Final Project

Due Date: **Tuesday, August 9, 2022** by 11:59 PM JST.

A **video link demonstrating the key features**, plus **all code**, is to be submitted online via Manaba by the due date.

Concepts to learn from this project:

- How to perform real-time audio analysis using block-based processing.
- How to convert DTFT magnitudes to a cool-looking bar chart (in Hz).
- How to create a GUI and an event-driven application in Matlab.

Important notes: The report and code you submit for this project must be your **own original work**.

Sharing of tips and/or a few lines of code is permitted. **Working in groups and/or copying of code is not**. This restriction includes using code from sources outside of the class (even open-source, public-domain code). Your code will be thoroughly checked using specialized software to ensure its originality.

Objective

In this project, you will create a full-featured audio player in Matlab. Through this process, you will learn how to perform real-time analysis and processing of audio and how to display the results graphically. In addition, you'll learn how to create a graphical user interface (GUI) in Matlab using event-driven-programming techniques.

The msound library

The standard audio facilities provided in Matlab do not easily allow block-based playback of audio. For this reason, in this project, use a third-party library called *msound*, which was described in class.

The *msound* library contains just one function: `msound()`. If you type `help msound` in the Matlab command window, you'll simply be given the README-type comments contained in MSOUND.M. To access the actual help file for *msound*, type `msound` or `msound('help')`. In this project, you'll be using only three facilities of msound: `'openwrite'`, `'putsamples'`, and `'close'`—you can ignore the rest.

An overview of the parts of this project

This project consists of four parts:

1. In Part 1, you'll create a time-based display of the audio that shows a time-domain plot corresponding to the block of audio that's currently being played back. This type of display allows one to easily see beats and volume changes; however, it doesn't provide an intuitive visualization of bass, mid-range, or treble.
2. In Part 2, you'll create a frequency-based display that shows the magnitude spectrum of the block of audio that's currently being played back.
3. In Part 3, you will implement real-time filtering.

4. In Part 4, you'll create the actual audio player using Matlab's GUI-building facilities. This part is more programming and less (but still a little) DSP. Nonetheless, collecting and implementing your code in an event-driven application will provide a greater appreciation of the techniques involved in real-time signal processing and how your backend code might possibly have to change to accommodate the front-end code.

Note that, despite the fact that you'll be using Matlab, the concepts that you'll learn in this project are very applicable to real-world, production-quality software applications. It's likely that you'll use these concepts again after you graduate (either in graduate school or on the job). For this reason, I highly recommend that you save your code in a safe location so that you may use it in the future. I also ask that you thoroughly comment your code; this not only helps us understand your code *now*, but it will also help you readily understand your code in the future.

Part 1: A live time-based display of audio

In the first part of the project, your goal is to write a Matlab function that will play and display a constantly updated time-domain plot. Call your function `rudsp_audio_time()`.

Writing your function

Your function should have the following parameters:

1. *Input parameter*: `fname`—a string containing the path+name of the wave file.
2. *Input parameter*: `blk_size`—an integer that specifies the number of samples in each block of audio that you plot/play in each iteration of your main processing loop. This parameter determines the rate at which your display is updated.

Your function signature should be as follows:

```
function rudsp_audio_time(fname, blk_size)

% TODO: Insert your code here.
```

The function should perform the following steps:

1. Load the appropriate samples (potentially all samples) of the wave file.

2. If the audio is stereo (two channels), convert stereo to mono.

3. Call the `msound()` function with the `'openwrite'` parameter to gain playback control of the soundcard. (Use `msound('help')` to learn the additional parameters to pass with `'openwrite'`.)

4. Loop over the length of the audio signal in increments of `blk_size`. Within your loop:
   a. Extract the appropriate block from the signal.
   b. Queue the block for playback by using `msound()` with the `'putsamples'` parameter.

c. Plot the time-domain samples of the block using Matlab's `plot()` function. Be sure to use the `xlim()` and `ylim()` functions to stabilize the *x* and *y* ranges of your plot.

5. Before your function exits, be sure to call `msound('close')`. Use a `try/catch` block to ensure that `msound('close')` is called in the event of an error.

Part 2: A live frequency-based display of audio

Now it's time to change the display to something that better corresponds to what we actually hear. In this second part of the project, your goal is to write a Matlab function that will play and display a constantly updated frequency-domain bar chart. Call your function `rudsp_audio_freq()`.

Writing your function

Your function should have the following parameters:

1. *Input parameter*: `fname`—a string containing the path+name of the wave file.
2. *Input parameter*: `use_wnd`—a Boolean (0 or 1) that specifies whether your function will perform windowing to combat edge-effects (see description of steps below).
3. *Input parameter*: `blk_size`—same as before (see Part 1).

Your function signature should be as follows:

```
function rudsp_audio_freq(fname, use_wnd, blk_size)

% TODO: Insert your code here.
```

The function should perform the following steps:

1. Load the appropriate samples (potentially all samples) of the wave file.
2. If the audio is stereo (two channels), convert stereo to mono.
3. 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'`.)
4. Loop over the length of the audio signal in increments of `blk_size`. Within your loop:
    a. Extract the appropriate block from the signal.
    b. Queue the block for playback by using the `msound()` function with the `'putsamples'` parameter.
    c. If the `use_wnd` parameter is true (non-zero), then multiply your signal block by a Hamming window of length `blk_size`.
    d. Compute and plot a bar chart of the magnitude spectrum of the block (in dB) over the range 20 Hz to 22,050 Hz. (See below for more info on this step.)

5. Before your function exits, be sure to call `msound('close')`. Use a `try/catch` block to ensure that `msound('close')` is called in the event of an error.

More info on the bar chart

Your bar chart should contain at least 16 bars in which each bar represents a range (band) of frequencies. An example is shown in **Figure 1**.

Table 1 lists the approximate ranges of frequencies that should be represented by each bar. Specifically, the first bar should correspond to a frequency range of roughly 20 Hz to 31 Hz. The second bar should correspond to a frequency range of roughly 31 Hz to 48 Hz. And, so on.

For reference, the center frequency of a band of frequencies is the mid-point on a logarithmic scale, which is given by the geometric mean of the lower and upper frequencies. For example, for the frequency range of 425-650 Hz, the center frequency is:

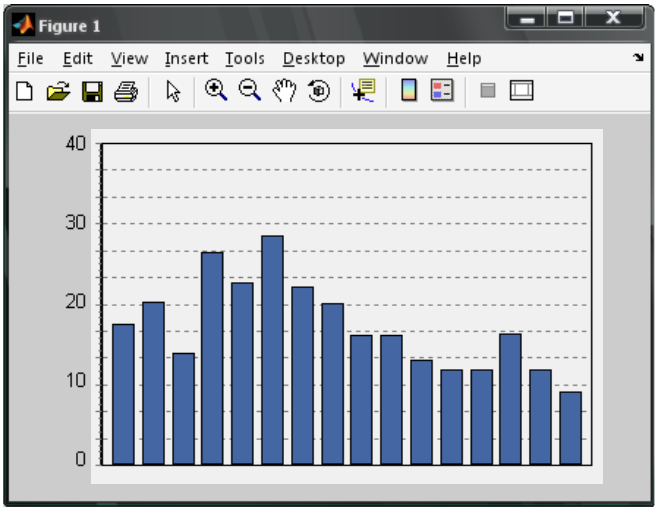

Figure 1: Snapshot of the bar chart that shows a real-time display of the audio's spectrum.

Bar	Frequency Range (Hz) f_{lower} to f_{upper}	Center Freq. (Hz) $\sqrt{f_{lower} \times f_{upper}}$	Freq. Ratio f_{upper}/f_{lower}	Num. Octaves $\log_2\left(\frac{f_{upper}}{f_{lower}}\right)$
1	20 to 31	25	1.55	0.63
2	31 to 48	39	1.55	0.63
3	48 to 74	60	1.54	0.62
4	74 to 115	92	1.55	0.64
5	115 to 175	142	1.52	0.61
6	175 to 275	219	1.57	0.65
7	275 to 425	342	1.55	0.63
8	425 to 650	526	1.53	0.61
9	650 to 1025	816	1.58	0.66
10	1025 to 1575	1271	1.54	0.62
11	1575 to 2450	1964	1.56	0.64
12	2450 to 3775	3041	1.54	0.62
13	3775 to 5850	4699	1.55	0.63
14	5850 to 9050	7276	1.55	0.63
15	9050 to 14000	11256	1.55	0.63
16	14000 to 22050	17570	1.58	0.66

Table 1: Suggested frequency range for each bar. You must determine which DTFT magnitudes to average for each bar in order to achieve the specified frequency range.

$$\sqrt{f_{upper} \times f_{lower}} = \sqrt{425 \times 650} = 526 \text{ Hz}.$$

We want each band of frequencies to have an approximately 0.5- to 0.75-octave bandwidth (generally useful when dealing with music). The number of octaves spanned by a band of frequencies is the $\log_2$ of the ratio of the upper to lower frequencies:

$$\log_2(f_{upper}/f_{lower}) = \log_2(650/425) = 0.61.$$

Thus, the 425-650 Hz band spans approximately 0.6 octaves (meaning $f_{upper}/f_{lower} \approx 2^{0.6} = 1.5$). Ideally, all 16 bands would have the same 0.6-octave bandwidth. However, as shown in **Table 1**, this is not quite the case, though it's arguably close enough for visualization purposes. You're free to tweak the ranges if you want to achieve more consistent bandwidths.

Testing your code

After you've coded the `rudsp_audio_freq()` function, run the function on the following signals:

> **Test Signals**: Sixteen pure tones (sine waves), each at a frequency at the center frequency for each bar (see **Table 1**), each of 3-second duration. You can create these tones in Matlab by generating an appropriate DT signal based on the desired CT frequency. For example, to create a tone at 526 Hz, you'd do:
>
> ```
> % desired frequency in Hz
> f0 = 526;
>
> % convert Hz to radians
> w0 = 2*pi*f0/fs;
>
> % a valid DT signal (not a CT wannabe)
> your_tone = sin(w0*n);
> ```
>
> where `n` is an appropriately sized vector of time indices (enough indices to yield 3 seconds).

Part 3: Real-time audio filtering

Now it's time to make your audio player much more valuable by adding real-time filtering capabilities.

(a) First, write a Matlab script to design a linear-phase FIR lowpass filter via windowing (see the docs for the `fir1()` function). As a test, try to design a filter with the following specs:
- A gain of $K_1 = -3$ dB at 200 Hz (the cutoff frequency).
- A gain of $K_2 = -60$ dB at 1000 Hz (we'll call this the beginning of the stopband).

(b) Next, extend your playback/visualization code to perform real-time audio filtering via the overlap-and-add method implemented via `conv()`, using your FIR lowpass filter as a testing filter.

(c) Finally, extend your code to perform real-time audio filtering via the overlap-and-add method implemented via the `fft()` and `ifft()` functions, using your FIR lowpass filter as a testing filter.

Part 4: The GUI

Finally, using all of the concepts/backend code described previously in this document, create an audio player in Matlab. The details of this project are too complex to describe via text. Instead, I will refer you to the lectures from class on creating a Matlab GUI.

Minimum requirements

Your audio player must contain the following minimum features to achieve a passing grade:

- The ability to support stereo output.
- The ability to specify the wave file to be played. This can be done via an edit box, or if you want some extra bling, use the `uigetfile()` function to display an Open dialog box.
- Play, Stop, and Pause buttons which operate correctly.
- A GUI display of the time-domain plot.
- A GUI display of the frequency-domain plot.
- A means of choosing between the time-domain plot and the frequency-domain plot (e.g., via a combo box). The user must be able to switch between plots as the song is being played back.
- The ability to filter the song in real-time (and switch filters on-the-fly).
- At least five filters: (1) allpass, (2) lowpass, (3) highpass, (4) bandpass, and (5) your best-sounding filter. The gains of the filters must be adjustable.

Extra feature suggestions

The above list specifies the **minimum requirements**. For a better grade, aim to impress. How? Add extra bling! Some examples include:

- A display of the current position in the song (in seconds).
- A slider that allows seeking to a different position within the song.
- A volume control.
- The ability to load a wave file in a block-based, as-needed fashion. This way, you won't have a huge delay when initially loading a large audio file.
- A slicker time-domain and/or frequency-domain display.
- Interesting background imagery.

What to hand in

Please submit the following to Manaba:

1. A report that describes the technical details of your audio player. (**Most of your grade will be determined based on this report, so please make it complete**; I suggest at least 6 pages.)
2. Your working code
3. A link to a video that demonstrates your audio player