National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

EE-330 Digital Signals Processing

Assignment 2 & 3

Student:

Muhammad Junaid Ali 290927

Muhammad Abdullah Khan 321711

Amur Saqib Pal 288334

Instructor: Dr. Jameel Nawaz

Class: BEE-11D

EE330: Digital Signal Processing

Assignment # 02 and 03

Issued on: 19/11/2021 Total Marks = 40 **Due Date:** 20/12/2021

Note: Submit your assignment on LMS folder.
Late submission will result in cancellation of 5 marks per day.
The assignments should be done in groups of 2 people at maximum.

Computing and Visualizing the Discrete-Time Fourier Transform

Objective

In this project, you will learn how to use MATLAB to compute the discrete-time Fourier transform (DTFT) and you will verify your result analytically. You will also learn how to perform real-time DTFT analysis of audio by computing multiple DTFTs of shorter blocks of the signal.

Introduction

In general, MATLAB can only estimate the DTFT of a DT signal because the DTFT is a continuous function of ω—i.e., the DTFT needs to be evaluated for all possible frequencies—something we can't do on the computer! We can estimate the DTFT of a DT signal by evaluating the DTFT at a finite number of frequencies. If we use a fine enough spacing of frequency values, our estimate of the DTFT can be quite good. In MATLAB, the function `freqz()` is used to estimate the DTFT (type `help freqz` in MATLAB to learn how to use `freqz()`).

Another issue which arises when using a computer to estimate the DTFT of a signal is the fact that we need an infinite amount of memory to store an infinite-duration signal. In reality, to estimate the DTFT of a signal efficiently, most applications use a procedure called windowing in which the DTFT algorithm is input only a finite-duration segment or "chunk" of the signal (e.g., a segment of length 32 samples). Consequently, even though `freqz()` can be quite good in estimating the DTFT, the estimate can become corrupted by this windowing operation.

The objective of this project is to provide you with experience in computing the DTFT in MATLAB. In Part 1, you'll compute the DTFT of a sinusoid both by hand and by using MATLAB. This part is meant to serve as a warm-up exercise since the DTFT of a sinusoid is very simple. In Part 2, you'll compute the DTFT of both pure tones and audio tracks, and you'll compare what you compute with what you hear. In Part 3, you'll create a real-time display in which the DTFT is shown graphically as the audio is played back.

Part 1: Warm-up—How to compute the DTFT in MATLAB

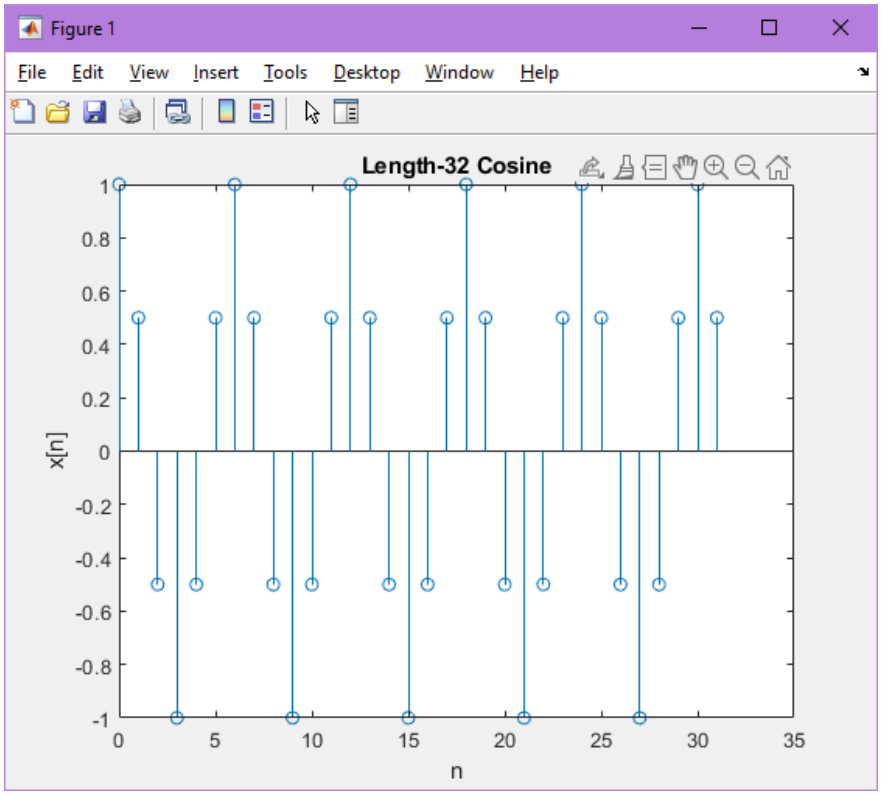
Given x[n] = cos(ω_0n) where $\omega_0 = \pi/3$, find the period of x(n) and give an expression for and sketch X($e^{j\omega}$). Now, let's estimate this DTFT in MATLAB by using freqz() and a length-32 chunk of x(n) . First, define the length-32 cosine via code similar to the following:

```
n = 0:31;       % length-32 chunk
w0 = pi/3;      % frequency of the cosine
x = cos(w0*n); % define the cosine
```

Code

```
n=0:31;          % n goes from 0 to 31 (Length 32)
w0=pi/3;         % frequency of cos
x=cos(w0*n);     % cosine function
stem(n,x);       % plotting cosine
xlabel('n'); ylabel('x[n]'); title ('Length-32 Cosine');
```

Screenshot



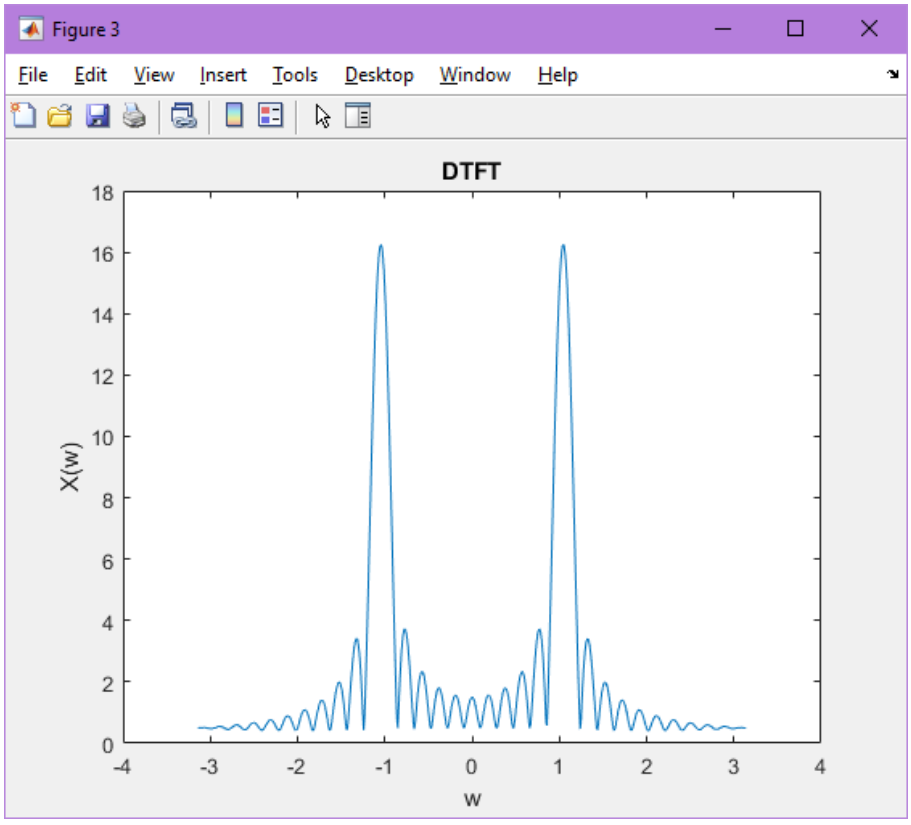
Plot the signal by using the stem() function (use help stem in MATLAB to learn how to use stem()). You should see samples ("lollipops") that follow the profile of a CT cosine. (NOTE: Remember to save all plots—you'll need to hand them in.)

Next, use freqz() to estimate the DTFT. Use code similar to the following:

```
% estimate the DTFT over the interval [-pi, pi]
w = -pi:0.01:pi
X = freqz(x, 1, w);

% plot the magnitude spectrum
figure; plot(w, abs(X));
```

```
n=0:31;          % n goes from 0 to 31
w0=pi/3;         % freq of cos
x=cos(w0*n);     % cosine function
stem(n,x);       % plotting cosine
xlabel('n'); ylabel('x[n]'); title ('Length-32 Cosine');
w=-pi:0.01:pi;  % [-pi,pi]
X=freqz(x,1,w); % taking the DTFT
figure; plot(w,abs(X)); % plotting the DTFT
xlabel('w'); ylabel('X(w)'); title ('DTFT');
```

Screenshot

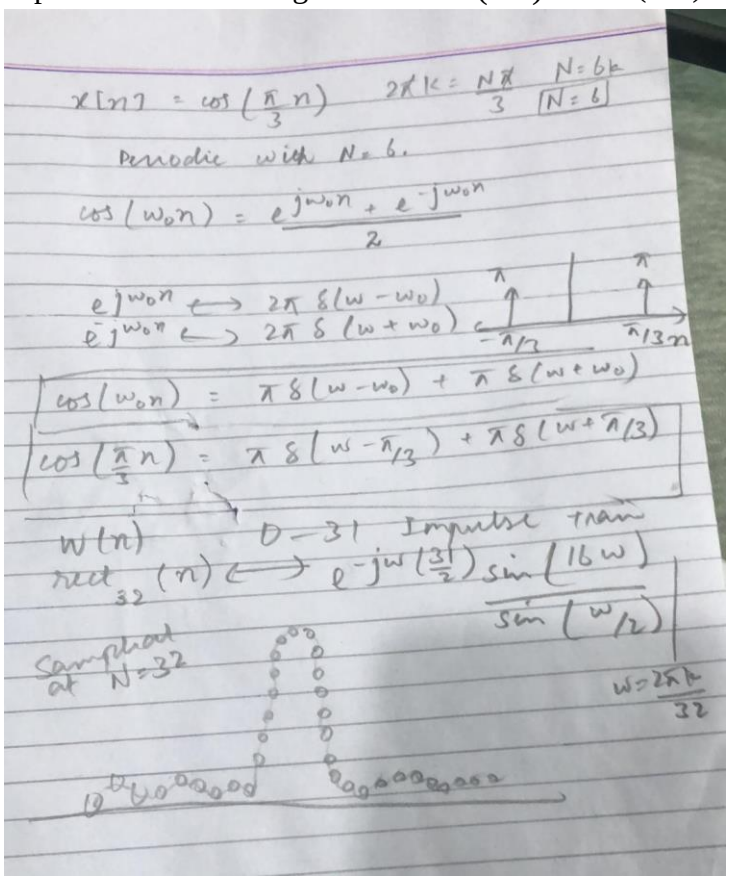


Your estimate of the DTFT won't be exactly the same as the analytical expression for $X(e^{j\omega})$ which you computed earlier. Describe the differences and similarities (e.g., do the locations of the peaks roughly coincide?).

Again, the reason the estimate of the DTFT isn't the same as the analytical expression is due to the fact that the input to freqz() was only a finite-length chunk of the cosine. We model this process as multiplication of the cosine with a DT rectangle

$$w(n) = \begin{cases} 1, & 0 \le n \le 31 \\ 0, & \text{else} \end{cases}$$

The resulting length-32 signal $\tilde{x} = x(n)w(n)$ is what we actually fed to freqz(). Derive expressions for the magnitudes of $\tilde{X}(e^{j\omega})$ and $W(e^{j\omega})$.



Next, write a function in MATLAB for your expression for the magnitude of $\tilde{X}(e^{j\omega})$; call it **X_tilde_mag()** and save it in your path with filename "X_tilde_mag.m". Here's some bare-bones code for "X_tilde_mag.m" to get you started:

```
function result = X_tilde_mag(w)

% TODO: Insert your function code here; be sure
% to use ./ (dot-divide) notation when computing
% the point-by-point division of two vectors
```

Next, evaluate your X_tilde_mag() function at the same frequencies which we earlier used with freqz()—i.e., frequencies w = -pi:0.01:pi. On the same graph, use the following code to plot your output from X_tilde_mag() along with the output from freqz() that was computed in Part 1:

Next, evaluate your X_tilde_mag() function at the same frequencies which we earlier used with freqz()—i.e., frequencies w = -pi:0.01:pi. On the same graph, use the following code to plot your output from X_tilde_mag() along with the output from freqz() that was computed in Part 1:

```
% plot the freqz magnitude spectrum and the analytical
% magnitude spectrum for the windowed cosine
figure; plot(w, abs(X)); hold on;
plot(w, X_tilde_mag(w), 'ro');
```

Code

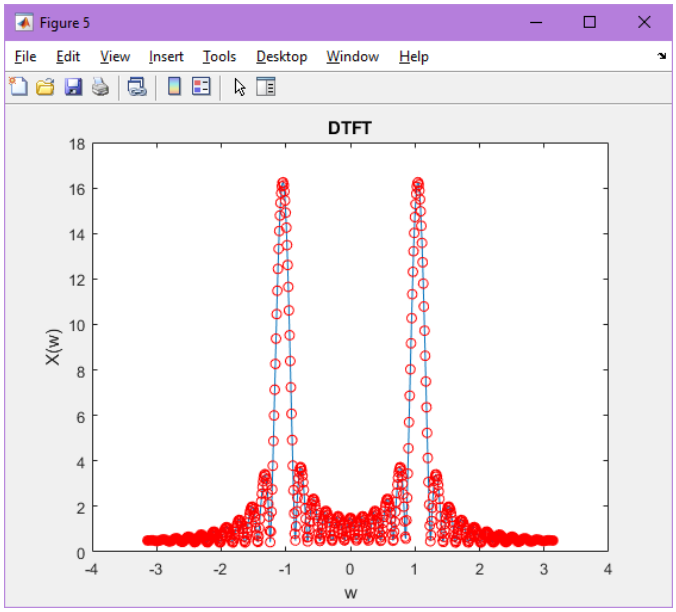
Function

```
function result = X_tilde_mag(x,n,w)
result=x*exp(1i*n'*w);
end
```

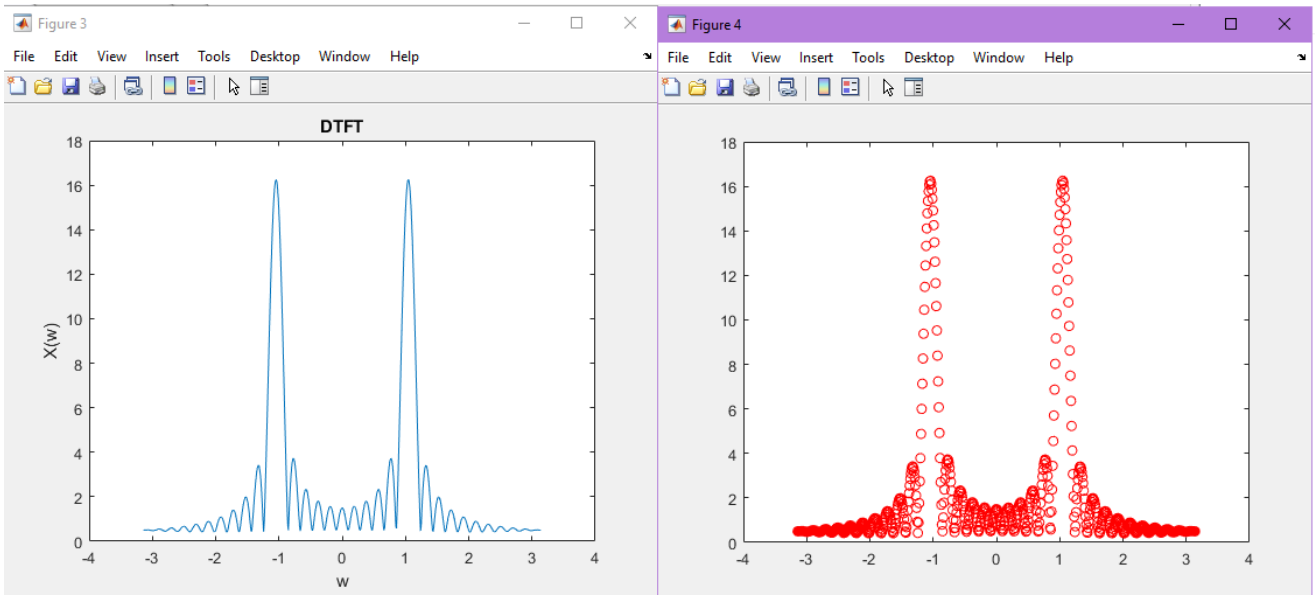
Implementation

```
% Muhammad Junaid Ali
% Assignment 2 & 3
% DSP
% Task 1
n=0:31;          % n goes from 0 to 31
w0=pi/3;         % freq of cos
x=cos(w0*n);     % cosine function
stem(n,x);       % plotting cosine
xlabel('n'); ylabel('x[n]'); title ('Length-32 Cosine');
w=-pi:0.01:pi;  % [-pi,pi]
X=freqz(x,1,w); % taking the DTFT
Y=X_tilde_mag(x,n,w); % applying the X_tilde function
figure; plot(w,abs(X)); hold on; % plotting the DTFT
xlabel('w'); ylabel('X(w)'); title ('DTFT');
plot(w,abs(Y),'ro');
```

Screenshot



Question 1
Answer
Seen side by side, we know that it is the sampled version of the DTFT. N – the number of samples can be increased such that they both become equal. $N \rightarrow \infty$. By taking N as infinite or by linearly interpolating we can make X_tilde more closely resemble with X.



Part 2: The DTFT of audio

Next, we will compute the DTFT of audio and verify that your previous analyses of the filters actually correspond to what you hear.

First, create wave files (.WAV) of two different sounds which you want to evaluate. Choose one sound that contains mainly highs and vocals. Choose the other sound to be relatively broadband, meaning that it should contain a good mix of bass, treble, and midrange. Your best bet would be to rip these from a CD, but if you don't have any CDs, MP3s will suffice.

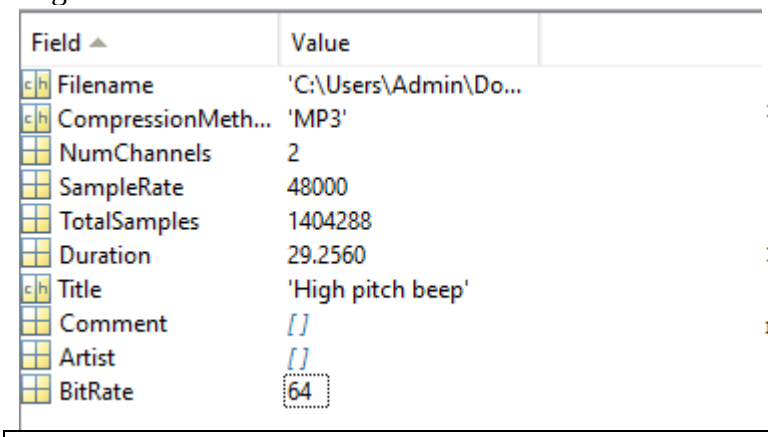
To read one of your wave files into MATLAB, use code similar to the following:

```
[x, fs, nbits] = wavread('c:/my_sound.wav');
x = 0.5 * (x(:, 1) + x(:, 2));
```

where x will contain the audio samples, fs will hold the sampling frequency in Hz, and nbits will hold the number of bits used for each sample. Make sure that you use CD-quality audio—i.e., fs should be 44100 and nbits should be at least 16. Note that the second line in the above code simply averages the two stereo channels to create a mono signal; remove this line if your wave file is already mono.

Task 2

High



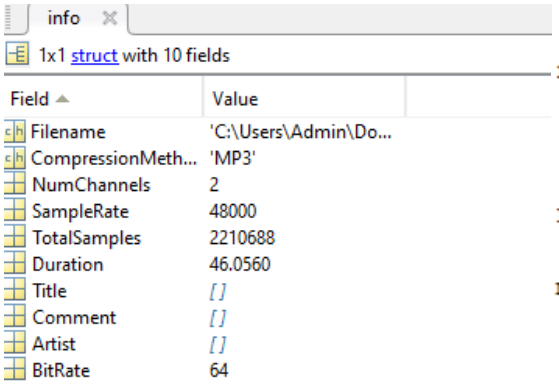
Field	Value
Filename	'C:\Users\Admin\Do...
CompressionMeth...	'MP3'
NumChannels	2
SampleRate	48000
TotalSamples	1404288
Duration	29.2560
Title	'High pitch beep'
Comment	[]
Artist	[]
BitRate	64

```
fs =

        48000

>> dsp23

nbits =

     64
```

```
[x,fs]=audioread('high.mp3');
x=0.5*(x(:,1)+x(:,2));
info=audioinfo('high.mp3');
nbits=info.BitRate
```

Low

```
[x,fs]=audioread('low.mp3');
x=0.5*(x(:,1)+x(:,2));
info=audioinfo('low.mp3');
nbits=info.BitRate
```




```
fs =

        48000

>> dsp23

nbits =

     64
```

Coding: Now, write the code for a function that will analyze and play the song. Call this function **analyze_audio()** and save it as **"analyze_audio.m"**. Your function should have the following signature:

```
function result = analyze_audio(x, fs)

% TODO: Insert your function code here
```

Your implementation of this function should perform the following tasks (in the following order):

1. Compute the DTFT of the first 10 seconds of the sound over the frequency range 0-16 kHz. (You will have to translate 10 seconds into an appropriate number of samples. You will also have to translate the CT frequency range 0-16 kHz into a proper range of DT frequencies ω.)

2. Plot the resulting magnitude spectrum. This should be a plot of CT frequency in Hz vs. magnitude in dB. Use semilogx() to make the frequency axis have a logarithmic spacing.

3. Play the sound. See the documentation for the wavplay() function or use the msound library's msound() function.

Code Implementation

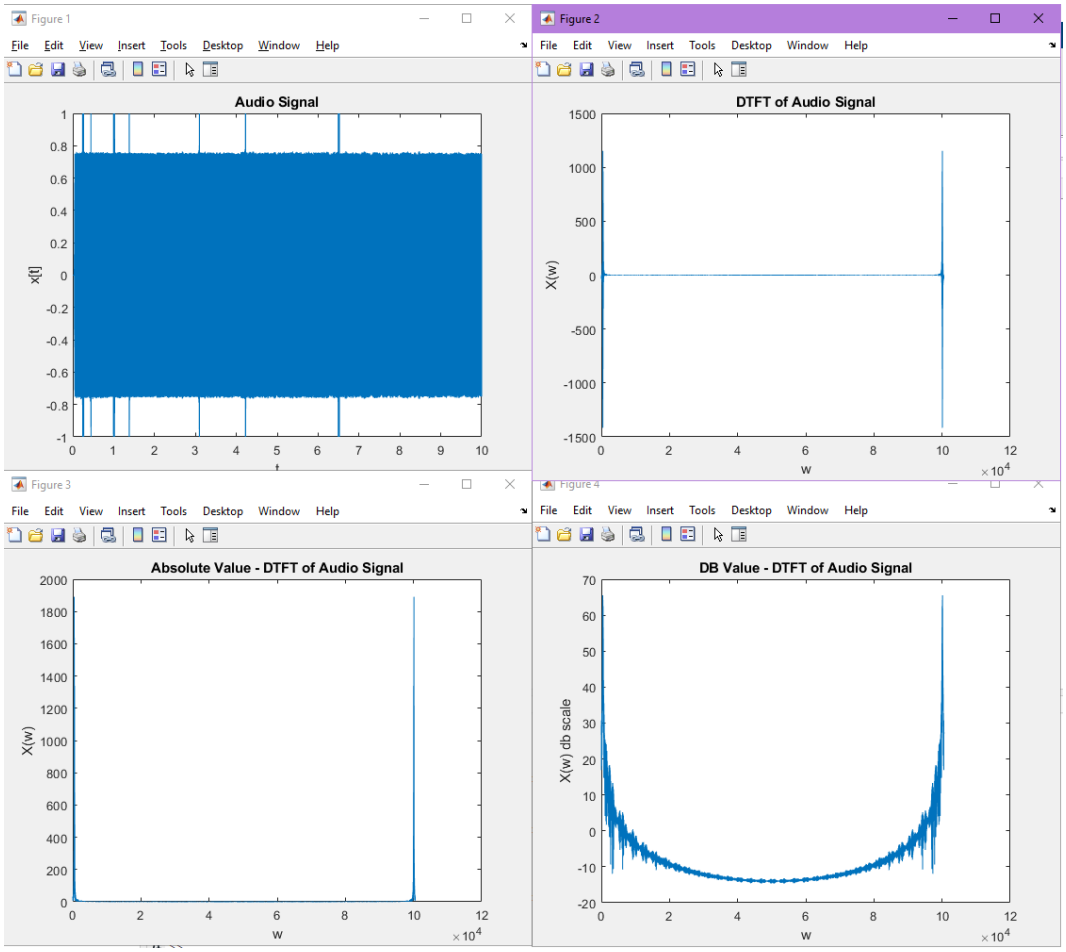
```
% Assignment 2 & 3
% DSP
% Task 2
[x,fs]=audioread('high.mp3');  % fs = 48000
info=audioinfo('high.mp3');
nbits=info.BitRate;   % bit rate = 64
```

```
analyze_audio(x,fs); % high pitch audio analysis

[x,fs]=audioread('low.mp3');  % fs = 48000
info=audioinfo('low.mp3');
nbits=info.BitRate;   % bit rate = 64
analyze_audio(x,fs); % low pitch audio analysis
```

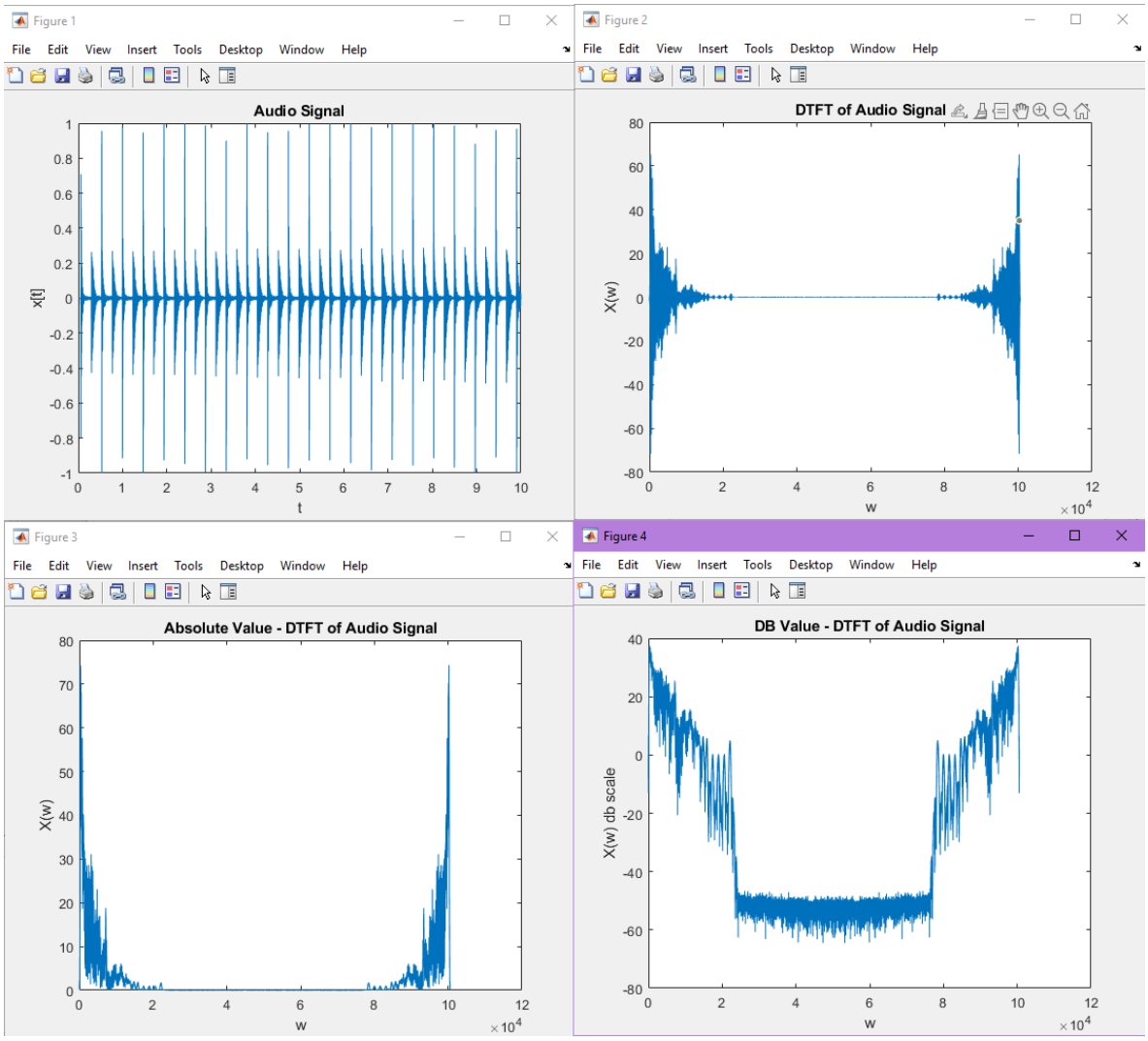
Function

```
function [] = analyze_audio(x,fs)
[m,n]=size(x);
fs=48000;  % sampling frequency
ts=1/fs;   % sampling time
t=ts*(0:m-1); % total time
x=0.5*(x(:,1)+x(:,2));
x=x(t<10); % x contains 480000 samples % x for the first 10 seconds
t=0:ts:10; t=t(1:480000); % 480001 samples to 480000 samples % t f
plot(t,x);
xlabel('t'); ylabel('x[t]'); title ('Audio Signal');
f=1:16000; % frequency ranging from 0 to 16kHz
w=2*pi*f;  % converting to DT frequency
%X=freqz(x,1,w);
X=fft(x,16000); % taking the Fourier Transform
figure;
plot(w,X);

xlabel('w'); ylabel('X(w)'); title ('DTFT of Audio Signal');
figure;
plot(w,abs(X));
xlabel('w'); ylabel('X(w)'); title ('Absolute Value - DTFT of Audio Signal');
figure;
plot(w,mag2db(abs(X)));
xlabel('w'); ylabel('X(w) db scale'); title ('DB Value - DTFT of Audio
Signal');
sound(x,fs);
end
```

Screenshot
Audio 1



Audio 2



Testing: After you've completed the code for the analyze_audio() function, run the function on the following signals:

a) Four pure tones (sine waves) at a frequency of 100, 1000, 5000, and 10000 Hz. Make each tone 10 seconds long. You should create these four tones in MATLAB, and be sure to hand in your code for this step.

b) Your two sounds that you converted to wave files.

For each signal, save the spectrum plots. Comment on how well the plots correspond to what you actually hear. Be

sure to provide very detailed comments—one-liners are not sufficient.

For Pure Tones
Code
Implementation

```
% Assignment 2 & 3
% DSP
% Task 2

fs = 48000;
t=0:1/fs:10;
fs1=100; fs2=1000; fs3=5000; fs4=10000;
x1=cos(2*pi*fs1*t);
analyze_audio(x1,fs)  % Task 2
% Uncomment which ever audio you want to analyse.
% x2=cos(2*pi*fs2*t);
% analyze_audio(x2,fs)
% x3=cos(2*pi*fs3*t);
% analyze_audio(x3,fs)
% x4=cos(2*pi*fs4*t);
% analyze_audio(x4,fs)
```

Function (Slightly Optimized, Hence a little different)

```
function [] = analyze_audio(x,fs)
ts=1/fs;
t=0:ts:10;
plot(t,x);
xlabel('t'); ylabel('x[t]'); title ('Audio Signal');
f=1:16000; % frequency ranging from 0 to 16kHz
w=2*pi*f;  % converting to DT frequency


X=fft(x,16000); % taking the Fourier Transform
figure;
plot(w,X);
xlabel('w'); ylabel('X(w)'); title ('DTFT of Audio Signal');
figure;
plot(w,abs(X));
xlabel('w'); ylabel('X(w)'); title ('Absolute Value - DTFT of Audio Signal');
figure;
plot(w,mag2db(abs(X)));
xlabel('w'); ylabel('X(w) db scale'); title ('DB Value - DTFT of Audio
Signal');
sound(x,fs);
end
```

Screenshots
In this case, there is a peak in the frequency domain in correspondence to the frequency f = 100 Hz (The fourier peaks at 2*pi*100)

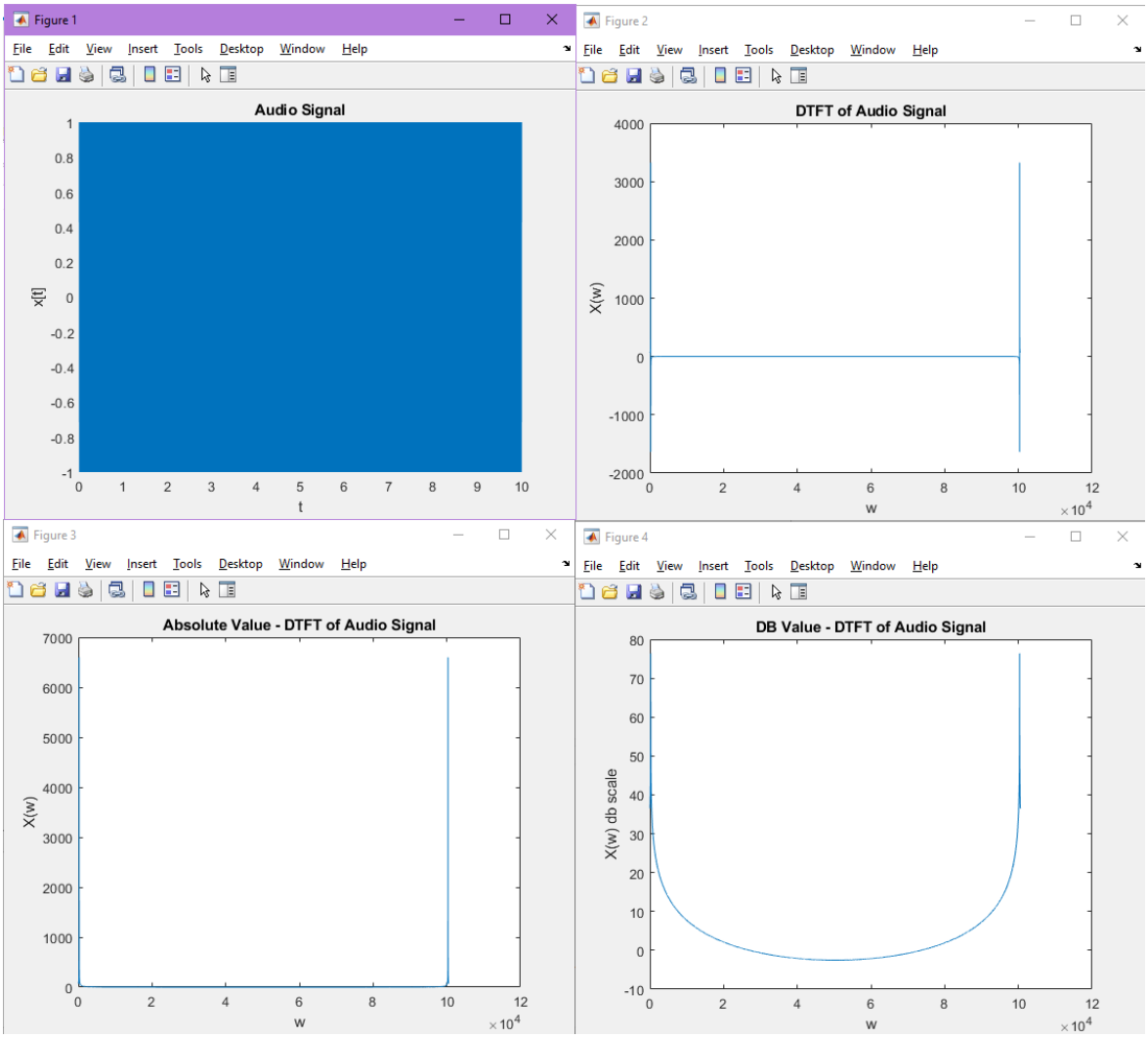
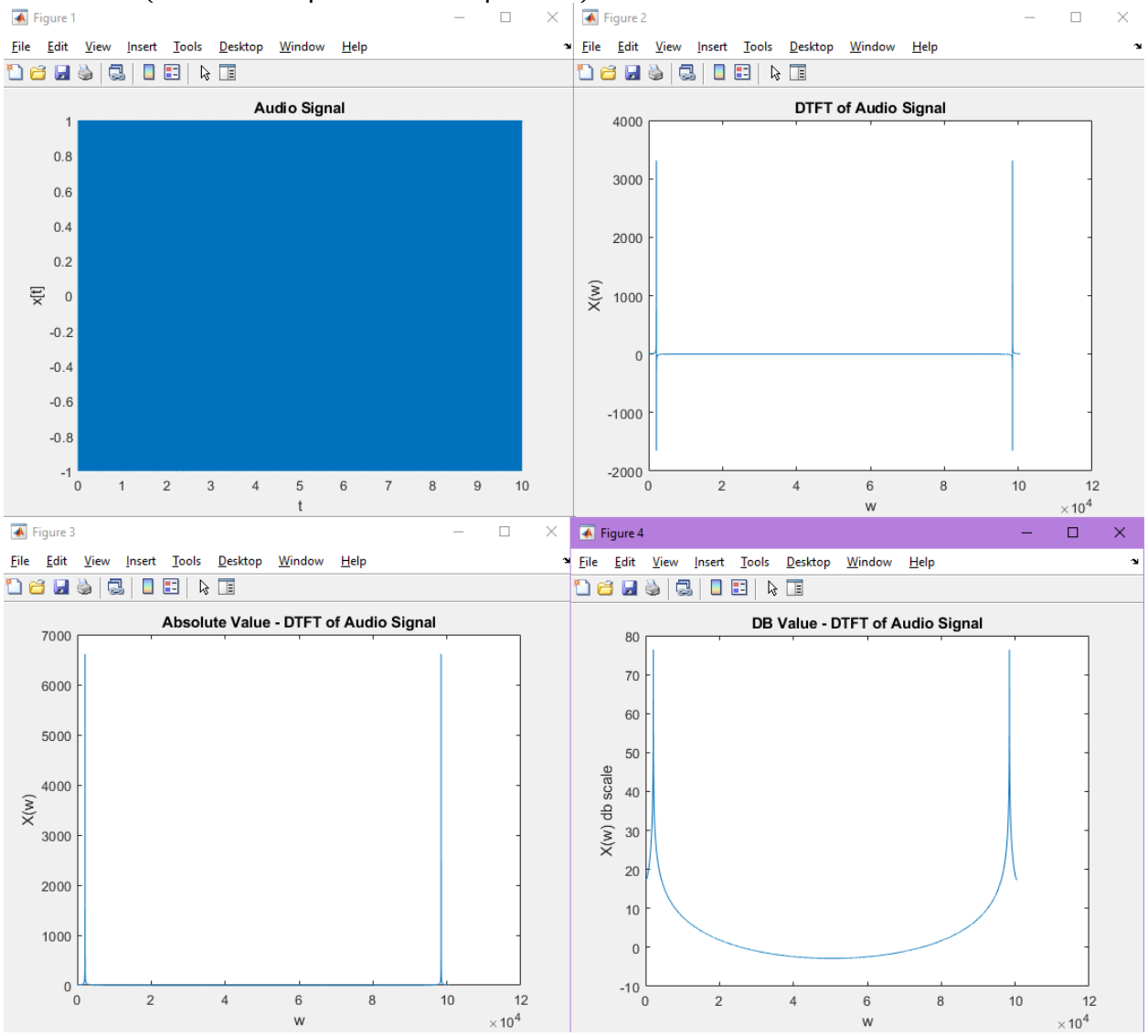
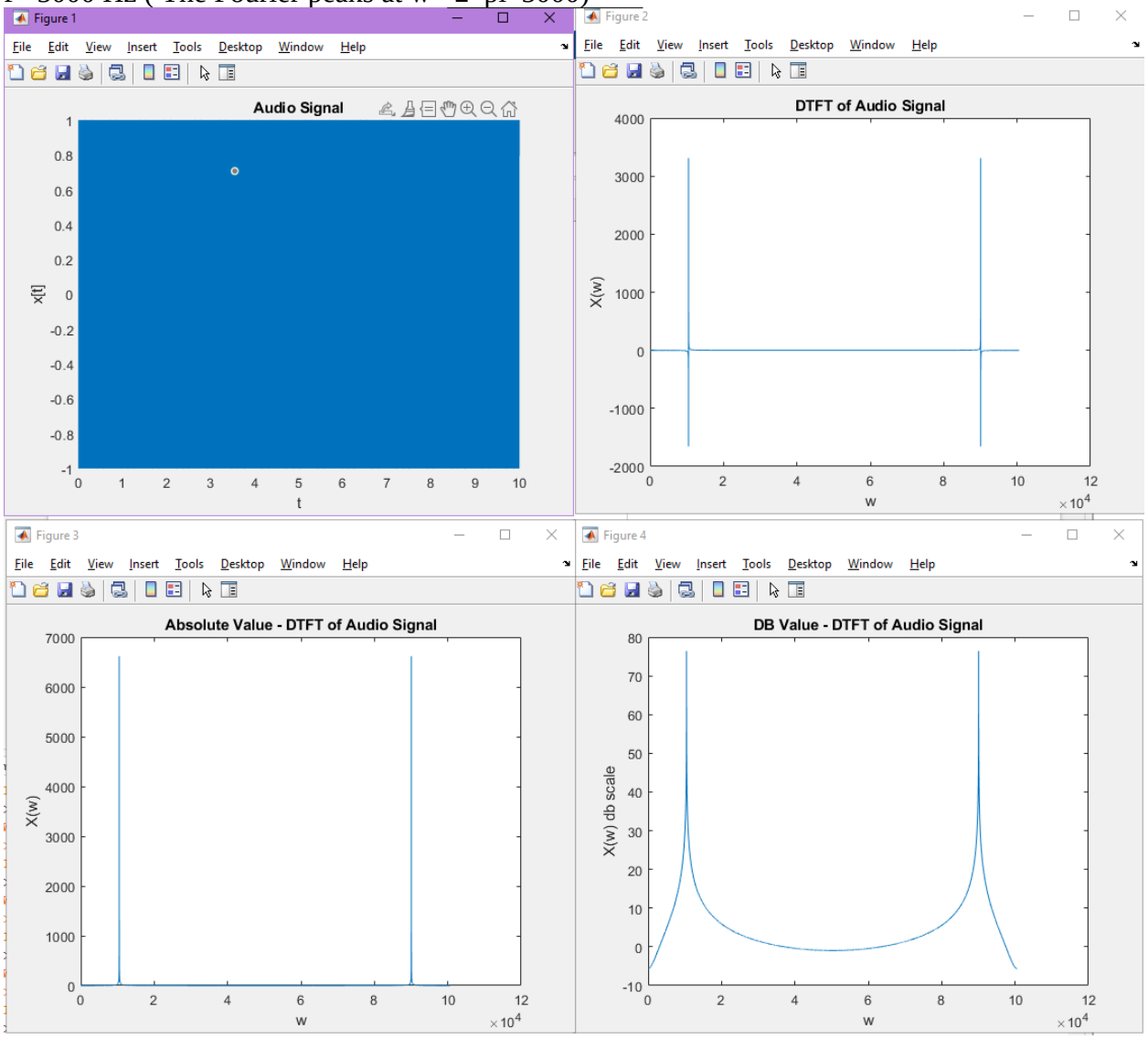


Figure 1
File Edit View Insert Tools Desktop Window Help
Audio Signal
x[t]
t
Figure 2
File Edit View Insert Tools Desktop Window Help
DTFT of Audio Signal
X(w)
w
×10^4
Figure 3
File Edit View Insert Tools Desktop Window Help
Absolute Value - DTFT of Audio Signal
X(w)
w
×10^4
Figure 4
File Edit View Insert Tools Desktop Window Help
DB Value - DTFT of Audio Signal
X(w) db scale
w
×10^4

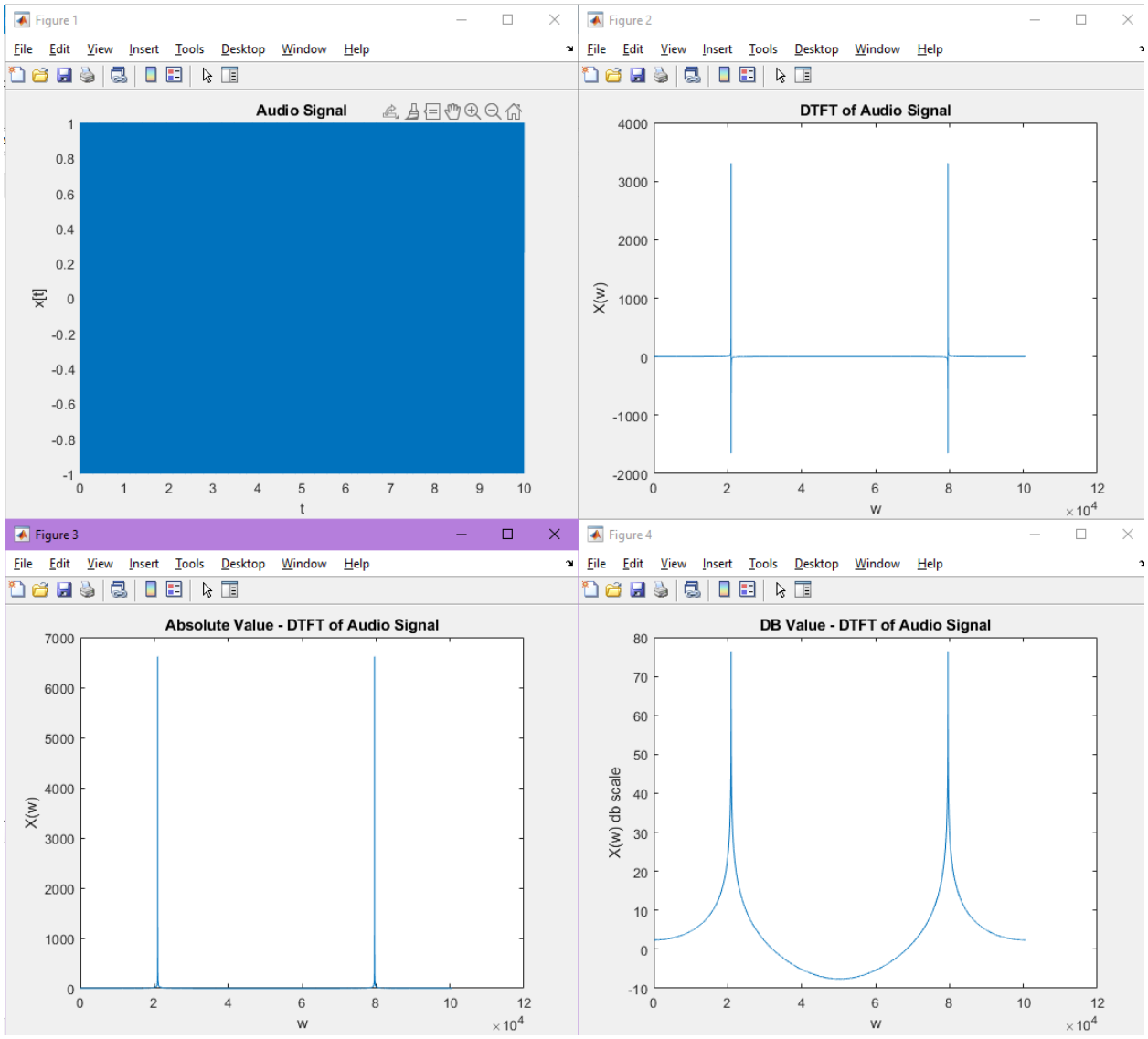
f= 1000 Hz (The Fourier peaks at w= 2*pi*1000)



f= 5000 Hz (The Fourier peaks at w= 2*pi*5000)



f= 10000 Hz (The Fourier peaks at w= 2*pi*10000)



Part 3: The DTFT of audio chunks (real-time analysis)

Finally, it's time to perform some real-time audio analysis. In this part of the project, you will write a MATLAB function—called analyze_audio_chunks()—that analyzes and plays an audio signal in a block-by-block manner and displays a constantly-updated bar-chart of the magnitude spectrum.

Coding: Your analyze_audio_chunks() function should have the following signature:

Note: Task 3 at the end.

```
function result = analyze_audio_chunks(x, fs, use_wnd)

blk_size = 2.5*1024;
N = length(x);

% TODO: Insert your function code here
```

where x is the (mono) audio signal, fs is the sampling frequency (in Hz), and use_wnd is a Boolean parameter that specifies whether or not an alternative windowing function will be used (more on this below). We will be using 2560-length chunks (blocks), which is stored in the variable blk_size. The variable N will hold the length of the audio signal.

Your implementation of this function should perform the following tasks:

1. Ensure that the audio signal's length is an integer multiple of the block size. If not, pad the audio signal with enough zeros so that its length is an integer multiple of the block size. (The MATLAB mod() function will prove useful.)

2. Call the msound() function with the 'openwrite' parameter to gain playback control of the soundcard. (Type msound('help') to get documentation on what additional parameters to pass in with 'openwrite'.)

3. Loop over the length of the audio signal in increments of the block size. Within your loop:

   a. Extract the appropriate block from the signal.
   b. If the use_wnd parameter is true (non-zero), then multiply your signal block by a length-2560 Gaussian window created via the MATLAB gausswin() function with default standard deviation (i.e., omit the second parameter to gausswin()).
   c. Compute and plot the magnitude spectrum of the block over the range 50-16000 Hz. (See below for more info on this step.)
   d. Play the block by using the msound() function with the 'putsamples' parameter.

4. Before your function exits, be sure to call msound('close') to close audio playback. Use a try/catch block to ensure that msound('close') is called in the event of an error. If you have to terminate your code by pressing Ctrl+C at the command line, be sure to also call msound('close') from the command line to ensure the audio device is closed.

Some more specifics on Step 3: In Step 3, your task is to analyze and play each 2560-length block of the audio. Playing the audio is simple enough since msound handles all of the work for you. Analyzing the signal in terms of its magnitude spectrum is the most time-consuming part—both for you and for MATLAB. To speed up processing, compute the DTFT only at the following 64 frequencies:

f = logspace(log10(50), log10(16000), 64); (Code Equation 1)

Specifically, given the above variable f, which represents the desired length-64 CT frequency range in Hz, compute a corresponding range of DT frequencies ω, and then pass this length-64 DT frequency range to freqz(). If you use a finer frequency spacing, freqz() might consume too many CPU cycles, thus resulting in choppy audio playback.

In addition, your real-time plot of the magnitude spectrum will be a bar chart with 14 bars, each bar representing the average magnitude in a particular frequency range. An example chart is shown in Figure 1. The corresponding frequency range for each bar should be as follows:

- Bar 1: 50 - 75 Hz
- Bar 2: 75 - 100 Hz
- Bar 3: 100 - 150 Hz
- Bar 4: 150 - 300 Hz
- Bar 5: 300 - 500 Hz
- Bar 6: 500 – 800 Hz
- Bar 7: 800-1200 Hz
- Bar 8: 1200-2000 Hz
- Bar 9: 2000-3000 Hz
- Bar 10: 3000-5000 Hz
- Bar 11: 5000-7000 Hz
- Bar 12: 7000-9000 Hz
- Bar 13: 10-12 kHz
- Bar 14: 12-16 kHz

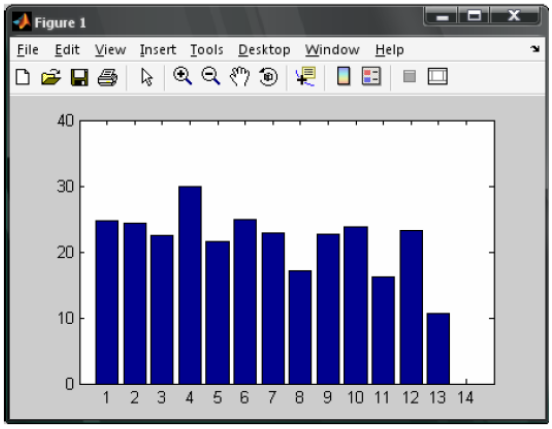


Figure 1: Snapshot of the bar chart that shows a real-time display of the audio's spectrum. The y-axis denotes magnitude in dB. The x-axis simply denotes bar index with greater indices corresponding to greater frequency bands. Note that in this snapshot, bar 14 (the 12-16 kHz range) just happens to be zero, but this will not be the case in general.

Use the following code to display the bar chart within your loop:

```
% make Figure 1 the current figure
figure(1);
% plot the bar chart
bar(X_avg_mag_in_dB);
% limit the range of the y-axis to within 0 and 40 (dB)
ylim('manual'); ylim([0, 40]);
```

In this code, `X_avg_mag_in_dB` is a length-14 vector that holds the average magnitude within each bar's corresponding frequency range. It's up to you to fill in `X_avg_mag_in_dB`. Specifically, after you use `freqz()` to compute the DTFT at the 64 frequencies specified in Code Equation 1, you must average the appropriate magnitudes to boil the result down to the 14 average magnitudes that will be stored in `X_avg_mag_in_dB` and ultimately plotted.

Testing: After you've completed the code for the `analyze_audio_chunks()` function, run the function on the following signals with the `use_wnd` parameter set to false (zero):

a) Fourteen pure tones (sine waves), each at a frequency within the center of each frequency range given in the above list. You should create these tones in MATLAB, and be sure to hand in your code for this step.
b) Your two sounds that you converted to wave files.

Next, repeat the above steps with the `use_wnd` parameter set to true (non-zero). For each signal, save snapshots of the spectrum plots. Comment on how well the plots correspond to what you actually hear, and comment on how the plots change when `use_wnd` is true (the changes will be frequency-dependent). Again, be sure to provide very detailed comments.

For those whose work reflects their sole effort and are willing to pledge by writing and signing the following statement

"No aid given, no aid received"

What to hand in

This lab requires that you hand in the following items:

1. All of your MATLAB code. Ensure that your code is well commented and works properly. Code that does not run will be given zero credit. *It is very important that your code is your own original work and not just a copy of your classmate's code with different variable names.*
2. All of the requested plots.
3. Answers to all of the questions, requested descriptions, and derivations in the lab. (Be sure to show your work for derivations, and be sure to re-read the lab to make sure you haven't missed any questions.)
4. Answers to the following questions:
   a. *From Part 1*: One way to make $\widetilde{X}(e^{j\omega})$ more closely resemble $X(e^{j\omega})$ is to use a longer window (e.g., a length-2048 rectangle). How else might we make $\widetilde{X}(e^{j\omega})$ more closely resemble $X(e^{j\omega})$?
   b. *From Part 3*: Why did the plots change when using the Gaussian window? Why were these changes different for low vs. high frequencies?
   c. *From Part 3*: Given a sampling rate of `fs = 44100` Hz, and a block length of 2560 samples, how often is the bar chart updating in frames/second? What block length would we need to use to achieve 30 frames/sec?

There is no need to prepare a formal lab report; however, I do ask that you collect your graphs, code, and solutions in a legible, easy-to-follow format.

Task 3

Code
Implementation

```
% Assignment 2 & 3
% DSP
% Task 2
[x,fs]=audioread('music.mp3');
info = audioinfo('music.mp3');

use_wnd=0; % use window variable 1 if need be.
analyze_audio_chunks(x,fs,use_wnd);
```

Function (analysis)

```
function [] = analyze_audio_chunks(x,fs,use_wnd)
blk_size=2560; % Bulk Size is 2560
[m,n]=size(x);
if(n==2)
x=0.5*(x(:,1)+x(:,2));    % if channels are 2, we need to convert it to one.
x=[x;zeros(blk_size-mod(length(x),blk_size),1)];
end
```

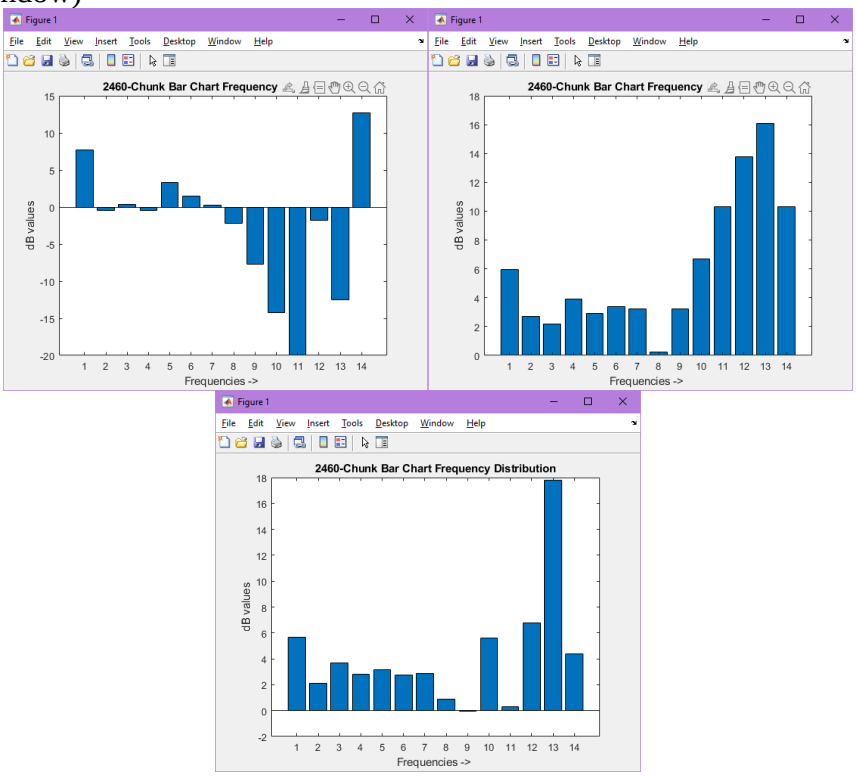
```
if(m==1)   % in pure tones, m==1 i.e. row vector and we have to pad zeros in
the columns
    x=[x,zeros(1,blk_size-mod(length(x),blk_size))];
end
% auto zero padding to the next nearest multiply of 2560.
N=length(x);
f = logspace(log10(50), log10(16000), 64);  % Log Spaced frequency values from
50 to 16000 Hz (64 Entries)
w = 2*pi*f; % 64 values DT frequency
for i=1:blk_size:(N-2560)     % for loop over the block size 2460.
sound(x(i:i+2560),fs);
X=freqz(x(i:i+2560),1,w);           % finding the fourier transform for the
chunk.
if(use_wnd==1)        % if window is used, multiply it with window.
    X=X.*gausswin(2560);
end
X_avg_mag_in_db=bar_chunks(X); % this gives us the 14 bars. another function
bar(X_avg_mag_in_db); % plotting the diagram
xlabel('Frequencies -> '); ylabel('dB values'); title('2460-Chunk Bar Chart
Frequency Distribution')
pause(0.5);   % real time plot (so that change can be observed for loop
iteration)
end
end
```

Function (chunks)

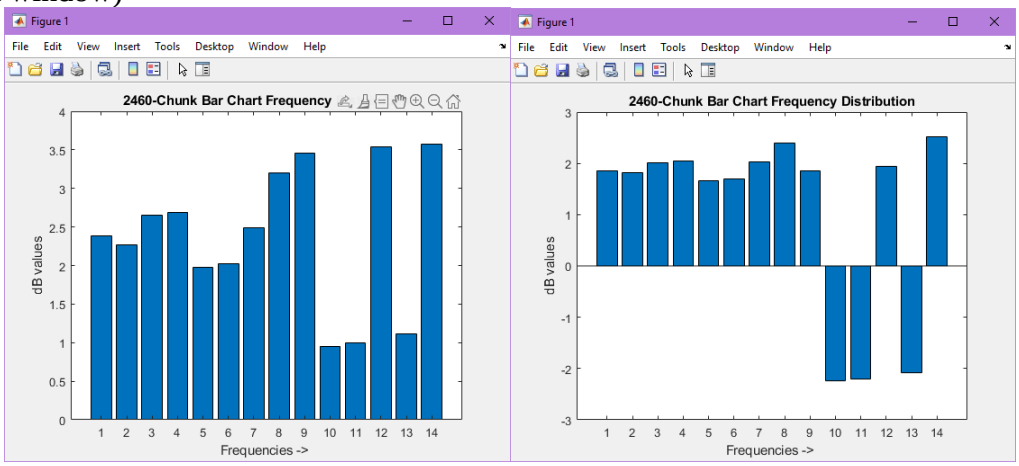
```
function y = bar_chunks(X)
y=zeros(14,1);
% Frequency Distribution among Bars
% 50 to 80 Hz - 1 to 6th Frequency Element
% 80 to 150 Hz - 7 to 13th Frequency Element
% 150 to 250 Hz - 14 to 18th Frequency Element
% 260 to 400 Hz - 19 to 23th Frequency Element
% 400 to 1000 Hz - 24 to 33th Frequency Element
% 1000 to 2000 Hz - 34 to 43th Frequency Element
% 2000 to 4500 Hz - 44 to 50th Frequency Element
% 4500 to 6500  Hz -51 to 54th Frequency Element
% 6500 to 7700 Hz - 55 to 56th Frequency Element
% 7700 to 8500 Hz - 57th Frequency Element
% 8500 to 10000 Hz - 58th Frequency Element
% 10000 to 11100 Hz - 59 to 60th Frequency Element
% 11100 to 12200 Hz - 61th Frequency Element
% 12200 to 16000 Hz - 62 to 64th Frequency Element
% Taking out the average and putting it to the corresponding value
y(1)=mag2db(sum(abs(X(1:6))))/6;
y(2)=mag2db(sum(abs(X(7:13))))/7;
y(3)=mag2db(sum(abs(X(14:18))))/5;
y(4)=mag2db(sum(abs(X(19:23))))/5;
y(5)=mag2db(sum(abs(X(24:33))))/10;
y(6)=mag2db(sum(abs(X(34:43))))/10;
y(7)=mag2db(sum(abs(X(44:50))))/7;
y(8)=mag2db(sum(abs(X(51:54))))/4;
y(9)=mag2db(sum(abs(X(55:56))))/2;
```

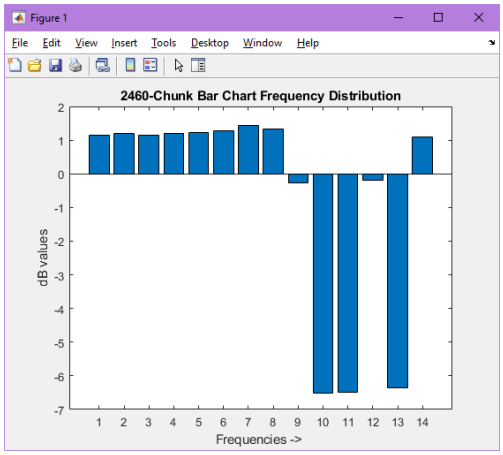
```
y(10)=mag2db(sum(abs(X(57))));
y(11)=mag2db(sum(abs(X(58))));
y(12)=mag2db(sum(abs(X(59:60))))/2;
y(13)=mag2db(sum(abs(X(61))));
y(14)=mag2db(sum(abs(X(62:64))))/3;
end
```

Screenshots
(without window)



(with window)

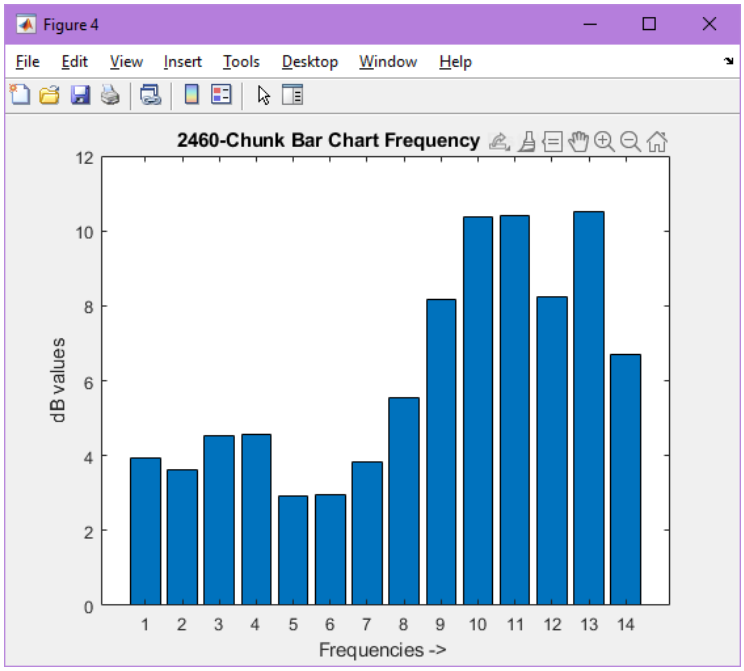




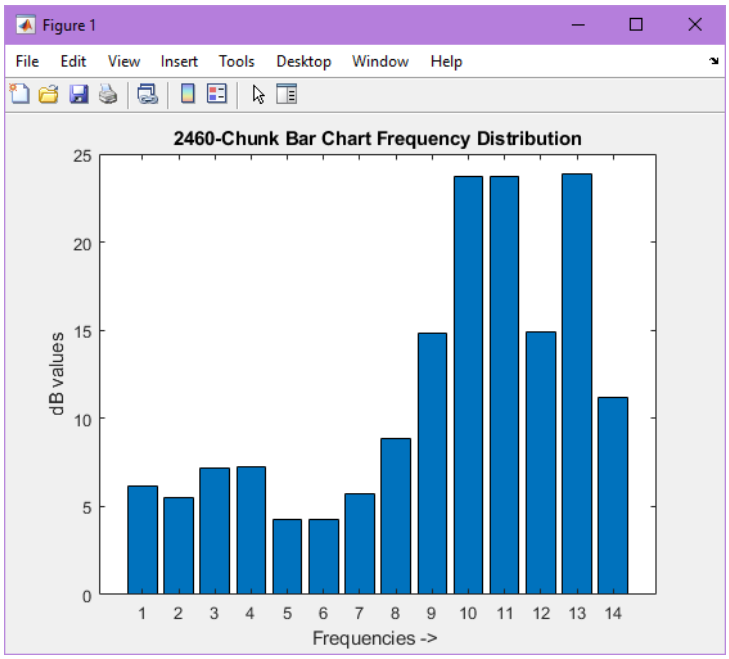
(tones)
Code
Implementation

```
% Assignment 2 & 3
% DSP
% Task 3
clear all


fs=44100; % sampling frequency
t=0:1/fs:5; % Total Time for pure tone.

x1=sin(2*pi*66*t);
x2=sin(2*pi*125*t);
x3=sin(2*pi*200*t);
x4=100*sin(2*pi*350*t);
x5=100*sin(2*pi*650*t);
x6=100*sin(2*pi*1500*t);
x7=100*sin(2*pi*2500*t);
x8=100*sin(2*pi*4000*t);
x9=100*sin(2*pi*6000*t);
x10=100*sin(2*pi*8000*t);
x11=100*sin(2*pi*9000*t);
x12=100*sin(2*pi*10500*t);
x13=100*sin(2*pi*11500*t);
x14=100*sin(2*pi*16000*t);
use_wnd=1;
analyze_audio_chunks(x8fs,use_wnd);
```

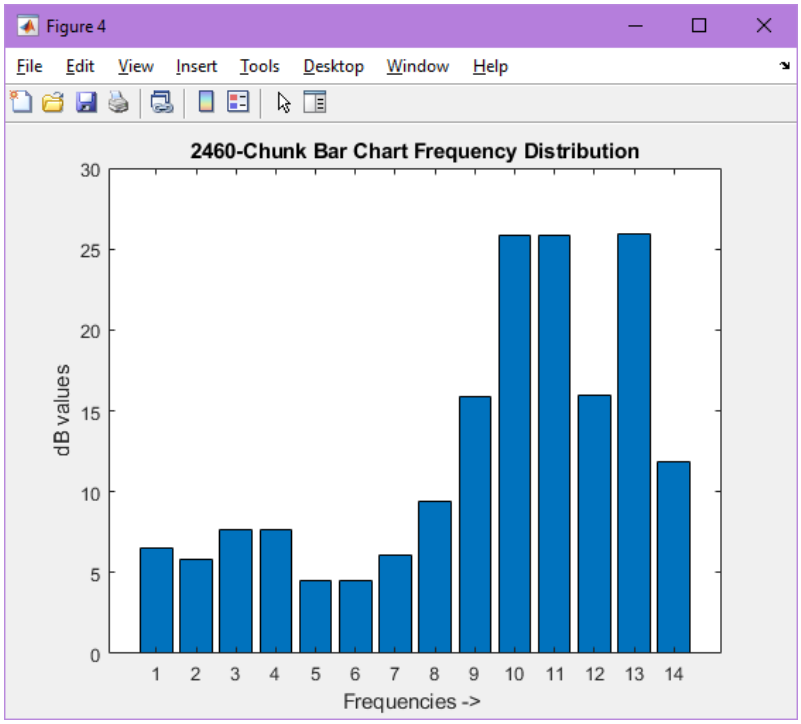
Low Frequency



MidRange



High Frequency



It is observed that as frequency of the tone is increased, the frequency components increase correspondingly.

Answers

b) *The Gaussian Window saturates the low and high freq components to some extent to the middle. The values were much closer to the average dB values they were getting. Without the windowing, this wasn't the case.*
c) *We're already getting 30fps but if frames/block was a problem, we could double the length of the chunk and fix our problem.*