Lab 3: Audio Effects and Real-Time Processing

Due Monday 4/5/18

Overview:

- This assignment should be completed with your assigned lab partner(s).
- Each group must turn in a report composed using a word processor (e.g., Word, Pages, LaTeX, etc...) including a cover page with full names of all group members. The remaining pages should contain (in order) the answers and MATLAB scripts for the exercises. MATLAB figures can be pasted into the document or saved as PDF files. When working on the project, please follow the instructions and respond to each item listed. Your project grade is based on: (1) your MATLAB scripts, (2) your report (plots, explanations, etc. as required), and (3) your final results. For all labs, you must clearly write the problem number next to your solution and label the axes on all plots to get full credit. Submission should be done electronically on Sakai. Please include the report in PDF format and any requested m-files and/or audio files.
- Plagiarism is a very serious offense in Academia. Any figures in the paper not generated by you should labeled "Reproduced from [...]". Any portions of any simulation code (e.g., MATLAB, C, etc...) not written by you be clearly marked in your source files. The original source of any mathematical derivation or proof should be explicitly cited.

1 Overview

In this lab, the goal is to understand and implement a variety of standard audio effects. Some effects will be implemented for real-time processing using Simulink while others will be implemented for off-line processing using standard Matlab scripts. To get started, we will first create a few simple models in Simulink that process real-time audio. These models require that the Matlab Audio System Toolbox be installed.

A few notes about simulink: (1) use ctrl-click to connect the middle of a wire to another block input, (2) if your mouse cursor starts disappearing (e.g., after rescaling with the track pad), then press the escape key to get it back and (3) if you click on the model window and start typing a block name, you can get a new block without using the library browser.

1.1 Real-Time Audio Warm-Up: Generating an Audio Signal

These exercises will demonstrate how one can use Matlab Simulink to process real-time audio. To begin, start Matlab and then type "simulink" to start Simulink. The following steps will allow to build the first model.

1. After starting simulink, a window should appear with an icon called "Blank Model". Double-click on this option to open a new empty model.
2. To get the list of the processing blocks supported by Simulink choose "Library Browser" from the "Tools" menu.
3. Inside the library browser, find the "Sine Wave" block in the section "Simulink -> Sources. Click and drag this block into your empty model. Double-click on the "Sine Wave" block to change its parameters. Change "Sine Type" to "Sample based", "Samples per period" to "200", and "Sample time" to "1/44100".
4. Inside the library browser, find the "Audio Device Writer" block in the section "Audio System Toolbox -> Sinks". Click and drag this block into your empty model.
5. Now, click on the output of the "Sine Wave" block and drag a new wire to the input of the Audio Device Writer" block.

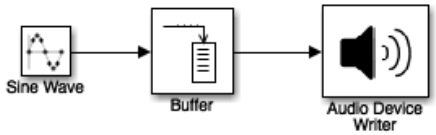
6. Try playing this model by clicking the green play button. How does it sound? Write down your first guess for what is going wrong.

7. Now, find the "Buffer" block in the library browser section "DSP System Toolbox -> Signal Management -> Buffers". Click and drag this block into your model.

8. Delete the wire connecting the "Sine Wave" block to the "Audio Device Writer" block by selecting the wire (by clicking) and then pressing the delete key. Next, connect the output of the "Sine Wave" block to the input of the "Buffer" block and connect the output of the "Buffer" block to the input of the "Audio Device Writer" block.

9. Try playing this model by clicking the green play button. How does it sound? Have we fixed the problem?

10. Double-click on the "Buffer" block to change its parameters. Change "Output buffer size" to "1024". This causes the model to use frame-based processing with 1024 samples per frame. Try pressing play again. How does the output sound now? Save this model as "warmup1" for future reference.



1.2 Real-Time Audio Warm-Up: Processing an Audio Signal

Now that we've figured out how to output an audio signal, let's try inputting an audio signal, processing it, and then outputting it.

1. Start with your "warmup1" model and first delete all the wires in that model.

2. Inside the library browser, find the "Audio Device Reader" block in the section "Audio System Toolbox -> Sources". Click and drag this block into your model.

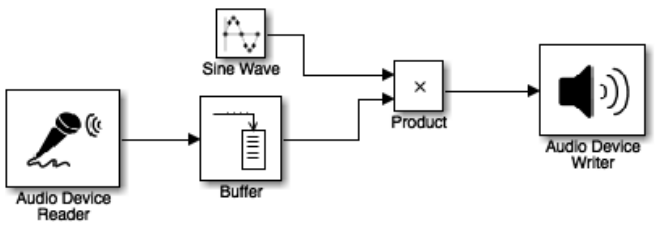
3. Double-click on the "Audio Device Reader" block to change its parameters. Change "Sample rate" to "44100". Double-click on the "Audio Device Writer" block to change its parameters. Uncheck the "Inherit sample rate from input" box and then change "Sample rate" to "44100".

4. Inside the library browser, find the "Product" block in the section "Simulink -> Commonly Used Blocks". Click and drag this block into your model.

5. Now, add wires to your model until it matches figure shown below.

6. Double-click on the "Sine Wave" block to change its parameters. Change "Bias" to "1", "Samples per period" to "80", and "Sample time" to "1024/44100". Save this model as "warmup2" for future reference.

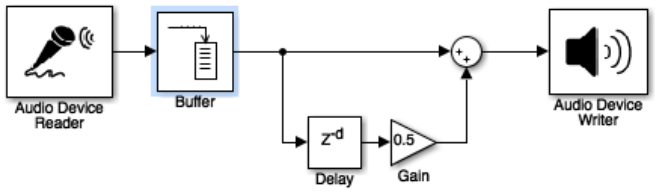
7. Use headphones to avoid feedback from the speaker to the microphone and try playing this model. If you talk into the input, you can hear that the output amplitude is modulated by a sine wave. What is the frequency of this modulation? Explain this based on the parameters you just entered into the "Sine Wave" block.



1.3 Real-Time Audio Warm-Up: Simple Echo

Now that we've figured out how to process an audio signal, let's try adding a simple echo.

1. Start with your "warmup2" model and delete the "Sine Wave" and "Product" blocks.
2. Inside the library browser, find the "Sum" block in the section "Simulink -> Commonly Used Blocks". Click and drag this block into your model.
3. Inside the library browser, find the "Delay" block in the section "Simulink -> Commonly Used Blocks". Click and drag this block into your model.
4. Inside the library browser, find the "Gain" block in the section "Simulink -> Commonly Used Blocks". Click and drag this block into your model.
5. Now, add wires to your model until it matches figure shown below.
6. Double-click on the "Gain" block to change its parameter to "0.5".
7. Double-click on the "Delay" block to change its parameters. Change "Delay length" to "44100" and "Input processing" to "Columns as channels".
8. Use headphones to avoid feedback from the speaker to the microphone and try playing this model. If you talk into the input, you should hear the input signal combined with an echo delayed by 1 second.
9. For real-time audio and effects, it is desirable to have the minimal latency possible. In our model, we can adjust the latency by changing the "Output buffer size" of the "Buffer" block. Determine the smallest buffer size that works on your system without buffer underflow (e.g., buffer underflow causes gaps in white noise at the signal output).



1.4 Real-Time Audio Warm-Up: Filtering Noise in Audio Signal

Now that we know how to add a simple effect to the audio signal, let's try implementing a digital filter. We will first learn to realize a filter with constant coefficients and then try extending it to make an externally controlled filter. In this process, we will learn about some useful signal processing blocks in Simulink. Before we proceed, note a convenient feature in the latest MATLAB versions: to add a block to your model, click anywhere in the white drawing area and just start typing the block's name, Simulink will give a list of suggestions right there and you can click on the desired block!

1. Start with your "warmup2" model and delete the "Audio Device Reader" and "Product" blocks. Also remove all wires in the model and save this new model as "sine_filter" for future reference.
2. Inside the library browser, find the "Biquad" block in the section "Simulink -> DSP System Toolbox HDL Support -> Filtering". Click and drag this block into your model. Alternatively, try typing "Biquad" using the above mentioned shortcut method.
3. Next add the "Time Scope" block from the section "Simulink -> DSP System Toolbox HDL Support -> Sinks". Right-click on the block, select "Signals & Ports -> Number of Input Ports" and click on "2".
4. Also add two "Constant" blocks from the section "Simulink -> Sources".
5. Double-click on the "Sine Wave" block to change its parameters. Change "Bias" to "0", "Samples per period" to "150", and "Sample time" to "1/44100".

6. Double-click on one of the "Constant" blocks and set the parameter "Constant Value" to [1 -1 0]' (note that it is a column vector). Similarly set the same parameter to [-0.9 0]' for the other "Constant" block. These are the filter coefficients corresponding to $B(z)$ and $A(z)$ respectively, where the leading "1" has been omitted for the $A(z)$ vector as required by the "Biquad" block. To understand the inputs for this block, try adding a "Transfer Fcn Direct Form II Time Varying" block and read its details by double-clicking on it. However, we will only use the "Biquad" block because it handles frame-based processing and therefore delete the "Transfer Fcn Direct Form II Time Varying" block once you are done reading about it.

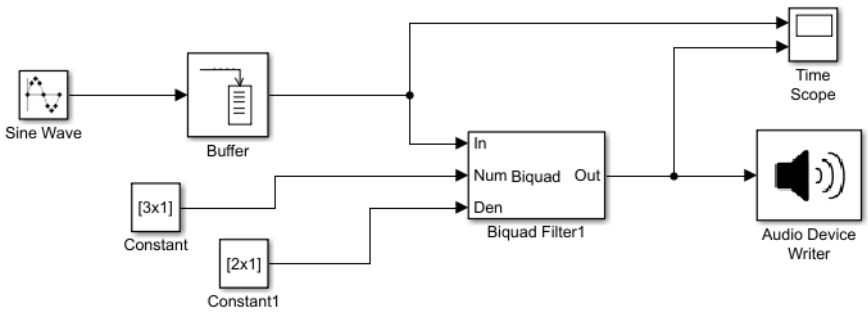
7. Double-click on the "Biquad" block to change its parameters. Set the "Coefficient source" option to "Input port(s)", the "Scale values mode" option to "Assume all are unity and optimize" and the "Input processing" option to "Columns as channels (frame based)".

8. Now complete the connections between these blocks as shown in the following figure.

9. This model implements a first-order IIR filter using the "Biquad" block with constant coefficients input through the "Constant" blocks. The "Time Scope" block can be double-clicked before running the model, so that its inputs are graphed in real-time and displayed after starting the simulation. In this case, you should see two sine waves with different amplitudes. Remember that it is standard to use column vectors for vector-valued block inputs.

10. Given the above filter coefficients and system sampling frequency, which frequencies should be attenuated by the filter and which ones should be passed almost as is? Demonstrate and verify your calculations by suitably varying the sine wave's frequency and observing the output of the "Time Scope" block.

11. Include this model file as part of your submission.



Now let us try to modify this model and implement a notch filter whose coefficients are controlled by the sine wave's frequency. To realize this, we need to use the "MATLAB Function" block in the above model.

1. Start with the above model, remove all wires and the "Time Scope" block. Save this new model as "audio_sine_notch" for future reference.

2. Change the values in the two "Constant" blocks to "300" and "20", and name them "Sine Frequency" and "Q" respectively as shown in the figure below. To rename, just click on the block's name below it and you should see a cursor then.

3. Add the block "From Multimedia File" from "Simulink -> Audio System Toolbox -> Sources". Double-click on it and give the full path of the file "guitar1.wav". Make sure that the "Samples per audio channel" parameter is set to "1024".

4. Add a "Sum" block from "Simulink -> Math Operations" and connect the outputs of the "From Multimedia File" block and "Buffer" blocks as shown in the figure below.

5. Next add the "MATLAB Function" block from the "Simulink -> User-Defined Functions" section. Double-click it and a function file should open in your MATLAB Editor. Rename the function as "iirnotchS" with inputs "f" and "Q", corresponding to the sine wave's frequency and the notch

filter's "Q" factor respectively. Also rename the outputs as "nr" and "dr", corresponding to the numerator and denominator coefficients of the notch filter respectively.

6. Now create a local copy of the input "f" into a new variable "lfo". This is because, in the general case of a time-varying input the function block cannot directly operate on "f" but only a "constant" copy of it. Although in this example, "f" is input from a constant block, it is important to note this behavior for future reference.

7. In your MATLAB Command Window, execute the command `type iirnotch`. Then you will see the set of commands executed by the function to generate the filter coefficients given the inputs "center frequency", denoted "Wo", and "Q-factor" or "3-dB bandwidth", denoted BW. You will execute the same set of commands inside the "MATLAB Function" block of your Simulink model. Then your function should look like this:

```
function [nr, dr] = iirnotchS(f,Q)
%#codegen
lfo = f;
Fs = 44100;
Wo = (2*lfo/Fs)*pi;
BW = (2*lfo/Fs/Q)*pi;
Ab = 3;
Gb = 10^(-Ab/20);
beta = (sqrt(1-Gb.^2)/Gb)*tan(BW/2);
gain = 1/(1+beta);

b  = gain*[1 -2*cos(Wo) 1];
a  = [1 -2*gain*cos(Wo) (2*gain-1)];

dr = a(2:end)';
nr = b';
end
```

   Again, it is important to return the vectors as column vectors.

8. Double-click on the "Sine Wave" block to change its parameters. Change "Samples per period" to "44100/300", and "Amplitude" to "0.5".

9. Now connect all the blocks until the "Biquad" block as shown in the figure below.

10. Add a "Digital Clock" block from the "Simulink -> Sources" section. During runtime, this block will output the current simulation time. Double-click it and set the "Sample Time" parameter to "1024/44100".

11. Next, add a "Switch" block, from the "Simulink -> Signal Routing" section, between the "Biquad" block and the "Audio Device Writer" block, and connect the "Digital Clock" output to its second input. This will act as the "control" input and whenever a specified condition is satisfied, the "Switch" block will output the first input; otherwise, it will output the third input. Double-click the "Switch" block and set the "Criteria for passing first input" parameter to "u2 > Threshold". Also set the "Threshold" parameter to "3", i.e. 3 seconds into the simulation this block will start outputting the first input.

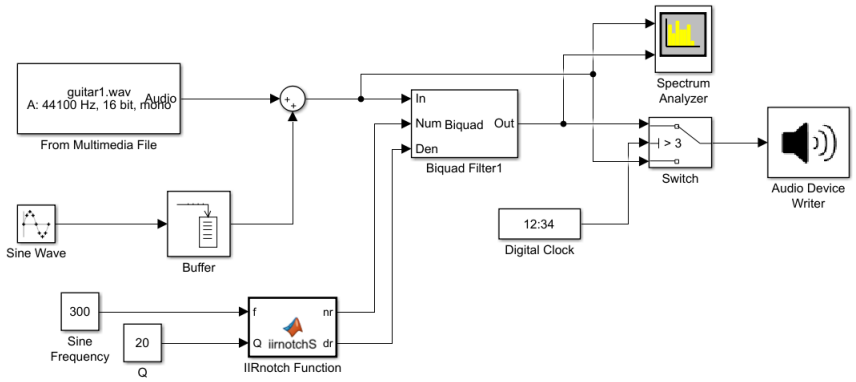
12. Finally add the "Spectrum Analyzer" block from the section "Simulink -> DSP System Toolbox HDL Support -> Sinks". As the name suggests, this will be useful to monitor the spectrum of its input signal(s) during run-time. As for the "Time Scope" block, right-click on the block, select "Signals & Ports -> Number of Input Ports" and click on "2".

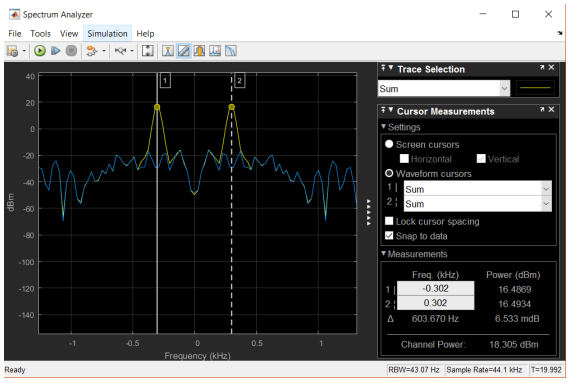
13. Complete all wire connections as shown in the figure below and start the simulation. You will first hear the guitar audio affected by the sinusoidal tone for about 3 seconds. Subsequently, you will hear the clean audio since this is now the output of the IIR filter executed by the "Biquad" block,

using the filter coefficients output by the "MATLAB Function" block. Note that this is in turn controlled by the sine wave's frequency.

14. In the "Spectrum Analyzer" window (double-click the block if the window doesn't appear automatically during run-time), select "Tools -> Measurements -> Cursor Measurements" or click on the shortcut in the toolbar. Adjust the measurement bars to measure the frequency of the two peaks of the yellow wave, which corresponds to the input to the "In" port of the "Biquad" block. The frequencies must be approximately 300 Hz in magnitude, exactly corresponding to the sinusoidal tone added to the guitar audio signal. This window should look something like the one shown below the model diagram.

15. Include this model file as part of your submission.





We have now finished exploring the basic blocks used in audio processing and we have seen some important things to keep in mind while designing on Simulink. Possibly the most salient feature is the frame-based processing of signals, i.e. operating on 1024x1 frames instead of individual samples, and because of this all parts of the system should operate at compatible sample rates. Otherwise, you will either receive a Simulink error (which is good) or the system will not work as expected (which is bad).

1.5 Real-Time Audio Warm-Up: Audio Plugins in Matlab

The audio plugin class in Matlab is designed to encapsulate the elements required to implement a virtual studio (VST) plugin. VST plugins can be loaded and used by many digital audio workstation (DAW) programs. In this section, we will explore the pitch-shifting example plug-in. Matlab includes an Audio Test Bench that can be used to apply the plug-in to both real-time audio and audio files.

1. Start Matlab and type "audiopluginexample.PitchShifter"
2. Type "audioTestBench"
3. When the window opens, set "Object Under Test" to "audiopluginexample.PitchShifter"
4. Press play to test the plug-in. It will use the default guitar file as input. Try changing the "Pitch Shift" during playback to hear the effect.

5. Now, click on the setup icon next to "Input" text. Change the "Name of audio file" box to the full path of the "singing44.wav" file from the website.

6. Now, change the "Output" to "Both" and set the "Pitch Shift" to "-1".

7. Press play to test the plug-in. This will create an "output.wav" file. Please submit this file with your lab.

8. You should also test this plug-in with real-time "Audio Device Reader" input and real-time "Audio Device Writer" output. Wear headphones to avoid feedback and enjoy auto-tuning your voice!

This plugin uses the time-domain overlap-add resampling discussed below. For details, see `https://www.mathworks.com/examples/audio-system/mw/audio_product-audioPitchShifterExample-delay-based-pitch-shifter`

2 Basic Effects

Historically, audio and guitar effects were implemented in circuits and optimized to give the right sound without being too complex (e.g., costing too much). As digital signal processing developed, it was desirable for some effects to be more accurate and pure (e.g., echo and delay) while other effects still try to capture the "warm" sound of the original analog effects (e.g., the distortion caused by overdriving tube amplifiers). Nowadays, one can approximate just about any effect with a moderate priced DSP and the main difficulty is designing the algorithm to give just the right sound. In this lab, we do not attempt to perfectly match any particular effect. Instead, the goal is to explore simple Matlab scripts that capture the essence of well-known effects. For a survey of other implementations and more advanced effects, see [1].

2.1 Echo, Delay, and Reverberation

An echo occurs when a sound reflects off a distant surface. Due to the longer path, the reflected sound arrives later than the original sound and can be heard as a distinct second copy of the original sound. A delay effect produces multiple echoes for musical effect. These echoes are often delayed by integer multiples of a fixed period of time (e.g., 1s, 2s, 3s, 4s). The following Matlab script implements a simple delay. Listen to the output of this effect for a few example input signals.

```
% Simple Delay Effect
repeats = 2;                        % Number of delays
atten = [0.9 0.5];                  % Attenuation of each delay
delay = [0.2 0.4];                  % Delays in seconds
index = round(delay*Fs);            % Delays in samples
y = x;                              % Initialize output
for i=1:repeats                     % For each delay
  xx = [zeros(index(i),1); x];      % Zero pad the beginning to add delay
  xx = atten(i)*xx(1:length(x));    % Cut vector to correct length
  y = y + xx;                       % Add delayed signal to output
end
```

Reverberation (or reverb) is the effect generated by many copies of a sound with different delays summing together due to reflections off the walls of a room. In contrast to echo, the individual copies are not distinguishable but overall effect typically makes the sound richer or fuller. Listen to the output of this effect for a few example input signals.

```
% Simple Reverb Effect
delay1 = round(Fs*0.008);           % FIR Delay
delay2 = round(Fs*0.025);           % IIR Delay
coef = 0.7;                         % IIR Decay rate
y = filter([1 zeros(1,delay1) coef],[1 zeros(1,delay2) -coef],x); % Filter
```

The quality of this simple reverberation is not particularly good. For example, the reverb has rather strong metallic artifacts that are due to the comb filtering effect of using fixed delays. This artifact remains even in the more complicated "Feedback Delay Network" approach described in [1, pp. 169–170]. To achieve more realistic reverb with minimal complexity, a good choice is the "freeverb" algorithm described here: `https://ccrma.stanford.edu/~jos/pasp/Freeverb.html`.

To get very realistic reverberation, a straightforward (but computationally complex) approach is to convolve the input with the impulse response of a room with good acoustical properties. The problem is that the room response time can easily be 1 second and that gives a 44,100 tap FIR for CD quality sound. A database of impulse responses can be found at `http://www.openairlib.net`.

2.2 Wah, Phaser, and Flanger

Most wah, phaser, and flanger effects are based on time-varying linear filters. For the wah, the center frequency of a band-pass filter (or peak filter) is varied periodically. The following Matlab script implements a simple wah effect. Listen to the output of this effect for a few example input signals.

```
% Simple Wah Effect
lfo_freq = 1;                                          % LFO Freq (Hz)
lfo_min = 200;                                         % LFO minval (Hz)
lfo_max = 2000;                                        % LFO maxval (HZ)
lfo = sawtooth(2*pi*lfo_freq*(1:length(x))/Fs,0.5);    % Generate triangle wave
lfo = 0.5*(lfo_max-lfo_min)*lfo+(lfo_min+lfo_max)/2;   % Shift/Scale Triangle wave
depth = 2;                                             % Q factor of peak filter
y = zeros(1,length(x));
for j=3:length(x);                                     % For each output
  [b,a] = iirpeak(2*lfo(j)/Fs,2*lfo(j)/Fs/depth);      % New filter coefs each time
  y(j) = b(1)*x(j)+b(2)*x(j-1)+b(3)*x(j-2) ...         % Compute 2nd order IIR output
         -a(2)*y(j-1)-a(3)*y(j-2);
end
```

For the phaser, the center frequencies of a bank of notch filters is varied periodically. The following Matlab script implements a simple phaser effect. Listen to the output of this effect for a few example input signals.

```
% Simple Two-Stage Phaser Effect
lfo_freq = 1;                                          % LFO Freq (Hz)
lfo_min = 200;                                         % LFO minval (Hz)
lfo_max = 2000;                                        % LFO maxval (HZ)
lfo = sawtooth(2*pi*lfo_freq*(1:length(x))/Fs,0.5);    % Generate triangle wave
lfo = 0.5*(lfo_max-lfo_min)*lfo+(lfo_min+lfo_max)/2;   % Shift/Scale Triangle wave
y = zeros(1,length(x));
for j=3:length(x);                                     % For each output
  [b,a] = iirnotch(2*lfo(j)/Fs,2*lfo(j)/Fs);           % New filter coefs each time
  y(j) = b(1)*x(j)+b(2)*x(j-1)+b(3)*x(j-2) ...         % Compute 2nd order IIR output
         -a(2)*y(j-1)-a(3)*y(j-2);
end
x = y;
y = zeros(1,length(x));
for j=3:length(x);                                     % For each output
  [b,a] = iirnotch(6*lfo(j)/Fs,6*lfo(j)/Fs);           % New filter coefs each time
  y(j) = b(1)*x(j)+b(2)*x(j-1)+b(3)*x(j-2) ...         % Compute 2nd order IIR output
         -a(2)*y(j-1)-a(3)*y(j-2);
end
```

For the flanger, the signal is mixed with delayed version of itself with a time-varying delay. This results in a time-varying comb filter. The following Matlab script implements a simple flanger effect.

Listen to the output of this effect for a few example input signals.

```
% Simple Flanger Effect
lfo_freq = 1/3;                                          % LFO Freq (Hz)
lfo_amp = 0.004;                                         % LFO Amp (sec)
lfo = 2+sawtooth(2*pi*lfo_freq*(1:length(x))/Fs,0.5); % Generate triangle wave
index = round((1:length(x))-Fs*lfo_amp*lfo);             % Read-out index
index(index<1) = 1;                                      % Clip delay
index(index>length(x)) = length(x);
y=x;                                                     % Input Signal
for j=1:length(x);                                       % For each sample
  y(j) = y(j)+x(index(j));                               % Add delayed signal
end
```

2.3 Exercise A

The goal of the this exercise is to implement an effect in real time. You may choose from the three effects described in Section 2.2 (i.e, wah, phaser, and flanger). You may implement the effect either using Simulink or as an Audio Plugin using the Audio Systems Toolbox.

Simulink. Implement the effect in simulink and try testing it in realtime and using a .wav file input. After you implement the effect, save the simulink model as "effect.slx" and include this file into your submission. Here are some implementation notes that you may find useful:

- The basic version of simulink does not have a triangle wave generator! However, there is a "Triangle Generator" block in the "Simscape" library. Feel free to substitute with a "Sine Wave" if you are unable to find this.
- To effect an affine mapping on the sine or triangle wave, you can combine the "Gain", "Sum", and "Constant" blocks.
- To effect a controlled delay, try using the "Variable Delay" or "Variable Fractional Delay" blocks.
- To implement a time-varying filter, try the block "Biquad" paired with the block "MATLAB Function".
- The coefficient formulas for notch and peak filters can be extracted from "iirnotch.m" and "iirpeak.m" using the "type" command. The "MATLAB Function" block can be used to compute these parameters.
- To test with an audio file input, try using the "From Multimedia File" source block from the Audio Systems Toolbox.

Once you complete the model for an effect, create a subsystem out of all the processing blocks in it excluding the input, i.e. the "From Multimedia File" block, and the output, i.e. the "Audio Device Writer" block. To do this, you can use the cursor to click-and-drag on the design area to create a rectangular selection of all blocks and wires inside that area. Then you should see three dots appear at the bottom-right corner of the selection and if you scroll the mouse-pointer over it, the first option it reveals is "create subsystem". This will collapse all those blocks into one composite block which is your "effect" block. You only have to connect an input and output to implement the effect (with fixed parameters). Similarly make a subsystem out of the "simple echo" model created in a previous section. Now try cascading the effect and echo blocks to implement a sequential task on the input audio signal. Connect the "From Multimedia File" and "Audio Device Writer" blocks as input and output, respectively, and observe the output. Include this model as "effect_echo" along with your submission and make sure you include the subsystem blocks appropriately. Mention any instructions to execute this composite model in your write-up.

What happens if you swap the order of the "effect" block and "echo" block? Do these operations commute?

Audio Plugin. The audio plugin class in Matlab is designed to encapsulate the elements required to implement a virtual studio (VST) plugin. VST plugins can be loaded and used by many digital audio workstation (DAW) programs. For example, the REAPER DAW has a free trial version (`http://www.reaper.fm/index.php`) that is compatible with the VST2 plugins generated by Matlab.

There is a tutorial on the MathWorks website that describes the steps required to write an audio plugin for Matlab (`https://www.mathworks.com/help/audio/gs/design-an-audio-plugin.html`). Audio plugins can be tested in Matlab using the Audio Test Bench.

- To get started, I suggest that you copy the code from step 7 of the web tutorial into an m-file "myEchoPlugin.m". Then, execute this file (to define the plugin) and test it with the Audio text bench by typing "audioTestBench(myEchoPlugin)". Set the echo gain to 1, the delay to 0.25 seconds, and press the green play button.
- By following the tutorial, you can understand what method in the m-file does and modify the plugin to implement a different effect.

3 A Simple Method of Pitch Shifting

One can double the pitch of a periodic sound simply by playing the samples twice as fast (note: to keep the same output sampling rate, one should actually decimate by 2). The problem with this approach is that the resulting sound is also compressed by a factor of two in time. For speech and singing, this sounds very unnatural and also accelerates the rate at which words are pronounced. Is there a simple way to avoid this coupled time-scale modification?

Indeed, there is. For integers $p \geq q$, we can map p-sample blocks to q-sample blocks. Each block can be resampled to achieve the change in pitch but the input/output stride is adjusted to prevent an overall time scaling. Thus, for $j = 0, 1, \ldots, q-1$ and $i = 0, 1, \ldots$, we define

$$\begin{aligned} y[qi + j] &= x[qi + pj/q] \\ &\approx x[\text{round}(qi + pj/q)], \end{aligned}$$

where the second formula uses rounding to handle non-integer sample values. Alternatively, one can use linear interpolation

$$x[qi + pj/q] \approx (\lceil qi + pj/q \rceil - (qi + pj/q))\, x\left[\lfloor qi + pj/q \rfloor\right] + (1 - \lceil qi + pj/q \rceil + qi + pj/q)\, x\left[\lceil qi + pj/q \rceil\right].$$

3.1 Exercise B

1. Try applying this method to "singing44.wav" using $p = 860$ and $q = 766$.
2. How does it sound? What is the problem with this method? By how many half-tones have we increased the frequency?
3. This artifact you're hearing can be reduced by doing the same operation twice as many times and adding the results together weighted by the window function

   $$w[m] = \sin^2\left(\frac{\pi}{q} m\right).$$

   In particular, for $j = 0, 1, \ldots, q/2 - 1$ and $i = 0, 1, \ldots$, try the formula

   $$y[qi/2 + j] = x[q(i-1)/2 + p(q/2 + j)/q]w[q/2 + j] + x[qi/2 + pj/q]w[j].$$

4. How does this sound? Think about this pitch-shifting process and where the artifacts might come from. Can you think if some possible ways to reduce the artifacts?

References

[1] U. Zölzer, *DAFX: Digital Audio Effects*. Wiley Publishing, 2nd ed., 2011.