

# CCIT4076 ENGINEERING AND INFORMATION SCIENCE

## LABORATORY 3: AUDIO SIGNAL PROCESSING

HKU SPACE Community College, Fall 2022

### 1 Audible Range

In this laboratory we are going to perform audio signal processing on Octave. Recall that audio signals admit frequencies ranging from 20Hz to 20kHz. Any monotonic sinusoid

$$x(t) = A \sin(2 \pi f_0 t)$$

with  $20 \leq f_0 \leq 20000$  can be heard in principle. This is merely an oversimplification. We will demonstrate your audible range in this section. Let us synthesize a T second 300 Hz sinusoid with the following command lines:

```
>> fs = 2*44100; T = 3;  
>> t = 0:1/fs:T;  
>> A = 0.1;  
>> f0 = 300;  
>> x = A*sin(2*pi*f0*t);
```

One may visualize the signal by using `plot(t, x)`. As a sound is meant to be heard, let us ask Octave to generate the sound by the command line:

```
>> sound(x, fs);
```

This shall produce a monotonic audio track buzzing from your speaker. Your task here is to complete the following table by listening to different frequencies given different amplitudes. Put a tick on the cell only if you can hear the tone clearly.

$A \backslash f_0$	5Hz	10Hz	50Hz	100Hz	1kHz	10kHz	15kHz	20kHz	25kHz
0.01									
0.1									
1									
10									

Sketch the graph of  $f_0$  versus smallest A to visualize your audible range.

### 2 Music Synthesis

Next, we will create a bank of music notes of the C<sub>4</sub> major and hence synthesize classical music from scratch. In coherence to lecture notes 4, below captures the frequencies corresponding to the music notes in the C<sub>4</sub> major:

Note	C4	D4	E4	F4	G4	A4	B4
Frequency	261.6256	293.6648	329.6276	349.2282	391.9954	440.0000	493.8833

Make use of the techniques learnt in Section 1 to create a  $1/2$  second long note for each of these notes. In other words, you need to create a total of seven  $1/2$  second long signals. You may name them as  $x_1, x_2, \dots, x_7$  respectively. Finally, create a huge array

$$X = [x_1, x_2, x_3, x_4, x_5, x_6, x_7];$$

to serve as your bank of music notes. Next, you may save the array  $X$  as a `.wav` file on your computer by the command line:

```
>> audiowrite('music_notes.wav', X, fs);
```

You may now see a new file appearing in your working directory. If you open that file by using any music player, that's exactly what you heard by typing `sound(X, fs)` in the command window!

We take one step further to create our own music frame. Let us take a classic Ode to Joy as written by Beethoven as an example. The note sequence

$$E_4 - E_4 - F_4 - G_4 - G_4 - F_4 - E_4 - D_4 - C_4 - C_4 - D_4 - E_4 - E_4 - D_4 - D_4$$

can store this into another array `Joy = [x_3, x_3, x_4, x_5, ...]`. You should be able to hear the classic upon completion of the array. Using this technique, you are now able to create any music frame you want.

We have learnt this powerful tool for music synthesis. However, we have left another essential component of music untouched: beat. Do you have any idea on how to create music notes with different beats?

### 3 Filtering

In the lectures we have discussed the filtering processes. In particular, we mentioned that there are three types of common filtering:

1. Low-pass filtering allows low frequencies to be kept at the output, and stops high frequency components to pass through the filter.
2. High-pass filtering is the exact contradiction of low-pass filtering, it erases low-pass (or baseband) components and only keeps high frequencies.
3. Band-pass filtering involves a specific type of filter that allows only a designated frequency range to pass through.

Roughly speaking, an ideal filter can be described by the equation

$$H(f) = \begin{cases} 1 & \text{if } f_L \leq f \leq f_H, \\ 0 & \text{otherwise.} \end{cases}$$

where the frequency range  $[f_L, f_H]$  is the pass-band. This description seems to be describing band-pass filters (BPF) at the first glance, but one may set  $f_L = 0$  for low-pass filters (LPF); or similarly  $f_H = \infty$  for high-pass filters (HPF).

In this section we are going to perform realistic filtering on Octave to give you some feelings on what filtering offers. Rigorously speaking, the training of the filtering process involves some more sophisticated mathematics and programming skills. Considering the course nature of CCIT4076 as a freshman course, we have built an Octave function `EIS_Filter.m` — in which we have done all the dirty work behind while providing you with the necessary degree of freedom to play with filtering.

The usage of the function is straightforward. Suppose the target signal is stored in the array `x` and the sampling frequency is `fs`, then the command:

```
>> y = EIS_Filter(x, fs, fl, fh);
```

where the variable `fl` stores the frequency  $f_L$  and `fh` stores  $f_H$ , shall return an array `y` as the filtered signal. The function will also plot the spectrum of the input signal, filter and output signal respectively. As a toy demo, suppose I have stored the classic Ode to Joy as built in Section 2 in the array `Joy`, then the command sequence:

```
>> fl = 0; fh = 340;
>> y = EIS_Filter(x, fs, fl, fh);
```

will result in a low-pass filtered version of the audio. Ideally speaking, all signals higher than 340Hz will then be erased, i.e. all “Fa” notes and “Sol” notes are blocked. However, since the filtering process here is a practical one, we observe that there are some “Fa” notes remaining in the filtered output, despite the fact that they are attenuated. This is shown in the subsequent figure. If you listen to the filtered output `y`, you will notice that some weak “Fa” notes but all the “Sol” notes are eliminated. This agrees with our observation on the output spectrum.

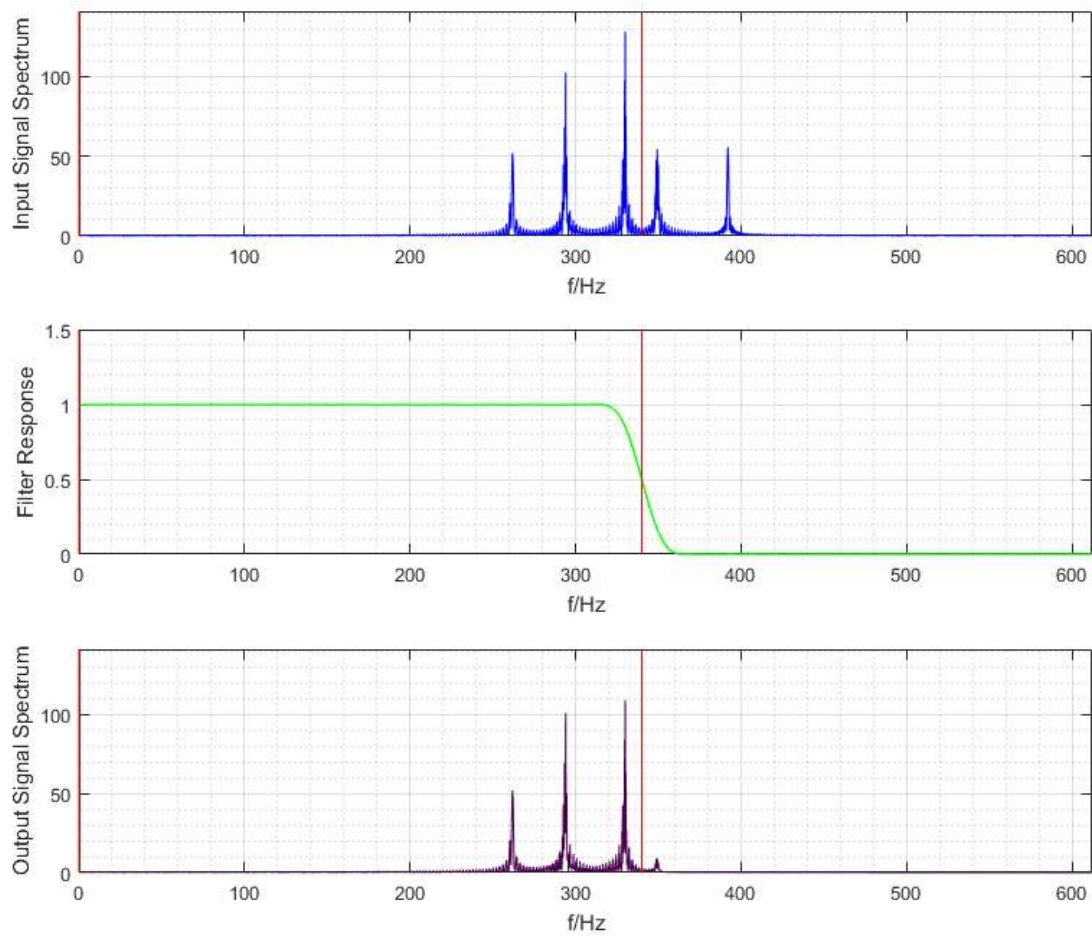


Figure: Function Output of running EIS\_Filter on Joy.

Top: Input spectrum. Middle: Designed Filter. Bottom: Filtered Output.