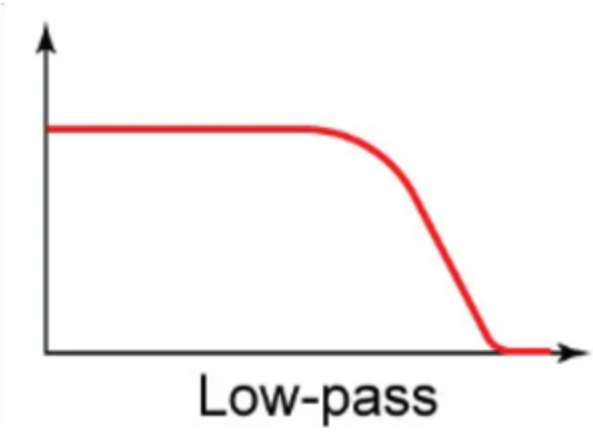
EENG-341 Signals and Systems Term Project
Term Project
Members: Christian Calvo (1297165), and Wayne Kells (1314568)
Due Date: 12/1/24

Git Repo: https://github.com/Tubliy/Proj1-Signals-and-Systems

Objective:

The objective of this project is to create a music track that's 5-10 seconds long with an IDE of our choosing. Then analyzing their time and frequency domain plot as well as their spectrograms. **EXTRA CREDIT**(Eventually, adding a noise disturbance and seeing how it affects the original plots. Finally, using a low pass filter to eliminate the disturbance and to revert the track to normal.)



Procedure:

LavendarTown.m (Music Track code)

The program Matlab was the IDE used to program the music track. A sampling frequency of 8000 hz was chosen because it's standard practice. With the general equation of a sine wave $y(t) = sin(2\pi \times freqofNote \times timeofNote)$. Breaking down this equation, $freqofNote$ was found with a note to frequency chart. The chart matched notes to their respective frequencies. $timeofNote$ is a time vector, starting at t = **0s** and progressing in steps of $Step\ Size = \frac{1}{FS} = \frac{1}{8000hz} = 0.000125s$ and ending at t = **whatever the note's duration is**. This then creates sine waves that oscillate at the frequencies of the given note. Arrays are created to section off specific parts of the track, including a melody, short notes, and long notes. However, these were supposed to be combined into one sine wave but it wasn't possible due to the silence and the added amplitudes (see below in analysis **p. 8** for more info). Concatenating the arrays is a way of controlling the order of what needs to be played, hence $song = [melody, shortnotes, longnotes]$;. Avoiding clipping is very important to analyze the time and frequency domain representations. The $audioWrite$ function has three parameters $(string = "Name\ of\ file.wav", signal\ used, FS\ (sampling\ frequency)$. Keeping it simple by naming the file "song.wav" , the combination signal "song", and defining FS as 8000hz.

audioMerge.m (Merging with noise disturbance)

Storing song.wav into a variable song which holds the audio data, and a Fs_s which contains the sampling size of the file. For example, the song variable will contain a matrix or a vector of data. This all depends on if the track is mono or stereo. The sampling rate of song.wav is shown below.

```
disp('Sampling rate of Fs_s');
disp(Fs_s);
```

Figure 1A: Display code.

```
Sampling rate of Fs_s
        8000
```

Figure 1B: Sampling rate of song.wav

This is due to the *audioWrite* function from the previous section. However, the sampling rate of audionoise.wav is significantly different.

```
disp('Sampling rate of Fs_n')
disp(Fs_n);
```

Figure 2A: Display code.

```
Sampling rate of Fs_n
       44100
```

Figure 2B: Sampling rate of audionoise.wav

A stereo signal has two columns left and right channels, hence the next part of the code is to check if the noise is not mono. If it's stereo it takes the mean of both of the columns leaving it with just one column making it a mono signal. The following code matches the sampling rates.

```
            if Fs_n ~= Fs_s
        try
            % Use resample if Signal Processing Toolbox is available
            noise = resample(noise, Fs_s, Fs_n);
        catch
            % Alternative resampling using interp1
            disp("Signal Processing Toolbox not installed");
            t_original = (0:length(noise)-1) / Fs_n; % Original time vector
            t_target = (0:1/Fs_s:(length(noise)-1)/Fs_n)'; % Target time vector
            noise = interp1(t_original, noise, t_target, 'linear'); % Linear
interpolation
```

```
        end
    end
```

Figure 3A: Code for matching sampling rates.

Using the not equal to operator "~=" in matlab, it uses the resample function from the Signal Processing Toolbox. This function has three parameters $resample(musicData,\ Sampling\ Frequency\ you\ want,\ Previous\ Sampling\ Frequency)$. This is then stored back into the noise variable. A try/catch was used in case the user doesn't have the toolbox installed, hence making it more user friendly. The alternative method uses linear interpolation which stores the original time vector, and a new target time vector. What linear interpolation does is that it's able to estimate unknown values that lie between known data points. Next, defining the starting time of the noise which is 10 seconds into the music track. The next block of code ensures that the noise signal fits within the music signal, if it doesn't then it pads it with zeros. After that, it checks if the song is stereo this time and makes sure that noise matches it. Below the noise is being added to the music track.

```
% Add noise to the music
noisySong = song;
noisySong(FS_Index:FS_Index + length(noise) - 1, :) = ...
   noisySong(FS_Index:FS_Index + length(noise) - 1, :) + noise;
```

Figure 3B: Adding the noise to the signals.

Storing the original audio data in a new variable named $noisySong$. FS_Index represents the starting index of noisysong where the noise will be added. The ":FS_Index + length(noise) - 1)" is the range in noisysong that will be affected by the noise. Finally "+ noise", just adds the provided noise to our original track. $audioWrite$ is used to save this file as .wav for playback and easy access.

audioCancel.m (Noise cancellation of disturbance)

A low pass filter is made to attenuate the higher frequency noise, so in this case $f_n > f_c$. The $f_c = 1000hz$, in our filter anything greater than that is attenuated. The code below creates a low pass filter in matlab.

```
filter_order = 12; % Order of the filter
lpFilter = designfilt('lowpassiir', 'FilterOrder', filter_order, ...
                      'HalfPowerFrequency', filter_cutoff, ...
                      'SampleRate', Fs_noisy);
% Apply the low-pass filter to the normalized noisy audio
```

```
y_processed = filtfilt(lpFilter, y_noisy); % filtfilt applies zero-phase filtering to avoid phase distortion
```

Figure 4A: Low pass filter design.

The *designfilt* function creates a low pass filter with the given filter order, and uses the filter's cutoff as well as the sampling rate. After that it's then turned into audioClean.wav.

timeandfreqPlot.m (Plotting the time and frequency domain)

This segment was used to plot the time and frequency domain graphs, and their respective spectrograms. In the analysis, the graphs' appearance is broken down as well as their purpose in this project. The following block of code demonstrates how they were plotted in Matlab.

```
% Create a new figure
figure;
% Plot for 'song.wav'
[y, Fs] = audioread('song.wav');
duration = length(y) / Fs;
t = linspace(0, duration, length(y));
Y = fft(y);
freq_resolution = Fs / length(Y);
f = linspace(0, Fs - freq_resolution, length(Y));
% Time domain of song.wav
subplot(4, 3, 1);
plot(t, y);
title('song.wav - Time Domain');
xlabel('Time (s)');
ylabel('Amplitude');
% Frequency domain of song.wav
subplot(4, 3, 2);
plot(f, abs(Y));
title('song.wav - Frequency Domain');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
xlim([0, Fs/2]);
% Spectrogram of song.wav
subplot(4, 3, 3);
spectrogram(y, 256, [], [], Fs, 'yaxis');
title('song.wav - Spectrogram');
colorbar;
```

Figure 5A: Plotting the time, frequency, and spectrograms.

In this case, y stores the audio data and Fs is the files sampling rate. The duration is calculated by the length of y divided by the sampling rate. Time "t" starts from 0 and moves in steps of the calculated duration, finally ending at the length of y. So the time domain plot is $y\ vs\ t$.

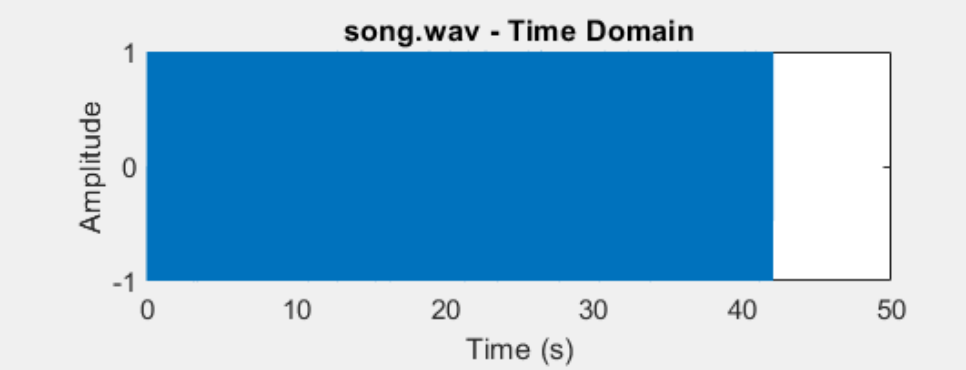


Figure 5B: Time domain plot.

The fft method is used to find the fourier transform of a function. This line of code $Y = fft(y)$; allows us to store the trans form in Y. Frequency resolution uses the same code from the "duration" variable. Frequency "f" starts at 0 and moves in steps of the calculated freq_resolution, finally ending at the length of $Y(jw)$. So the time frequency domain plot is $|Y(jw)|\ vs\ f$.

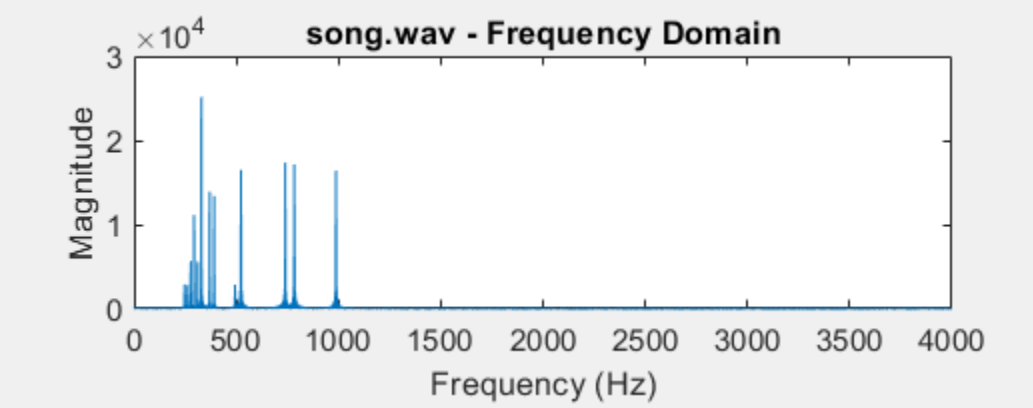


Figure 5C: Frequency domain plot.

Finally, the spectrogram plot includes x as the time and y as frequency. The color in the graph represents the amplitude or power of a signal at a specific time. Brighter or darker colors indicate higher or lower amplitudes respectively. The code for this plot doesn't have anything of importance, the 256 is the window size and the []'s are all default matlab necessities.

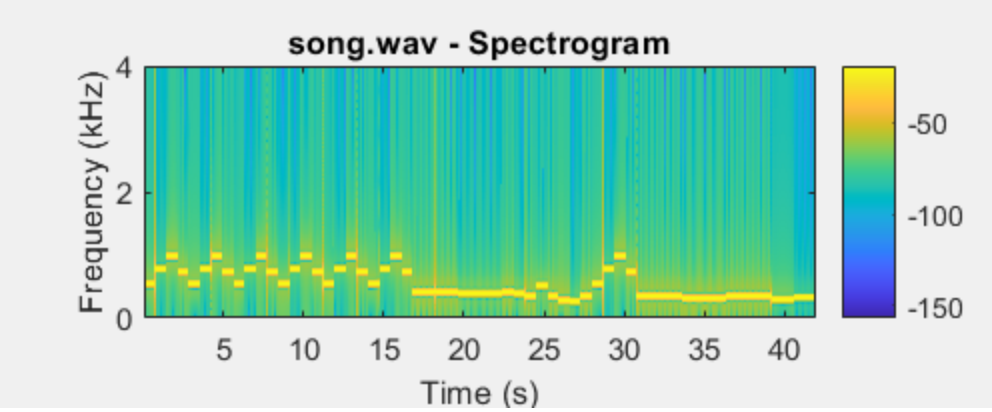


Figure 5E: Spectrogram.

Member Contributions:

Christian Calvo - Wrote the objective,procedure, audioMerge.m code, README.txt, and Appendix.

Wayne Kells - Wrote the analysis of the graphs, audioCancel.m code, and LavendarTown.M code.

Works Cited:

Musescore. “Lavender Town.” *Musescore.com*, Musescore, 24 May 2012, musescore.com/thenewmaestro/scores/50012. Accessed 30 Nov. 2024.

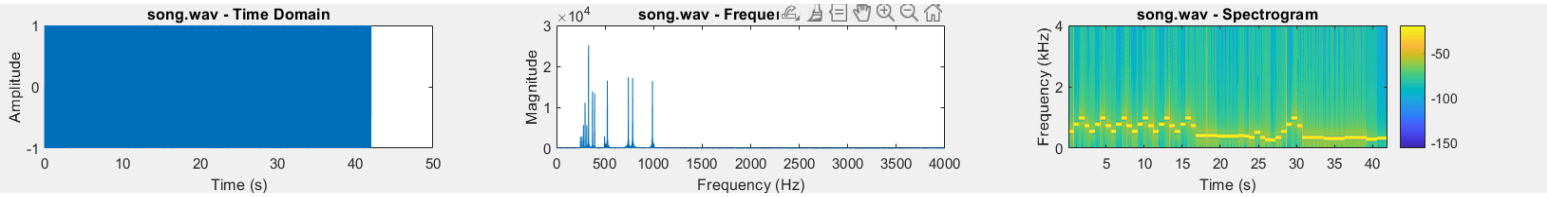
"Note Frequencies." *Muted.io*, www.muted.io/note-frequencies/. Accessed 30 Nov. 2024.

"Spectrogram." *MathWorks*, www.mathworks.com/help/signal/ref/spectrogram.html. Accessed 30 Nov. 2024.

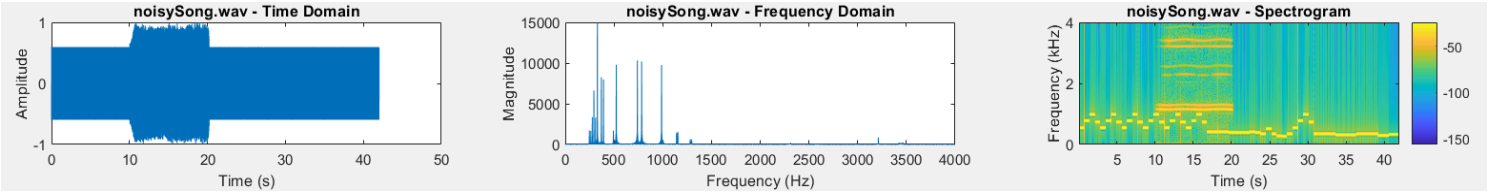
"Signal Processing Tutorial Launchpad." *MathWorks*, www.mathworks.com/academia/student_center/tutorials/signal-processing-tutorial-launchpad.html. Accessed 30 Nov. 2024.

"MATLAB Tutorial Launchpad." *MathWorks*, www.mathworks.com/academia/student_center/tutorials/mltutorial_launchpad.html?confirmation_page#. Accessed 30 Nov. 2024.

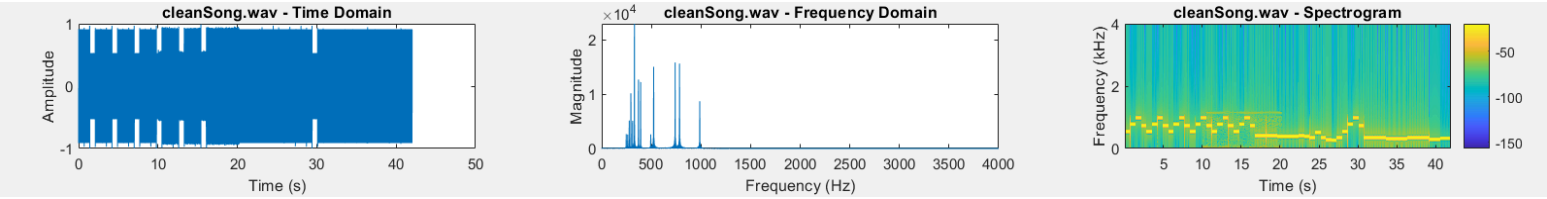
Analysis:



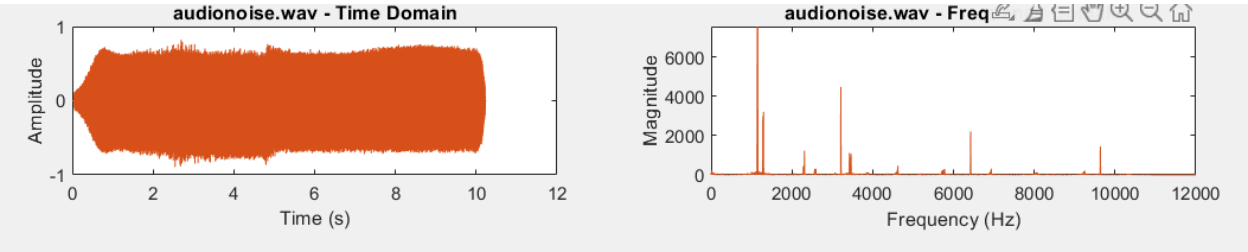
Time,Frequency, and Spectrogram of song.wav (**Figure 6A**)



Time,Frequency, and Spectrogram of noisySong.wav (**Figure 6B**)



Time,Frequency, and Spectrogram of cleanSong.wav (**Figure 6C**)



Time and Frequency domain of audionoise.wav (**Figure 6D**)

The time domain graph of figure 6A shows a wide, interrupted signal over 40 seconds. This graph demonstrates how this graph is clean and contains no problems that would lead to disruptions or other problems. Figure 6B on the other hand contains a very noticeable imperfection in between the 10 and 20 second mark of the music. This irregularity is due to the noise we added to the music. In figure 6C, we applied noise reduction to remove this irregularity. However, the result of that wasn't perfect, and some artifacts of the added noise are still present, even though the music sounds fine.

The music created contained both a harmony and melody. The difference between the two being that melodies are the portion of music in which one note is played after another, while the harmony is a group of notes. For the frequency domain, you can see that in 6A, there are very notable peaks at certain points of the laid out song. These peaks represent the notes played and how many times they were played. Notably, the 4 most played notes (C, G High B, F#) was a majority of the melody, while the lesser played notes were contained in most of the harmonics. Figure 6B contains much of the same frequency domain as 6A. The difference being now there's a very strong frequency that was played more than any other note. This frequency is the noise that was added. The peak magnitude for 6A was 3000, while 6B had a peak of 16000. This difference in magnitude shows just how strong the noise is in comparison to the music. Figure 6C managed to remove a majority of the noise, with the peak magnitude going down to 2000. However it is not permanently removed, as the highest magnitude frequency is still the noise. Since the noise is no longer at such a high magnitude, it's essentially doing nothing. The spectrograms do a better job at demonstrating the removal of the noise.

All 3 spectrograms show a clear distinction of each section of the music. With the notes at the end being the harmony and the notes at the beginning being the melody. Spectrogram 6A has no imperfections and shows the music playing as normal. Spectrogram 6B however showcases noise the same way as the time domain. There is a clear distinction at where the noise starts and ends. Spectrogram 6C shows the removal of the noise, but a little bit of residue is left behind. Near the notes there are significantly more glowing parts. Showing what was left from the filter.

Appendix:

Matlab Code 1: (LavendarTown.m)

```
clear; % Remove all variables from the workspace
% Sampling frequency
FS = 8000; % Sampling frequency in Hz
% Generate notes
a = sin(2*pi*440*(0:0.000125:0.2));
b = sin(2*pi*493.88*(0:0.000125:0.7));
c = sin(2*pi*523.24*(0:0.000125:0.7));
chalfnote = sin(2*pi*523.24*(0:0.000125:1.4));
cs = sin(2*pi*554.37*(0:0.000125:0.2));
d = sin(2*pi*587.33*(0:0.000125:0.2));
e = sin(2*pi*659.26*(0:0.000125:0.7));
g = sin(2*pi*783.99*(0:0.000125:0.7));
highb = sin(2*pi*987.767*(0:0.000125:0.7));
fs = sin(2*pi*739.988*(0:0.000125:0.7));
lowfs = sin(2*pi*369.994*(0:0.000125:0.7));
lowfshalfnote = sin(2*pi*369.994*(0:0.000125:1.4));
lowg = sin(2*pi*391.995*(0:0.000125:0.7));
lowghalfnote = sin(2*pi*391.995*(0:0.000125:1.4));
lowe = sin(2*pi*329.628*(0:0.000125:0.7));
lowewholenote = sin(2*pi*329.628*(0:0.000125:2.8));
lowc = sin(2*pi*261.626*(0:0.000125:0.7));
lowchalfnote = sin(2*pi*261.626*(0:0.000125:1.4));
lowb = sin(2*pi*246.942*(0:0.000125:0.7));
lowdwholenote = sin(2*pi*293.665*(0:0.000125:2.8));
lowdhalfnote = sin(2*pi*293.665*(0:0.000125:1.4));
lowcsharphalfnote = sin(2*pi*277.183*(0:0.000125:1.4));
lowdsharphalfnote = sin(2*pi*311.127*(0:0.000125:1.4));
% Create song segments
melody = [c, g, highb, fs, c, g, highb, fs,c, g, highb, fs, c, g, highb, fs,c,
g, highb, fs,];
shortnotes = [c, g, highb, fs,lowghalfnote,
lowghalfnote,lowfshalfnote,lowfshalfnote,lowg,lowfs,lowe,b,lowe,lowc,lowb,lowe
];
longnotes = [c, g, highb, fs,lowewholenote, lowdwholenote,
lowewholenote,lowcsharphalfnote,lowdsharphalfnote];
% Combine parts by adding them together
song = [melody, shortnotes, longnotes];
% Normalize song to avoid clipping
song = song / max(abs(song));
% Write the final music to a .wav file
audiowrite('song.wav', song, FS);
% Playback
audio = audioplayer(song, FS);
play(audio);
```

Matlab Code 2: (audioMerge.m)

```
% Load audio files
[song, Fs_s] = audioread('song.wav');
[noise, Fs_n] = audioread('audionoise.wav');
% Convert noise to mono if necessary
if size(noise, 2) > 1
   noise = mean(noise, 2);
end
% Resample the disturbance signal to match the song's sampling rate
if Fs_n ~= Fs_s
   try
       % Use resample if Signal Processing Toolbox is available
       noise = resample(noise, Fs_s, Fs_n);
   catch
       % Alternative resampling using interp1
       disp("Signal Processing Toolbox not installed");
       t_original = (0:length(noise)-1) / Fs_n; % Original time vector
       t_target = (0:1/Fs_s:(length(noise)-1)/Fs_n)'; % Target time vector
       noise = interp1(t_original, noise, t_target, 'linear'); % Linear
interpolation
   end
end
% Specify the starting time for noise in seconds
startTimeofNoise = 10;
% Calculate the starting index for adding the disturbance
FS_Index = round(startTimeofNoise * Fs_s);
% Ensure the disturbance signal fits within the music signal length
if FS_Index + length(noise) > length(song)
   noise = noise(1:length(song) - FS_Index); % Trim noise
elseif FS_Index + length(noise) < length(song)
   noise = [noise; zeros(length(song) - (FS_Index + length(noise)), 1)]; % Pad
noise
end
% Match channels if music is stereo
if size(song, 2) > 1 && size(noise, 2) == 1
   noise = [noise, noise]; % Convert mono to stereo
end
% Add noise to the music
noisySong = song;
noisySong(FS_Index:FS_Index + length(noise) - 1, :) = ...
   noisySong(FS_Index:FS_Index + length(noise) - 1, :) + noise;
% Normalize the combined audio to prevent clipping
noisySong = noisySong / max(abs(noisySong), [], 'all');
% Save the combined audio as a new file
audiowrite('noisySong.wav', noisySong, Fs_m);
% Displaying Sampling rates for Procedure
disp('Sampling rate of Fs_s');
disp(Fs_s);
disp('Sampling rate of Fs_n')
```

```
disp(Fs_n);
disp('Audio processing completed and saved as noisySong.wav');
```

Matlab Code 3: (audioCancel.m)

```
% Load audio files
[y_clean, Fs_clean] = audioread('song.wav'); % Reference clean audio
(optional)
[y_noisy, Fs_noisy] = audioread('noisySong.wav'); % Noisy audio to be cleaned
% Ensure both audio files have the same sampling rate
if Fs_clean ~= Fs_noisy
   error('Sampling rates of the clean and noisy audio files do not match.');
end
% Normalize the noisy audio
y_noisy = y_noisy / max(abs(y_noisy)); % Scale the signal to the range [-1, 1]
% Validate and set the cutoff frequency
nyquist_frequency = Fs_noisy / 2;
filter_cutoff = 1000; % Example cutoff frequency in Hz
if filter_cutoff >= nyquist_frequency
   error('Cutoff frequency must be less than the Nyquist frequency (%f Hz).',
nyquist_frequency);
end
% Design a low-pass filter
filter_order = 12; % Order of the filter
lpFilter = designfilt('lowpassiir', 'FilterOrder', filter_order, ...
                      'HalfPowerFrequency', filter_cutoff, ...
                      'SampleRate', Fs_noisy);
% Apply the low-pass filter to the normalized noisy audio
y_processed = filtfilt(lpFilter, y_noisy); % filtfilt applies zero-phase
filtering to avoid phase distortion
% Normalize the filtered audio again to avoid clipping
y_processed = y_processed / max(abs(y_processed));
% Write the final processed audio to a new file
output_filename = 'cleanSong.wav'; % Name for the output file
audiowrite(output_filename, y_processed, Fs_noisy);
% Display a message indicating the completion of the process
disp('Processed and normalized audio file saved as cleanSong.wav');
```

Matlab Code 4: (timeandfreqPlot.m)

```
% Create a new figure
figure;
% Plot for 'song.wav'
[y, Fs] = audioread('song.wav');
duration = length(y) / Fs;
t = linspace(0, duration, length(y));
Y = fft(y);
freq_resolution = Fs / length(Y);
f = linspace(0, Fs - freq_resolution, length(Y));
% Time domain of song.wav
subplot(4, 3, 1);
plot(t, y);
title('song.wav - Time Domain');
xlabel('Time (s)');
ylabel('Amplitude');
% Frequency domain of song.wav
subplot(4, 3, 2);
plot(f, abs(Y));
title('song.wav - Frequency Domain');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
xlim([0, Fs/2]);
% Spectrogram of song.wav
subplot(4, 3, 3);
spectrogram(y, 256, [], [], Fs, 'yaxis');
title('song.wav - Spectrogram');
colorbar;


% Plot for 'noisySong.wav'
[y, Fs] = audioread('noisySong.wav');
duration = length(y) / Fs;
t = linspace(0, duration, length(y));
Y = fft(y);
freq_resolution = Fs / length(Y);
f = linspace(0, Fs - freq_resolution, length(Y));
% Time domain of noisySong.wav
subplot(4, 3, 4);
plot(t, y);
title('noisySong.wav - Time Domain');
xlabel('Time (s)');
ylabel('Amplitude');
% Frequency domain of noisySong.wav
subplot(4, 3, 5);
plot(f, abs(Y));
title('noisySong.wav - Frequency Domain');
xlabel('Frequency (Hz)');
```

```
ylabel('Magnitude');
xlim([0, Fs/2]);
% Spectrogram of noisySong.wav
subplot(4, 3, 6);
spectrogram(y, 256, [], [], Fs, 'yaxis');
title('noisySong.wav - Spectrogram');
colorbar;
% Plot for 'cleanSong.wav'
[y, Fs] = audioread('cleanSong.wav');
duration = length(y) / Fs;
t = linspace(0, duration, length(y));
Y = fft(y);
freq_resolution = Fs / length(Y);
f = linspace(0, Fs - freq_resolution, length(Y));
% Time domain of cleanSong.wav
subplot(4, 3, 7);
plot(t, y);
title('cleanSong.wav - Time Domain');
xlabel('Time (s)');
ylabel('Amplitude');
% Frequency domain of cleanSong.wav
subplot(4, 3, 8);
plot(f, abs(Y));
title('cleanSong.wav - Frequency Domain');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
xlim([0, Fs/2]);
% Spectrogram of cleanSong.wav
subplot(4, 3, 9);


spectrogram(y, 256, [], [], Fs, 'yaxis');
title('cleanSong.wav - Spectrogram');
colorbar;
% Plot for 'audiodisturb.wav'
[y, Fs] = audioread('audionoise.wav');
duration = length(y) / Fs;
t = linspace(0, duration, length(y));
Y = fft(y);
freq_resolution = Fs / length(Y);
f = linspace(0, Fs - freq_resolution, length(Y));
% Time domain of audionoise.wav
subplot(4, 3, 10);
plot(t, y);
title('audionoise.wav - Time Domain');
xlabel('Time (s)');
ylabel('Amplitude');
```

```
% Frequency domain of audionoise.wav
subplot(4, 3, 11);
plot(f, abs(Y));
title('audionoise.wav - Frequency Domain');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
xlim([0, Fs/2]);
% Add a global title for the entire figure
sgtitle('Audio Signal Analysis: Time, Frequency, and Spectrogram Domains');
```