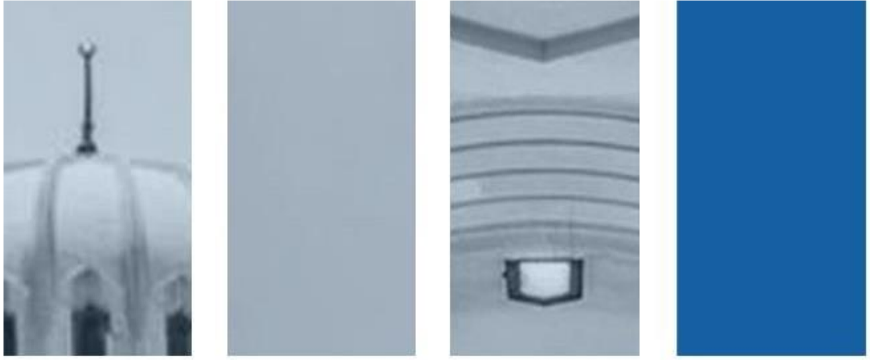


Fundamentals of Communication: Final project report

ECE252s

Ain Shams University
Faculty Of Engineering

Spring Semester - 2023

Contents:

Report structure:

This report will be organized as follows: First off, all the required figures of all the generated plots will be attached, along with their function calls from main.m, and afterwards, each class and each function's implementation will be explored in detail.

Project file structure:

main.m: Where the main runs, screenshots will be attached in the "Function calls and figures" section.

transmitter.m: Where transmitter class is defined, details in "Code dive" section. This is used in parts 1 and 2.

receiver.m: Where receiver class is defined, details in "Code dive" section. This is used in parts 1 and 2.

sweep_over_sigma.m: Function file, details in "Code dive" section. This is used in part 1, point 9.

ber_plot.m: Function file, details in "Code dive" section. This is used in part 1, point 10.

The GitHub repository contains three more files, and two of them were only used for testing and do not count towards this submission. The third file is:

overlapping_ber_plots.m: The script that produces the figure with the overlapping Sigma/BER plots. See Figure (26).

Note: The first two lines of main.m are only there to clear previous outputs and fix the generated figures' positions.

```
clear all; clc;
figure_position = [(get(0,"screensize")(3) * 13 / 136) (get(0,"screensize")(4) * 1 / 4)...
                   (get(0,"screensize")(3) * 55 / 68 ) (get(0,"screensize")(4) * 1 / 2)];
```

A. Function calls and figures:

1. Part 1: Transmitter:

General format that will be followed till the end of this section:

Snapshot from main.m

#TX object construction
#Generating stream of random bits of size 10,000
#Line coding the stream with desired format and at VCC 1.2

#Figure 1: Subplots 1 & 2
#Figure 2: Subplots 1 & 2

Figure 1: Subplot 1

For part 1 point 1, "Generate stream of random bits (10,000 bit) (This bit stream should be selected to be random, which means that the type of each bit is randomly selected by the program code to be either '1' or '0')."

Figure 1: Subplot 2

For part 1 point 2, "Line code the stream of bits (pulse shape) according to XXXXXXXXXXX (Supply voltages are: +1.2 V and -1.2V).", where XXX is the line-coding format.

Figure 2: Subplot 1

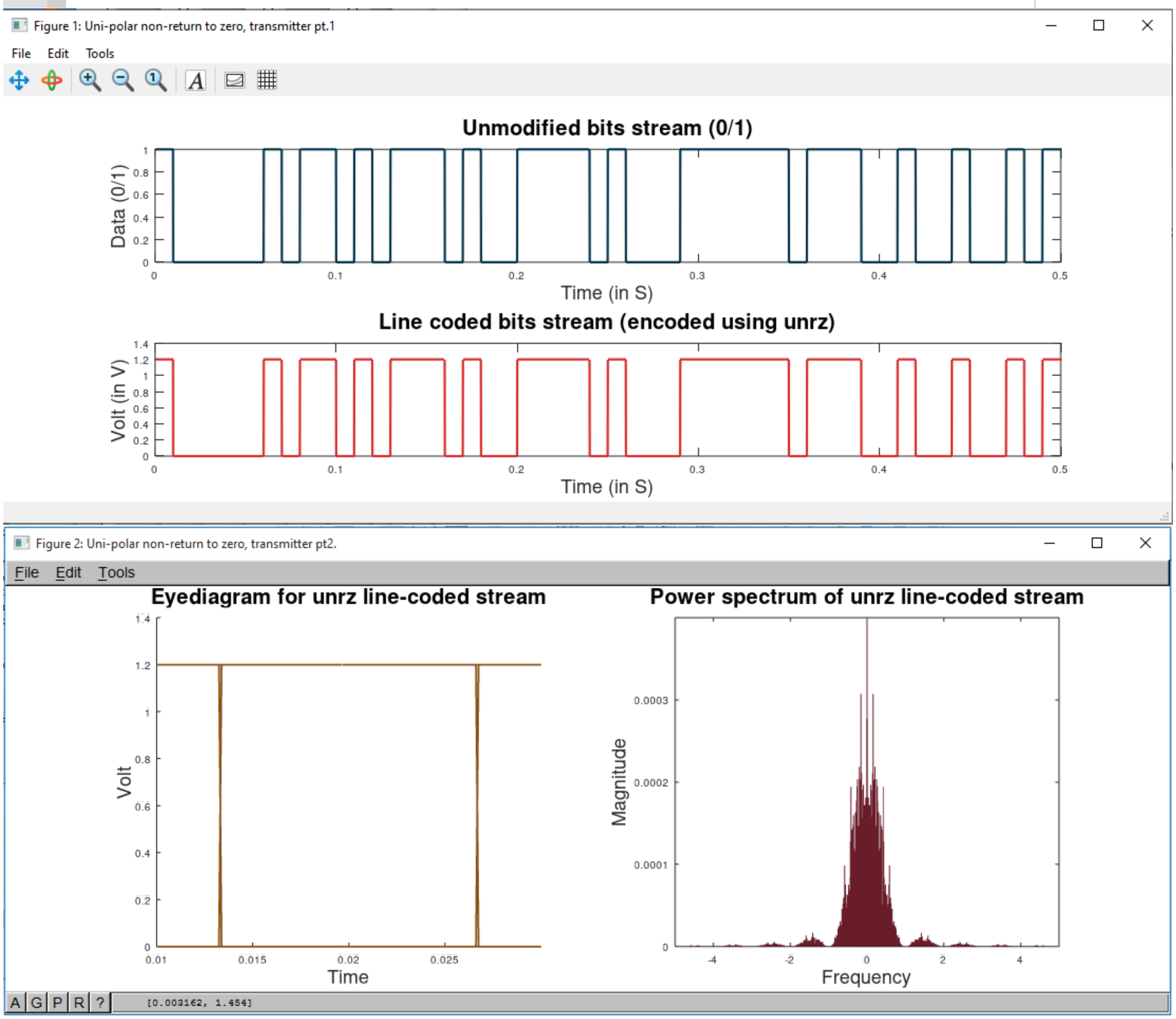
For part 1 point 3, "Plot the corresponding Eye diagram."

Figure 2: Subplot 2

For part 1 point 4, "Plot the spectral domains of the pulses (square of the Fourier transform)."

a. Unipolar non-return to zero:

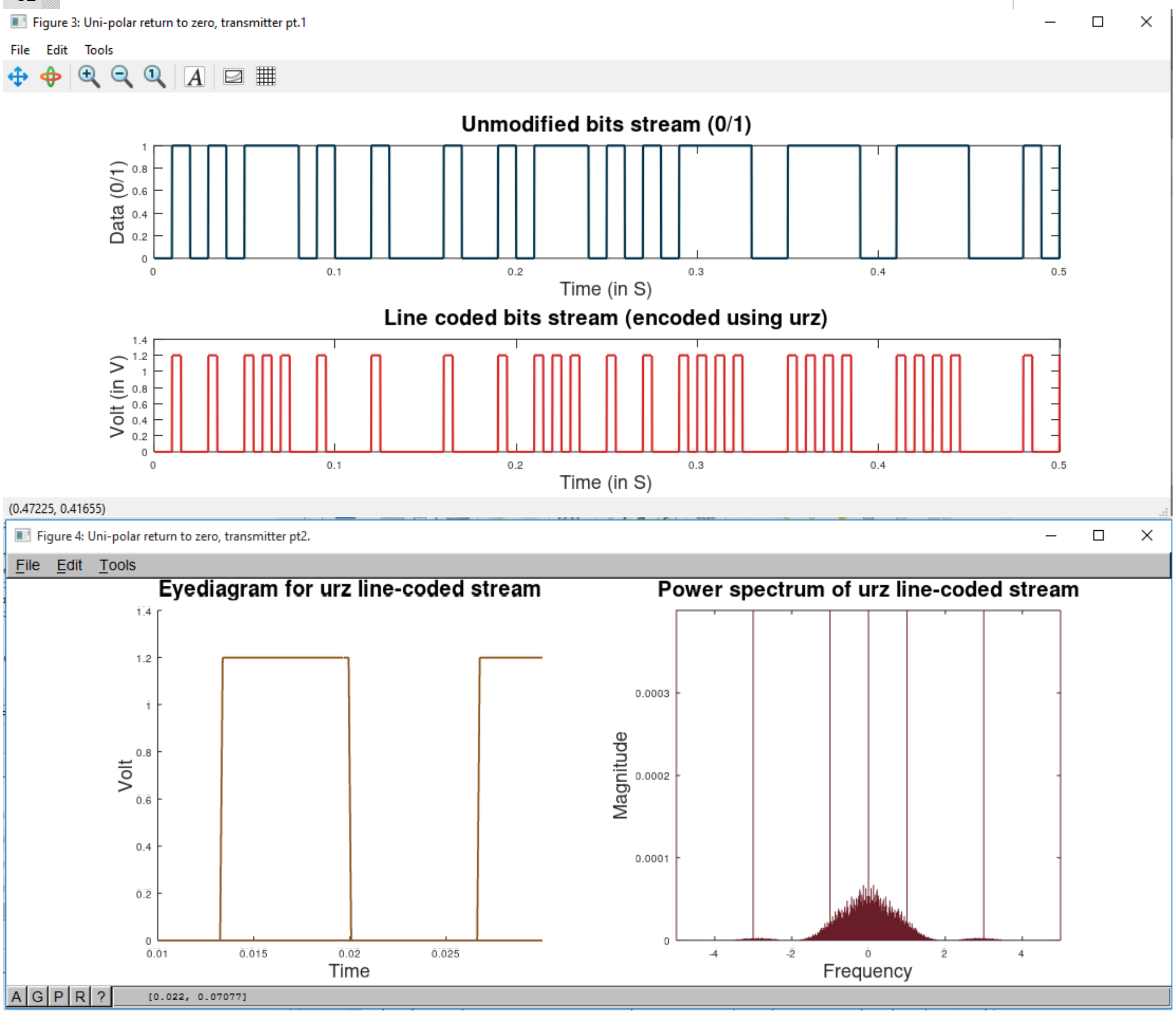
```
## 1. Uni-polar non-return to zero
tx1 = transmitter();
tx1 = tx1.create_stream(10000);
tx1 = tx1.line_code('unrz', 1.2);

figure('Name', 'Uni-polar non-return to zero, transmitter pt.1', 'Position', figure_position);
subplot(2, 1, 1);
tx1.plot('stream');
subplot(2, 1, 2);
tx1.plot('line_coded_stream');

figure('Name', 'Uni-polar non-return to zero, transmitter pt2.', 'Position', figure_position);
subplot(1, 2, 1);
tx1.plot_eyediagram('line_coded_stream');
subplot(1, 2, 2);
tx1.plot_line_code_power_spectrum();
```



Figures (1) & (2): Unipolar non-return to zero transmitter plots

b. Unipolar return to zero:

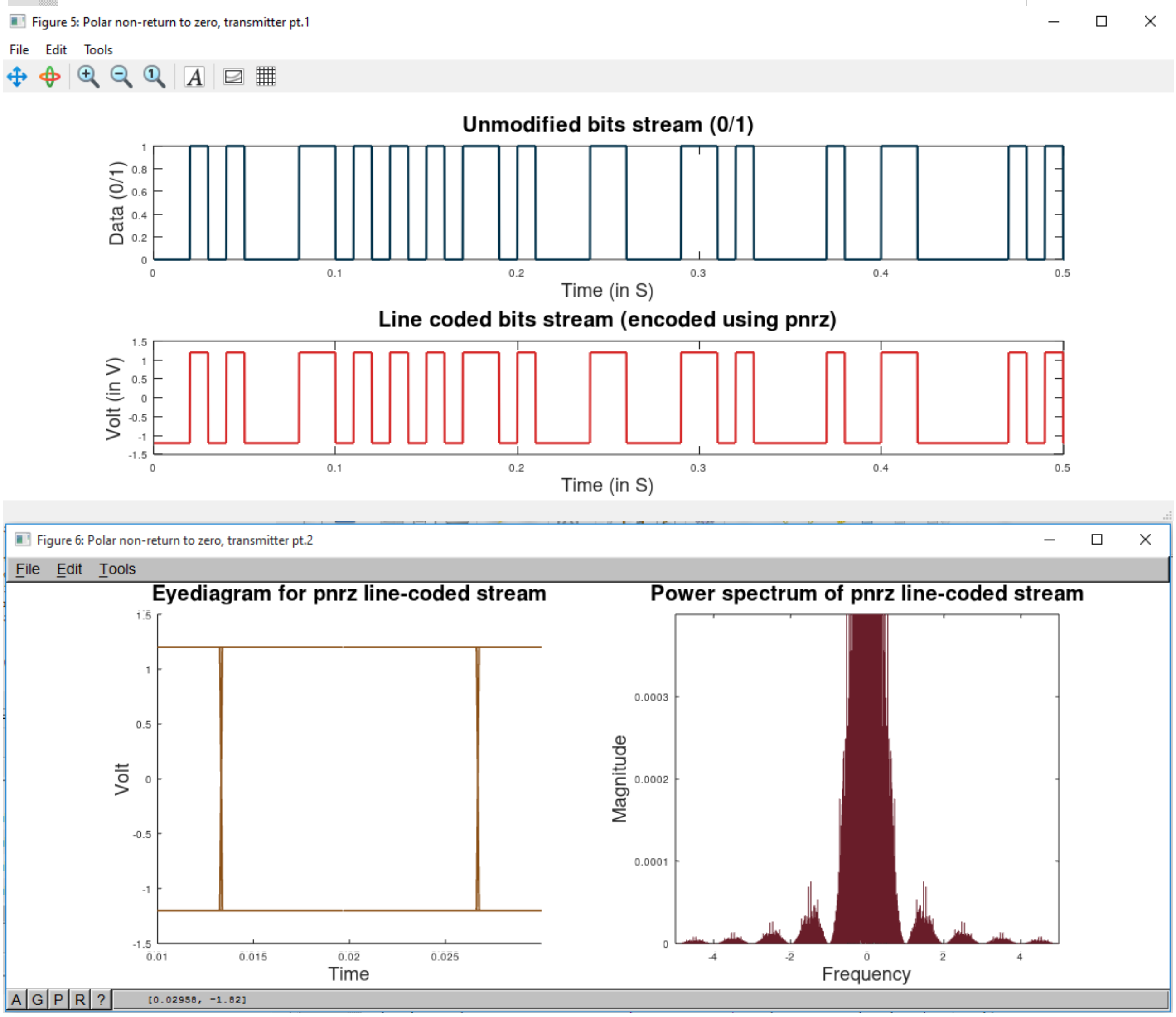
```
## 2. Uni-polar return to zero
tx2 = transmitter();
tx2 = tx2.create_stream(10000);
tx2 = tx2.line_code('urz', 1.2);

figure('Name', 'Uni-polar return to zero, transmitter pt.1', 'Position', figure_position);
subplot(2, 1, 1);
tx2.plot('stream');
subplot(2, 1, 2);
tx2.plot('line_coded_stream');

figure('Name', 'Uni-polar return to zero, transmitter pt2.', 'Position', figure_position);
subplot(1, 2, 1);
tx2.plot_eyediagram('line_coded_stream');
subplot(1, 2, 2);
tx2.plot_line_code_power_spectrum();

```



Figures (3) & (4): Unipolar return to zero transmitter plots

c. Polar non-return to zero:

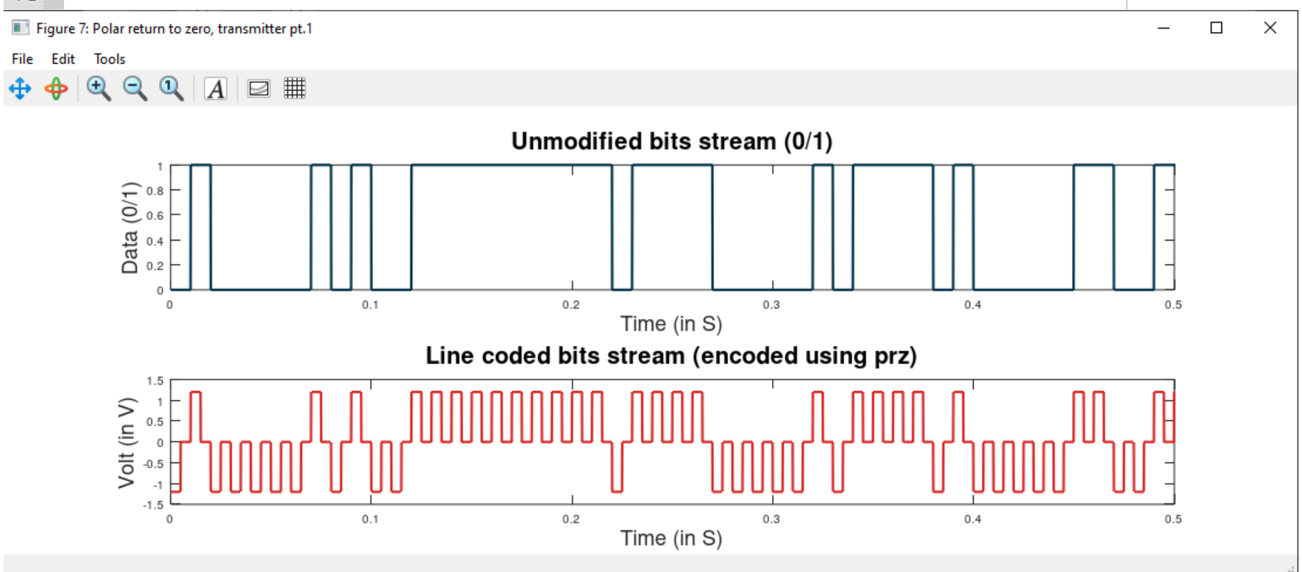
```
## 3. Polar non-return to zero
tx3 = transmitter();
tx3 = tx3.create_stream(10000);
tx3 = tx3.line_code('pnrz', 1.2);

figure('Name', 'Polar non-return to zero, transmitter pt.1', 'Position', figure_position);
subplot(2, 1, 1);
tx3.plot('stream');
subplot(2, 1, 2);
tx3.plot('line_coded_stream');

figure('Name', 'Polar non-return to zero, transmitter pt.2', 'Position', figure_position);
subplot(1, 2, 1);
tx3.plot_eyediagram('line_coded_stream');
subplot(1, 2, 2);
tx3.plot_line_code_power_spectrum();
```

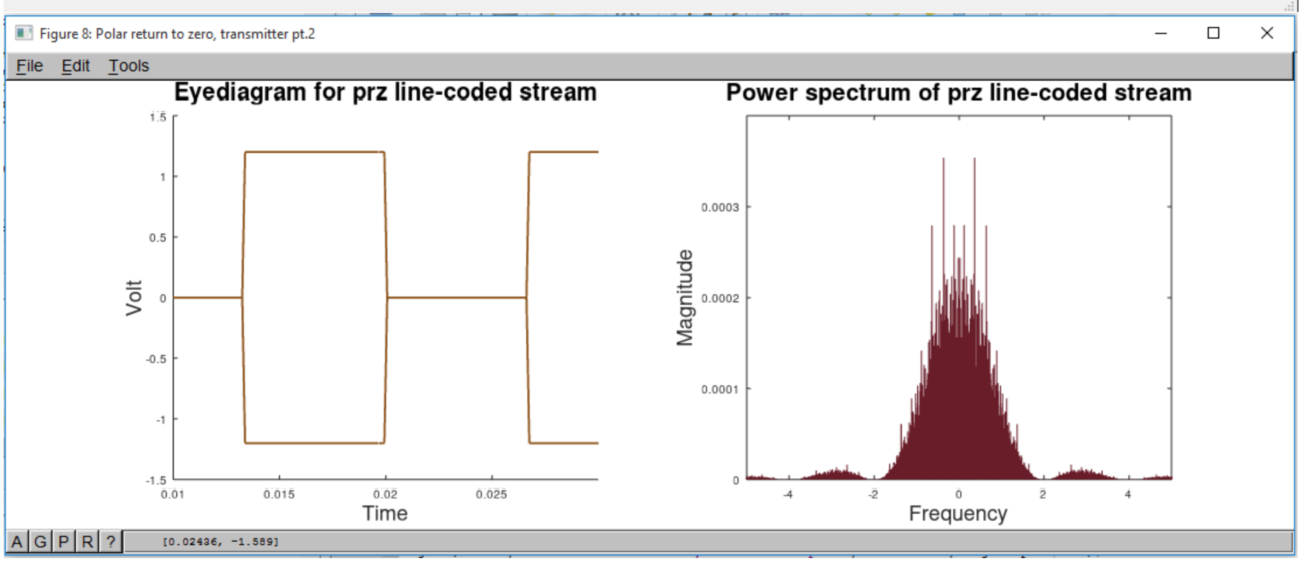


Figures (5) & (6): Polar non-return to zero transmitter plots

d. Polar return to zero:

```
## 4. Polar return to zero
tx4 = transmitter();
tx4 = tx4.create_stream(10000);
tx4 = tx4.line_code('prz', 1.2);

figure('Name', 'Polar return to zero, transmitter pt.1', 'Position', figure_position);
subplot(2, 1, 1);
tx4.plot('stream');
subplot(2, 1, 2);
tx4.plot('line_coded_stream');

figure('Name', 'Polar return to zero, transmitter pt.2', 'Position', figure_position);
subplot(1, 2, 1);
tx4.plot_eyediagram('line_coded_stream');
subplot(1, 2, 2);
tx4.plot_line_code_power_spectrum();
```

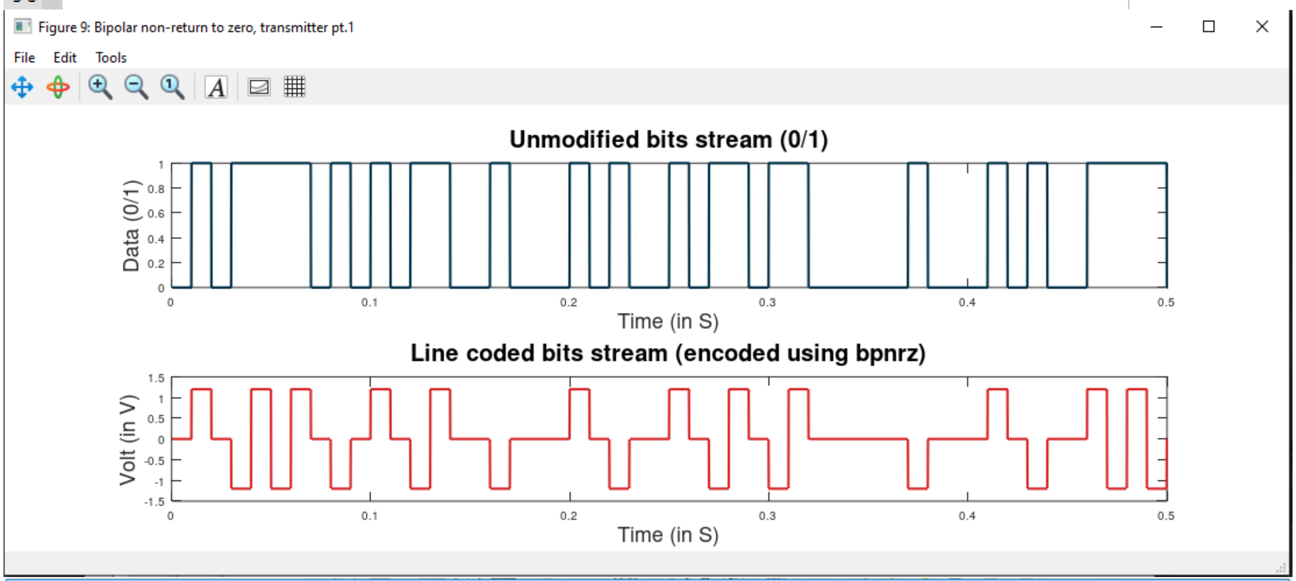


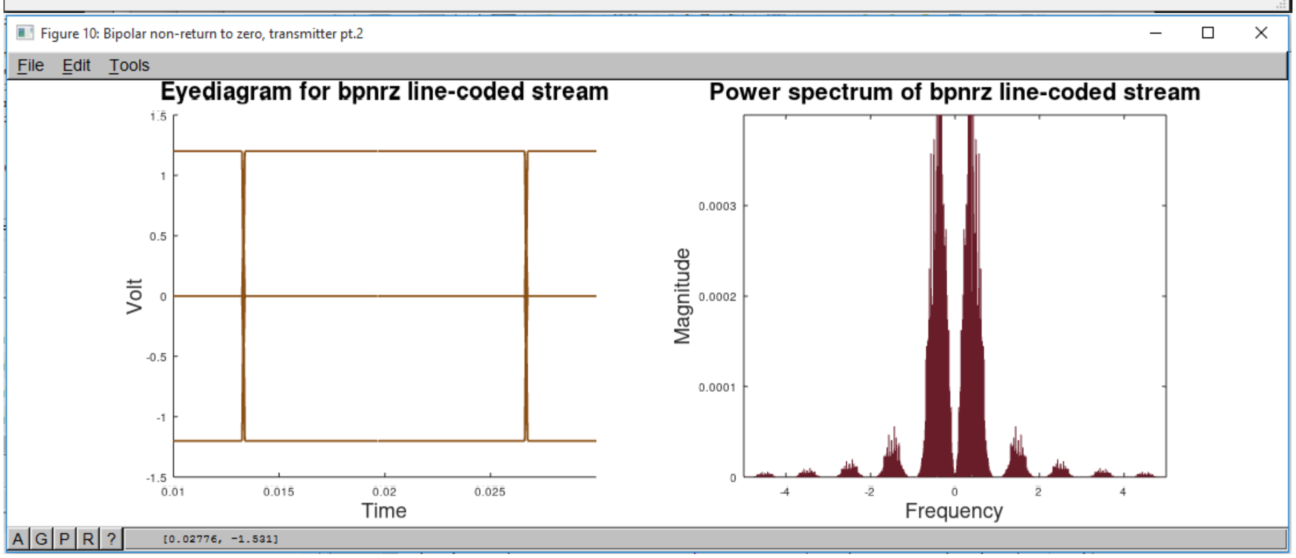


Figures (7) & (8): Polar return to zero transmitter plots

e. Bipolar non-return to zero plots:

```
## 5. Bipolar non-return to zero
tx5 = transmitter();
tx5 = tx5.create_stream(10000);
tx5 = tx5.line_code('bpnrz', 1.2);

figure('Name', 'Bipolar non-return to zero, transmitter pt.1', 'Position', figure_position);
subplot(2, 1, 1);
tx5.plot('stream');
subplot(2, 1, 2);
tx5.plot('line_coded_stream');

figure('Name', 'Bipolar non-return to zero, transmitter pt.2', 'Position', figure_position);
subplot(1, 2, 1);
tx5.plot_eyediagram('line_coded_stream');
subplot(1, 2, 2);
tx5.plot_line_code_power_spectrum();
```

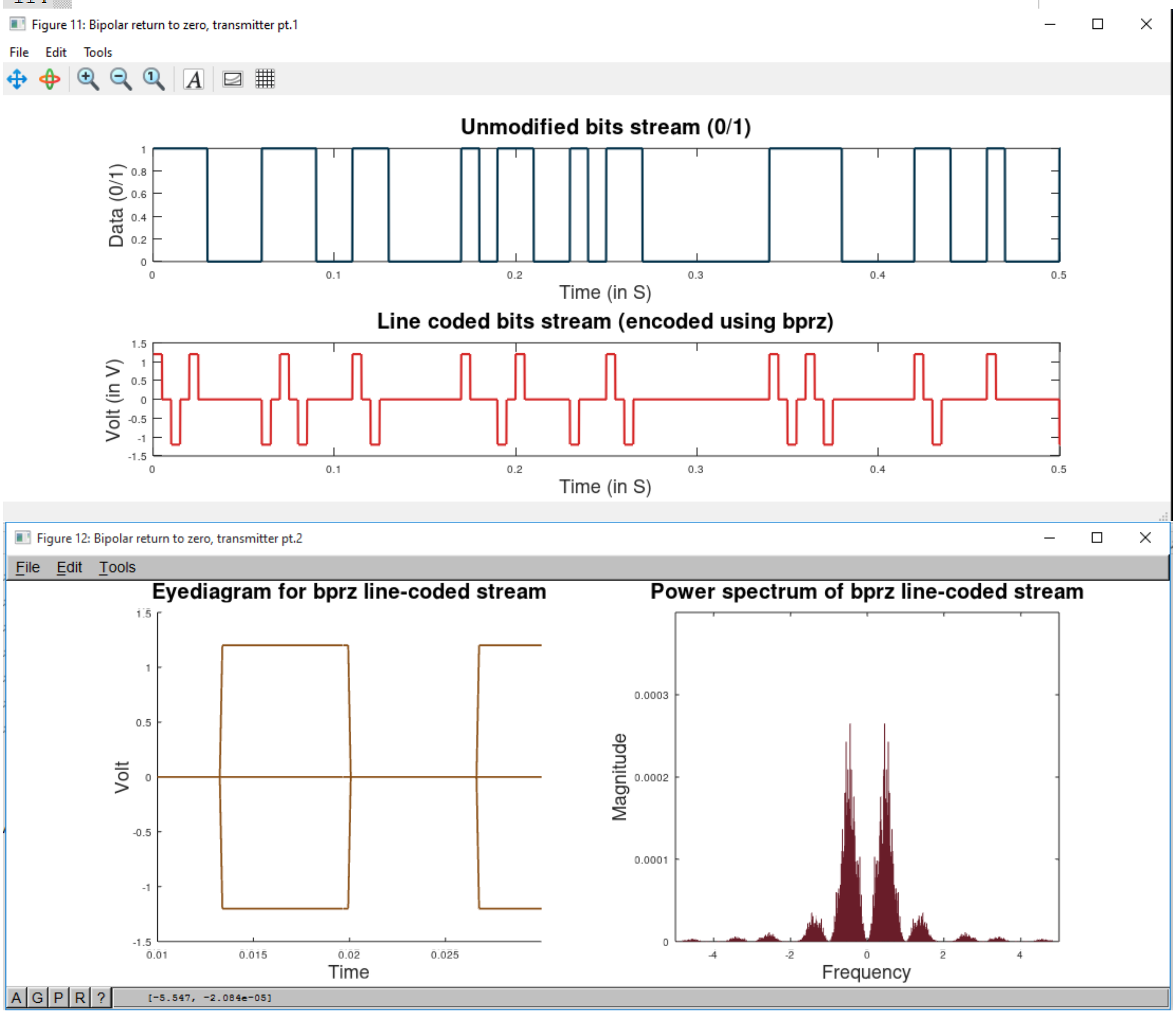




Figures (9) & (10): Bipolar non-return to zero transmitter plots

f. Bipolar return to zero:

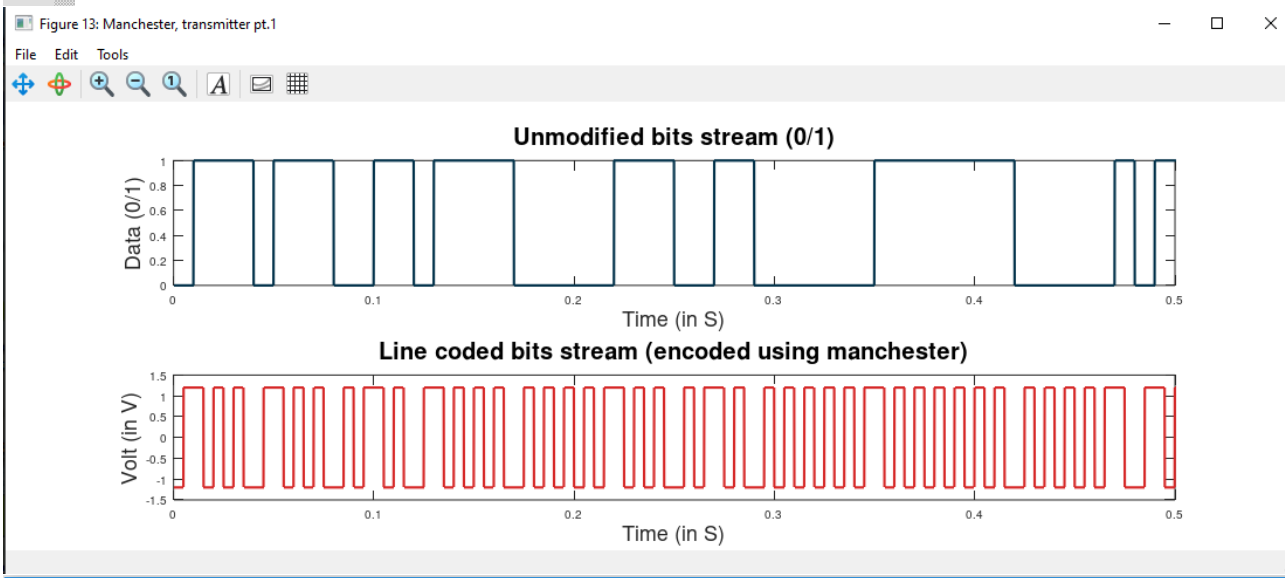
```

## 6. Bipolar return to zero
tx6 = transmitter();
tx6 = tx6.create_stream(10000);
tx6 = tx6.line_code('bprz', 1.2);

figure('Name', 'Bipolar return to zero, transmitter pt.1', 'Position', figure_position);
subplot(2, 1, 1);
tx6.plot('stream');
subplot(2, 1, 2);
tx6.plot('line_coded_stream');

figure('Name', 'Bipolar return to zero, transmitter pt.2', 'Position', figure_position);
subplot(1, 2, 1);
tx6.plot_eyediagram('line_coded_stream');
subplot(1, 2, 2);
tx6.plot_line_code_power_spectrum();

```

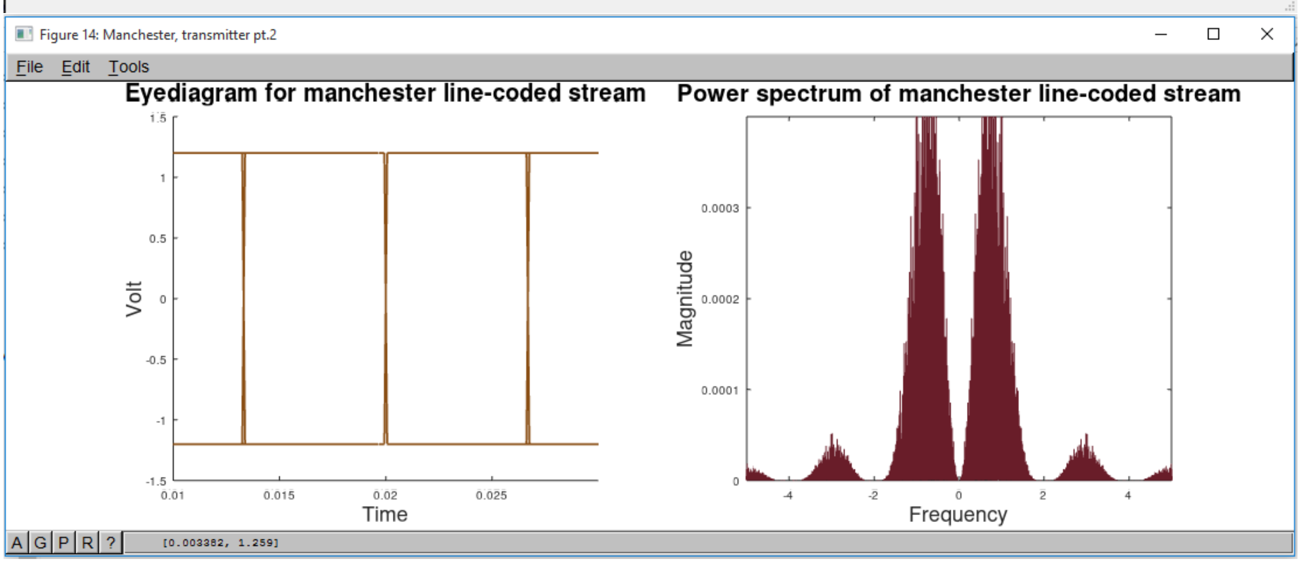


Figures (11) & (12): Bipolar return to zero transmitter plots

g. Manchester:

```
## 7. Manchester
tx7 = transmitter();
tx7 = tx7.create_stream(10000);
tx7 = tx7.line_code('manchester', 1.2);

figure('Name', 'Manchester, transmitter pt.1', 'Position', figure_position);
subplot(2, 1, 1);
tx7.plot('stream');
subplot(2, 1, 2);
tx7.plot('line_coded_stream');

figure('Name', 'Manchester, transmitter pt.2', 'Position', figure_position);
subplot(1, 2, 1);
tx7.plot_eyediagram('line_coded_stream');
subplot(1, 2, 2);
tx7.plot_line_code_power_spectrum();
```





Figures (13) & (14): Manchester transmitter plots

2. Part 1: Receiver:

General format that will be followed till the end of this section (over two pages):

Snapshot from main.m

RX object construction from the previously constructed non-return to zero TX object
Function call to extract data from TX object's line code

#Figure 1, subplots 1 & 2

#RX object construction from the previously constructed return to zero TX object
#Function call to extract data from TX object's line code

#Figure 2, subplots 1 & 2

#Figure 3 construction, subplots 1 & 2, constructed after function call to sweep_over_sigma and plot_ber, both explained in detail in the "Code dive" section.

Figure 1: Subplot 1

For part 1, point 5, "Design a receiver which consists of a decision device."
Subplot 1 is the received non-return to zero waveform.

Figure 1: Subplot 2

For part 1, point 5 as well.
Subplot 2 is the extracted data from the received waveform.

General format that will be followed till the end of this section (continued):

Figure 2: Subplot 1

For part 1, point 5, "Design a receiver which consists of a decision device."
Subplot 1 is the received return to zero waveform.

Figure 2: Subplot 2

For part 1, point 5 as well.
Subplot 2 is the extracted data from the received waveform.

Figure 3: Subplot 1 Figure 3: Subplot 2

For part 1 points 6, 7, 8, 9, and 11,

"6. Compare the output of the decision level with the generated stream of bits in the transmitter and count number of errors. Then calculate bit error rate (BER).
7. Repeat the previous steps for different line coding.
8. Add noise to the received.
9. Sweep on the value of sigma (10 values ranges from 0 to the maximum supply voltage) and calculate the corresponding BER for each value of sigma."

a. Unipolar line-coding:

```
# 1. Uni-polar:
rx1 = receiver(tx1);
rx1 = rx1.extract_stream_from_line_code();

figure('Name', 'Uni-polar non-return to zero, receiver', 'Position', figure_position);
subplot(2, 1, 1);
rx1.plot('rx_line_coded_stream');
subplot(2, 1, 2);
rx1.plot('extracted_stream');

rx2 = receiver(tx2);
rx2 = rx2.extract_stream_from_line_code();

figure('Name', 'Uni-polar return to zero, receiver', 'Position', figure_position);
subplot(2, 1, 1);
rx2.plot('rx_line_coded_stream');
subplot(2, 1, 2);
rx2.plot('extracted_stream');

figure('Name', 'Uni-polar BER plots', 'Position', figure_position);
subplot(1, 2, 1);
[sigma_array, ber_array] = sweep_over_sigma(tx1, rx1, 40);
plot_ber(sigma_array, ber_array, tx1.line_coding_style);

subplot(1, 2, 2)
[sigma_array, ber_array] = sweep_over_sigma(tx2, rx2, 40);
plot_ber(sigma_array, ber_array, tx2.line_coding_style);
```

This number is explained in “code dive”. It’s the number of values sigma sweeps over. 40 produces a smoother graph than the required 10.

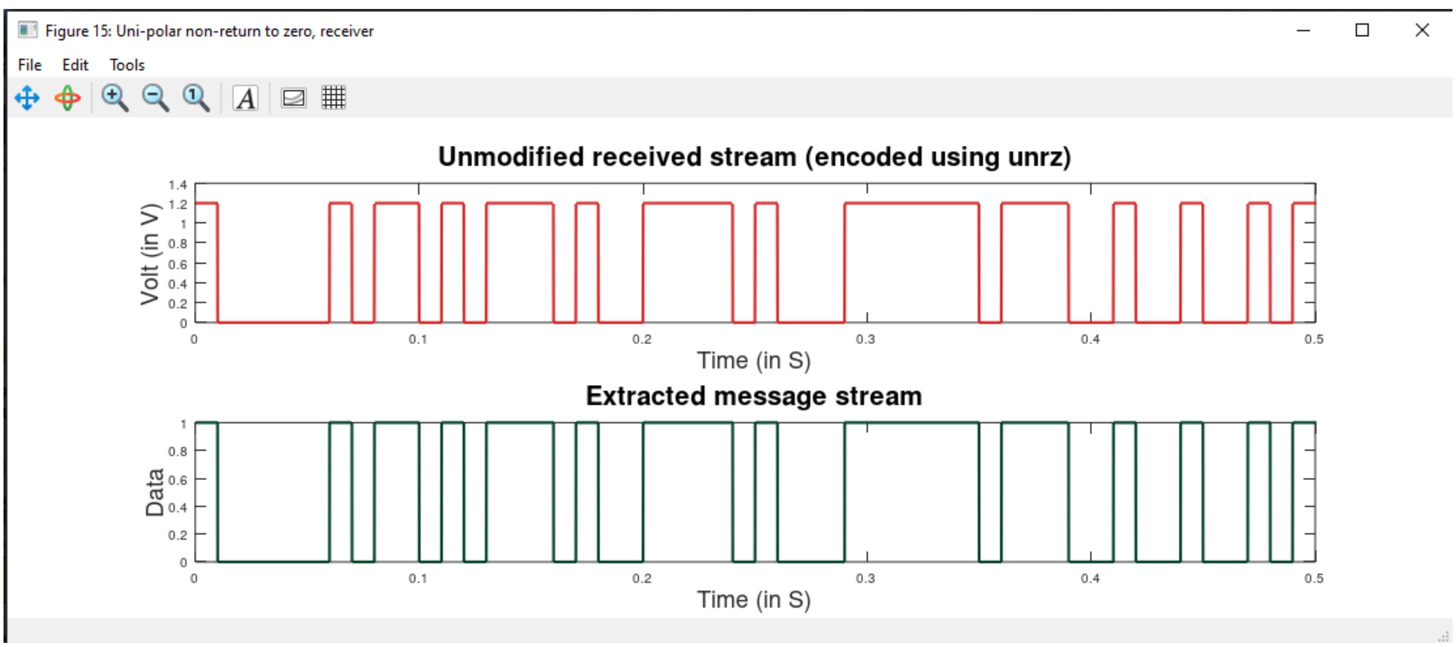


Figure (15): Unipolar non-return to zero receiver plots

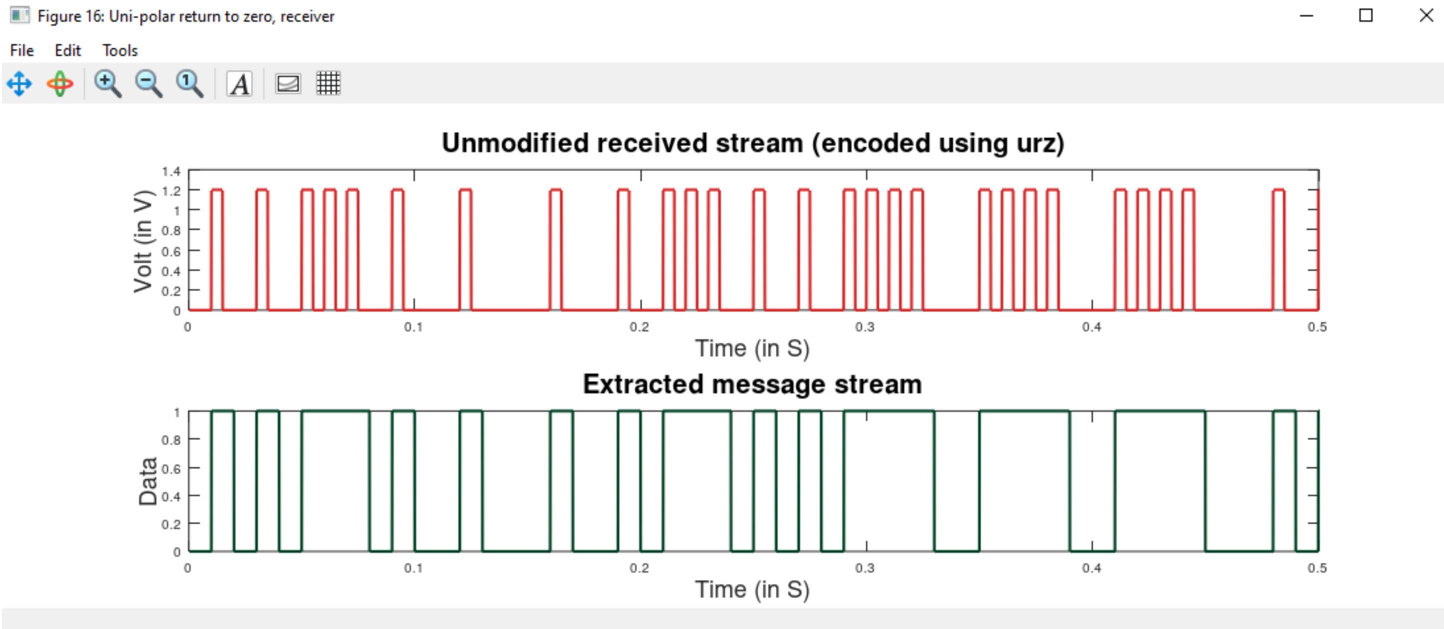


Figure (16): Unipolar return to zero receiver plots

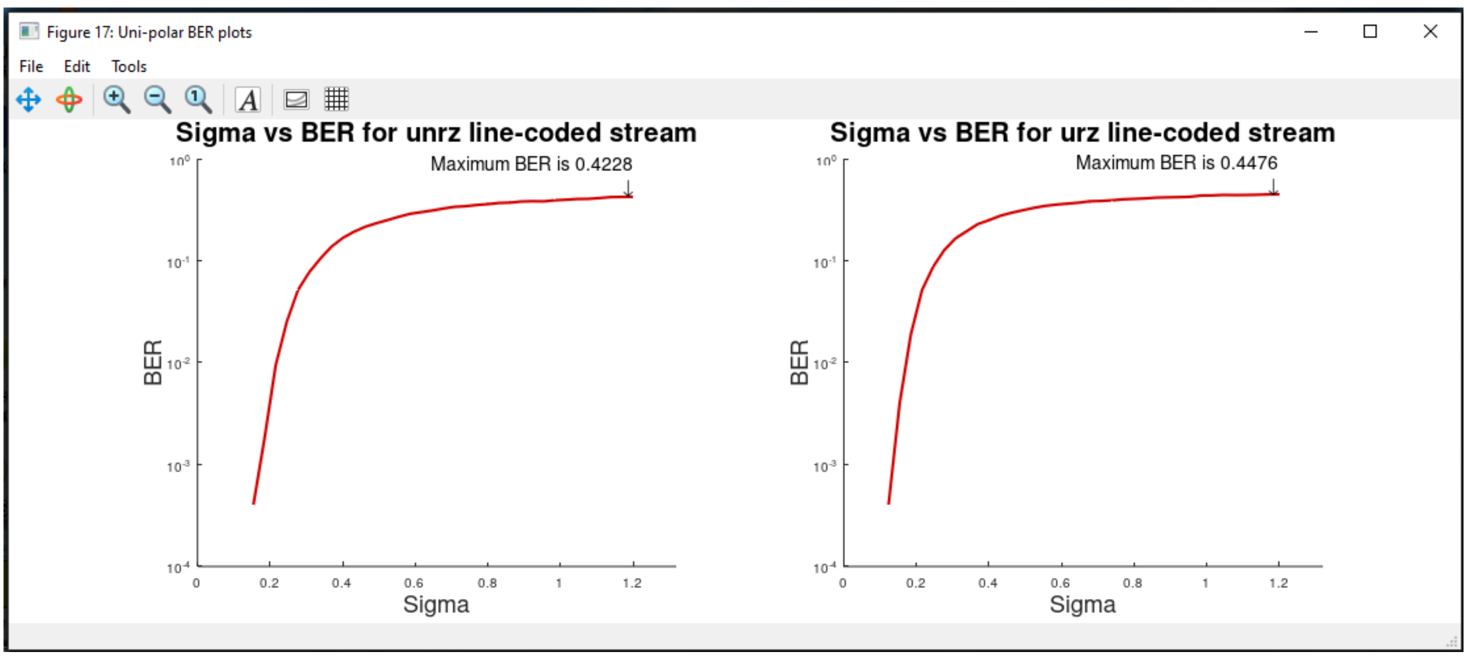


Figure (17): Unipolar Sigma VS BER plots

b. Polar line-coding:

```
# 2. Polar:
rx3 = receiver(tx3);
rx3 = rx3.extract_stream_from_line_code();

figure('Name', 'Polar non-return to zero, receiver', 'Position', figure_position);
subplot(2, 1, 1);
rx3.plot('rx_line_coded_stream');
subplot(2, 1, 2);
rx3.plot('extracted_stream');

rx4 = receiver(tx4);
rx4 = rx4.extract_stream_from_line_code();

figure('Name', 'Polar return to zero, receiver', 'Position', figure_position);
subplot(2, 1, 1);
rx4.plot('rx_line_coded_stream');
subplot(2, 1, 2);
rx4.plot('extracted_stream');

figure('Name', 'Polar BER plots', 'Position', figure_position);
subplot(1, 2, 1);
[sigma_array, ber_array] = sweep_over_sigma(tx3, rx3, 40);
plot_ber(sigma_array, ber_array, tx3.line_coding_style);

subplot(1, 2, 2)
[sigma_array, ber_array] = sweep_over_sigma(tx4, rx4, 40);
plot_ber(sigma_array, ber_array, tx4.line_coding_style);

```

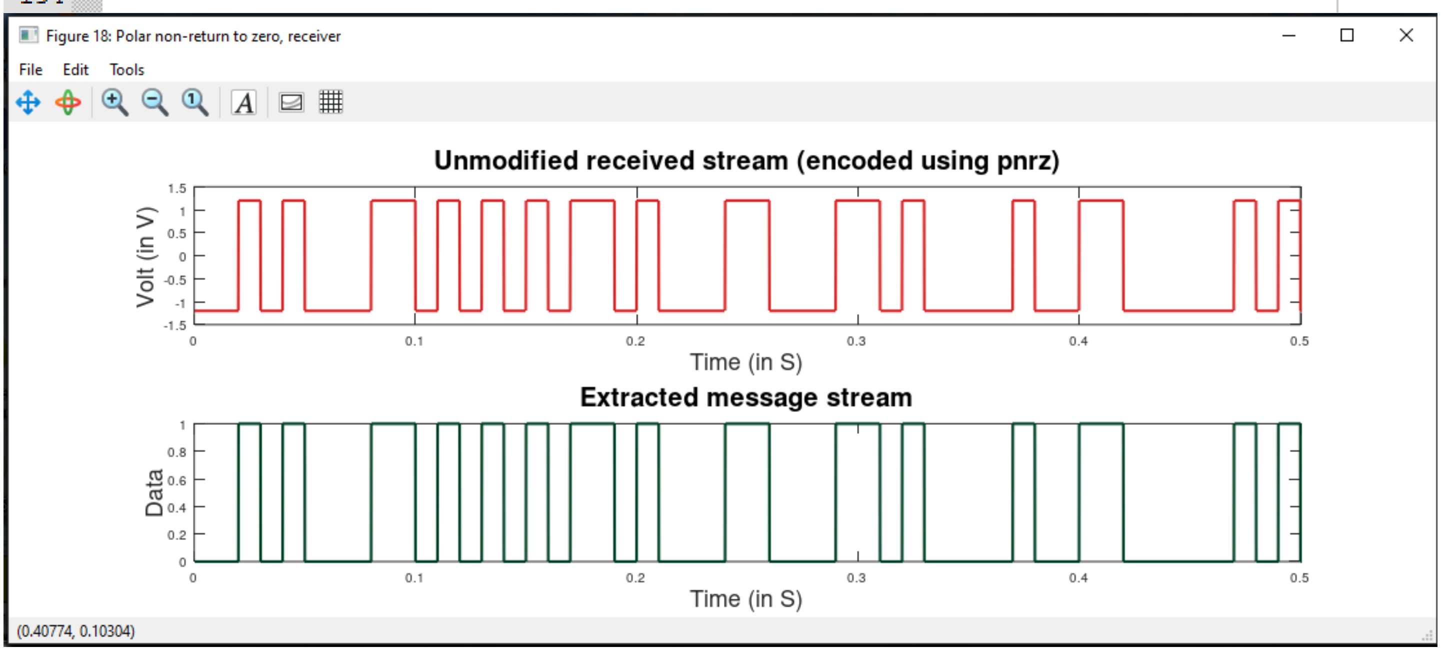


Figure (18): Polar non-return to zero receiver plots

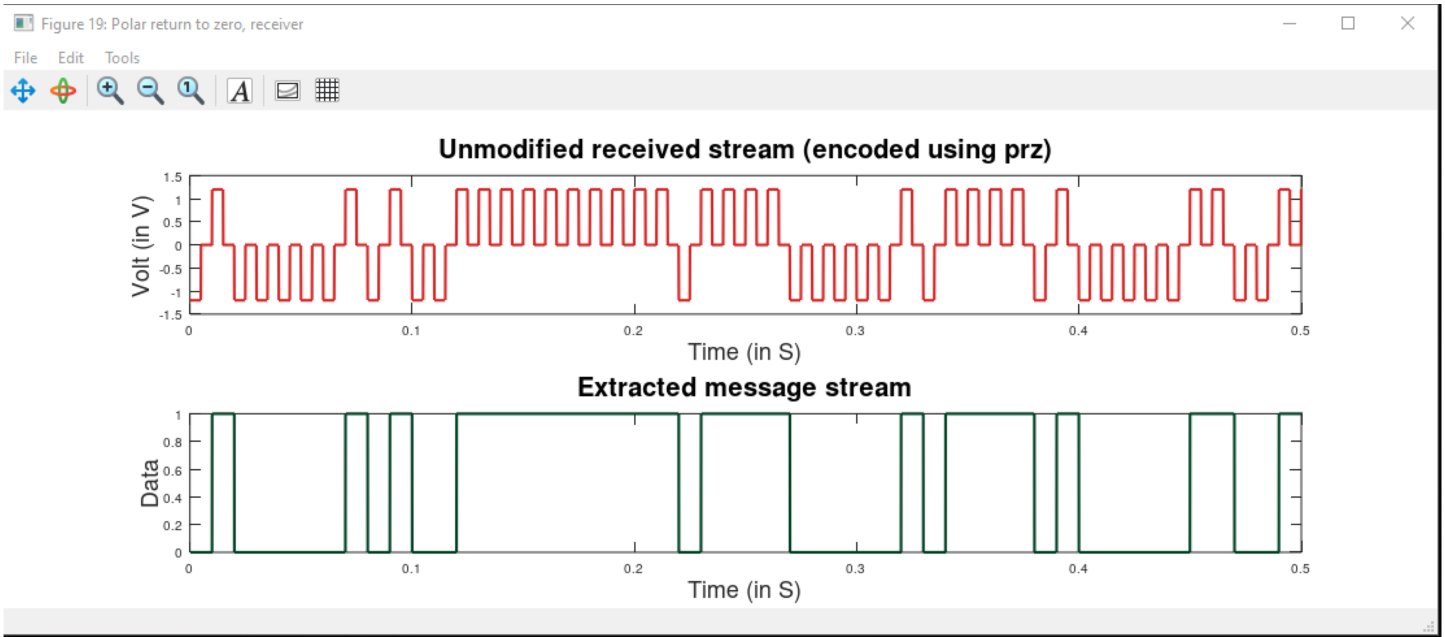


Figure (19): Polar return to zero receiver plots

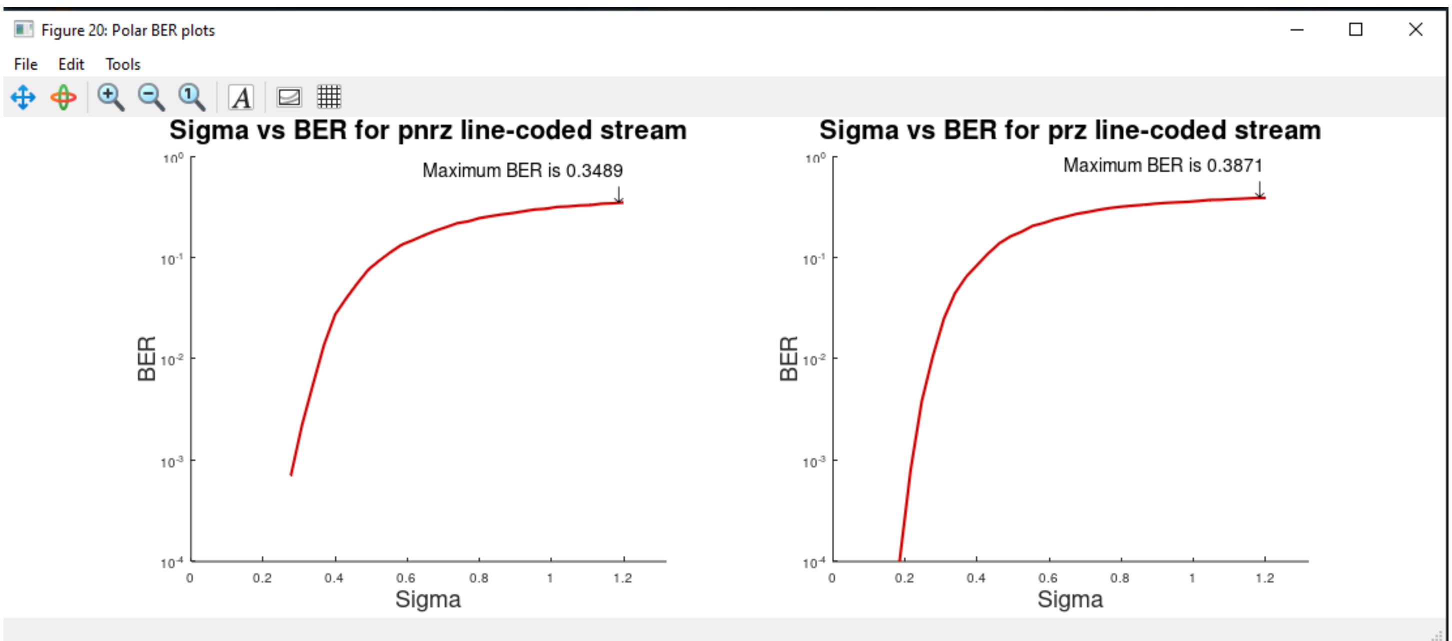


Figure (20): Polar Sigma VS BER plots

c. Bipolar line-coding:

```
# 3. Bipolar:
rx5 = receiver(tx5);
rx5 = rx5.extract_stream_from_line_code();

figure('Name', 'Bipolar non-return to zero, receiver', 'Position', figure_position);
subplot(2, 1, 1);
rx5.plot('rx_line_coded_stream');
subplot(2, 1, 2);
rx5.plot('extracted_stream');

rx6 = receiver(tx6);
rx6 = rx6.extract_stream_from_line_code();

figure('Name', 'Bipolar return to zero, receiver', 'Position', figure_position);
subplot(2, 1, 1);
rx6.plot('rx_line_coded_stream');
subplot(2, 1, 2);
rx6.plot('extracted_stream');

figure('Name', 'Bipolar BER plots', 'Position', figure_position);
subplot(1, 2, 1);
[sigma_array, ber_array, detected_ber_array] = sweep_over_sigma(tx5, rx5, 40);
plot_ber(sigma_array, ber_array, tx5.line_coding_style, detected_ber_array);

subplot(1, 2, 2)
[sigma_array, ber_array, detected_ber_array] = sweep_over_sigma(tx6, rx6, 40);
plot_ber(sigma_array, ber_array, tx6.line_coding_style, detected_ber_array);
```

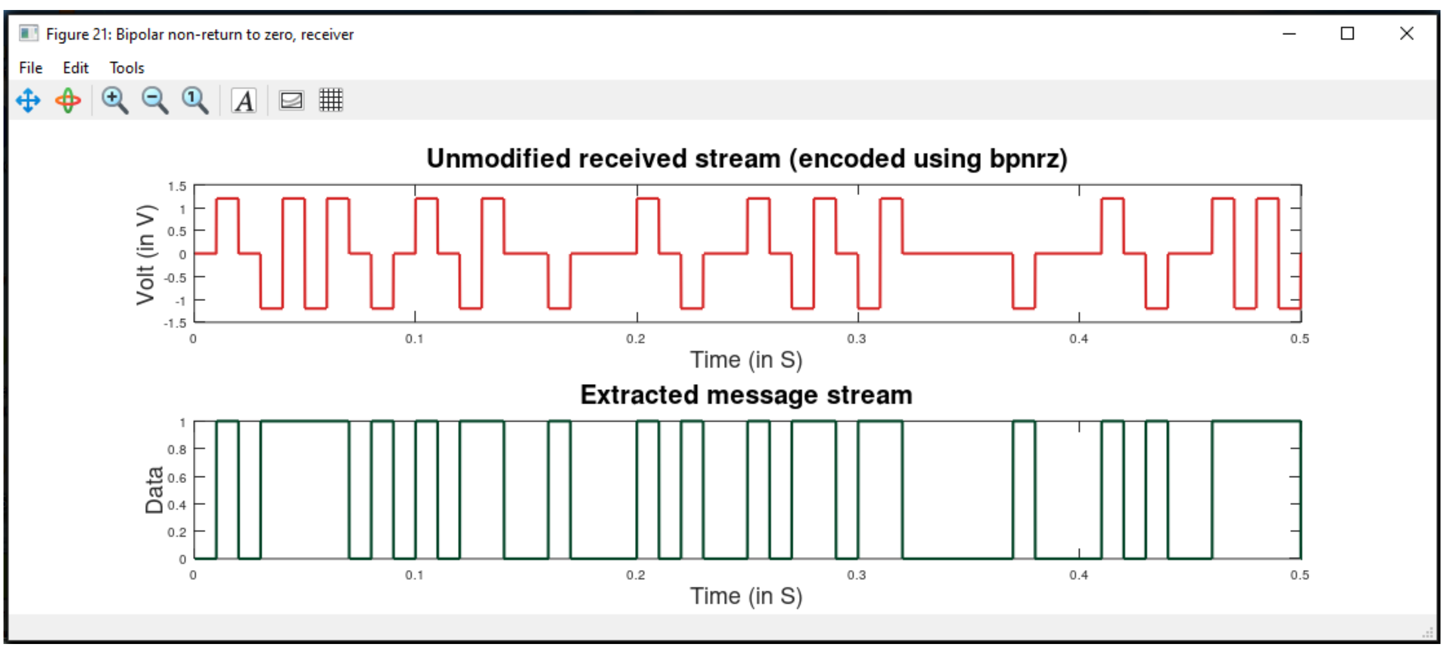


Figure (21): Bipolar non-return to zero receiver plots

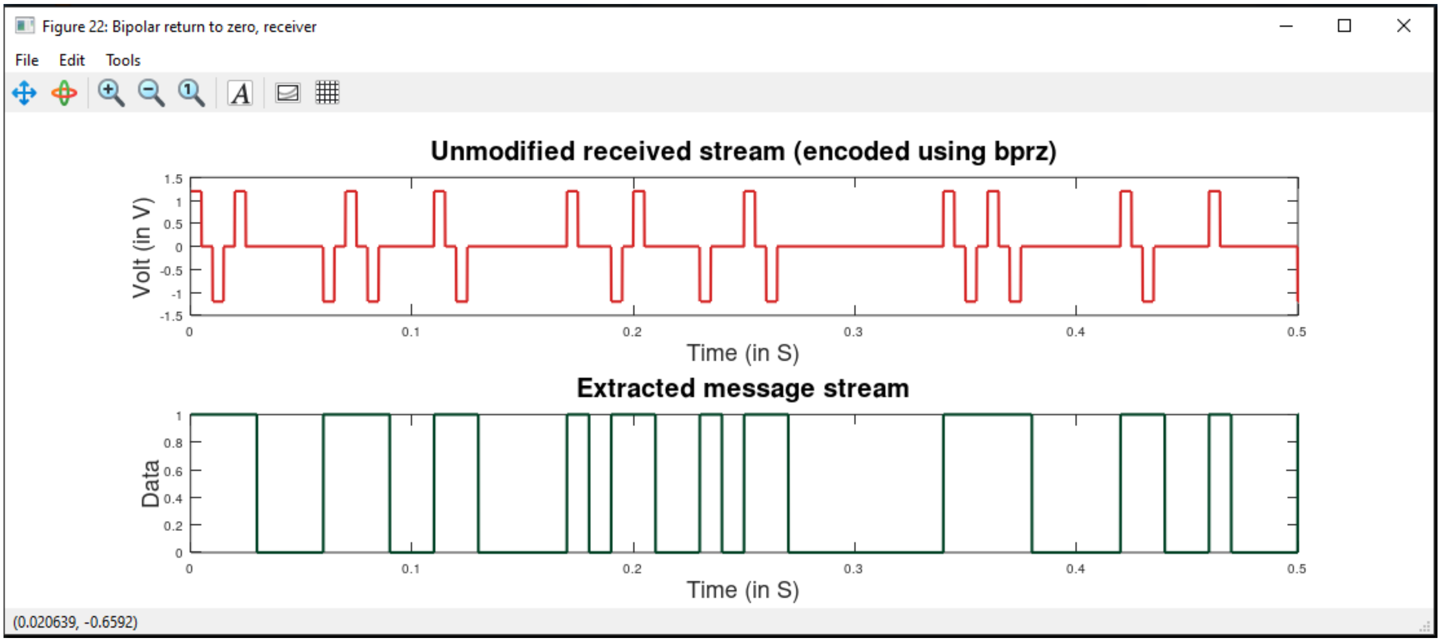


Figure (22): Bipolar non-return to zero receiver plots

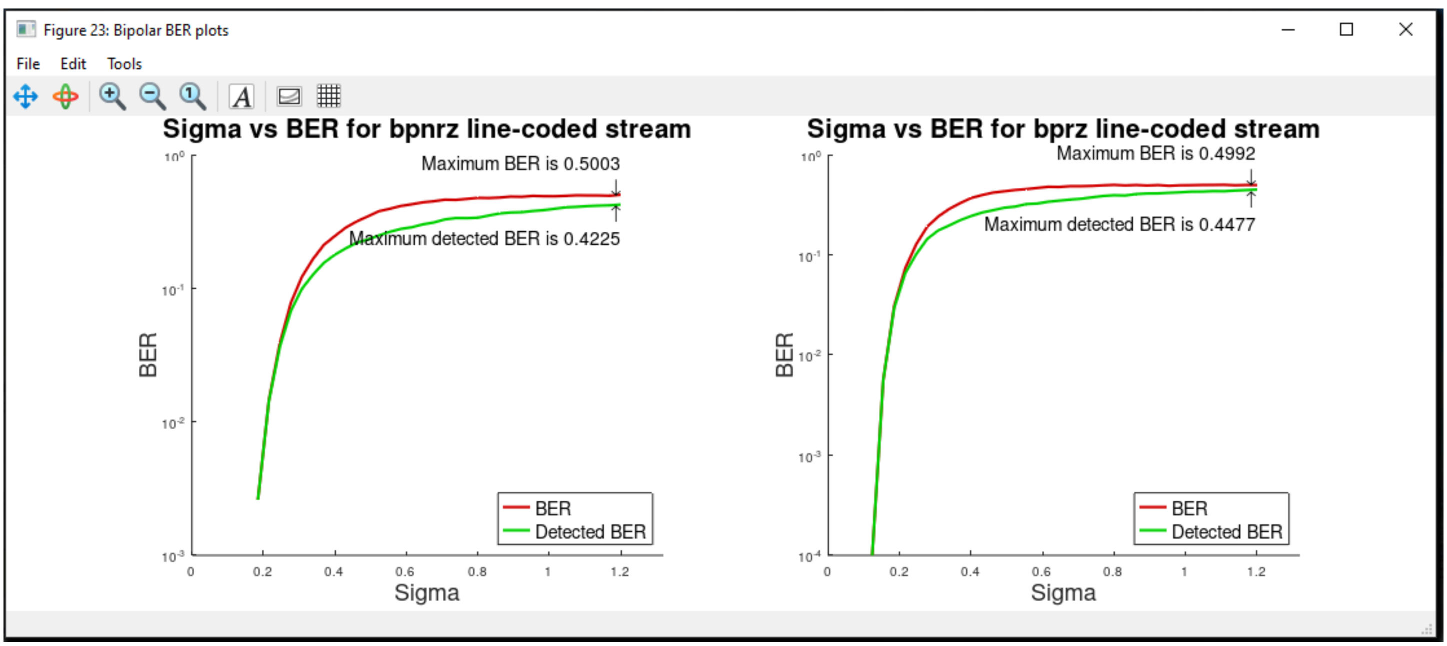
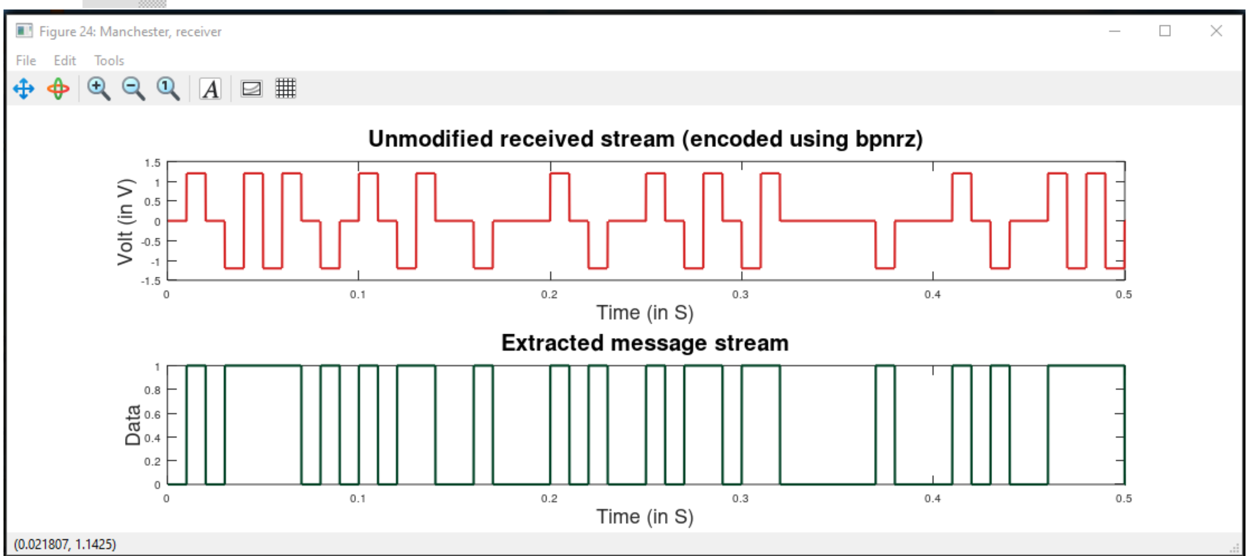


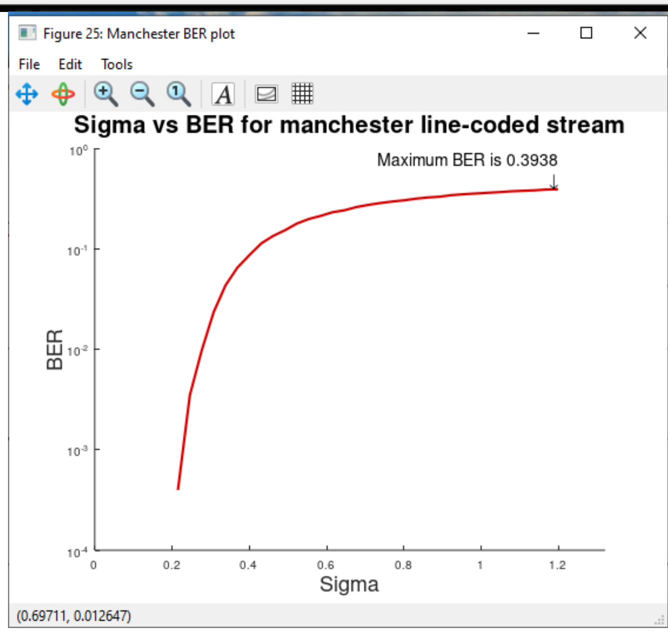
Figure (23): Bipolar Sigma VS BER plots

Note that "Detected BER" is the BER based on the detected bits from the bonus error detection circuit. This is highlighted in "Bonus, highlighted" section.

d. Manchester:

```
# 4. Manchester:
rx7 = receiver(tx7);
rx7 = rx7.extract_stream_from_line_code();

figure('Name', 'Manchester, receiver', 'Position', figure_position);
subplot(2, 1, 1);
rx5.plot('rx_line_coded_stream');
subplot(2, 1, 2);
rx5.plot('extracted_stream');

figure('Name', 'Manchester BER plot');
[sigma_array, ber_array] = sweep_over_sigma(tx7, rx7, 40);
plot_ber(sigma_array, ber_array, tx7.line_coding_style);
```





Figures (24) & (25): Manchester receiver and BER plots

e. Sigma vs. BER for different line-coding styles in the same figure:

This code is not in main.m since it would slow down the current program tremendously. This is from the file overlapping_ber_plots.m. This is for part 1, point 7.

```

#All BER plots
figure;
hold on;

[sigma_array, ber_array] = sweep_over_sigma(tx1, rx1, 40);
semilogy(sigma_array, ber_array, 'LineWidth', 1.5);

[sigma_array, ber_array] = sweep_over_sigma(tx2, rx2, 40);
semilogy(sigma_array, ber_array, 'LineWidth', 1.5);

[sigma_array, ber_array] = sweep_over_sigma(tx3, rx3, 40);
semilogy(sigma_array, ber_array, 'LineWidth', 1.5);

[sigma_array, ber_array] = sweep_over_sigma(tx4, rx4, 40);
semilogy(sigma_array, ber_array, 'LineWidth', 1.5);

[sigma_array, ber_array] = sweep_over_sigma(tx5, rx5, 40);
semilogy(sigma_array, ber_array, 'LineWidth', 1.5);

[sigma_array, ber_array, detected_ber_array] = sweep_over_sigma(tx6, rx6, 40);
semilogy(sigma_array, ber_array, 'LineWidth', 1.5);

[sigma_array, ber_array] = sweep_over_sigma(tx7, rx7, 40);
semilogy(sigma_array, ber_array, 'LineWidth', 1.5);

legend('UNRZ', 'URZ', 'PNRZ', 'PRZ', 'BPNRZ', 'BPRZ', 'MNCHSTR',...
       'Location', 'southeast', 'FontSize', 14);
title(['Sigma vs BER for all line-coding styles'], 'FontSize', 20);
xlabel('Sigma', 'FontSize', 18);
ylabel('BER', 'FontSize', 18);

hold off;

```

Plot in following page.

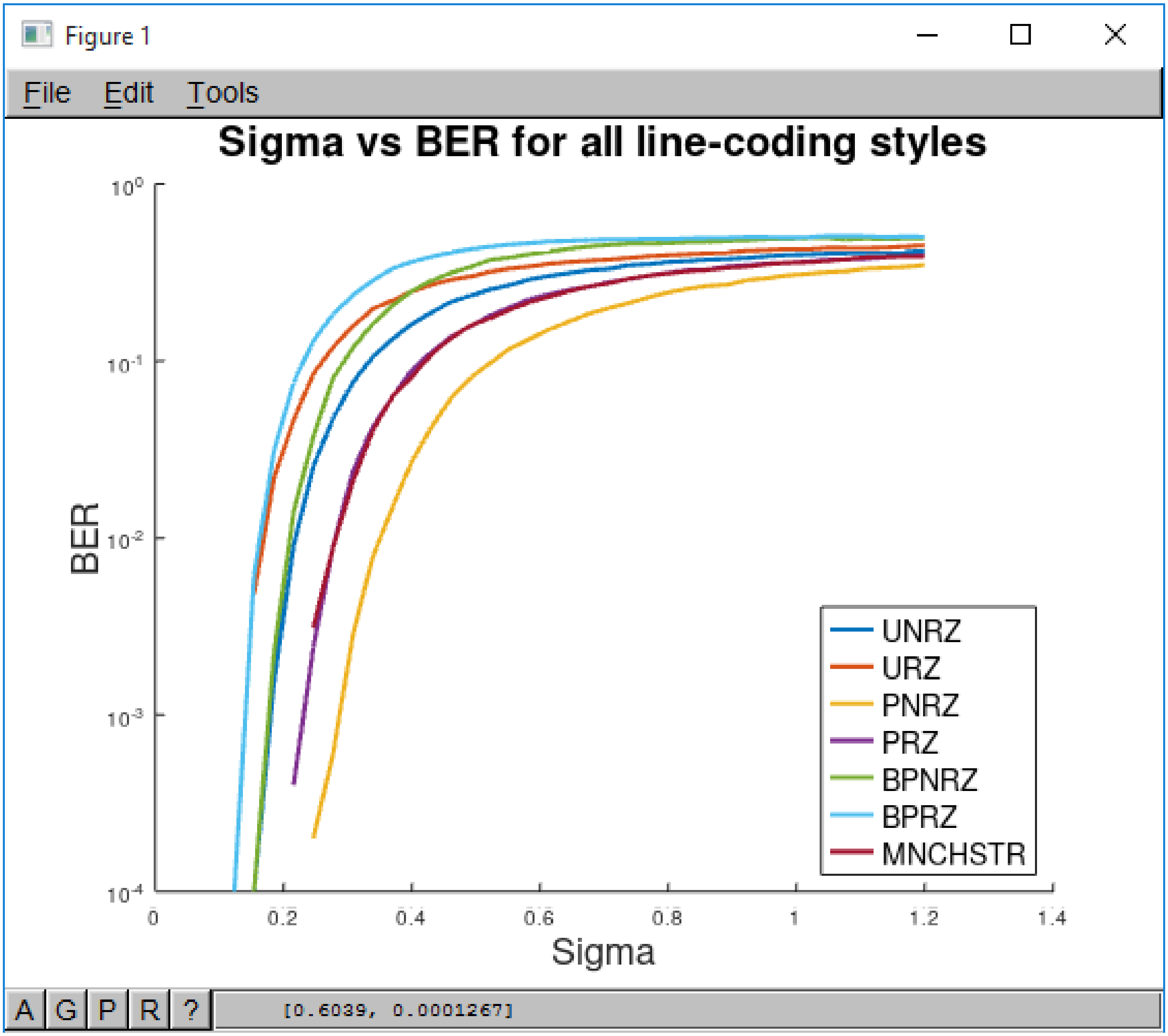


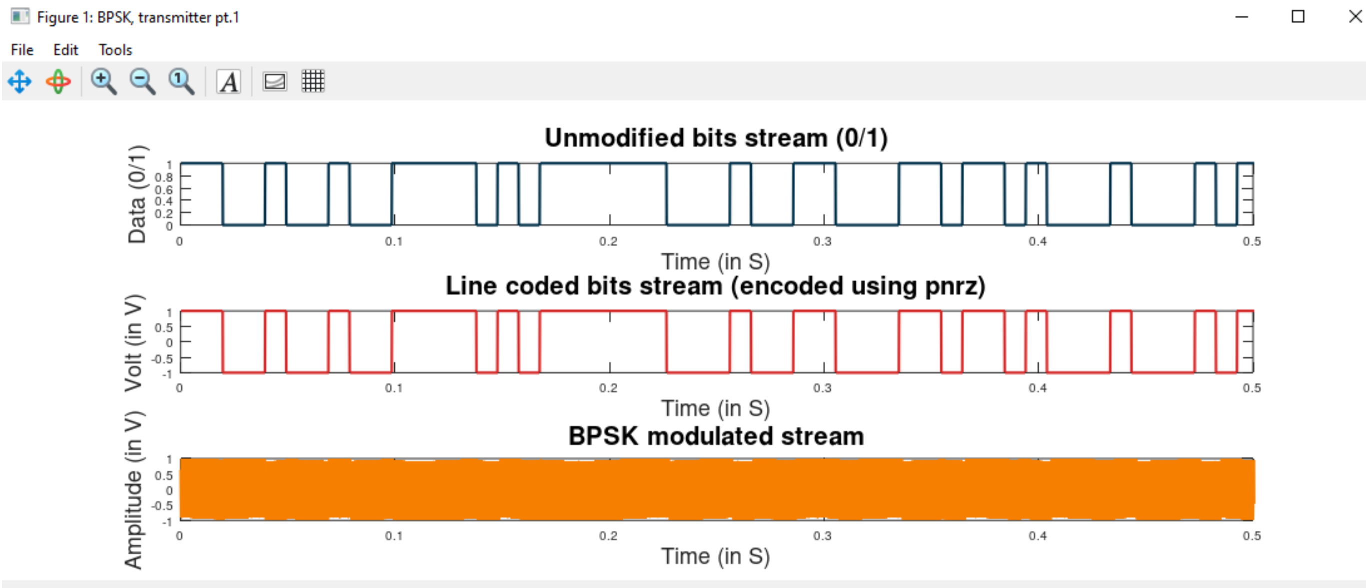
Figure (26): Overlapping Sigma vs BER plots

3. Part 2: Transmitter:

The following code and figures are the solutions to part 2 points 1, 2, 3, 4 and 5.

1. Generate a stream of random bits (100 bit).
2. Line code the stream of bits (pulse shape) according to Polar non return to zero (Maximum voltage +1, Minimum voltage -1).
3. Plot the spectral domains.
4. Plot the time domain of the modulated BPSK signal (fc=1GHz).
5. Plot the spectrum of the modulated BPSK signal.

```

################################################################################

# Part 2: Transmitter:

tx_bpsk = transmitter();
tx_bpsk = tx_bpsk.create_stream(100);
tx_bpsk = tx_bpsk.line_code('pnrz', 1);
tx_bpsk = tx_bpsk.bpsk();

figure('Name', 'BPSK, transmitter pt.1', 'Position', figure_position);
subplot(3, 1, 1);
tx_bpsk.plot('stream');
subplot(3, 1, 2);
tx_bpsk.plot('line_coded_stream');
subplot(3, 1, 3);
tx_bpsk.plot('bpsk_modulated');

figure('Name', 'BPSK, transmitter pt.2', 'Position', figure_position);
subplot(1, 2, 1);
tx_bpsk.plot_line_code_power_spectrum();
subplot(1, 2, 2);
tx_bpsk.plot_bpsk_power_spectrum();

```



The previous plot is, obviously, not the best to gauge how a BPSK modulator works, due to the high frequency of the carrier. When lowering the frequency (for illustration purposes only) the following plot is produced. (Note that these two plots were done on two different random bit streams.)

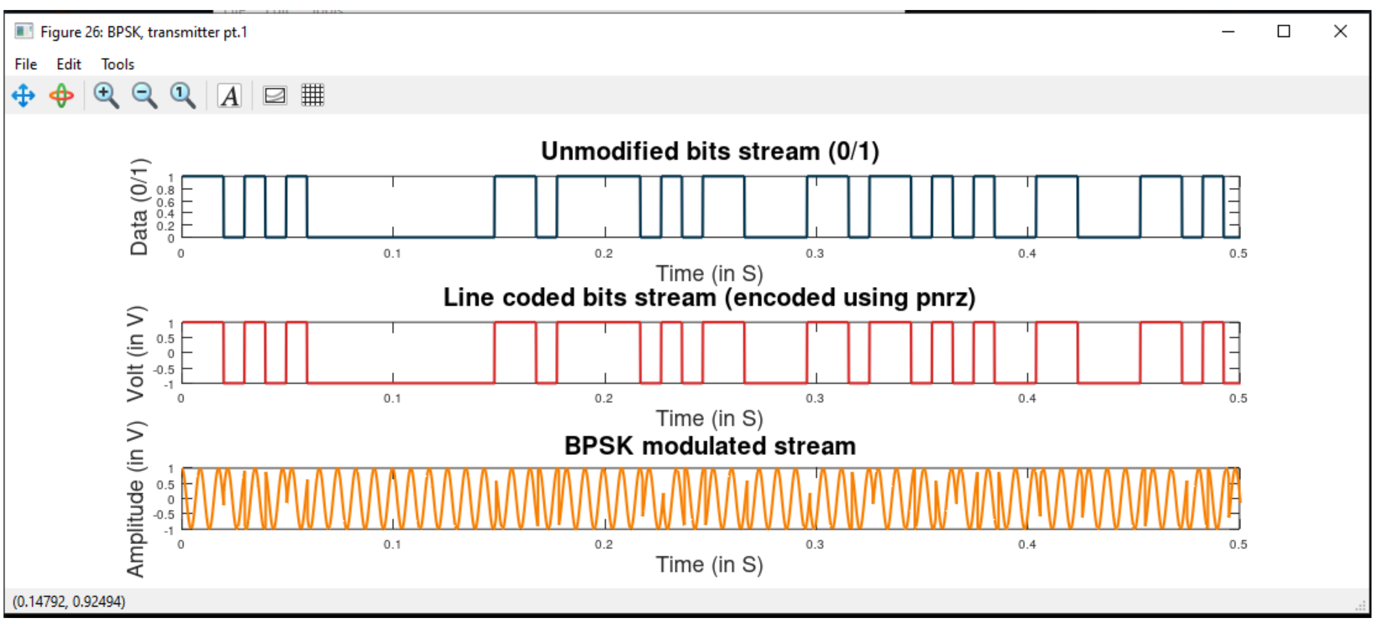


Figure (27): BPSK modulated stream, transmitter side

These plots specifically are the solutions to points 1, 2, and 4.

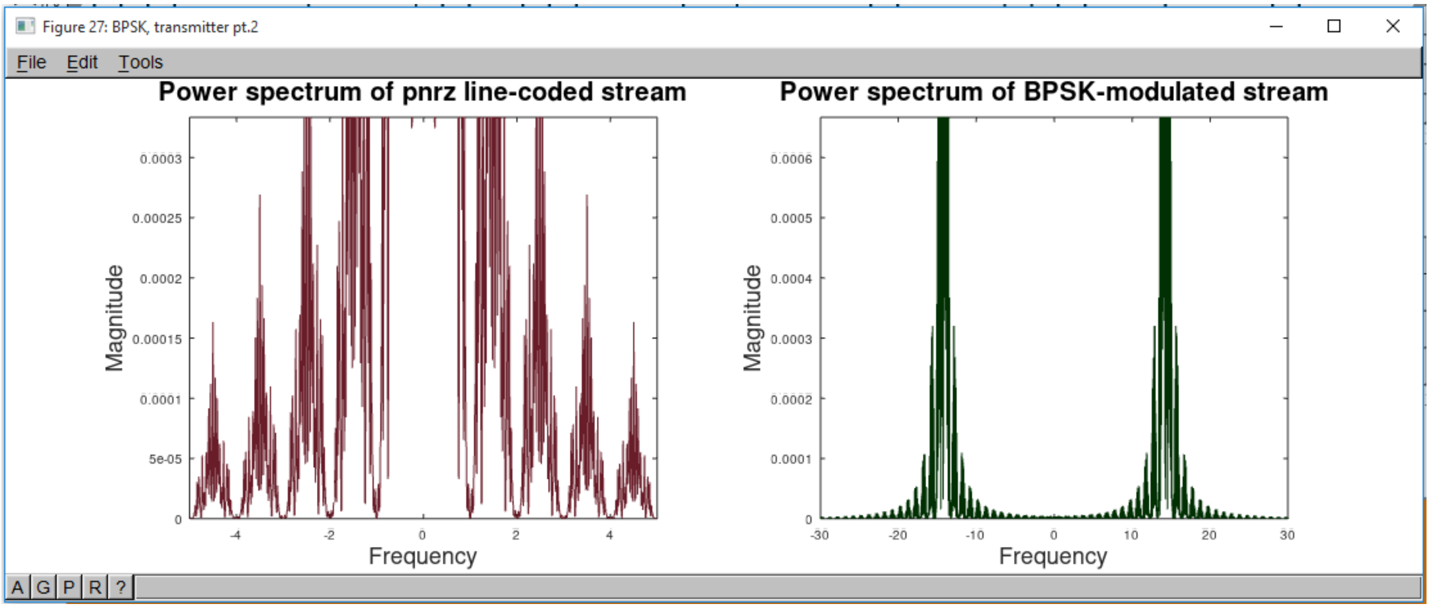


Figure (28): Spectra of the line-coded stream as well as the BPSK-modulated stream

This one is the solution to points 3 and 5. Note the difference between Figure (28)'s plot and Figure (6)'s, due to the huge difference in the sizes of the streams (one is a hundred times the size of the other!)

4. Part 2: Receiver:

```
################################################################################

# Part 2: Receiver:

rx_bpsk = receiver(tx_bpsk);
rx_bpsk = rx_bpsk.extract_line_code_from_bpsk_modulated();
rx_bpsk = rx_bpsk.extract_stream_from_line_code();

figure('Name', 'BPSK, receiver', 'Position', figure_position);
subplot(3, 1, 1);
rx_bpsk.plot('rx_line_coded_stream');
subplot(3, 1, 2);
plot(rx_bpsk, 'noisy_rx_stream');
subplot(3, 1, 3);
rx_bpsk.plot('extracted_stream');

BER = rx_bpsk.get_bit_error_rate(tx_bpsk);
BER
```

```
Command Window
BER = 0
BER_test = 0.044900
>>
```

BER_test will be mentioned later in Code Dive!

The following plot illustrates the last two points, part 2 points 6 and 7.

6. Design a receiver which consists of modulator, integrator (simply LPF) and decision device.
7. Compare the output of decision level with the generated stream of bits in the transmitter. (BER) = number of error bits/ Total number of bits.

The calculated BER is already displayed in the command window above, and all that remains is the output from the demodulator (and integrator), as well as the plot of the resulting stream after going through the decision device.

Note, it's only called noisy because that is the same variable that's used to store the streams after adding noise to them, hence the 'sigma = 0' part.

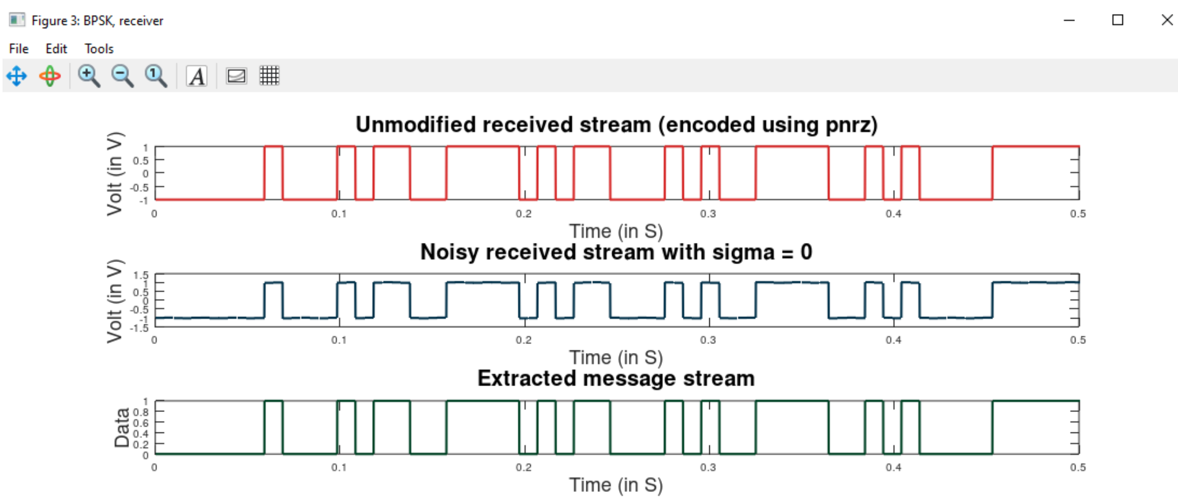
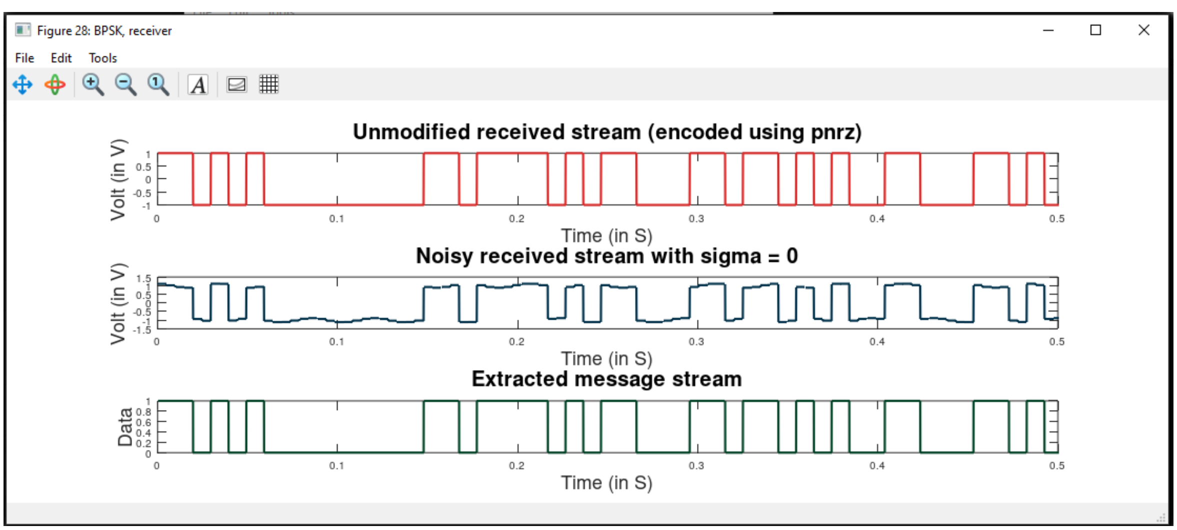


Figure (29): BPSK-modulated stream, receiver side

Here's a fun bonus plot: The modulator with the lower frequency mentioned above for illustration purposes produces the following plots, and it's interesting to note how the output of the integrator is much closer to the original signal when the frequency is higher.



B. Code dive:

This section will go over the code files submitted, and quickly review each function.

1. Transmitter class:

The transmitter.m file contains the classdef that creates an object of the type transmitter, an object whose properties are listed below.

```
classdef transmitter
  properties
    stream = [];
    time_limit = 0.0;
    line_coded_stream = [];
    bpsk_modulated = [];
    line_coding_style = '';
    stream_size = 0;
    vcc_positive = 0.0; %VCC+ and VCC- are later used by the receiver class
    vcc_negative = 0.0;
  endproperties
```

- **stream** is the original bits stream, all zeroes and ones.
- **time_limit** is how long that signal will last according to the size of the stream as well as the bitrate.
- **line_coded_stream** is the stream after encoding.
- **bpsk_modulated** is the stream after the line coded stream gets BPSK modulated.
- **line_coding_style** is the string that contains the encoding style for the line code.
- **stream_size** is the size of the stream (in number of bits).
- **vcc_positive** is the maximum power supply for the stream.
- **vcc_negative** is the maximum negative power supply for the stream.

In the following pages the methods of the class are listed. Aside from the constructor, there are 7 total functions in the transmitter class.

- create_stream: creates bits stream, returns an updated transmitter object.
- line_code: line codes the bits stream, returns an updated transmitter object.
- bpsk: modulates the line code using BPSK, returns an updated transmitter object.
- plot: plots a transmitter stream (bits stream, line coded stream, or BPSK modulated stream).
- plot_line_code_power_spectrum: plots power spectrum of line coded stream.
- plot_bpsk_power_spectrum: plots the power spectrum of the BPSK-modulated stream.
- plot_eyediagram: plots the eyediagram of a transmitter stream.

```
methods
  function obj = transmitter (input_stream, bitrate)
    obj.stream = [];
    time_domain_vector = [];
    obj.line_coded_stream = [];
    obj.bpsk_modulated = [];
    obj.line_coding_style = '';
    obj.stream_size = 0;
    if (nargin >= 1)
      if size(input_stream)(1) ~= 1 || size(input_stream(2)) == 0
        error("Array dimensions don't conform to transmitter class's specifications. Array must be 1xN, and N must be larger than 0.");
      endif

      for i = 1 : length(input_stream)
        if input_stream(i) ~= 0 && input_stream(i) ~= 1
          error(["Data entered must have only 0's and 1's. The entry at index " string(i) " is neither."]);
        endif
      endfor

      obj.stream = repelem(input_stream, 2);
      obj.stream_size = length(input_stream);

      if (nargin < 2)
        bitrate = 100;
      endif

      obj.time_limit = obj.stream_size * 1/bitrate - 1/bitrate;
    endif
  endfunction
```

The constructor first initializes values of the object's fields, and depending on whether there are input arguments or not, the object can be constructed using an already existing stream from somewhere else in the program (input_stream), and if a bitrate is not given with the existing stream as a second argument, it defaults to 100.

```
function obj = create_stream (obj, stream_size, bitrate)
  if ~isa(obj, 'transmitter')
    error("Passed object is not of the transmitter type.");
  endif
  if nargin < 3
      bitrate = 100;
  endif
  if nargin == 0 || nargin == 1 || (nargin >= 2 && stream_size == 0)
    stream_size = 10000;
  endif

  temp = randi([0 1], 1, stream_size);
  obj.stream = repelem(temp, 2);
  obj.stream_size = stream_size;
  obj.time_limit = obj.stream_size * 1/bitrate - 1/bitrate;
endfunction
```

The first function that should be called after the constructor is "create_stream", which creates the random bits stream according to a couple of parameters, stream_size and bitrate, both of which are optional and default to 10,000 and 100 respectively. Each bit is repeated twice, and the two elements of the array represent a single bit's duration.
Note that this function doesn't need to be called if a stream was already provided to the constructor method.

```
function obj = line_code (obj, line_coding_style, vcc_positive, vcc_negative)
  if ~isa(obj, 'transmitter')
    error("Passed object is not of the transmitter type.");
  endif

  if obj.stream_size == 0
    error("You need to call create_stream first!");
  endif

  if nargin < 3
    error("Not enough arguments. Make sure to enter both line coding style and vcc.");
  endif

  if nargin < 4
    vcc_negative = vcc_positive * -1;
  endif

  obj.vcc_positive = vcc_positive;
  obj.vcc_negative = vcc_negative;
  obj.line_coding_style = line_coding_style;
  obj.line_coded_stream = zeros (1, obj.stream_size * 2);

  styles = {'unrz' 'urz' 'pnrz' 'prz' 'bpnrz' 'bprz' 'manchester'};

  index = find(strcmp(styles, line_coding_style));
```

This function is the one responsible for all the line coding, the snapshot above is only its start and error handling. It takes a few arguments: the transmitter object, the ID for the desired line coding style, the maximum power supply, as well as the negative one (optional). It then goes on to initialize a few of the object's properties, and then creates the array of styles that will be used later in a switch-case to determine which line-coding style should be followed.

```
  switch index
    case 1 %unipolar non-return to zero
      obj.line_coded_stream = (obj.stream == 1) .* vcc_positive;

    case 2  %unipolar return to zero
      for i = 1 : obj.stream_size * 2
        if (obj.stream(i) == 1 && (mod(i, 2) == 1))
          obj.line_coded_stream(i) = vcc_positive;
        endif
      endfor
```

For the unipolar styles, one was done using list comprehension, and the other was done using a regular for-loop. Note that the return to zero takes advantage of the fact that the full bit duration is represented by two elements of the array, so the value of the given bit from the data stream is only considered for the line coded stream when the element's index is odd.

```
case 3 %polar non-return to zero
  obj.line_coded_stream = (obj.stream == 1) .* vcc_positive + (obj.stream ~= 1) .* vcc_negative;

case 4 %polar return to zero
  for i = 1 : obj.stream_size * 2
    if (obj.stream(i) == 1 && (mod(i, 2) == 1))
      obj.line_coded_stream(i) = vcc_positive;
    elseif mod(i, 2) == 1
      obj.line_coded_stream(i) = vcc_negative;
    endif
  endfor
```

This was done the same way unipolar encoding was done, and since all other line-coding styles follow almost the same formula, the code will be attached without further comments.

```
case 5 %bipolar non-return to zero
  flag = 1;
  for i = 1 : obj.stream_size * 2
    if (obj.stream(i) == 1 && flag)
      obj.line_coded_stream(i) = vcc_positive;
      if (mod(i, 2) == 0)
        flag = 0;
      endif
    elseif (obj.stream(i) == 1 && ~flag)
      obj.line_coded_stream(i) = vcc_negative;
      if (mod(i, 2) == 0)
        flag = 1;
      endif
    endif
  endfor

case 6 %bipolar return to zero
  flag = 1;
  for i = 1 : obj.stream_size * 2
    if (obj.stream(i) == 1 && flag && (mod(i, 2) == 1))
      obj.line_coded_stream(i) = vcc_positive;
      flag = 0;
    elseif (obj.stream(i) == 1 && ~flag && (mod(i, 2) == 1))
      obj.line_coded_stream(i) = vcc_negative;
      flag = 1;
    endif
  endfor

case 7 %manchester
  for i = 1 : obj.stream_size * 2
    if (obj.stream(i) == 1)
      if (mod(i, 2) == 1)
        obj.line_coded_stream(i) = vcc_positive;
      else
        obj.line_coded_stream(i) = vcc_negative;
      endif
    else
      if (mod(i, 2) == 1)
        obj.line_coded_stream(i) = vcc_negative;
      else
        obj.line_coded_stream(i) = vcc_positive;
      endif
    endif
  endfor
```

```
function obj = bpsk (obj)
  if ~isa(obj, 'transmitter')
    error("Passed object is not of the transmitter type.");
  endif
  if isnull(obj.line_coded_stream) || strcmp(obj.line_coding_style, 'pnrz') ~= 1
    error("transmitter_object.line_code('pnrz', vcc) must be called first.");
  endif

  obj.bpsk_modulated = zeros(1, obj.stream_size/0.01);
  temp = repelem(obj.line_coded_stream, 50);
  for i = 1 : length(temp)
    obj.bpsk_modulated (i) = cos(2 * 3.14159265  * 1e9 * i) * temp(i);
  endfor
endfunction
```

The next function in the class file is the bpsk() function, which takes the line-coded stream from a transmitter object and creates the BPSK stream. Note that each element in the array was repeated 50 times so that the total number of elements in the array per one bit duration is 100, this allows for a smooth sine/cosine when multiplying the stream by the carrier.

The following two functions have to do with plotting the spectral domains of the pulses.

```
function plot_line_code_power_spectrum(obj)
  if ~isa(obj, 'transmitter')
    error("Passed object is not of the transmitter type.");
  endif
  if isnull(obj.line_coded_stream)
    error("transmitter_object.line_code('line_coding_style', vcc) must be called first.");
  endif

  stream = repelem(obj.line_coded_stream, 50);
  N = length(stream);
  ts = 0.01;
  T = N * ts ;
  fs = 1 / ts;
  df = 1 / T;

  if(rem(N ,2)==0)
    frequencies = -(0.5*fs) : df : (0.5*fs - df);            %% Frequency vector if x/f is even
  else
    frequencies = -(0.5*fs - 0.5*df) : df : (0.5*fs - 0.5*df);%% Frequency vector if x/f is odd
  endif

  S =  fftshift((fft(stream)))/N;

  plot(frequencies, abs(S.^2), 'Color', "#691d29");
  title(['Power spectrum of '  obj.line_coding_style  ' line-coded stream'], 'FontSize', 20);
  xlabel('Frequency', 'FontSize', 18);
  ylabel('Magnitude', 'FontSize', 18);
  axis([-5 5 0 (max(obj.line_coded_stream)/3000)]); %heuristic
endfunction
```

Note that every element is repeated 50 times, so the total bit duration is 100 elements in the array.

```
function plot_bpsk_power_spectrum(obj)
  if ~isa(obj, 'transmitter')
    error("Passed object is not of the transmitter type.");
  endif
  if isnull(obj.bpsk_modulated)
    error("transmitter_object.bpsk() must be called first.");
  endif

  stream = obj.bpsk_modulated;
  N = length(stream);
  ts = 0.01;
  T = N * ts ;
  fs = 1 / ts;
  df = 1 / T;

  if(rem(N ,2)==0)
    frequencies = -(0.5*fs) : df : (0.5*fs - df);              %% Frequency vector if x/f is even
  else
    frequencies = -(0.5*fs - 0.5*df) : df : (0.5*fs - 0.5*df);%% Frequency vector if x/f is odd
  endif

  S =  (fftshift(fft(stream)))/N;

  plot(frequencies, abs(S.^2), 'Color', "#003003");
  title(['Power spectrum of BPSK-modulated stream'], 'FontSize', 20);
  xlabel('Frequency', 'FontSize', 18);
  ylabel('Magnitude', 'FontSize', 18);
  axis([-30 30 0 (max(obj.line_coded_stream)/1500)]); %heuristic
endfunction
```

Two more functions remain, plotting eyediagrams and plotting the streams themselves. In both functions each element in the array is repeated 50 times to have the stream appear square-ish when plotting.

```
function plot_eyediagram(obj, chosen_stream)
  if (nargin < 2)
    chosen_stream = line_coded_stream
  elseif strcmp(chosen_stream, 'line_coded_stream') ~= 1 && strcmp(chosen_stream, 'stream') ~= 1
    error("The given parameter is not supported by this function. This function only supports 'stream' and 'line_coded_stream'");
  endif
  if (length(obj.(chosen_stream)) < 40)
    warning("plot_eyediagram doesn't work properly with a stream size of less than 20 bits.");
  endif
  if (obj.stream_size > 1000)
    warning("Stream size was capped to 1000 bits to speed up eyediagram generation.");
  endif
  hold on
  stream = obj.(chosen_stream)(1:min(obj.stream_size * 2, 1000));
  stream = [stream stream(length(stream))];
  stream = repelem(stream, 50);
  bit_time = obj.time_limit / (obj.stream_size - 1);
  for i = 1 : 300 : length(stream) - 300
    plot(linspace(0, bit_time*4, 300), stream(i : i + 299), 'Color', "#8a4f15", 'LineWidth', 1.25);
  endfor

  if strcmp(chosen_stream, 'line_coded_stream')
    title(['Eyediagram for ' obj.line_coding_style ' line-coded stream'], 'FontSize', 20);
    ylabel('Volt', 'FontSize', 18);
  else
    title('Eyediagram for transmitted 0/1 stream', 'FontSize', 20);
    ylabel('Data (0/1)', 'FontSize', 18);
  endif

  xlabel('Time', 'FontSize', 18);
  axis([(bit_time) (3 * bit_time)])
  hold off
endfunction
```

This is the final plotting function. The main goal of creating this function was to create pretty figures with custom properties that would clutter main.m unnecessarily, and making a function out of it avoids repetition in code, which is always a good thing.

```
function plot(obj, param)
  if ~isa(obj, 'transmitter')
    error("Passed object is not of the transmitter type.");
  endif
  if nargin < 2
    error("You must include the parameter you want to plot.");
  endif

  if strcmp(param, 'stream') == 1
    stream = [obj.stream  obj.stream(length(obj.stream))];
    stream = repelem(stream, 50);
    plot(linspace(0, obj.time_limit, length(stream)), stream, 'LineWidth', 1.55, 'Color', "#003049");
    title('Unmodified bits stream (0/1)', 'FontSize', 20);
    xlabel('Time (in S)', 'FontSize', 18);
    ylabel('Data (0/1)', 'FontSize', 18);
    axis([0 min(0.5, obj.time_limit)]);

  elseif strcmp(param, 'line_coded_stream') == 1
    line_coded_stream = [obj.line_coded_stream  obj.line_coded_stream(length(obj.line_coded_stream))];
    line_coded_stream = repelem(line_coded_stream, 50);
    plot(linspace(0, obj.time_limit, length(line_coded_stream)), line_coded_stream, 'LineWidth',1.55, 'Color', "#d62828");
    title(['Line coded bits stream (encoded using '  obj.line_coding_style  ')'], 'FontSize', 20);
    xlabel('Time (in S)', 'FontSize', 18);
    ylabel('Volt (in V)', 'FontSize', 18);
    axis([0 min(0.5, obj.time_limit)]);

  elseif strcmp(param, 'bpsk_modulated') == 1
    plot(linspace(0, obj.time_limit, length(obj.bpsk_modulated)), obj.bpsk_modulated, 'LineWidth',1.5, 'Color', "#f77f00");
    title('BPSK modulated stream', 'FontSize', 20);
    xlabel('Time (in S)', 'FontSize', 18);
    ylabel('Amplitude (in V)', 'FontSize', 18);
    axis([0 min(0.5, obj.time_limit)]);

  else
    error(["The parameter passed to the function" param " doesn't exist."]);
  endif
endfunction
```

2. Receiver class:

The receiver.m file contains the classdef that creates an object of the type receiver, an object whose properties are listed below.

```
classdef receiver
  properties
    rx_line_coded_stream = []; %unmodified transmitter stream
    noisy_rx_stream = [];
    sigma = 0.0;
    detected_errors = 0.0; %Used only for line_coding_style bpnrz/bprz
    stream_size = 0.0;
    time_limit = 0.0;

    line_coding_style = '';
    rx_bpsk_stream = [];

    vcc_positive = 0.0;
    vcc_negative = 0.0;
    extracted_stream = []; %0's and 1's

  endproperties
```

- **rx_line_coded_stream** is the original received stream, no noise added.
- **noisy_rx_stream** is the stream with optionally added noise, or the stream after demodulation from the received BPSK modulated signal (Also optional).
- **sigma** is the sigma for the added noise, this is only kept as a property of the object for the sake of plotting later.
- **detected_errors** is the number of detected errors for the BONUS requirement.
- **stream_size** is the size of the stream (in number of bits).
- **time_limit** is how long that signal will last according to the size of the stream as well as the bitrate.
- **line_coding_style** is the string that contains the encoding style for the line code.
- **rx_bpsk_stream** is the received BPSK modulated stream.
- **vcc_positive** is the maximum power supply for the stream.
- **vcc_negative** is the maximum negative power supply for the stream.
- **extracted_stream** is the output, the result of decoding the encoded bits.

In the following pages the methods of the class are listed. Aside from the constructor, there are 5 total functions in the receiver class.

- add_noise: adds noise to rx_line_coded_stream, returns an updated receiver object.
- extract_stream_from_line_code: extracts original stream, returns an updated transmitter object.
- extract_line_code_from_bpsk_modulated: returns an updated transmitter object.
- get_bit_error_rate: returns BER according to given receiver and transmitter objects.
- plot: plots a receiver object’s stream (rx_line_coded_stream, extracted_stream, noisy_rx_stream, rx_bpsk_stream).

```
function obj = receiver (transmitter_object)
  if nargin < 1
    error("A transmitter object must be created first and passed into this constructor.");
  endif
  if (isnull(transmitter_object.line_coded_stream))
    error("transmitter object's line_code function must be called before initializing the receiver's values.");
  endif

  obj.rx_line_coded_stream = transmitter_object.line_coded_stream;
  obj.rx_bpsk_stream = transmitter_object.bpsk_modulated;
  obj.noisy_rx_stream = obj.rx_line_coded_stream;
  obj.extracted_stream = []; %0's and 1's
  obj.stream_size = transmitter_object.stream_size;
  obj.line_coding_style = transmitter_object.line_coding_style;
  obj.vcc_positive = transmitter_object.vcc_positive;
  obj.vcc_negative = transmitter_object.vcc_negative;
  obj.time_limit = transmitter_object.time_limit;
  obj.detected_errors = 0;

endfunction
```

Starting with the constructor, to create a receiver object a transmitter object **must** be passed as a parameter, and the transmitter object will be used to initialize all of the receiver's fields.

```
function ber = get_bit_error_rate(obj, transmitter_object)
  if nargin < 2
    error("The transmitter object whose stream will be compared must be passed as a second argument.");
  endif
  if ~isa(obj, 'receiver') || ~isa(transmitter_object, 'transmitter')
    error("The function must be used as follows -> receiver_object.get_bit_error_rate(transmitter_object).");
  endif
  if isnull(obj.extracted_stream)
    error("receiver_object.extract_stream_from_line_code() or extract_stream_from_bpsk_modulated must be called first!");
  endif
  if isnull(transmitter_object.stream)
    error("transmitter_object's stream must first be initialized at construction time or by calling create_stream().");
  endif

  ber = 0;
  for i = 2 : 2 : obj.stream_size * 2
    if obj.extracted_stream(i / 2) ~= transmitter_object.stream(i)
      ber += 1;
    endif
  endfor
  ber /= obj.stream_size;

endfunction
```

The function get_bit_error_rate does exactly that, it compares a transmitter object's stream with the noisy receiver object's, calculates the error, divides by stream size to get BER then returns it.

This function will be shown in action at the end of this section!

As for the line-code decoding, the only thing worth mentioning is that for the non-return to zero styles, the average of the two elements of the array (that represent a single bit duration) is taken and is the only value considered for the decision device.

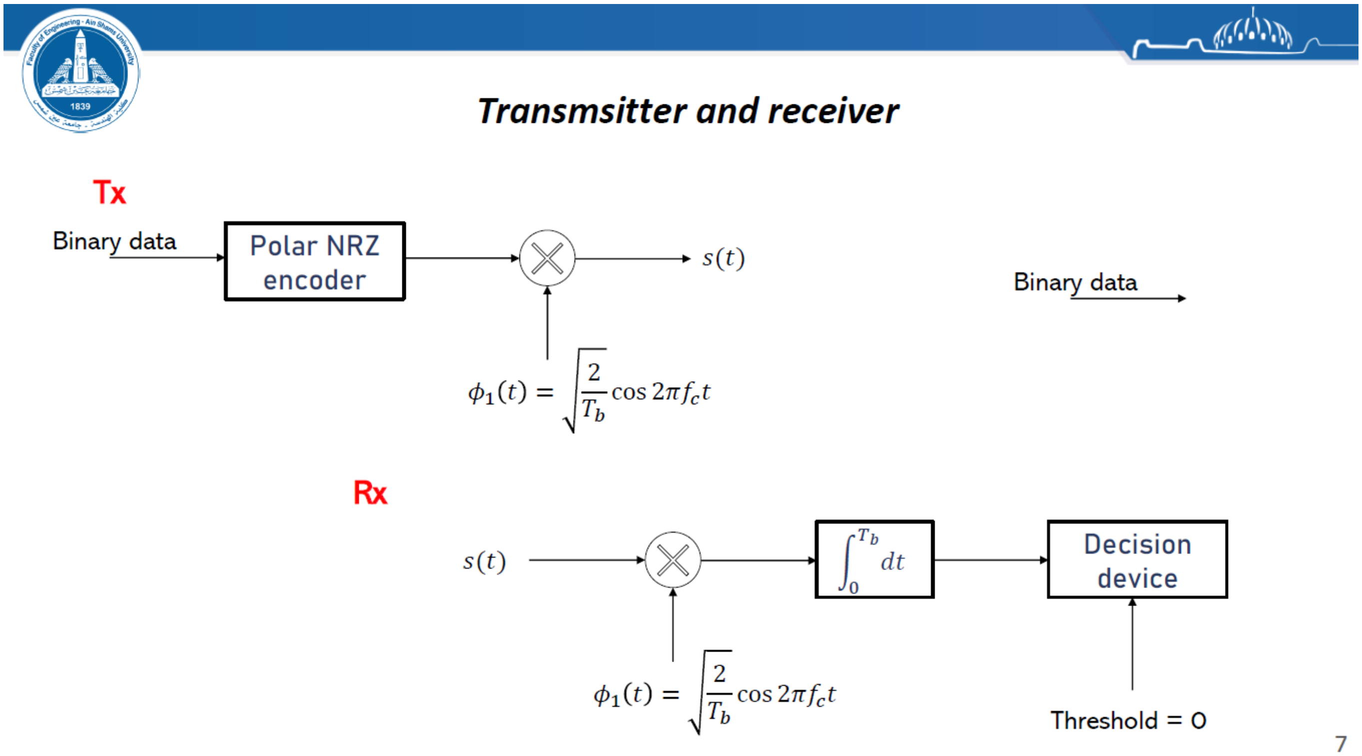
```
function obj = extract_stream_from_line_code (obj)
  obj.extracted_stream = zeros(1, obj.stream_size);
  obj.detected_errors = 0;

  if (strcmp(obj.line_coding_style,'unrz') == 1)
    decision_level = obj.vcc_positive / 2;
    for i = 2 : 2 : obj.stream_size * 2
      if (obj.noisy_rx_stream(i - 1) + obj.noisy_rx_stream(i)) / 2 > decision_level
        obj.extracted_stream(i / 2) = 1;
      endif
    endfor
  endif

  if (strcmp(obj.line_coding_style,'urz')==1)
      decision_level = obj.vcc_positive / 2;
      for i = 2 : 2 : obj.stream_size*2
        if obj.noisy_rx_stream(i - 1) > decision_level
          obj.extracted_stream(i / 2) = 1;
        endif
    endfor
  endif

  if (strcmp(obj.line_coding_style,'pnrz') == 1)
    decision_level = (obj.vcc_positive + obj.vcc_negative) / 2;
    for i = 2 : 2 : obj.stream_size*2
      if (obj.noisy_rx_stream(i - 1) + obj.noisy_rx_stream(i)) / 2 > decision_level
        obj.extracted_stream(i / 2) = 1;
      endif
    endfor
  endif
  if (strcmp(obj.line_coding_style,'prz')==1)
    decision_level = (obj.vcc_positive + obj.vcc_negative) / 2;
    for i = 2 : 2 : obj.stream_size * 2
      if obj.noisy_rx_stream(i - 1) > decision_level
      obj.extracted_stream(i / 2) = 1;
      endif
    endfor
  endif
  if (strcmp(obj.line_coding_style,'manchester')==1)
    decision_level = (obj.vcc_positive + obj.vcc_negative) / 2;
    for i = 2 : 2 : obj.stream_size * 2
     if obj.noisy_rx_stream(i - 1) > decision_level && obj.noisy_rx_stream(i) < decision_level
       obj.extracted_stream(i / 2) = 1;
     endif
    endfor
  endif
```

The bipolar styles' sections are a bit bigger because they work on part 1's BONUS, which will be documented in detail in the section "Bonus, highlighted".

```
if (strcmp(obj.line_coding_style,'bpnrz')==1)
  decision_level_high = obj.vcc_positive / 2;
  decision_level_low  = obj.vcc_negative / 2;
  flag = 0.5;
  for i = 2 : 2 : obj.stream_size * 2
      if ((obj.noisy_rx_stream(i - 1) + obj.noisy_rx_stream(i)) / 2 > decision_level_high ||
        (obj.noisy_rx_stream(i - 1) + obj.noisy_rx_stream(i)) / 2 < decision_level_low)
        obj.extracted_stream(i / 2) = 1;
      endif
      if (obj.noisy_rx_stream(i - 1) + obj.noisy_rx_stream(i)) / 2 > decision_level_high
        if flag == 1
          obj.detected_errors += 1;
        endif
        flag = 1;
      elseif (obj.noisy_rx_stream(i - 1) + obj.noisy_rx_stream(i)) / 2 < decision_level_low
        if flag == 0
          obj.detected_errors += 1;
        endif
        flag = 0;
      endif
  endfor
endif
if (strcmp(obj.line_coding_style,'bprz')==1)
  decision_level_high = obj.vcc_positive / 2;
  decision_level_low  = obj.vcc_negative / 2;
  flag = 0.5;
  for i = 2 : 2 : obj.stream_size * 2
      if obj.noisy_rx_stream(i - 1) > decision_level_high || obj.noisy_rx_stream(i - 1) < decision_level_low
        obj.extracted_stream(i / 2) = 1;
      endif
      if obj.noisy_rx_stream(i - 1) > decision_level_high
        if flag == 1
          obj.detected_errors += 1;
        endif
        flag = 1;
      elseif obj.noisy_rx_stream(i - 1) < decision_level_low
        if flag == 0
          obj.detected_errors += 1;
        endif
        flag = 0;
      endif
  endfor
endif
```

List comprehension could've probably been used for most of these, but the team found that for loops and if conditions were much easier to trace and understand.

As for extract_line_code_from_bpsk_modulated, its role is to use the transmitter's BPSK stream to update the receiver's noisy_rx_stream by demodulating and integrating, faithful to this slide from the course's lectures.




```
function obj = extract_line_code_from_bpsk_modulated (obj)
  if ~isa(obj, 'receiver')
    error("Passed object is not of the receiver type.");
  endif
  if isnull(obj.rx_bpsk_stream)
    error("This receiver object does not have a BPSK stream. Make sure it was initialized with the correct transmitter object.");
  endif

  temp = zeros(1, obj.stream_size /0.01);
  for i = 1 : length(temp)
    temp(i) = cos(2 * 3.14159265  * 1e9 * i) * obj.rx_bpsk_stream(i);
  endfor

  bitrate = (obj.stream_size - 1) / obj.time_limit;
  obj.noisy_rx_stream = zeros(1, obj.stream_size);

  for i = 1 : 50 : length(temp)
    obj.noisy_rx_stream((i - 1) / 50 + 1) = sum(temp(i : i + 49)) * (4 / bitrate);
  endfor
endfunction
```

Because the constant was neglected on transmitter's side!

The extracted line code is later passed through extract_stream_from_line_code, the decision device, as shown in main.m earlier.

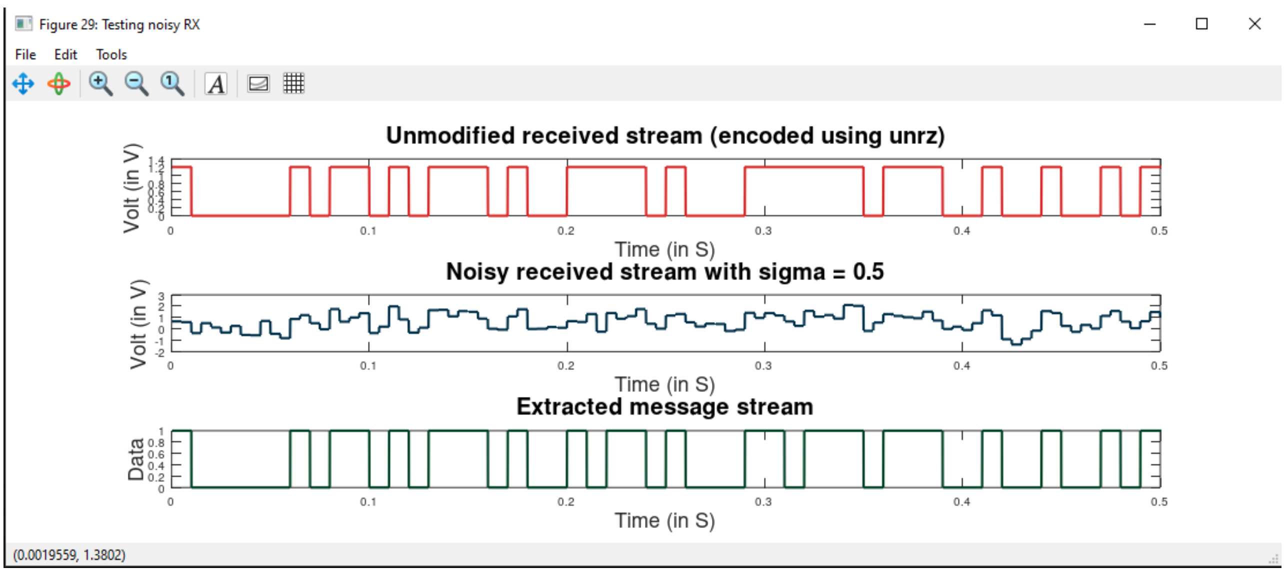
The fourth function is plot, and it's similar to the transmitter's plot function, so it will be attached with no further comments.

```
function plot (obj, param)
  if ~isa(obj, 'receiver')
    error("Passed object is not of the receiver type.");
  endif
  if nargin < 2
    error("You must include the parameter you want to plot.");
  endif

  if strcmp(param, 'noisy_rx_stream') == 1
    stream = [obj.noisy_rx_stream  obj.noisy_rx_stream(length(obj.noisy_rx_stream))];
    stream = repelem(stream, 50);
    plot(linspace(0, obj.time_limit, length(stream)), stream, 'LineWidth', 1.5, 'Color', "#003049");
    title(['Noisy received stream with sigma = ' num2str(obj.sigma)], 'FontSize', 20);
    xlabel('Time (in S)', 'FontSize', 18);
    ylabel('Volt (in V)', 'FontSize', 18);
    axis([0 min(0.5, obj.time_limit)]);

  elseif strcmp(param, 'rx_line_coded_stream') == 1
    line_coded_stream = [obj.rx_line_coded_stream  obj.rx_line_coded_stream(length(obj.rx_line_coded_stream))];
    line_coded_stream = repelem(line_coded_stream, 50);
    plot(linspace(0, obj.time_limit, length(line_coded_stream)), line_coded_stream, 'LineWidth',1.5, 'Color', "#d62828");
    title(['Unmodified received stream (encoded using '  obj.line_coding_style  ')'], 'FontSize', 20);
    xlabel('Time (in S)', 'FontSize', 18);
    ylabel('Volt (in V)', 'FontSize', 18);
    axis([0 min(0.5, obj.time_limit)]);

  elseif strcmp(param, 'rx_bpsk_stream') == 1
    plot(linspace(0, obj.time_limit, length(obj.bpsk_modulated)), obj.bpsk_modulated, 'LineWidth',1.5, 'Color', "#f77f00");
    title('Unmodified BPSK modulated received stream', 'FontSize', 20);
    xlabel('Time (in S)', 'FontSize', 18);
    ylabel('Amplitude (in V)', 'FontSize', 18);
    axis([0 min(0.5, obj.time_limit)]);

  elseif strcmp(param, 'extracted_stream')
    stream = repelem(obj.extracted_stream, 2);
    stream = [stream  stream(length(stream))];
    stream = repelem(stream, 100);
    plot(linspace(0, obj.time_limit, length(stream)), stream, 'LineWidth', 1.5, 'Color', "#004225");
    title('Extracted message stream', 'FontSize', 20);
    xlabel('Time (in S)', 'FontSize', 18);
    ylabel('Data', 'FontSize', 18);
    axis([0 min(0.5, obj.time_limit)]);

  else
    error(["The parameter passed to the function " param " doesn't exist."]);
  endif
endfunction
```

Then comes the final function, add_noise, which takes in a specified sigma and adds that noise to the original rx_line_coded_stream, then updates noisy_rx_stream with the new values.

```
function obj = add_noise(obj, sigma)
  if ~isa(obj, 'receiver')
    error("Passed object is not of the receiver type.");
  endif
  if nargin < 2
    error("Sigma (standard deviation) for the added noise must be provided as an argument to the function.");
  endif

  noise = sigma * randn(1, length(obj.rx_line_coded_stream));
  obj.sigma = sigma;
  obj.noisy_rx_stream += noise;
endfunction
```

Below are a few plots that show this function's effects in action:



And when calling get_bit_error_rate:

```
# Adding noise without sweeping example:

rx_noise = receiver(tx1);
rx_noise = rx_noise.add_noise(0.5);
rx_noise = rx_noise.extract_stream_from_line_code();

figure('Name', 'Testing noisy RX', 'Position', figure_position);
subplot(3, 1, 1);
rx_noise.plot('rx_line_coded_stream');
subplot(3, 1, 2);
rx_noise.plot('noisy_rx_stream');
subplot(3, 1, 3);
rx_noise.plot('extracted_stream');

BER_test = rx_noise.get_bit_error_rate(tx1);
BER_test
```

```
Command Window
BER_test = 0.044900
>>
```

And that concludes the two main functions that this digital communication system depends on. Next up are a couple of helper functions whose one goal is to plot sigma vs BER plots.

3. Sweep over sigma function:

Sweep over sigma does exactly that, it sweeps over the values of sigma as per the requirements for part 1 point 9.

As shown, it takes **four arguments** (**two** of them optional), and they are:

1. The transmitter object (needed for comparing the streams to get the error).
2. The receiver object whose extracted stream is to be compared.
3. The number of sigma values to sweep over (basically determines the step for the sigma array)
4. Sigma limit (sigma was limited to the value of positive VCC, 1.2V, so that's the default value for that parameter).

And it **returns three** values:

1. Sigma array, which is used in the plots later to be put on the x-axis.
2. BER array, which is used in the plots later to be put on the y-axis.
3. Detected BER array, which is an **optional** return value and is only used for the BONUS segment of part 1, when the line coding style is bi-polar.

```
function [sigma_array, ber_array, detected_ber_array] = sweep_over_sigma(tx_object, rx_object, number_of_sigma_values, sigma_limit)
  if nargin < 2 || ~isa(tx_object, 'transmitter') || ~isa(rx_object, 'receiver')
    error("Both a transmitter object and a receiver object must be passed to this function, in that order.");
  endif
  if nargin < 3
    number_of_sigma_values = 100;
  endif
  if nargin < 4
    sigma_limit = tx_object.vcc_positive;
  endif

  sigma_array = linspace(0, sigma_limit, number_of_sigma_values);
  ber_array = zeros(1, number_of_sigma_values);
  detected_ber_array = zeros(1, number_of_sigma_values);

  for i = 1 : number_of_sigma_values
    rx_object = add_noise(rx_object, sigma_array(i));
    rx_object = rx_object.extract_stream_from_line_code();
    ber_array(i) = get_bit_error_rate(rx_object, tx_object);
    detected_ber_array(i) = rx_object.detected_errors / rx_object.stream_size;
  endfor
endfunction
```

4. BER plot function:

This function's one goal is to display the BER plots in a nice layout, using semilogy and custom plotting properties.

It takes **four** arguments (**one** of them optional):

1. Sigma array, to represent x-axis values
2. BER array, to represent y-axis values.
3. The line coding style that was used for the stream whose BER was calculated.
4. **(optional)** the detected BER array, detected from the BONUS requirement.

```
function plot_ber(sigma_array, ber_array, line_coding_style, detected_ber_array)
  hold on;

  semilogy(sigma_array, ber_array, 'LineWidth', 1.5, 'Color', "#D10000");
  txt = {strjoin({'Maximum BER is' num2str(max(ber_array))}, ' ') '\downarrow'};
  text(sigma_array(length(sigma_array)), ber_array(length(ber_array)), txt, 'FontSize', 14,
       'HorizontalAlignment','right', 'VerticalAlignment','bottom');

  if nargin >= 4 && ~isnull(detected_ber_array)
    semilogy(sigma_array, detected_ber_array, 'LineWidth', 1.5, 'Color', "#00D100");
    txt = {'\uparrow' strjoin({'Maximum detected BER is' num2str(max(detected_ber_array))}, ' ')};
    text(sigma_array(length(sigma_array)), detected_ber_array(length(detected_ber_array)), txt, 'FontSize', 14,
       'HorizontalAlignment','right', 'VerticalAlignment','top');
    legend('BER', 'Detected BER', 'Location', 'southeast', 'FontSize', 14);
  endif

  title(['Sigma vs BER for ' line_coding_style ' line-coded stream'], 'FontSize', 20);
  xlabel('Sigma', 'FontSize', 18);
  ylabel('BER', 'FontSize', 18);
  axis([min(sigma_array) (1.1 * max(sigma_array))]);

  hold off;
endfunction
```

5. Bonus, highlighted:

```
if (strcmp(obj.line_coding_style,'bprz')==1)
  decision_level_high = obj.vcc_positive / 2;
  decision_level_low  = obj.vcc_negative / 2;
  flag = 0.5;
  for i = 2 : 2 : obj.stream_size * 2
      if obj.noisy_rx_stream(i - 1) > decision_level_high || obj.noisy_rx_stream(i - 1) < decision_level_low
        obj.extracted_stream(i / 2) = 1;
      endif
      if obj.noisy_rx_stream(i - 1) > decision_level_high
        if flag == 1
          obj.detected_errors += 1;
        endif
        flag = 1;
      elseif obj.noisy_rx_stream(i - 1) < decision_level_low
        if flag == 0
          obj.detected_errors += 1;
        endif
        flag = 0;
      endif
  endfor
endif
```

Part 1, point 11: (Bonus) For the case of Bipolar return to zero, design an error detection circuit. Count the number of detected errors in case of different number of sigma (Use the output of step 8). –Step 8 stated, “Add noise to the received signal”.

It is known that bipolar encoding alternates its 1-bits’ values between +VCC and -VCC.

The error detection circuit works by checking whether a +VCC bit was followed by another +VCC bit, or if a -VCC bit was followed by another -VCC bit. When either of these cases is detected, the object’s detected_errors property is incremented. That property’s sole purpose is to keep track of the number of detected errors for bipolar encoding styles (bipolar non-return to zero, and bipolar return to zero).

```
detected_ber_array(i) = rx_object.detected_errors / rx_object.stream_size;
```

So, this line from sweep_over_sigma uses that value to later create the figure below.

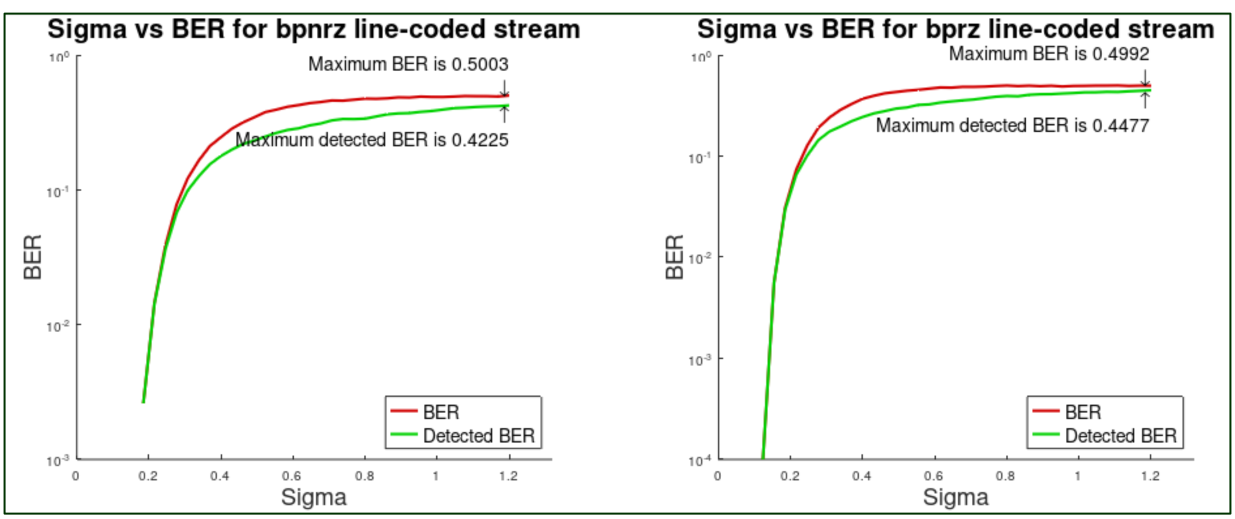


Table of figures: