Tecnológico de Monterrey **Campus Ciudad de México** **School of Engineering and Sciences** **Department of Mechatronics**	**TE2019 Digital Signal Processing Laboratory** Student's 1 name and ID: Alfredo Zhu Chen - A01651980 Student's 2 name and ID: Luis Arturo Dan Fong - A016506720 Lab # 3 Elementary discrete-time signals and signal operations and transformations Date: 05/03/2020

1 INTRODUCTION

The need of knowing to use simple algorithms and basic signal processing is fundamental for a telecommunications and electronic engineer, to be able to interact and modify given signals to match with a desirable result. In the laboratory we were required to practice our knowledge of algorithms and signal processing. The practice was divided into several exercises in which the usage of compression, expansion, windowing and energy calculations were required to transform a given signal into a specific result.

OBJECTIVES

1. To implement basic signals transformations and operations in MATLAB
2. To implement low-complexity digital signal processing algorithms in MATLAB

2 MATERIALS & METHODS

The materials used in this laboratory were:

- PC or laptop
- MATLAB R2019a plus the Audio and Signal Processing Toolboxes
- Personal headphones

2.1 Finite-energy signals

Considering the given discrete-time signal from equation (1), first we define the interval of 26 values from -5 to 20 using the function *linspace*. Since the signal has a unit step function, a vector having the same size of the interval is created, it starts with 5 zeros because of the 5 negative values. Then, the signal is defined as figure 2.1.1 and it is considered a shifting of 6 because of MATLAB non negative or 0 index consideration. The signal is plotted using *stem* function.

$$x[n] = 0.5^{n} \cdot u[n] \quad (1)$$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%a) Creating the discrete signal by the definition
%Defining x[n]=(0.5^n)*u[n] from n=-5 to 20
    %1.Defining the size of point
n = linspace(-5,20,26);

    %2. unit step
%Since te function has unit step(u[n]), initialize 0 values for
%first 5 negative values and 1 for the rest of the values
u = [zeros(1,5) ones(1,21)];
%It can be also done by using the next line
%u =[0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1];

    %3. Define descrete signal
%Discrete Signal (With the addition of n+6 for adjusting the graph
%to the desirable location, starting from -5 and having 26 points.
x(6+n) = (0.5.^n).*u(6+n);

    %4.Plotting with Stem
stem(n ,x)
xlabel("n")
ylabel("x[n]")
title("x[n] vs n")
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 2.1.1 Defining signal from equation(1)

To determine if the signal from equation (1) is a finite-energy signal, we implement equation (2) to determine the energy of the signal using a for loop as shown in figure 2.1.2. We use *fprintf* to see the value.

$$E_x = \sum_{-\infty}^{\infty} |x[n]|^2 \quad [J] \tag{2}$$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%b) Computing infinite sum by definition
%Iterate over the size and add squared of the magnitude
E_x = 0;
for (z=1:1:26)
    E_x = E_x+(abs(x(z)))^2;
end
fprintf("The values of energy by definition is : "+E_x+"\n")
```

Figure 2.1.2 Calculating from scratch the energy of the signal

To verify our result, a symbolic MATLAB of function *symsum* is used. The first argument is the function, the second one is the independent variable, and the two final ones are the interval of k in which the function is evaluated.

```
38      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
39      %c) - Use SymSum to verify point b)
40
41 -    syms k;
42 -    test_Ex= double( symsum(((.5)^k)^2,k,0,20));
43 -    fprintf("The values of energy using symsum is : "+test_Ex+"\n")
44 -    fprintf("The values are the same: "+(E_x==test_Ex))
```

Figure 2.1.3 Verifying energy value result using symsum

2.2 Even and odd decomposition and energy

Considering X[n] to be (3), we started by defining and plotting it so we could see the whole graph and compare it with its even and odd components (Figure 2.2.1). Then we reflect the signal by inverting the order of values n and creating X[-n] as shown in the lines from 17 to 19.

$$x(t) = 1 - t \quad 0 \leq t \leq 1 \tag{3}$$

```
 6      %% Defining the function
 7 -    Ts=0.25;    %Sampling period
 8 -    n= linspace(-1,1,2/0.25+1); %Interval with by sampling period step
 9 -    u=[zeros(1,4) ones(1,5)];  %defining unit to make negative values 0
10 -    X=(1-n).*u; %defining function X[n]
11 -    subplot(5,1,1)
12 -    stem(n,X)   %plot the function X[n] with stem
13 -    xlabel("n")
14 -    ylabel("X[n]")
15 -    title("X[n]")
16
17 -    minusN= linspace(1,-1,2/0.25+1); %interval reversal
18 -    minusU=[ones(1,5) zeros(1,4)]; %unit step reversal
19 -    minusX=(1-minusN).* minusU; %defining function X[-n]
20 -    subplot(5,1,2)
21 -    stem(n,minusX)  %plot the function X[-n] with stem
22 -    xlabel("n")
23 -    ylabel("X[-n]")
24 -    title("X[-n]")
```

Figure 2.2.1 Defining X[n] and X[-n]

We do some mathematical operations using the previous part to create the even and odd components of the original function(Figure 2.2.2). After this, both components are plotted for later analysis purposes.

```
%% Even and odd components
X_even=0.5*(X+minusX);    %even component of X[n]
X_odd=0.5*(X-minusX);     %odd component of X[n]
subplot(5,1,3)
stem(n,X_even)
xlabel("n")
ylabel("X[n](even)")
title("X[n](even) vs n")
subplot(5,1,4)
stem(n,X_odd)
xlabel("n")
ylabel("X[n](odd)")
title("X[n](odd) vs n")

X_sum=X_even+X_odd;  %sum even and odd component
subplot(5,1,5)
stem(n,X_sum)
xlabel("n")
ylabel("X[n](sum)")
title("X[n](sum) vs n")
```

Figure 2.2.2 Creating even and odd components of the original signal

Finally we used the energy formula (2) from the experiment in section 2.1 to obtain the energy values for the odd component, the even component, the original signal and the sum of both components (figure 2.2.3).

```
%% Energy
%Initialize energy values
E_x_even=0;
E_x_odd=0;
E_x_sum=0;
E_x=0;
%Iterate over size of X and sum squared value at each step
for i=1:size(minusX,2)
    E_x_even = E_x_even+(X_even(1,i))^2;
    E_x_odd= E_x_odd+(X_odd(1,i))^2;
    E_x_sum = E_x_sum+(X_sum(1,i))^2;
    E_x = E_x+(X(1,i))^2;
end

fprintf("The value of E_x_even is: "+E_x_even+"\n")
fprintf("The value of E_x_odd is: "+E_x_odd+"\n")
fprintf("The value of the sum of E_x_even and E_x_odd is: "+E_x_sum+"\n")
fprintf("The value of original E_x is: "+E_x+"\n")
```

Figure 2.2.3 Obtaining energy values for both components, the sum of them and original signal

2.3 Expansion and compression of discrete-time signals

In this part we started by defining and plotting the discrete-time signal (4) for using it as a reference for the compression and expansion processes.

$$x[n] = cos(2\pi n/7) \tag{4}$$

```
6       %Defining the function
7 -     nvalues=21;
8 -     n= linspace(-10,9,nvalues);    %Interval
9 -     X=cos(2*pi*n/7);
10 -    subplot(3,1,1)
11 -    stem(n,X)
12 -    xlabel("n")
13 -    ylabel("X[n]")
14 -    title("X[n](original) vs n")
15
```

Figure 2.3.1 Defining and plotting original signal

For the compression process we were required to avoid the usage of the downsample and upsample commands, so we decided to use a *for* loop to iterate over x by step size of 2 (Figure 2.3.2). The step size of 2 has the purpose to sample one value and skip another. It is also important to notice that each even index n should be sampled, but matlab has no 0 indexation, so it starts it samples each odd index to adjust this.

```
15
16      %% Compression(Downsampling)
17      %Iterate over X by step of 2 and take sample with n value
18 -    for i=1:2:size(X,2)
19 -        X_compression((i+1)/2)=X(i);
20 -        n_compression((i+1)/2)=n(i);
21 -    end
22
23 -    subplot(3,1,2)
24 -    stem(n_compression,X_compression)
25 -    xlabel("n")
26 -    ylabel("X[2n]")
27 -    title("X[2n](compression) vs n")
28
```

Figure 2.3.2 Compression of the signal

In the case of the expansion we iterate over x adding 0 between samples(Figure 2.3.3). The interval n_expansion values is defined with *linspace*, the size should be 2 times bigger than the original.

```
%% Upsample
%Iterate over X and add 0 for each sample
X_expansion=[];
n_expansion= linspace(-10,9,size(n,2)*2);
for i=1:size(X,2)
    X_expansion=[X_expansion X(i) 0];
end

subplot(3,1,3)
stem(n_expansion,X_expansion)
xlabel("n")
ylabel("X[n/2]")
title("X[n/2](expansion) vs n")

```

Figure 2.3.3 Expansion of the signal

2.4 Echo of music

For this part we were required to use a “handel.mat” file to obtain its music file and simulate an echo in a specific situation. In this case we began by loading the “handel.mat” file and plotting its “y” function, as well as playing its sound(Figure 2.4.1).

```
%Load sample and produce sound
load handel.mat
sound(y,Fs)
pause(10)
subplot(2,1,1)
plot(y)
xlabel("n")
ylabel("y[n]")
title("Sound y[n] vs n")
```

Figura 2.4.1 Loading “handel.mat” file, plotting and sound testing

Once the original signal was plotted, we calculated the delay of N1 and N2 considering (5), where v = 245 m/s and the distance was given by the instructions (distance for N1 = 17 m, while distance for N2 = 34 m) (figure 2.4.2).

$$N = \frac{d}{v} \quad [s] \tag{5}$$

```
%Calculate delay N1 and N2 for echo effect distance over velocity(345m/s)is
%the time it takes to bounce back, divide that with the sampling period to
%know at which point starts to play the echo. Fs is already define as we
%load the handel.mat
N1=round((17/345)/(1/Fs))
N2=round((34/345)/(1/Fs))
```

Figure 2.4.2 N1 and N2 values

Then, the two echo signals are adjusted to match the original sound plus the time the echo was done by adding zeros to fill the gaps until the size matched. Finally after proving that both the echoes and the

original sound were the same length, we create our “r” function in which the echoes were added to the original sound with different amplitudes. *audiowrite* function is used to store both signals to the computer.

```
22 -    Nmax=max(N1,N2);    %Verify which delay is larger in order to know how long
23      %will be the total time
24
25      %The echos must match the size of the original sound+the time of echo
26 -    echo1=[zeros(N1,1);y;zeros(Nmax-N1+1,1)];  %define echo1
27 -    echo2=[zeros(N2,1);y;zeros(Nmax-N2+1,1)];   %define echo2
28
29      %Adding original sound padded with 0 and echo effects, you can change
30      %amplitudes to experiment other results
31 -    r=1*[y;zeros(Nmax,1);0]+0.6*echo1+0.2*echo2;
32
33      %Produce the sound
34 -    sound(r,Fs)
35
36 -    subplot(2,1,2)
37 -    plot(r)
38 -    xlabel("n")
39 -    ylabel("r[n]")
40 -    title("Sound with echo r[n] vs n")
41 -    audiowrite("withoutEcho.wav",y,Fs)
42 -    audiowrite("withEcho.wav",r,Fs)
```

Figure 2.4.3 Creating echo1 and echo2 for final the sound

2.5 Rectangular windowing

In this experiment, we are looking at the effect of applying a window to highlight a specific part of a signal. First, we define an interval *n* from -10 to 10. The input signal x1 is defined as n times the heaviside function(unit step function) in line 8. The rectangular window signal *w* is highlighting from 0 to N=6 in this case, it is important to consider that the heaviside evaluated at 0 is equal to 0.5 in MATLAB, so we change the values to 1 when n=0 and n=N to follow the unit step arbitration from the lab briefing. y1 is calculated as the multiplication of w and x1. A second case of x2 is implemented and it is also computed.

$$x[n] = n \cdot u[n] \tag{6}$$

$$w[n] = u[n] - u[n - N] \tag{7}$$

```
 7 -    n=-10:10; %Define interval
 8 -    x1=n.*heaviside(n);  %Define X1 function
 9 -    w=heaviside(n)-heaviside(n-N);  %Window function
10 -    w(find(n==0 | n==N))=1; %set w[0] and w[N] to 1
11 -    y1=x1.*w;   %define output y1 of the windowing system
12
13 -    x2=(n-5).*heaviside(n-5);   %let input x2 be x[n-5]
14 -    y2=x2.*w;   %compute output y2 when using x2
```

Figure 2.5.1 Windowing process for input signal

```
16      %% Plotting all
17 -    subplot(5,1,1)
18 -    stem(n,x1)
19 -    xlabel("n")
20 -    ylabel("x1[n]")
21 -    title("x1[n] vs n")
22 -    subplot(5,1,2)
23 -    stem(n,w)
24 -    xlabel("n")
25 -    ylabel("w[n]")
26 -    title("w[n] vs n")
27 -    subplot(5,1,3)
28 -    stem(n,y1)
29 -    xlabel("n")
30 -    ylabel("y1[n]")
31 -    title("y1[n] vs n")
32 -    subplot(5,1,1)
33 -    stem(n,x2)
34 -    xlabel("n")
35 -    ylabel("x2[n]")
36 -    title("x2[n] vs n")
37 -    subplot(5,1,4)
38 -    stem(n,y2)
39 -    xlabel("n")
40 -    ylabel("y2[n]")
41 -    title("y2[n] vs n")
```

Figure 2.5.2 Plotting all signals

3 RESULTS AND DISCUSSION

3.1 Finite Energy signals

The next plot shows x[n] within the interested interval from -5 to 20. Negative values are 0 because of the unit step function u[n].

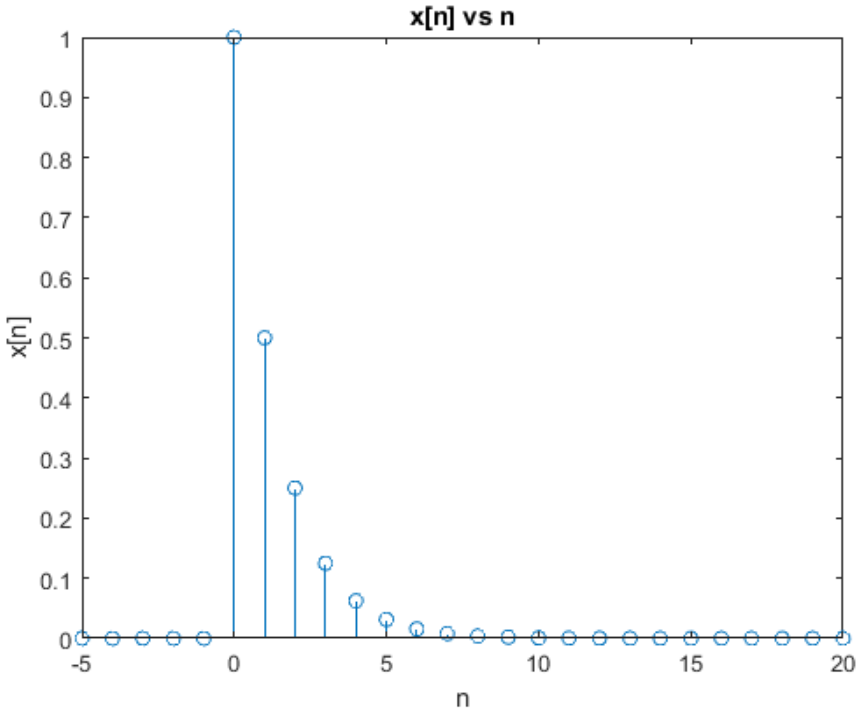

Figure 3.1.1 Signal x[n] vs n

In figure 3.2, we can see that the value of the energy calculated by definition from scratch and the one using function symsum are equal, both results are 1.3333 J and we can say that the method we implemented is correct. We can also find out that the discrete-time signal which we used in this section of the experiment is classified as a finite.energy discrete-time signal because of its finite energy value 1.3333 J.

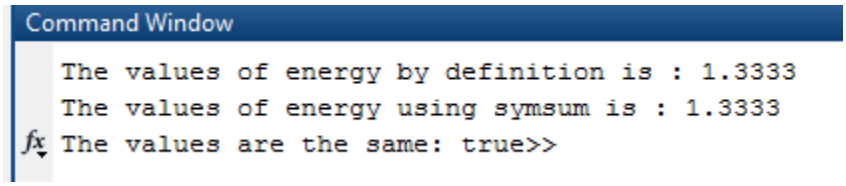

Figure 3.1.2 Showing the values of energy in Command Window

3.2 Even and odd decomposition and energy

As it can be seen from the next figure 3.2.1, it can be seen that the sum of the even and odd components of the original signal is the same as the original signal, this verifies that we obtained correct results

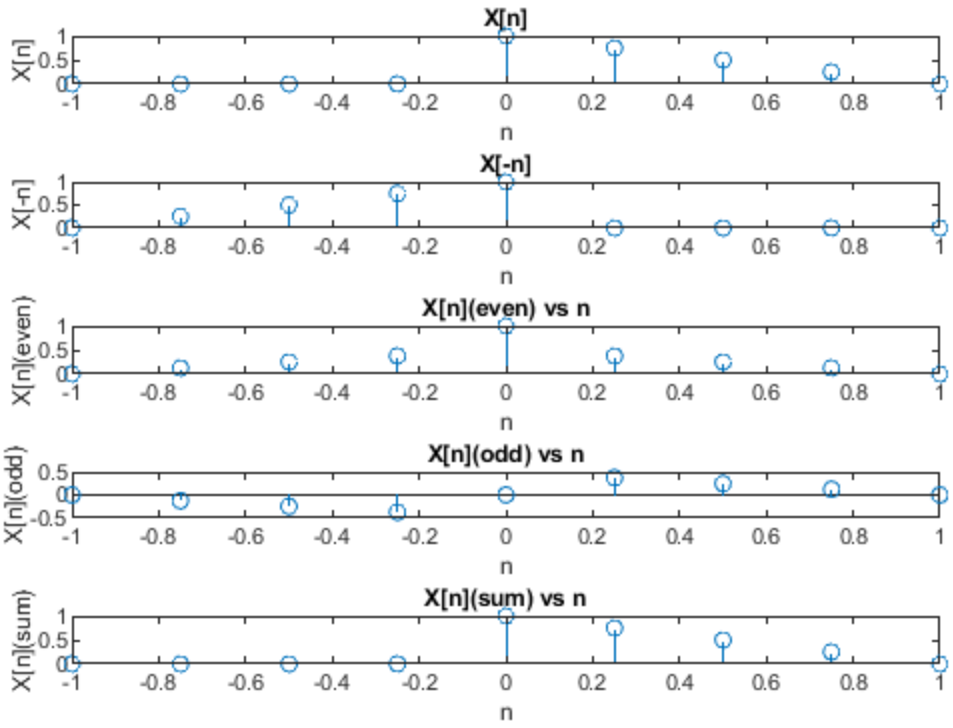

Figure 3.2.1 plots for all signals vs n

From the Command Window in the next figure, we print out the energy values. It can be observed that the sum of the energy of both components is equal to the value of energy from the original signal.

```
Command Window
The value of E_x_even is: 1.4375
The value of E_x_odd is: 0.4375
The value of the sum of E_x_even and E_x_odd is: 1.875
The value of original E_x is: 1.875
fx >>
```

Figure 3.2.3 Showing the values of energy in Command Window

3.3 Expansion and compression of discrete-time signals

At one hand, considering the down-sampling by 2 of the original signal, the number of samples are reduced by a factor of 2, this is because the sampling takes place in each 2 positions of the signal until the process is completed.
On the other hand, The expansion for the signal yields twice the size of the original signal. The signal y is such that y[n]=x[n/2] for n even, and y[n]=0 otherwise. When n/2 is not defined in the original sample, it takes the value of 0, so between each defined sample in the original signal, which can later be used for interpolation and having a likely continuous result.

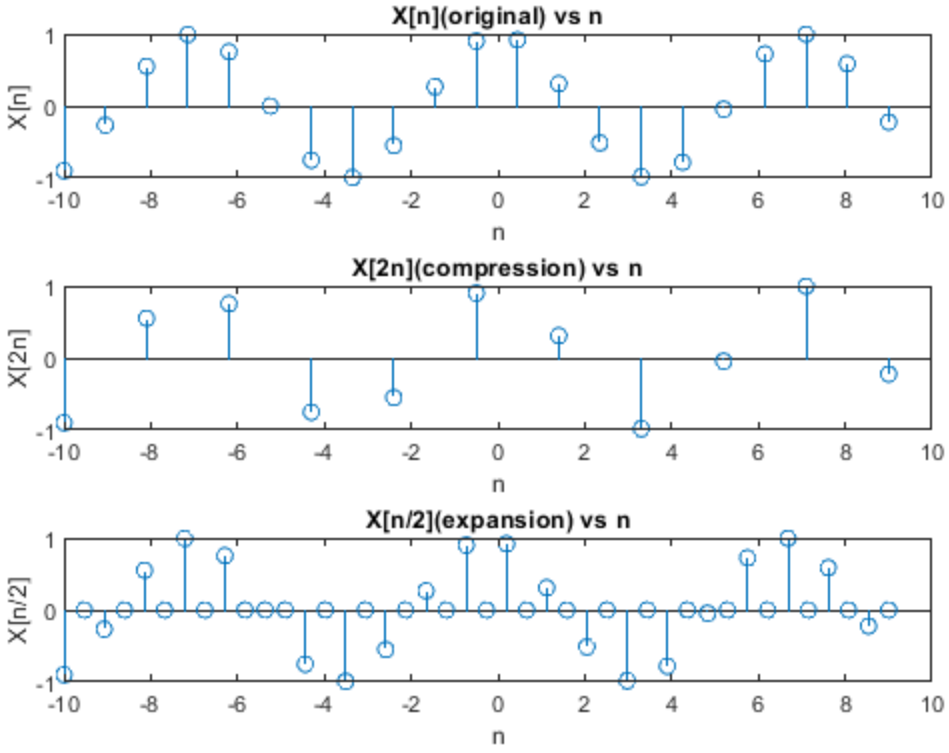

Figure 3.3.2 plot of the original, compressed and expanded signal

3.4 Echo of music

The delay values result as figure 3.4.1, N_1=404 s and N_2 =807 s. After all the process, the echoed effect can be heard and appreciated. The original music signal and the echoed music signal are plotted in figure 3.4.2, there is lightly difference because of the amplitudes we used.

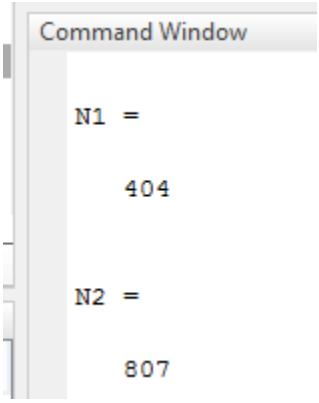

Figure 3.4.1 delay values N_1 and N_2

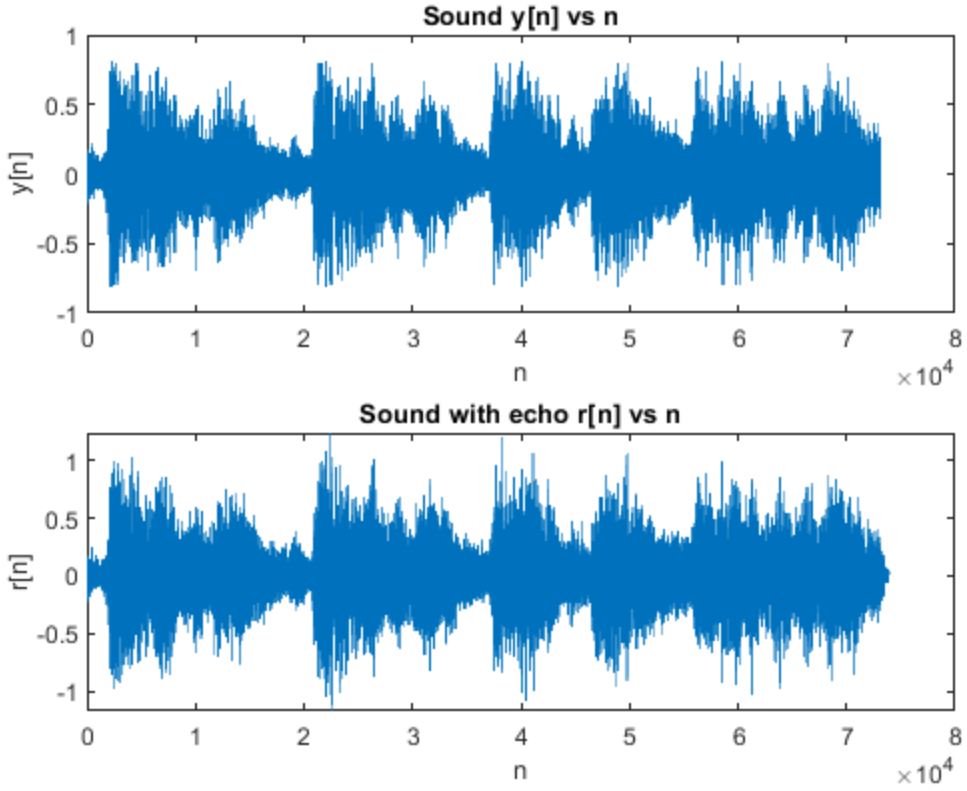

Figure 3.4.2 Original music and Echoed music

3.5 Rectangular windowing

The next image shows the resulting plots of x1[n] - input signal, w[n] - rectangular window signal, y1[n] - resulting signal from the multiplication of x1[n] and w[n], y2[n] - resulting signal from the multiplication of x2[n] and w[n]. The rectangular window system is linear because it satisfies the superposition principle as follows(considering $\alpha$and β two different constans):

$$T\{\alpha x_1[n] + \beta x_2[n]\} = \alpha T\{x_1[n]\} + \beta T\{x_2[n]\} \quad (8)$$

$$(\alpha x_1[n] + \beta x_2[n])w[n] = \alpha x_1[n]w[n] + \beta x_2[n]w[n] \quad (9)$$

$$\alpha x_1[n]w[n] + \beta x_2[n]w[n] = \alpha x_1[n]w[n] + \beta x_2[n]w[n] \quad (10)$$

The rectangular windowing system is not time-invariant, since it changes with time. When input x1[n] produces output y1[n], a shifted input x1[n-5](also known as x2[n] in our code) doesn't produce a shifted output y1[n-5](also known as x2[n] in our code). This can be illustrated with the figure 2.5.1.

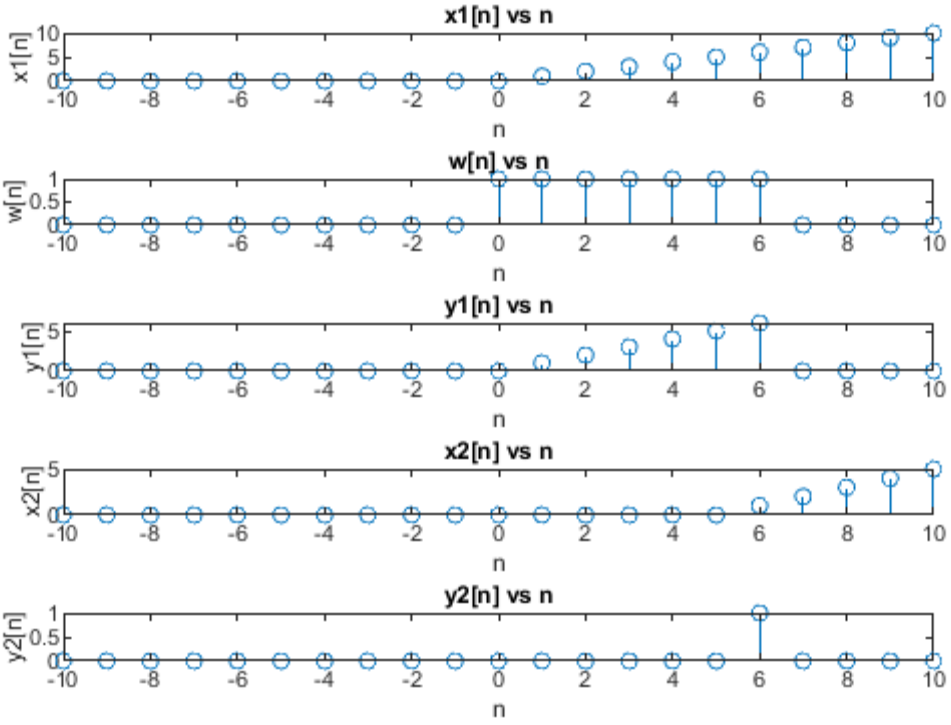

Figure 3.5.1 plots of all signals

4 CONCLUSION

We can conclude that the usage of signal processing and simple algorithms are fundamental for modifying signals into desirable results. In this particular case we achieved our objectives, by processing the given data into its even and odd components, generating an energy function, compressing and expanding signals, widowing the given signal and echoing a sound from a file.