents the regenerative repeater block used in telephone lines.
3:     % When entering the receiver to select the type of line coding by coding scheme.
4:     switch (coding_scheme)
5:         case 'BiPolarRZ'
6:             Repeater_output = r_bipolarRZ(received_signal_with_noise, voltage, timeVec,
noSamplesPerBit, noOfBits);
7:         end
8:     end

10: function [Repeater_output] = r_bipolarRZ(received_signal_with_noise, voltage, timeVec,
noSamplesPerBit, noOfBits)
11:     % Pre-allocate a vector to store the Receiver_output
12:     Repeater_output = zeros(1, length(timeVec));
13:     % L decision level of timing circuit
14:     % M, P time of bit in our time vector
15:     threshold_1 = voltage / 2;
16:     threshold_2 = (-1 * (voltage)) / 2;
17:     L = noSamplesPerBit / 4;
18:     M = 1;
19:     P = noSamplesPerBit / 2;
```



```

20:     for i = 1:1:noOfBits
21:         if (received_signal_with_noise(L) < threshold_2)
22:             for k = M:1:P
23:                 Repeater_output(k) = -1 * voltage;
24:             end
25:         elseif (received_signal_with_noise(L) > threshold_1)
26:             for k = M:1:P
27:                 Repeater_output(k) = voltage;
28:             end
29:         elseif (received_signal_with_noise(L) < threshold_1)
30:             for k = M:1:P
31:                 Repeater_output(k) = 0;
32:             end
33:         end
34:         M = M + noSamplesPerBit;
35:         P = P + noSamplesPerBit;
36:         L = L + noSamplesPerBit;
37:     end
38: end

```

File: Error_Detection_Circuit.m

```

1: function [number_of_detected_errors] = Error_Detection_Circuit(Repeater_output,
voltage, noSamplesPerBit, noOfBits)
2:     % Error detection circuit for Bipolar return to zero
3:     number_of_detected_errors = 0;
4:     L = noSamplesPerBit / 4;
5:
6:     for i = 1:1:noOfBits
7:         if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
8:             break;
9:         end
10:
11:         % 1000000000...01
12:         if (Repeater_output(L) == voltage && Repeater_output(L + noSamplesPerBit) ==
0)
13:             if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
14:                 break;
15:             else
16:                 L = L + noSamplesPerBit;
17:             end
18:
19:             while (Repeater_output(L) == 0)
20:                 if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
21:                     break;
22:                 else
23:                     L = L + noSamplesPerBit;
24:                 end
25:             end

```

```

26:
27:     if (Repeater_output(L) == voltage)
28:         number_of_detected_errors = number_of_detected_errors + 1;
29:     end
30:     % -100000000...0-1
31:     elseif (Repeater_output(L) == -1 * voltage && Repeater_output(L +
noSamplesPerBit) == 0)
32:         if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
33:             break;
34:         else
35:             L = L + noSamplesPerBit;
36:         end
37:
38:         while (Repeater_output(L) == 0)
39:             if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >=
noOfBits * noSamplesPerBit)
40:                 break;
41:             else
42:                 L = L + noSamplesPerBit;
43:             end
44:         end
45:
46:         if (Repeater_output(L) == -1 * voltage)
47:             number_of_detected_errors = number_of_detected_errors + 1;
48:         end
49:
50:         % -1-1
51:         elseif (Repeater_output(L) == -1 * voltage && Repeater_output(L +
noSamplesPerBit) == -1 * voltage)
52:             number_of_detected_errors = number_of_detected_errors + 1;
53:             if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
54:                 break;
55:             else
56:                 L = L + noSamplesPerBit;
57:             end
58:
59:             % 11
60:             elseif (Repeater_output(L) == voltage && Repeater_output(L + noSamplesPerBit)
== voltage)
61:                 number_of_detected_errors = number_of_detected_errors + 1;
62:                 if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
63:                     break;
64:                 else
65:                     L = L + noSamplesPerBit;
66:                 end
67:             end
68:

```

```

69:         if (L >= noOfBits * noSamplesPerBit || (L + noSamplesPerBit) >= noOfBits *
noSamplesPerBit)
70:             break;
71:         else
72:             L = L + noSamplesPerBit;
73:         end
74:     end
75: end

```

File: Bonus_Sweep_on_value_of_sigma.m

```

1: function [number_of_detected_errors] = Bonus_Sweep_on_value_of_sigma(lineCodeVec,
voltage, timeVec, coding_scheme, noSamplesPerBit, noOfBits)
2:     % Generate 10 ranges for sigma that range from 0 to the maximum supply voltage
3:     sigma_ranges = linspace(0, voltage, 10);
4:
5:     % Pre-allocate a vector to store the BER values & num_errors
6:     number_of_detected_errors = zeros(1, 10);
7:
8:     % Loop chooses each value of sigma by index i
9:     for i = 1:10
10:         % Add noise to signal
11:         received_signal_with_noise = add_noise_to_linecoding(lineCodeVec,
sigma_ranges(i), timeVec);
12:
13:         % Path signal through Regenerative Repeater
14:         [Repeater_output] = Regenerative_Repeater(received_signal_with_noise,
coding_scheme, voltage, timeVec, noSamplesPerBit, noOfBits);
15:
16:         % Calculate number_of_detected_errors for this value of sigma
17:         number_of_detected_errors(i) = Error_Detection_Circuit(Repeater_output,
voltage, noSamplesPerBit, noOfBits);
18:     end
19: end

```