ECE252s-Fundamentals of Communication Systems

Spring 2025

Course Project

Name	ID	Contribution
Shady Tarek	2101513	Analog Q1-4
Mahmoud Nabil Abdelrhman	2201051	Analog Q5-7
Ganna Khaled Abd ElNasser	2200089	Analog Q8-12
Abdallah Hossam Ragab	2100844	Analog Q8-12
Nour Ayman Mahmoud	2200105	Digital part1
Mostafa Ahmed Abd ELSattar	2201092	Digital Part2

Supervised By:

Dr. Mazen Erfan
Eng. Ahmed khaled Nasr

Dr.Alaa Eldin Fathy
Eng. Ahmed El Sayed

Table Of Contents:

Analog communication

Q1-4:

For the signal shown in the Figure 1:

1. Plot the function x(t) on Octave.

2. Derive an analytical expression for its Fourier transform.

3. Use Octave to calculate the Fourier transform of the signal with sampling frequency fs =100 Hz and resolution equal to 0.01 Hz, and then plot it together with the analytical expression on one graph.

4. Estimate the BW defined as the frequency band after which the power spectrum of the signal drops to 5% of its maximum value.

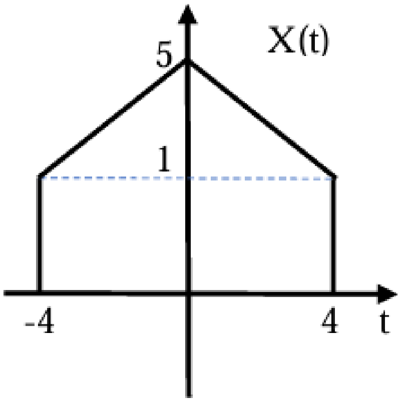
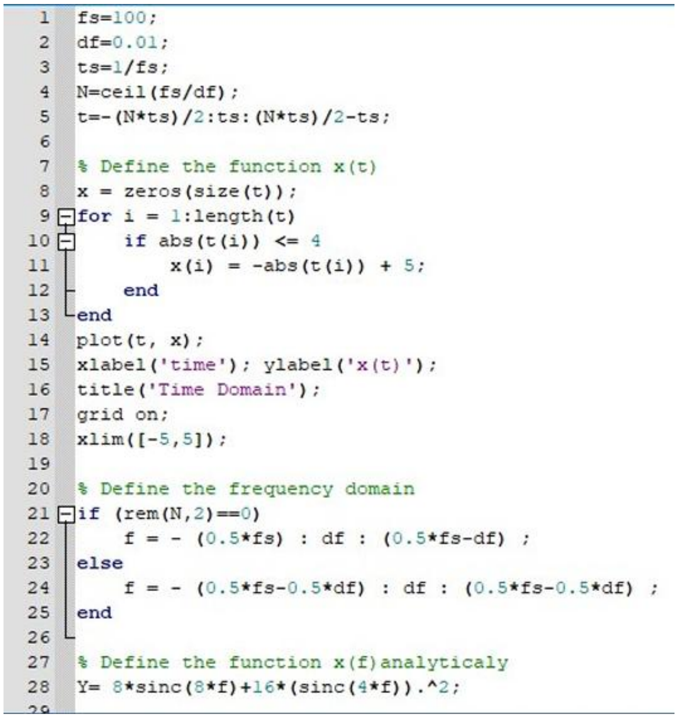
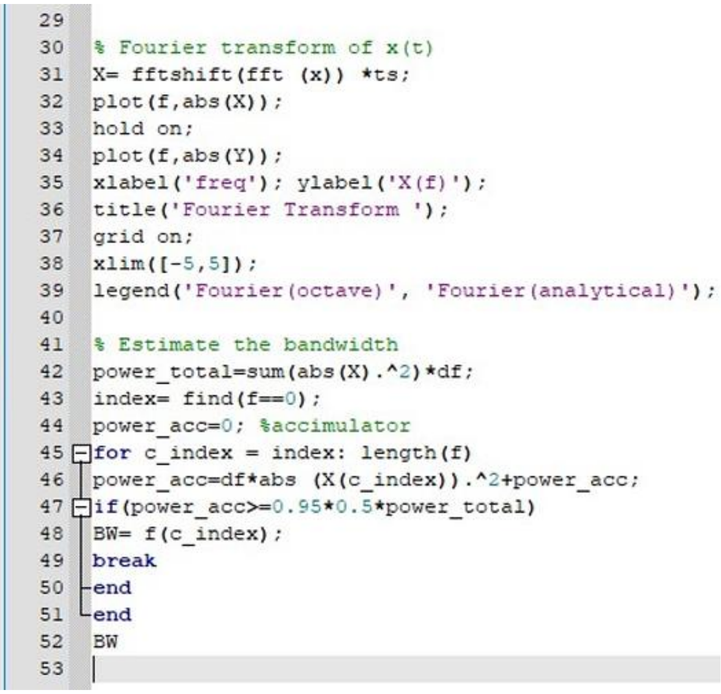


Figure (1)

- ***Code:***

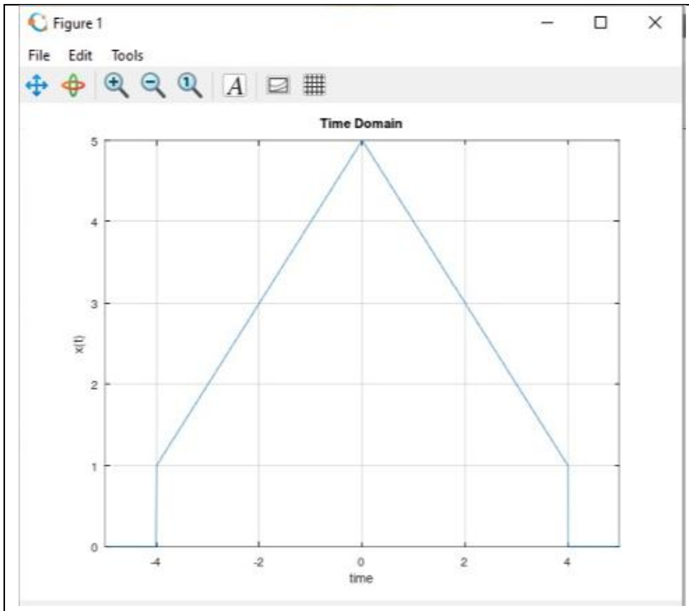
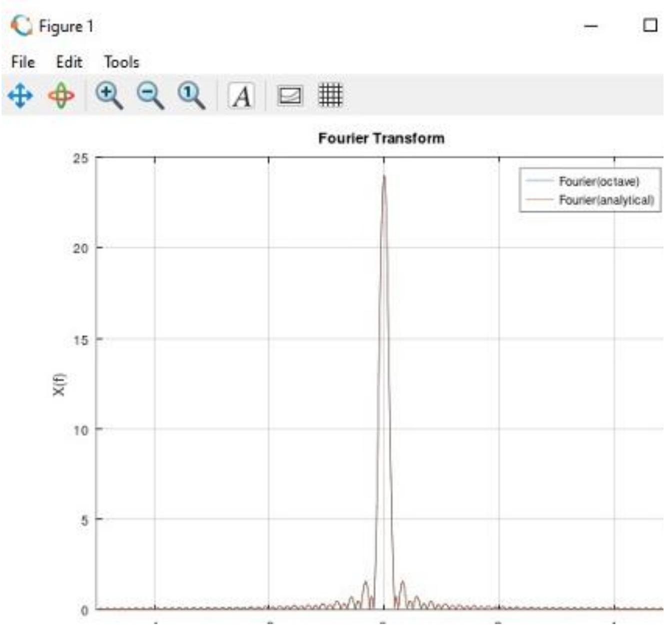


```
fs=100;
df=0.01;
ts=1/fs;
N=ceil(fs/df);
t=-(N*ts)/2:ts:(N*ts)/2-ts;

% Define the function x(t)
x = zeros(size(t));
for i = 1:length(t)
    if abs(t(i)) <= 4
        x(i) = -abs(t(i)) + 5;
    end
end
plot(t, x);
xlabel('time'); ylabel('x(t)');
title('Time Domain');
grid on;
xlim([-5,5]);

% Define the frequency domain
if (rem(N,2)==0)
    f = - (0.5*fs) : df : (0.5*fs-df) ;
else
    f = - (0.5*fs-0.5*df) : df : (0.5*fs-0.5*df) ;
end

% Define the function x(f)analyticaly
Y= 8*sinc(8*f)+16*(sinc(4*f)).^2;
```

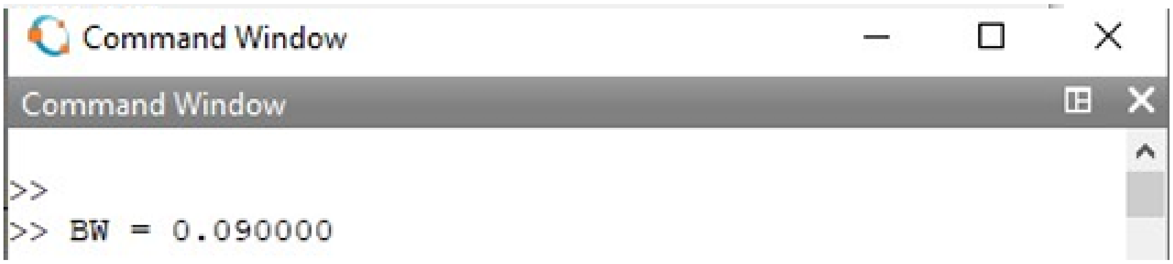


```

% Fourier transform of x(t)
X= fftshift(fft (x)) *ts;
plot(f,abs(X));
hold on;
plot(f,abs(Y));
xlabel('freq'); ylabel('X(f)');
title('Fourier Transform ');
grid on;
xlim([-5,5]);
legend('Fourier(octave)', 'Fourier(analytical)');

% Estimate the bandwidth
power_total=sum(abs(X).^2)*df;
index= find(f==0);
power_acc=0; %accimulator
for c_index = index: length(f)
power_acc=df*abs (X(c_index)).^2+power_acc;
if(power_acc>=0.95*0.5*power_total)
BW= f(c_index);
break
end
end
BW

```

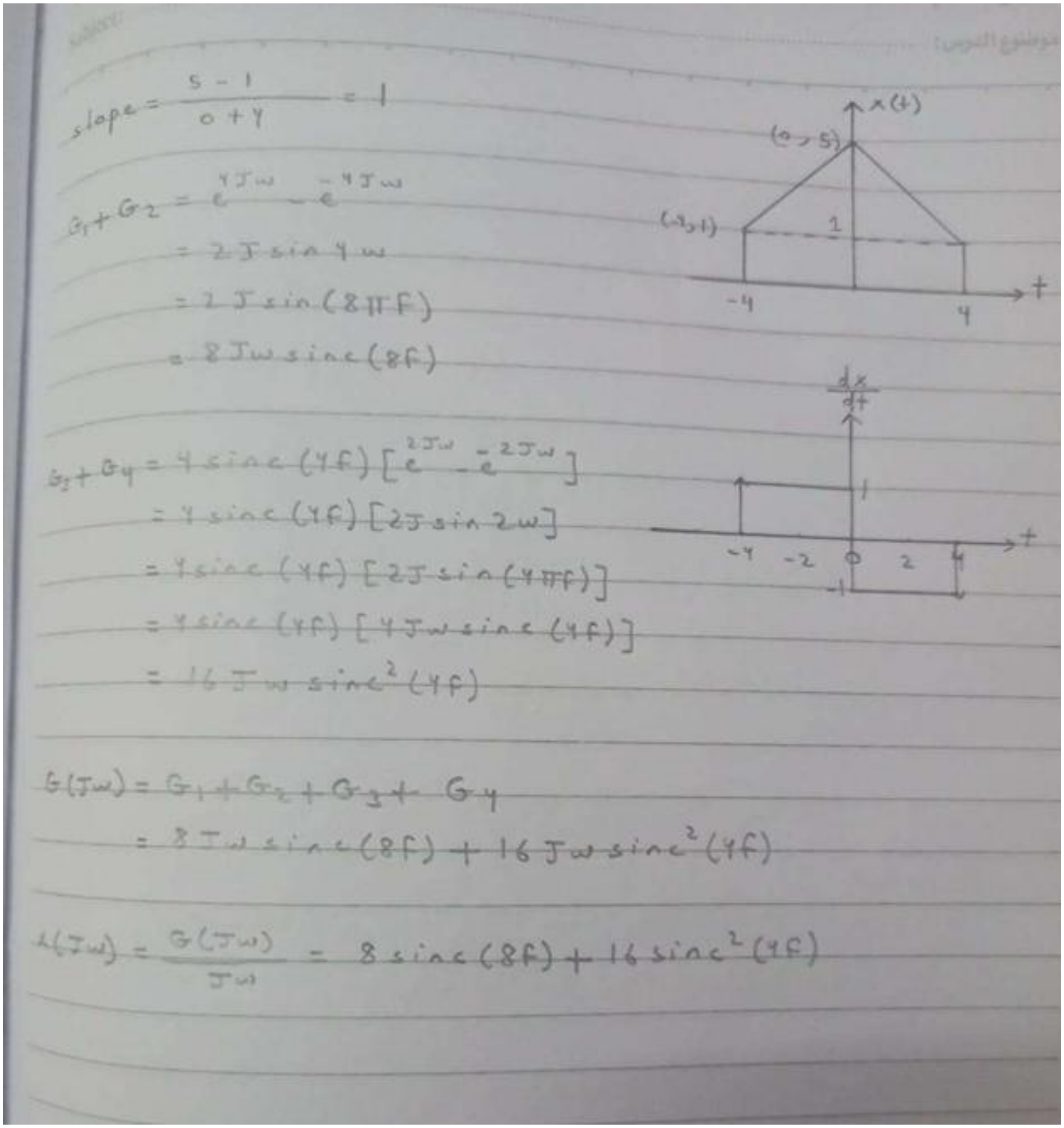
- ***Snapshots:***

	
Plot of signal in time domain	***Plot of fourier transform of signal***

- ***Estimate BW:***



```
>>
>> BW = 0.090000
```



$$\text{slope} = \frac{5-1}{0+4} = 1$$

$$G_1 + G_2 = e^{4J\omega} - e^{-4J\omega}$$

$$= 2J\sin 4\omega$$

$$= 2J\sin(8\pi F)$$

$$= 8J\omega \operatorname{sinc}(8f)$$

$$G_3 + G_4 = 4\operatorname{sinc}(4f)\left[e^{2J\omega} - e^{-2J\omega}\right]$$

$$= 4\operatorname{sinc}(4f)\left[2J\sin 2\omega\right]$$

$$= 4\operatorname{sinc}(4f)\left[2J\sin(4\pi f)\right]$$

$$= 4\operatorname{sinc}(4f)\left[4J\omega \operatorname{sinc}(4f)\right]$$

$$= 16J\omega \operatorname{sinc}^2(4f)$$

$$G(J\omega) = G_1 + G_2 + G_3 + G_4$$

$$= 8J\omega \operatorname{sinc}(8f) + 16J\omega \operatorname{sinc}^2(4f)$$

$$X(J\omega) = \frac{G(J\omega)}{J\omega} = 8\operatorname{sinc}(8f) + 16\operatorname{sinc}^2(4f)$$

Q5-7:

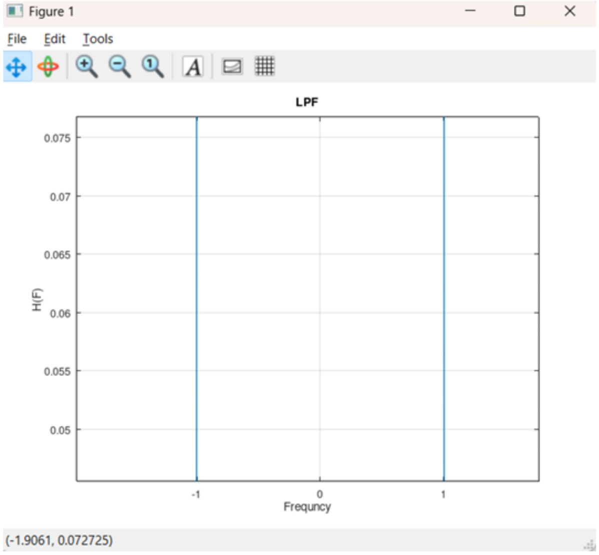
5. If this signal is to pass through a perfect LPF with BW= 1Hz. Plot the output of the filter in the time domain along with the input signal.

6. Repeat (5) if the LPF BW is reduced to be = 0.3 Hz.

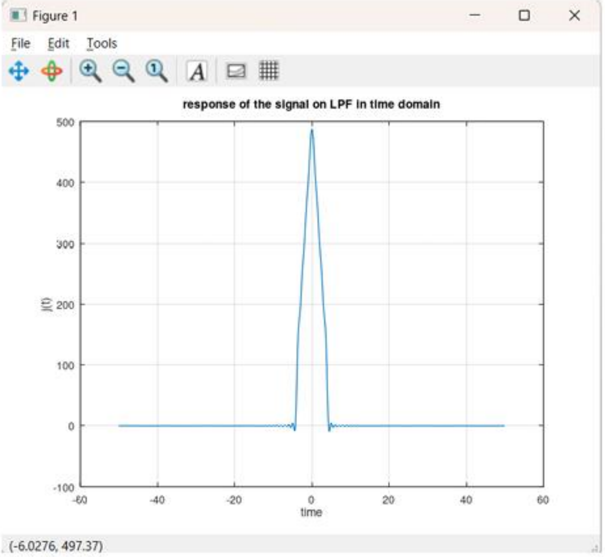
7. For m(t) defined as below, Repeat steps 1-4.

$$m(t) = \left\{ \begin{matrix} \cos(2\,\pi * 0.5 * t) & 0 < t < 4 \\ 0 & otherwise \end{matrix} \right\}$$

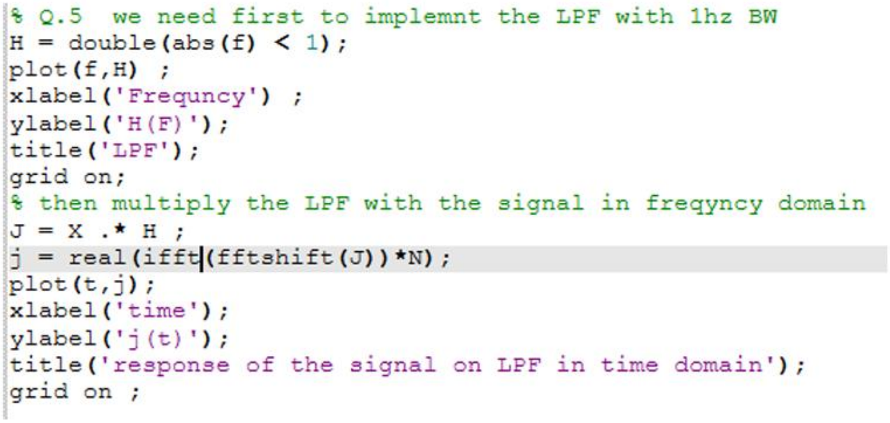
- ***LPF implementation with 1 HZ***



- ***Multiplying with the signal we have and plotting in time domain***

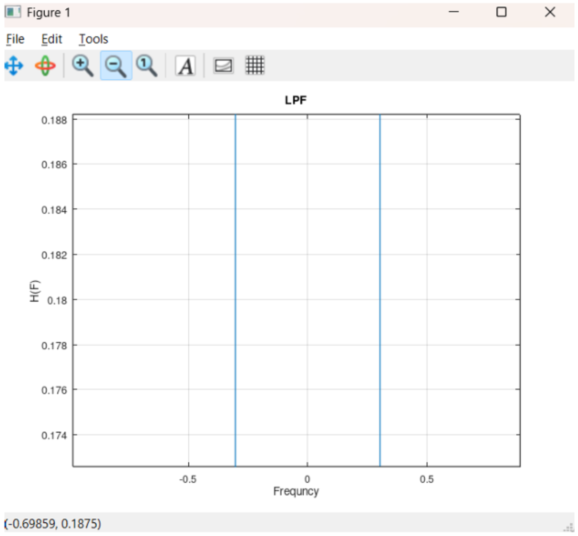


- ***Code implemented in Q5***

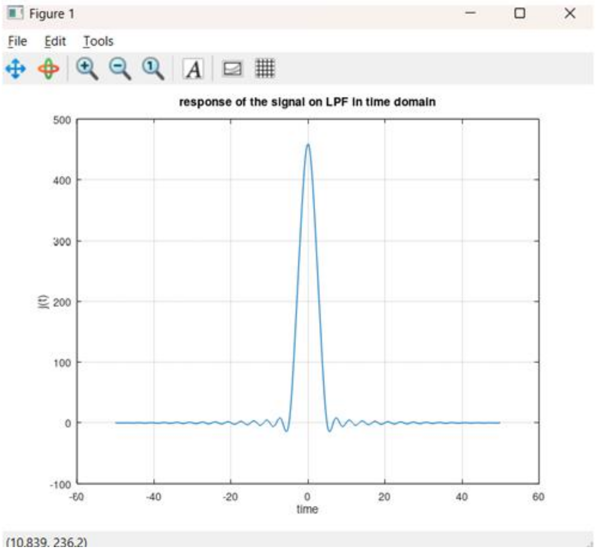


```
% Q.5  we need first to implemnt the LPF with 1hz BW
H = double(abs(f) < 1);
plot(f,H) ;
xlabel('Frequncy') ;
ylabel('H(F)');
title('LPF');
grid on;
% then multiply the LPF with the signal in freqyncy domain
J = X .* H ;
j = real(ifft(fftshift(J))*N);
plot(t,j);
xlabel('time');
ylabel('j(t)');
title('response of the signal on LPF in time domain');
grid on ;
```

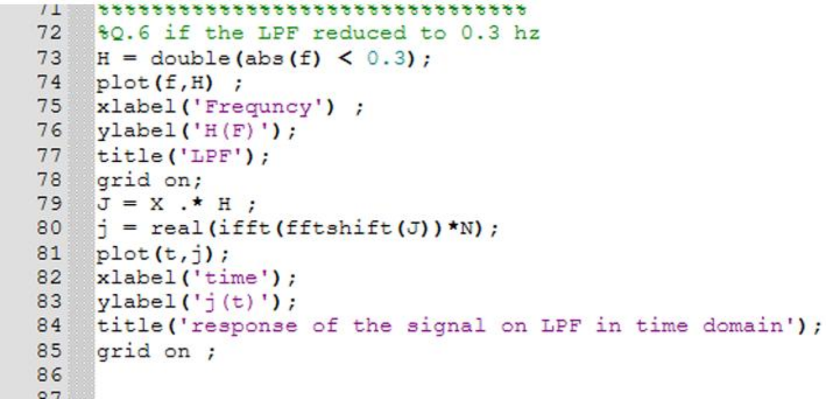
- ***Repeat for Q6***



- ***Output***

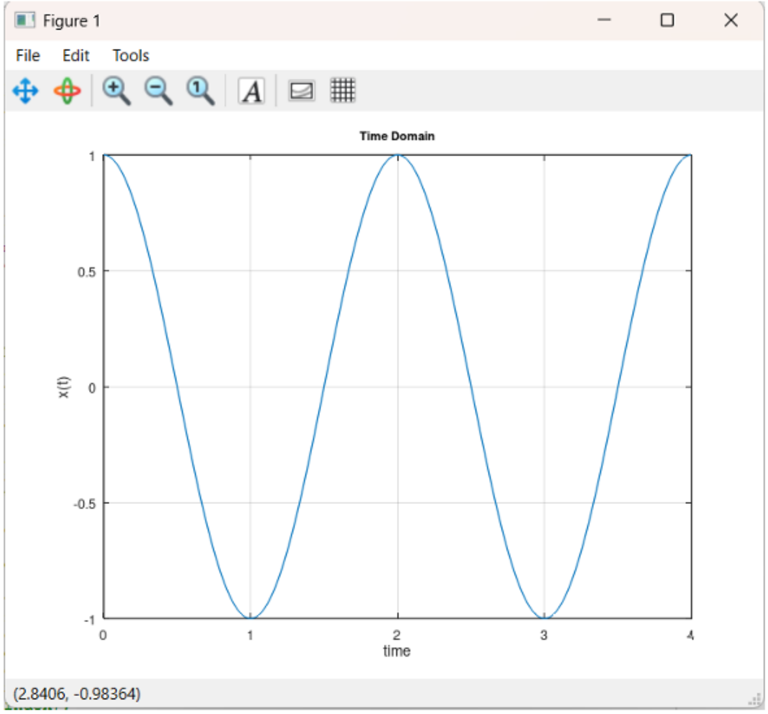


- ***Code implemented in Q6***



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Q.6 if the LPF reduced to 0.3 hz
H = double(abs(f) < 0.3);
plot(f,H) ;
xlabel('Frequncy') ;
ylabel('H(F)');
title('LPF');
grid on;
J = X .* H ;
j = real(ifft(fftshift(J))*N);
plot(t,j);
xlabel('time');
ylabel('j(t)');
title('response of the signal on LPF in time domain');
grid on ;
```

- ***For question 7 we must repeat the same question 1 – 4 for this signal***



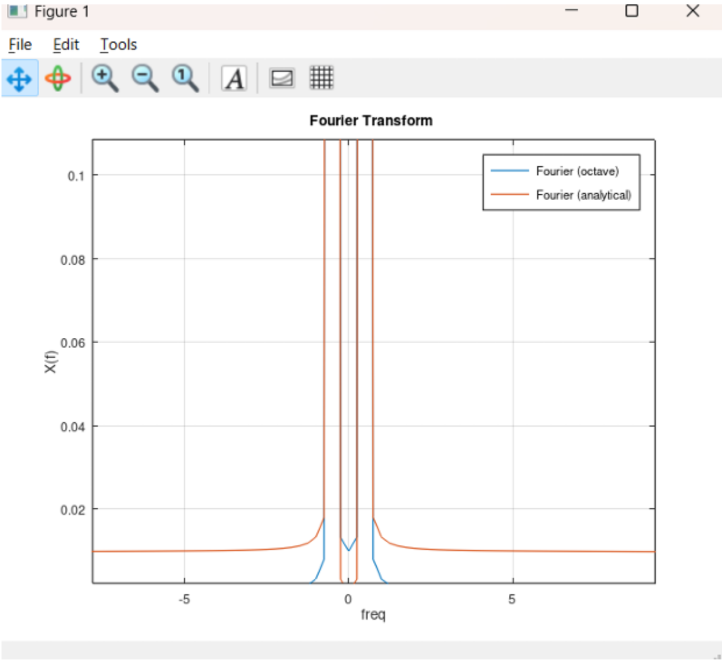
1- Plot of signal in time domain

2- this is a cosine signal that its fourier transform is common and that it is

$$0.5*\delta(f-f_m)+0.5*\delta(f+f_m)$$

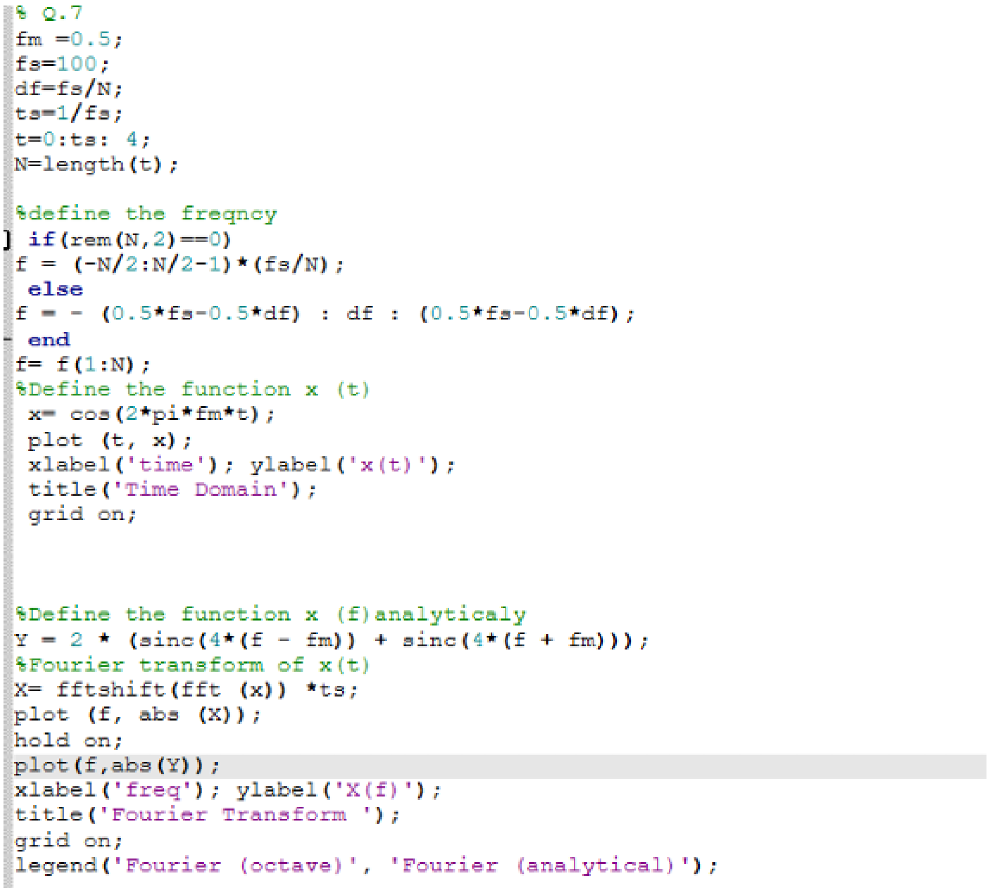
But there was a problem to write this on octave as there was not a dirac function in octave and when load this package there was a error so I have write in octave like this

$$2 * (sinc(4*(f - fm)) + sinc(4*(f + fm)))$$



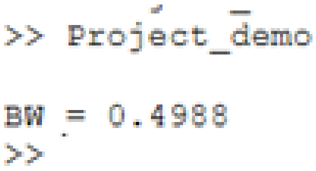
3-Fourier Transform of signal

- ***Code implemented***



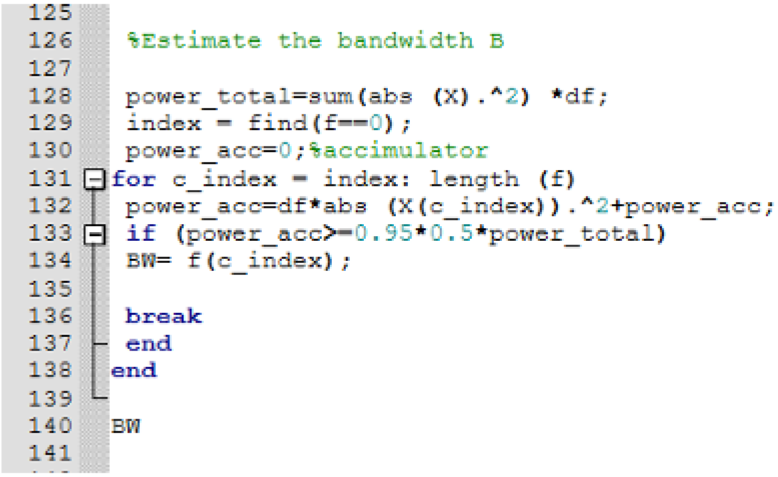
```
% Q.7
fm =0.5;
fs=100;
df=fs/N;
ts=1/fs;
t=0:ts: 4;
N=length(t);

%define the freqncy
 if(rem(N,2)==0)
f = (-N/2:N/2-1)*(fs/N);
 else
f = - (0.5*fs-0.5*df) : df : (0.5*fs-0.5*df);
 end
f= f(1:N);
%Define the function x (t)
 x= cos(2*pi*fm*t);
 plot (t, x);
 xlabel('time'); ylabel('x(t)');
 title('Time Domain');
 grid on;



%Define the function x (f)analyticaly
Y = 2 * (sinc(4*(f - fm)) + sinc(4*(f + fm)));
%Fourier transform of x(t)
X= fftshift(fft (x)) *ts;
plot (f, abs (X));
hold on;
plot(f,abs(Y));
xlabel('freq'); ylabel('X(f)');
title('Fourier Transform ');
grid on;
legend('Fourier (octave)', 'Fourier (analytical)');
```

- ***Estimated BW***



```
>> Project_demo

BW = 0.4988
>>
```

- ***Code implemented***



```
 %Estimate the bandwidth B

 power_total=sum(abs (X).^2) *df;
 index = find(f==0);
 power_acc=0;%accimulator
for c_index = index: length (f)
 power_acc=df*abs (X(c_index)).^2+power_acc;
 if (power_acc>=0.95*0.5*power_total)
 BW= f(c_index);

 break
 end
end

BW
```

Q8-12:

8. FDM Modulation Scheme:

It is required to transmit x(t) and m(t) on different channels.
Where x(t) is modulated in DSB-SC, and m(t) is modulated by SSB. Each channel bandwidth is 2 Hz.

- The modulated signal is s1(t): x(t) is to modulate a carrier signal $c1(t) = \cos(\omega ct)$ with carrier frequency $fc = 20Hz$, Use x(t) from step 5.
- The modulated signal is s2(t): m(t) is to modulate a carrier signal c2(t), such that there is only 2 Hz guard (empty) band between the two channels.

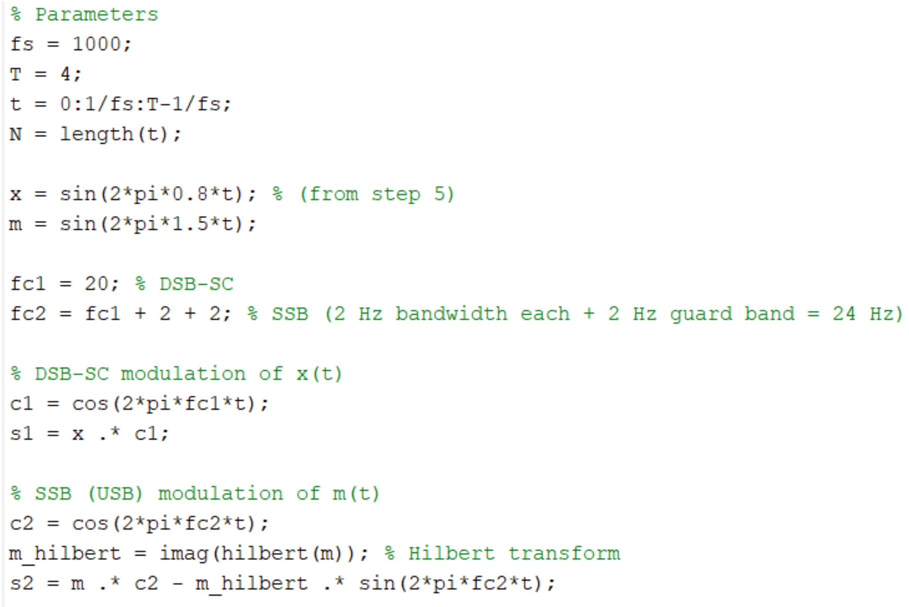
9. State whether you will use USB or LSB.

10. Write an appropriate value for c2(t)

11. Plot s(t) which is s1(t) + s2(t) then Plot S(f).

12. Create a coherent demodulator for each channel and plot the received messages and the input messages on the same figure.

- ***Code implemented for Q8***

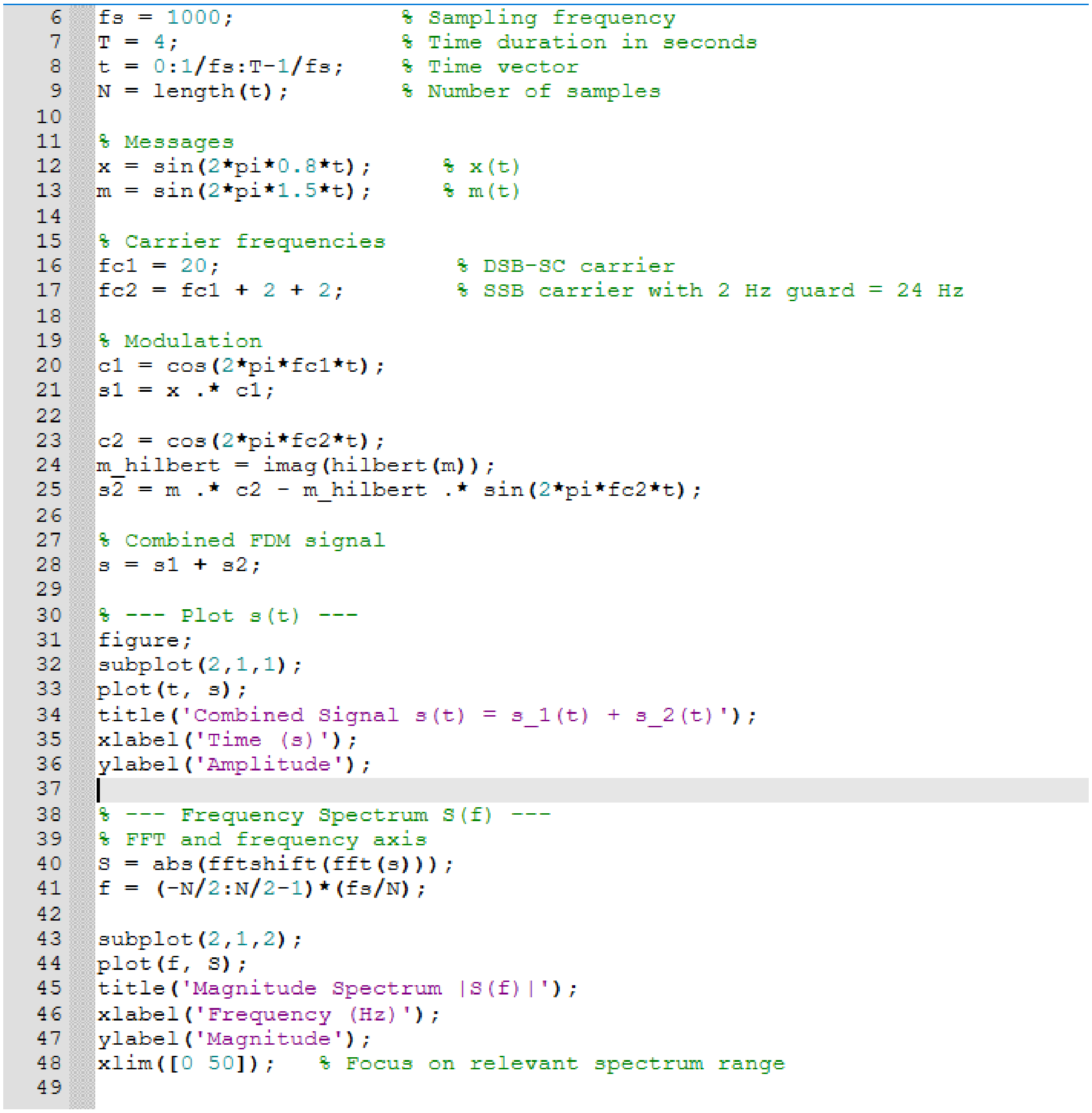


```
% Parameters
fs = 1000;
T = 4;
t = 0:1/fs:T-1/fs;
N = length(t);

x = sin(2*pi*0.8*t); % (from step 5)
m = sin(2*pi*1.5*t);

fc1 = 20; % DSB-SC
fc2 = fc1 + 2 + 2; % SSB (2 Hz bandwidth each + 2 Hz guard band = 24 Hz)

% DSB-SC modulation of x(t)
c1 = cos(2*pi*fc1*t);
s1 = x .* c1;

% SSB (USB) modulation of m(t)
c2 = cos(2*pi*fc2*t);
m_hilbert = imag(hilbert(m)); % Hilbert transform
s2 = m .* c2 - m_hilbert .* sin(2*pi*fc2*t);
```

9- We use USB to place the signal above the DSB-SC spectrum with 2Hz guard band

10- C2(t) = cos(2π*24*t)

11- Code and graphs:



```
fs = 1000;            % Sampling frequency
T = 4;                % Time duration in seconds
t = 0:1/fs:T-1/fs;    % Time vector
N = length(t);        % Number of samples

% Messages
x = sin(2*pi*0.8*t);     % x(t)
m = sin(2*pi*1.5*t);     % m(t)

% Carrier frequencies
fc1 = 20;                 % DSB-SC carrier
fc2 = fc1 + 2 + 2;        % SSB carrier with 2 Hz guard = 24 Hz

% Modulation
c1 = cos(2*pi*fc1*t);
s1 = x .* c1;

c2 = cos(2*pi*fc2*t);
m_hilbert = imag(hilbert(m));
s2 = m .* c2 - m_hilbert .* sin(2*pi*fc2*t);

% Combined FDM signal
s = s1 + s2;

% --- Plot s(t) ---
figure;
subplot(2,1,1);
plot(t, s);
title('Combined Signal s(t) = s_1(t) + s_2(t)');
xlabel('Time (s)');
ylabel('Amplitude');

% --- Frequency Spectrum S(f) ---
% FFT and frequency axis
S = abs(fftshift(fft(s)));
f = (-N/2:N/2-1)*(fs/N);

subplot(2,1,2);
plot(f, S);
title('Magnitude Spectrum |S(f)|');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
xlim([0 50]);   % Focus on relevant spectrum range
```

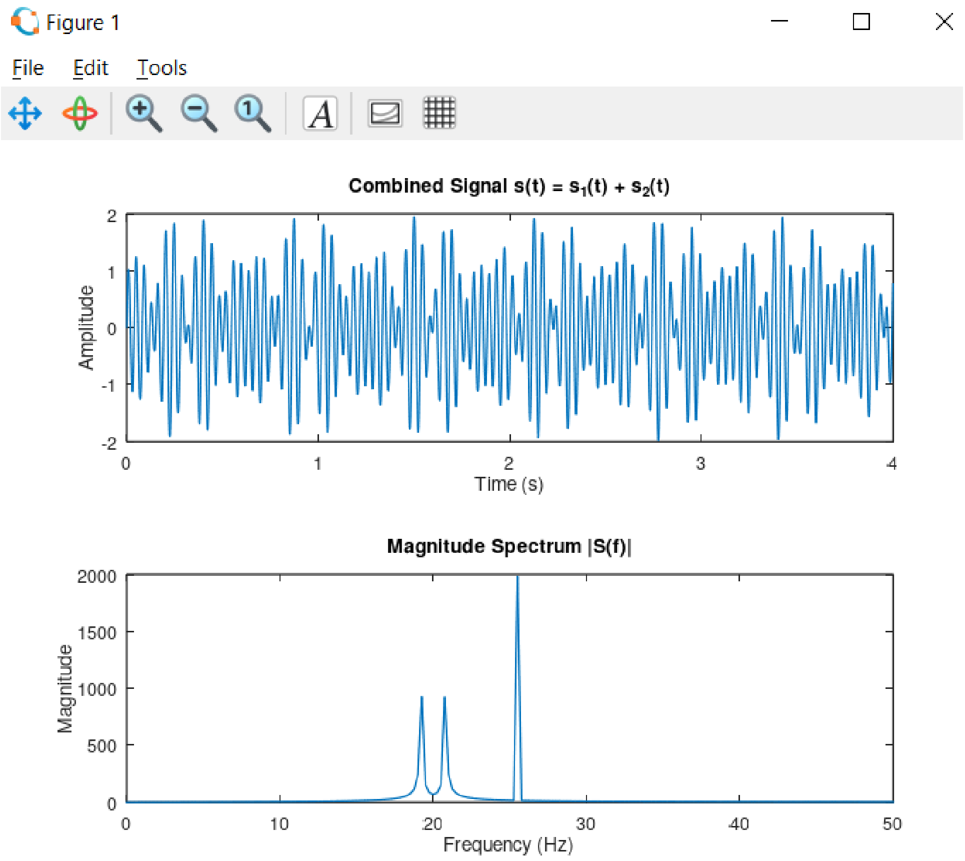
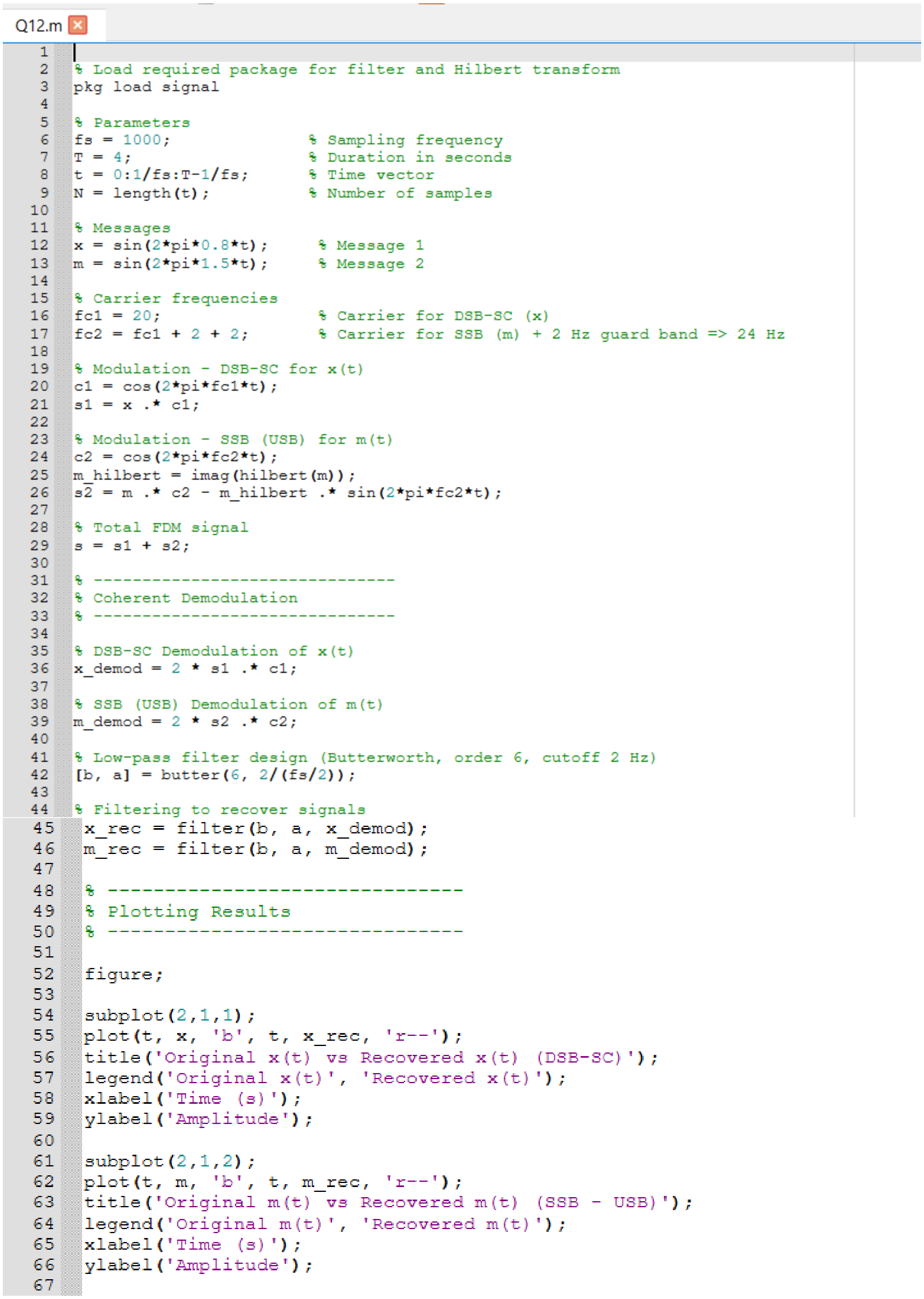


Figure 1
File
Edit
Tools
Combined Signal s(t) = s1(t) + s2(t)
Amplitude
2
1
0
-1
-2
0
1
2
3
4
Time (s)
Magnitude Spectrum |S(f)|
Magnitude
2000
1500
1000
500
0
0
10
20
30
40
50
Frequency (Hz)

12- Code and graphs:



```
% Load required package for filter and Hilbert transform
pkg load signal

% Parameters
fs = 1000;              % Sampling frequency
T = 4;                  % Duration in seconds
t = 0:1/fs:T-1/fs;      % Time vector
N = length(t);          % Number of samples

% Messages
x = sin(2*pi*0.8*t);     % Message 1
m = sin(2*pi*1.5*t);     % Message 2

% Carrier frequencies
fc1 = 20;                % Carrier for DSB-SC (x)
fc2 = fc1 + 2 + 2;       % Carrier for SSB (m) + 2 Hz guard band => 24 Hz

% Modulation - DSB-SC for x(t)
c1 = cos(2*pi*fc1*t);
s1 = x .* c1;

% Modulation - SSB (USB) for m(t)
c2 = cos(2*pi*fc2*t);
m_hilbert = imag(hilbert(m));
s2 = m .* c2 - m_hilbert .* sin(2*pi*fc2*t);

% Total FDM signal
s = s1 + s2;

% -------------------------------
% Coherent Demodulation
% -------------------------------

% DSB-SC Demodulation of x(t)
x_demod = 2 * s1 .* c1;

% SSB (USB) Demodulation of m(t)
m_demod = 2 * s2 .* c2;

% Low-pass filter design (Butterworth, order 6, cutoff 2 Hz)
[b, a] = butter(6, 2/(fs/2));

% Filtering to recover signals
x_rec = filter(b, a, x_demod);
m_rec = filter(b, a, m_demod);

% -------------------------------
% Plotting Results
% -------------------------------

figure;

subplot(2,1,1);
plot(t, x, 'b', t, x_rec, 'r--');
title('Original x(t) vs Recovered x(t) (DSB-SC)');
legend('Original x(t)', 'Recovered x(t)');
xlabel('Time (s)');
ylabel('Amplitude');

subplot(2,1,2);
plot(t, m, 'b', t, m_rec, 'r--');
title('Original m(t) vs Recovered m(t) (SSB - USB)');
legend('Original m(t)', 'Recovered m(t)');
xlabel('Time (s)');
ylabel('Amplitude');
```

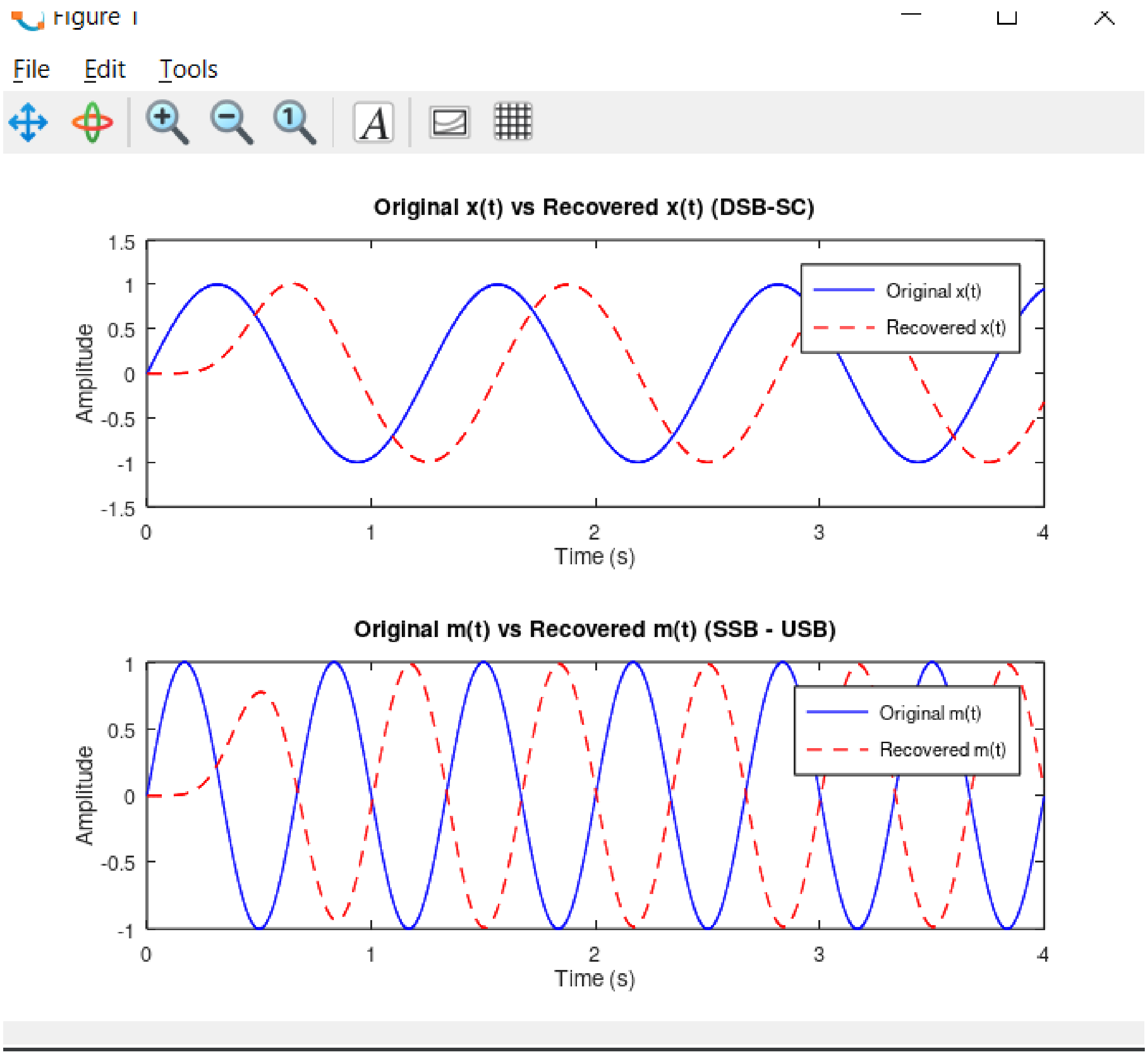


Figure 1
File
Edit
Tools
Original x(t) vs Recovered x(t) (DSB-SC)
Amplitude
Time (s)
1.5
1
0.5
0
-0.5
-1
-1.5
0
1
2
3
4
Original x(t)
Recovered x(t)
Original m(t) vs Recovered m(t) (SSB - USB)
Amplitude
Time (s)
1
0.5
0
-0.5
-1
0
1
2
3
4
Original m(t)
Recovered m(t)

Digital Communication

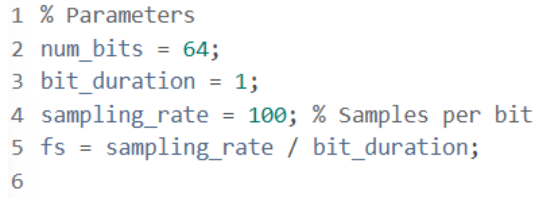
Part I

- ***Objective***

To develop and compare two line coding techniques using the Octave simulator by generating a random bit stream of at least 64 bits. One line code will be selected from AMI, CMI, or Manchester, and the other from unipolar NRZ or polar NRZ. Then plotting both the time and frequency domains for each code and analyzing their characteristics, with explanations and code provided.

- ***Chosen techniques:*** Manchester vs. Polar NRZ
- ***Code with explanation:***

1. Initialization & Parameters Setup



```
% Parameters
num_bits = 64;
bit_duration = 1;
sampling_rate = 100; % Samples per bit
fs = sampling_rate / bit_duration;

```

num_bits = 64;

Number of bits to be generated (minimum 64 as required).

bit_duration = 1;

Duration of each bit (1 second).

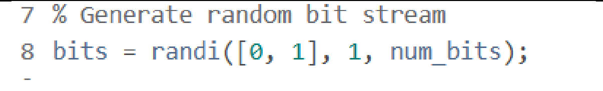
sampling_rate = 100;

Number of samples per bit (higher sampling rate = smoother waveform).

fs = sampling_rate / bit_duration;

Sampling frequency (samples per second).

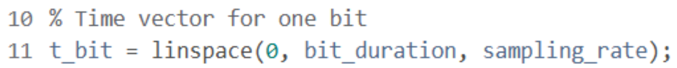
2. Random Bit Stream Generation



```
% Generate random bit stream
bits = randi([0, 1], 1, num_bits);
```

Generates a random binary sequence of length num_bits (64 bits). Each bit is either 0 or 1.

3. Time Vector for One Bit

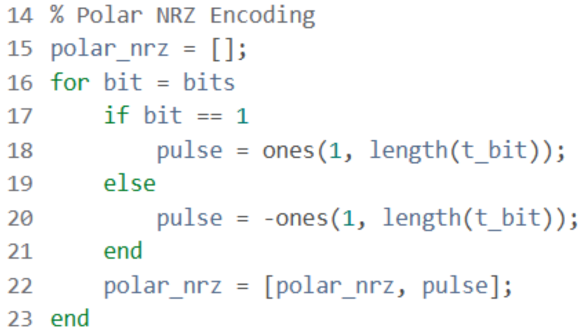


```
% Time vector for one bit
t_bit = linspace(0, bit_duration, sampling_rate);
```

Creates a linearly spaced time vector for one bit duration.

If bit_duration = 1 and sampling_rate = 100, it generates 100 points between 0 and 1.

4. Polar NRZ Encoding



```
% Polar NRZ Encoding
polar_nrz = [];
for bit = bits
    if bit == 1
        pulse = ones(1, length(t_bit));
    else
        pulse = -ones(1, length(t_bit));
    end
    polar_nrz = [polar_nrz, pulse];
end
```

<u>Polar NRZ Logic:</u>

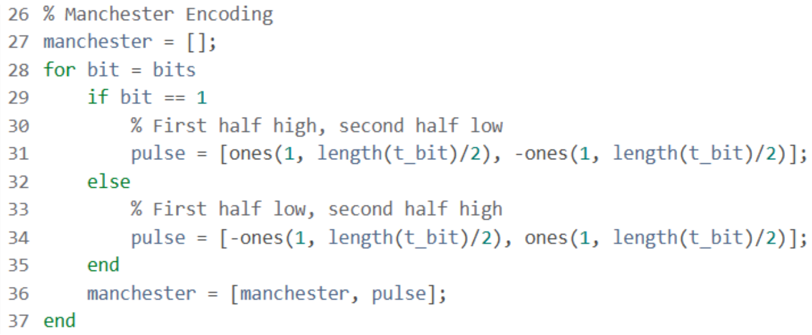
1 → +1V (constant for entire bit duration).

0 → -1V (constant for entire bit duration).

polar_nrz = [polar_nrz, pulse];

Adds each encoded bit to the ***polar_nrz*** array.

5. Manchester Encoding



```
% Manchester Encoding
manchester = [];
for bit = bits
    if bit == 1
        % First half high, second half low
        pulse = [ones(1, length(t_bit)/2), -ones(1, length(t_bit)/2)];
    else
        % First half low, second half high
        pulse = [-ones(1, length(t_bit)/2), ones(1, length(t_bit)/2)];
    end
    manchester = [manchester, pulse];
end
```

<u>Manchester Logic:</u>

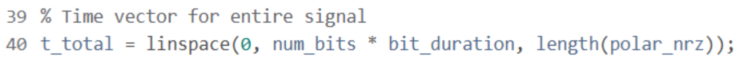
1 → High-to-Low transition (first half +1, second half -1).

0 → Low-to-High transition (first half -1, second half +1).

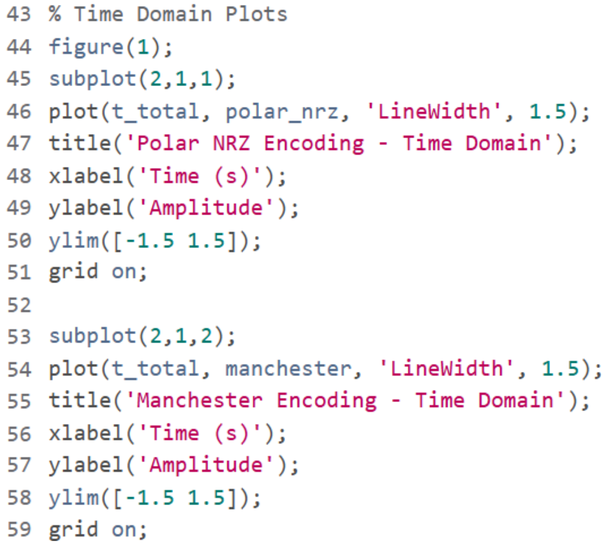
manchester = [manchester, pulse];

Adds each encoded bit to the ***manchester*** array.

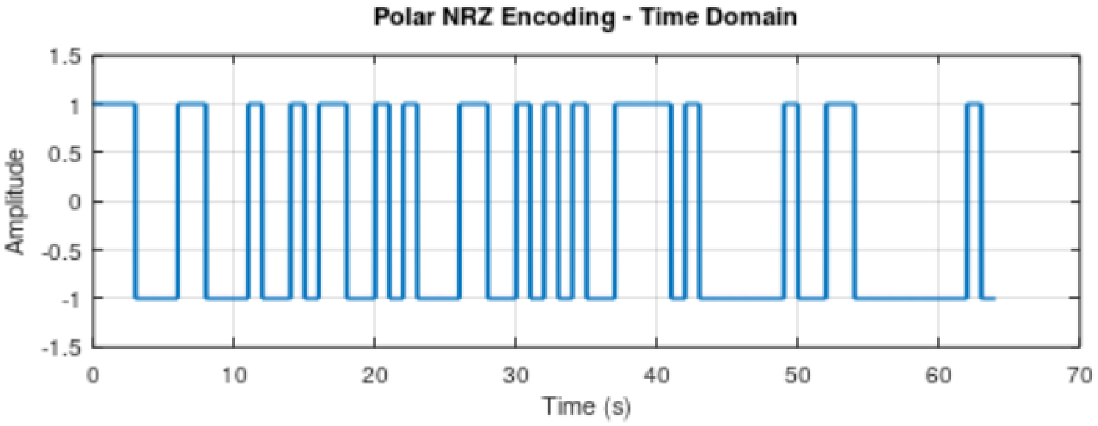
6. Time Vector for Entire Signal

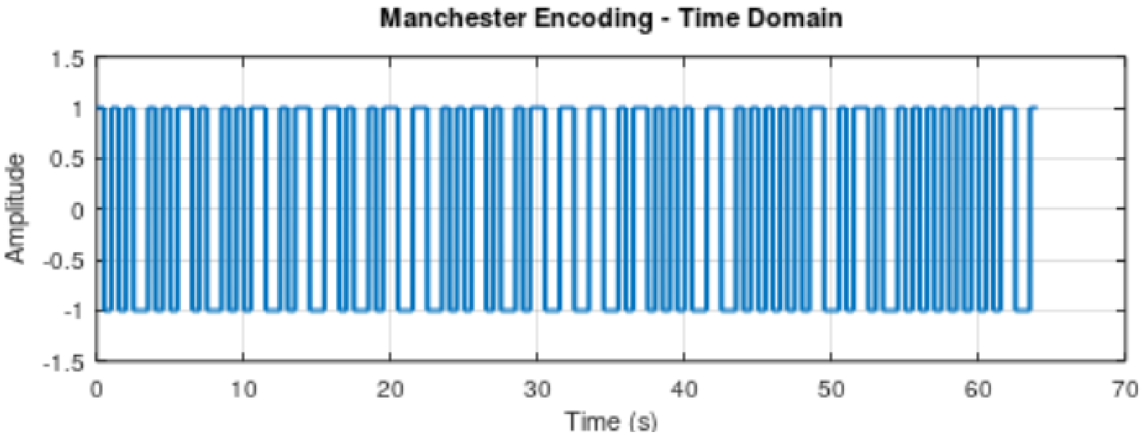

`% Time vector for entire signal`
`t_total = linspace(0, num_bits * bit_duration, length(polar_nrz));`
	Creates a time vector for the entire encoded signal.
	For 64 bits, it generates a time axis from 0 to 64 seconds.

7. Time Domain Plots

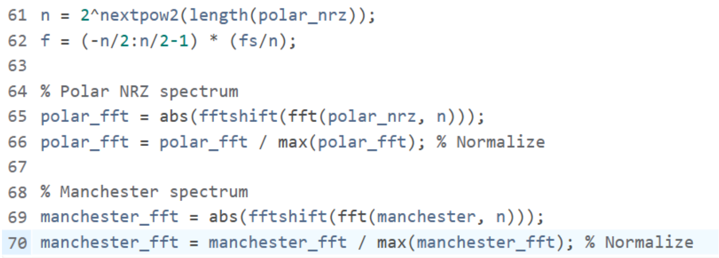

`% Time Domain Plots`
`figure(1);`
`subplot(2,1,1);`
`plot(t_total, polar_nrz, 'LineWidth', 1.5);`
`title('Polar NRZ Encoding - Time Domain');`
`xlabel('Time (s)');`
`ylabel('Amplitude');`
`ylim([-1.5 1.5]);`
`grid on;`
 	
`subplot(2,1,2);`
`plot(t_total, manchester, 'LineWidth', 1.5);`
`title('Manchester Encoding - Time Domain');`
`xlabel('Time (s)');`
`ylabel('Amplitude');`
`ylim([-1.5 1.5]);`
`grid on;`
	figure(1); → Creates a new figure window.
	subplot(2,1,1); → Divides the figure into 2 rows, 1 column, and selects the first subplot.
	plot(t_total, polar_nrz, 'LineWidth', 1.5); → Plots Polar NRZ in time domain.
	title(), xlabel(), ylabel() → Adds labels.
	ylim([-1.5 1.5]); → Sets y-axis limits.
	grid on; → Displays grid lines.

- ***Plots:***





8. Frequency Domain Analysis (FFT)



```
n = 2^nextpow2(length(polar_nrz));
f = (-n/2:n/2-1) * (fs/n);

% Polar NRZ spectrum
polar_fft = abs(fftshift(fft(polar_nrz, n)));
polar_fft = polar_fft / max(polar_fft); % Normalize

% Manchester spectrum
manchester_fft = abs(fftshift(fft(manchester, n)));
manchester_fft = manchester_fft / max(manchester_fft); % Normalize
```

n =2^nextpow2(length(polar_nrz)); Computes the next power of 2 for FFT efficiency.

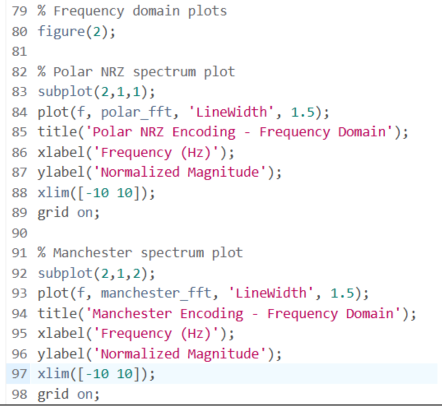
f = (-n/2:n/2-1) * (fs/n); Generates the frequency axis.

fft() → Computes the Fast Fourier Transform.

fftshift() → Centers the FFT around 0 Hz.

Normalization (/ max(polar_fft)) → Scales the spectrum to [0, 1].

9. Frequency Domain Plots



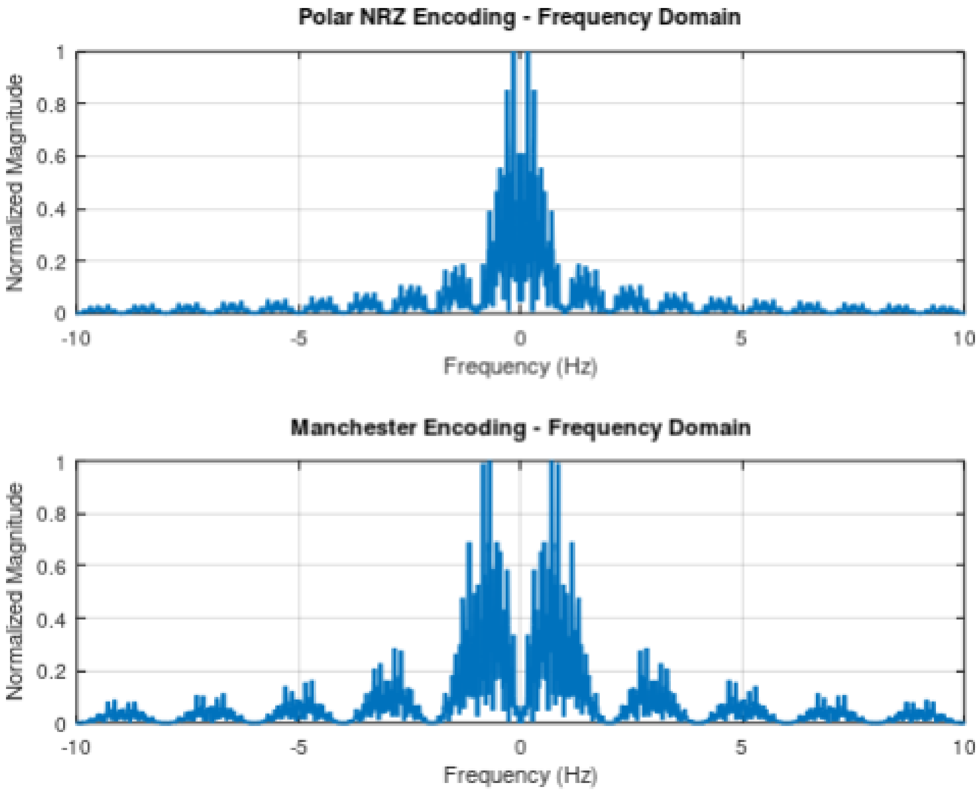
```
% Frequency domain plots
figure(2);

% Polar NRZ spectrum plot
subplot(2,1,1);
plot(f, polar_fft, 'LineWidth', 1.5);
title('Polar NRZ Encoding - Frequency Domain');
xlabel('Frequency (Hz)');
ylabel('Normalized Magnitude');
xlim([-10 10]);
grid on;

% Manchester spectrum plot
subplot(2,1,2);
plot(f, manchester_fft, 'LineWidth', 1.5);
title('Manchester Encoding - Frequency Domain');
xlabel('Frequency (Hz)');
ylabel('Normalized Magnitude');
xlim([-10 10]);
grid on;
```

f contains all our frequency values (both positive and negative)

polar_fft contains how much signal exists at each frequency

'LineWidth', 1.5 makes the line thicker and easier to see

- ***Plots***



❖ Analysis of Results

Time Domain Characteristics

Polar NRZ	Manchester
Represents 1s as a positive voltage level and 0s as negative voltage **Constant voltage level throughout the bit duration** **No DC component due to symmetrical positive and negative levels** **Long sequences of identical bits can cause synchronization issues**	**Each bit is represented by a transition in the middle of the bit period** **1 is represented by high-to-low transition** **0 is represented by low-to-high transition** **Always has a transition in the middle of each bit period** **No DC component due to equal positive and negative durations**

- Spectral Domain Characterstics

Polar NRZ	Manchester
Has a DC component (centered at 0 Hz). **Energy concentrated in lower frequencies.** **Efficient bandwidth use but poor synchronization for long similar bits.**	**No DC component due to balanced positive and negative parts.** **Spectral energy is centered away from 0 Hz.** **Higher bandwidth, but better for synchronization and error detection.**

- Advantages

	Polar NRZ	Manchester
Synchronization	Poor (long 0s/1s)	Good : Guaranteed transitions in each bit period make clock recovery reliable.
Bandwidth	Low Requires only half the bandwidth of Manchester coding	High
Implementation	Simple	Moderate
DC Component	Present	Absent
Error Detection	Weak	Strong: Violations of the coding rules can be automatically detected.

When to use which code:

<u>Use Manchester coding when:</u>

- You really need the receiver to easily sync up with your signal.
- You have enough room in your frequency band.

<u>Use Polar NRZ when:</u>

- Saving bandwidth is super important (like for sending data over long distances)
- You can handle a little more complexity in syncing up the signal

Conclusion

The simulation demonstrates key differences between Manchester and Polar NRZ coding. Manchester provides excellent synchronization capabilities at the cost of higher bandwidth requirements, while Polar NRZ offers better bandwidth efficiency but requires additional synchronization mechanisms. The choice depends on the specific requirements of the communication channel and the importance of clock recovery versus bandwidth efficiency.

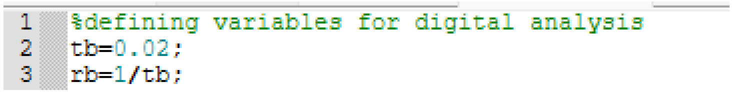
Part II

ASK System

We choose the BPSK system out of the three options available.

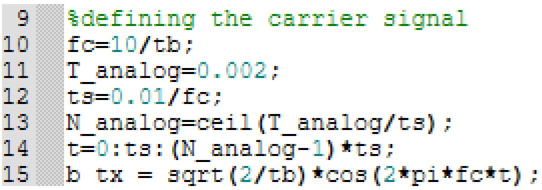
Transmitter

1- Defining Variables for Digital Analysis



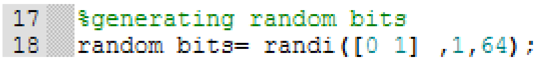
```
%defining variables for digital analysis
tb=0.02;
rb=1/tb;
```

2- Defining the Carrier Signal



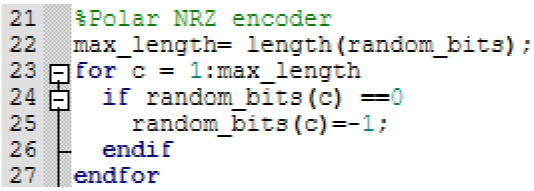
```
%defining the carrier signal
fc=10/tb;
T_analog=0.002;
ts=0.01/fc;
N_analog=ceil(T_analog/ts);
t=0:ts:(N_analog-1)*ts;
b tx = sqrt(2/tb)*cos(2*pi*fc*t);
```

3- Generating random bits



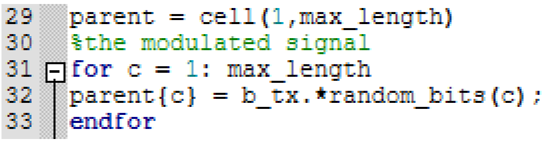
```
%generating random bits
random bits= randi([0 1] ,1,64);
```

4- Polar NRZ Encoder



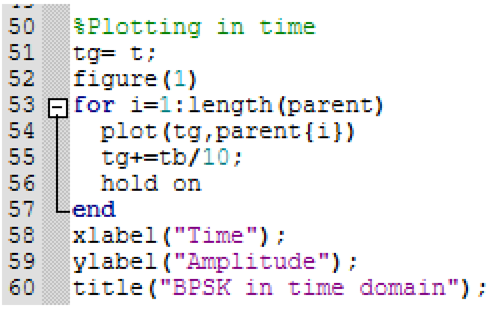
```
%Polar NRZ encoder
max_length= length(random_bits);
for c = 1:max_length
  if random_bits(c) ==0
    random_bits(c)=-1;
  endif
endfor
```

5- Getting the modulated Signal

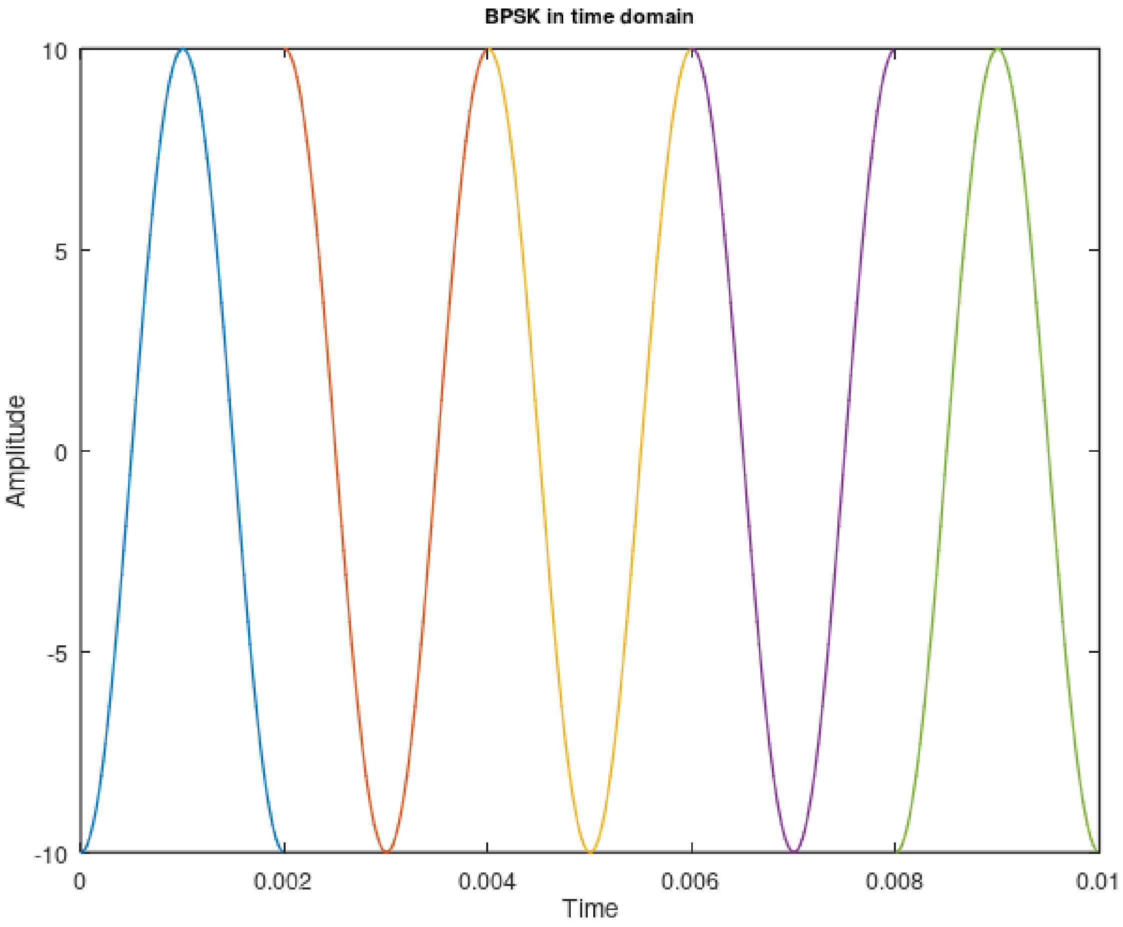


```
parent = cell(1,max_length)
%the modulated signal
for c = 1: max_length
parent{c} = b_tx.*random_bits(c);
endfor
```

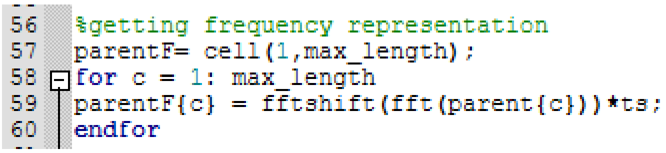
6- Plotting versus time (Choosing 5 bits only to plot to be visible)



```
%Plotting in time
tg= t;
figure(1)
for i=1:length(parent)
  plot(tg,parent{i})
  tg+=tb/10;
  hold on
end
xlabel("Time");
ylabel("Amplitude");
title("BPSK in time domain");
```

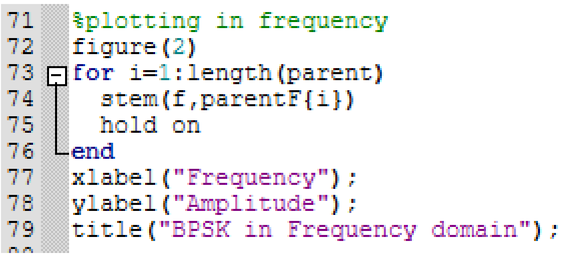


7- Getting frequency representation

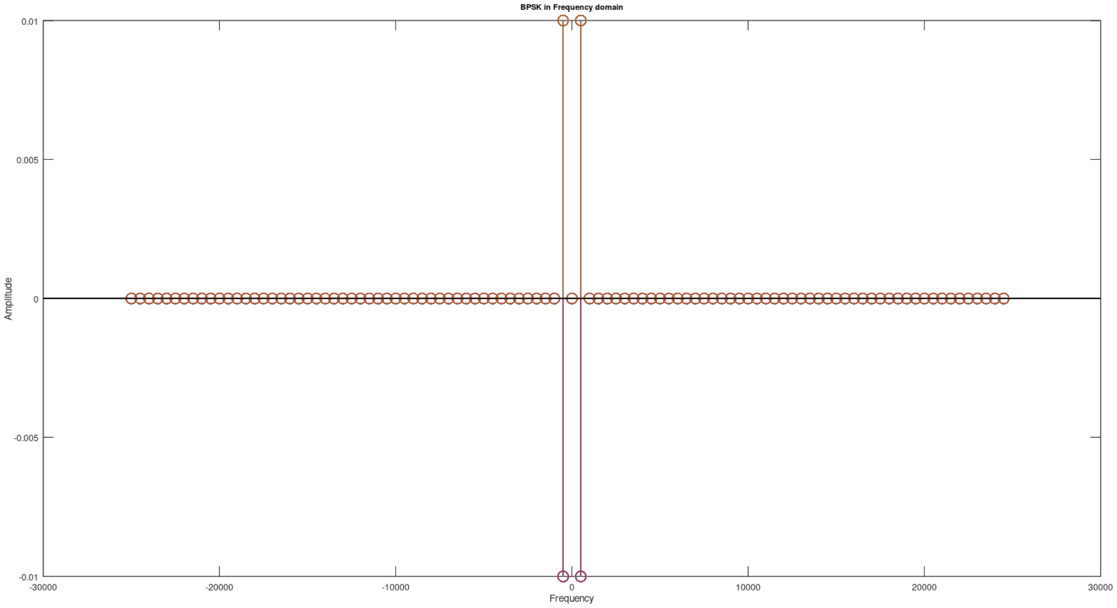


```
%getting frequency representation
parentF= cell(1,max_length);
for c = 1: max_length
parentF{c} = fftshift(fft(parent{c}))*ts;
endfor
```

8- Plotting in Frequency

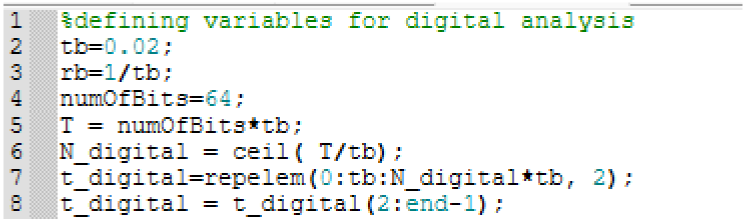


```
%plotting in frequency
figure(2)
for i=1:length(parent)
  stem(f,parentF{i})
  hold on
end
xlabel("Frequency");
ylabel("Amplitude");
title("BPSK in Frequency domain");
```



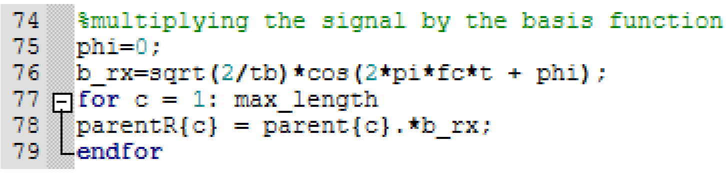
Receiver

1- Defining Variables for digital analysis



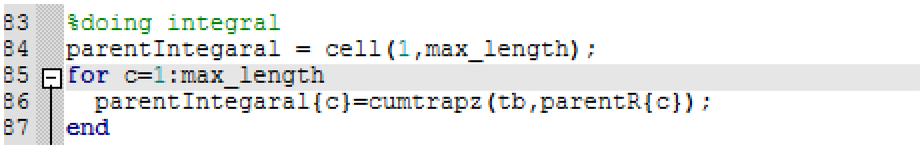
```
%defining variables for digital analysis
tb=0.02;
rb=1/tb;
numOfBits=64;
T = numOfBits*tb;
N_digital = ceil( T/tb);
t_digital=repelem(0:tb:N_digital*tb, 2);
t_digital = t_digital(2:end-1);
```

2- Multiplying the signal by the basis function



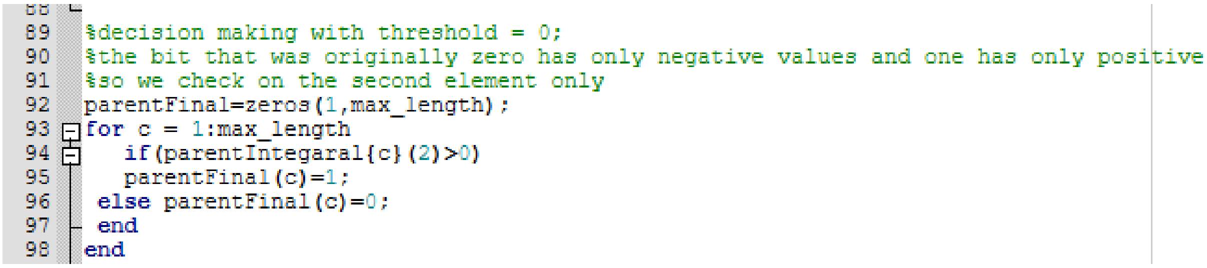
```
%multiplying the signal by the basis function
phi=0;
b_rx=sqrt(2/tb)*cos(2*pi*fc*t + phi);
for c = 1: max_length
parentR{c} = parent{c}.*b_rx;
endfor
```

3- Doing integral



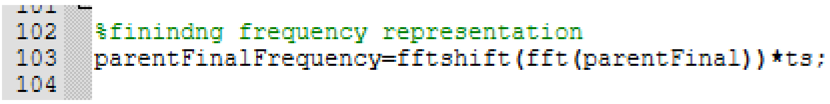
```
%doing integral
parentIntegaral = cell(1,max_length);
for c=1:max_length
  parentIntegaral{c}=cumtrapz(tb,parentR{c});
end
```

4- Decision Making



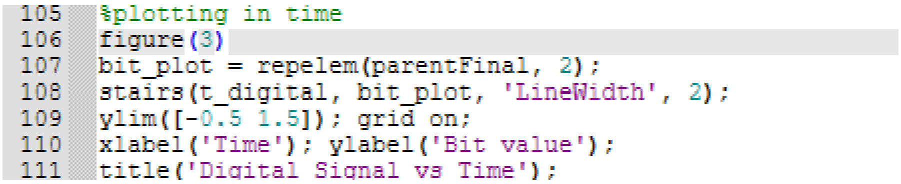
```
%decision making with threshold = 0;
%the bit that was originally zero has only negative values and one has only positive
%so we check on the second element only
parentFinal=zeros(1,max_length);
for c = 1:max_length
   if(parentIntegaral{c}(2)>0)
   parentFinal(c)=1;
 else parentFinal(c)=0;
 end
end
```

5- Finding frequency representation:

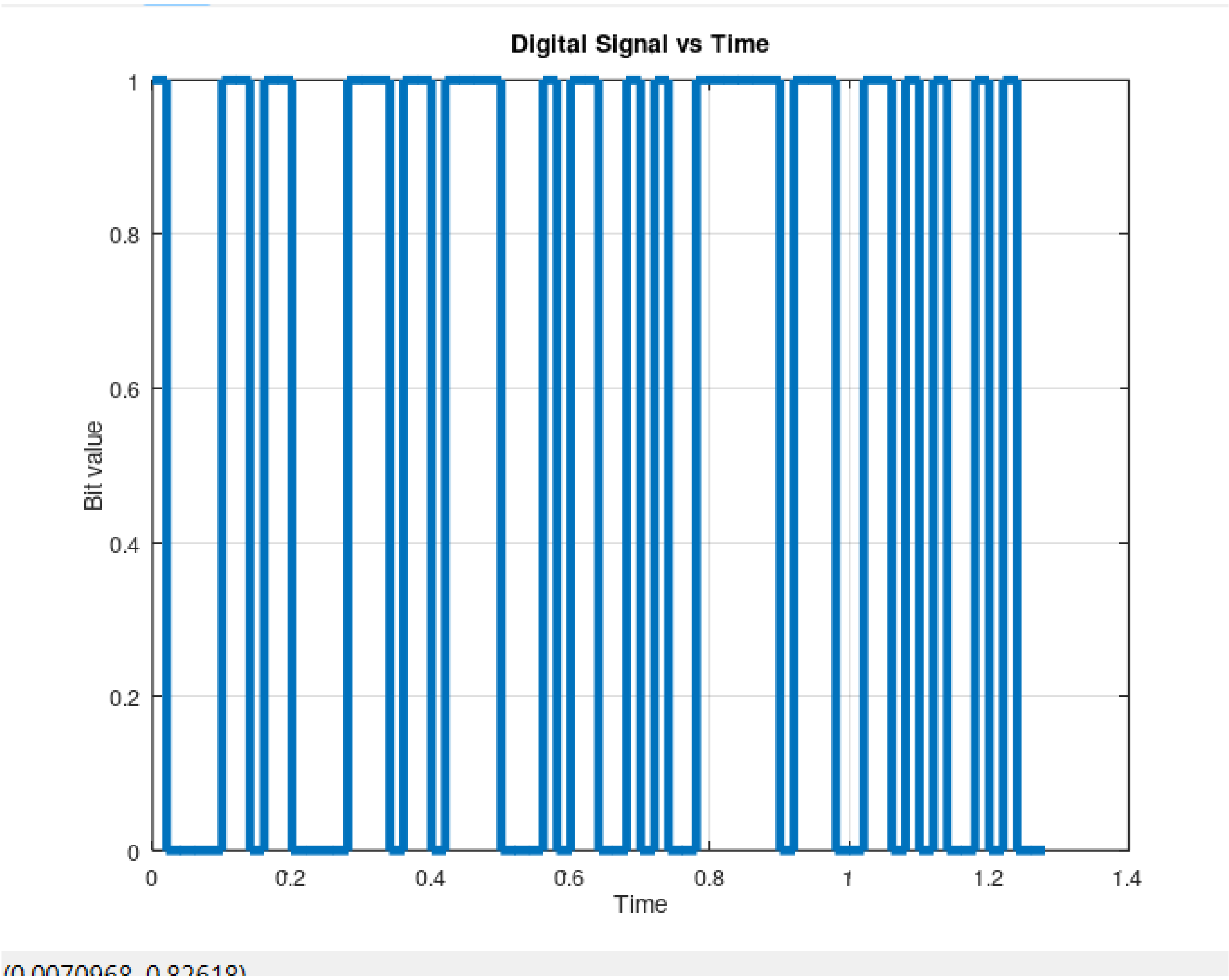


```
%finindng frequency representation
parentFinalFrequency=fftshift(fft(parentFinal))*ts;
```

6- Plotting in time

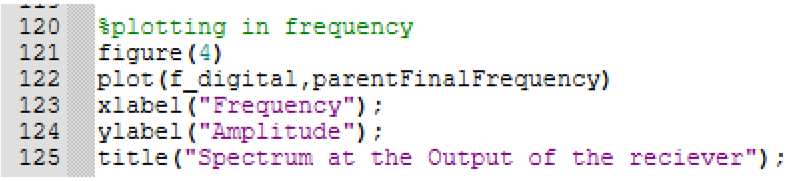


```
%plotting in time
figure(3)
bit_plot = repelem(parentFinal, 2);
stairs(t_digital, bit_plot, 'LineWidth', 2);
ylim([-0.5 1.5]); grid on;
xlabel('Time'); ylabel('Bit value');
title('Digital Signal vs Time');
```

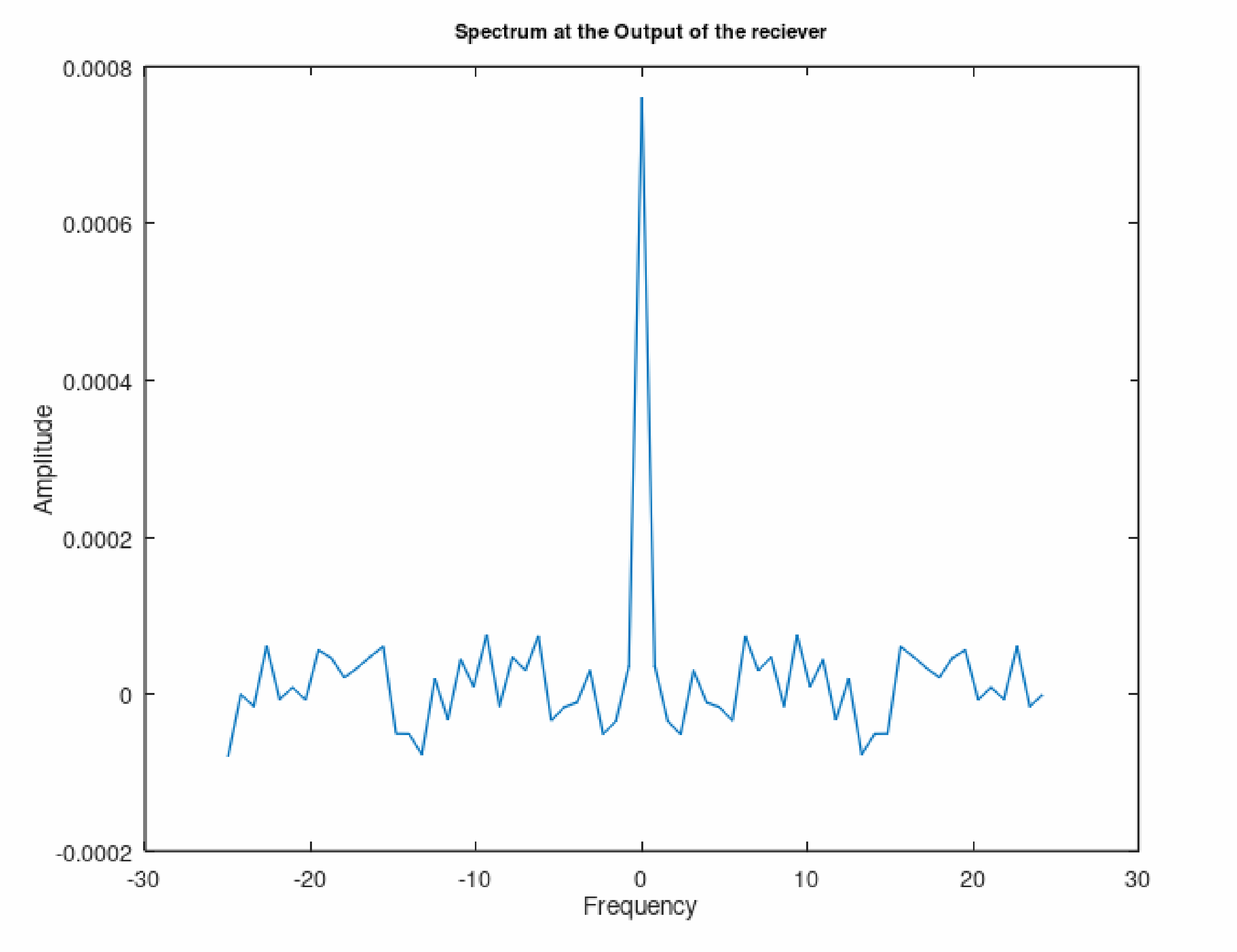


(0.0070968, 0.82618)

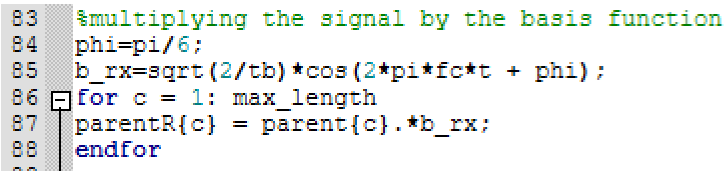
7- Plotting in frequency



```
%plotting in frequency
figure(4)
plot(f_digital,parentFinalFrequency)
xlabel("Frequency");
ylabel("Amplitude");
title("Spectrum at the Output of the reciever");
```



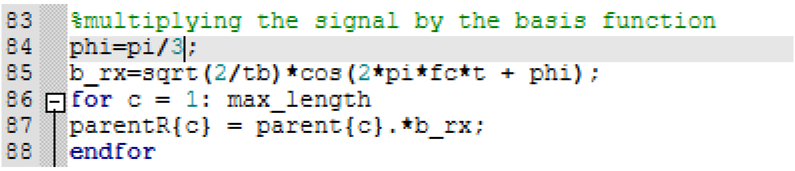
8- Making phi =30



```
%multiplying the signal by the basis function
phi=pi/6;
b_rx=sqrt(2/tb)*cos(2*pi*fc*t + phi);
for c = 1: max_length
parentR{c} = parent{c}.*b_rx;
endfor
```

Comment: It weaken the signal and reduces the SNR, but still detectable.

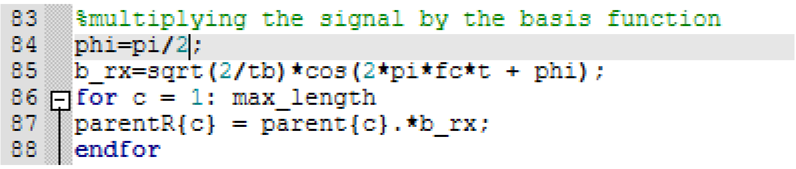
9- Making phi=60



```
%multiplying the signal by the basis function
phi=pi/3;
b_rx=sqrt(2/tb)*cos(2*pi*fc*t + phi);
for c = 1: max_length
parentR{c} = parent{c}.*b_rx;
endfor
```

Comment: It weaken the signal even more, getting it to 50% loss of the amplitude of the signal.

10- Making phi = 90



```
%multiplying the signal by the basis function
phi=pi/2;
b_rx=sqrt(2/tb)*cos(2*pi*fc*t + phi);
for c = 1: max_length
parentR{c} = parent{c}.*b_rx;
endfor
```

Comment: This inverts all bits, and hence data is lost.

Conclusion

By applying the concepts of BPSK, we gained valuable knowledge and practical approaches to how Digital Communications work in real life. Also, we got an insightful and important introduction to the world of Digital Signal Processing (DSP).