Reverberation Removal

Now we have been given an audio file with the noise of a highly reverberant room that we want to remove. Because the acoustics of room would be hard to model in terms of exact delays, we send a short pulse p(n), to mimic an impulse, into the room and record the response of the room r(n) to this input. Having the input and output of a system we can determine the transfer function of the system, i.e. the acoustics of the room, and we can use this to run an inverse filtering operation to recover the originally desired audio in the recording.

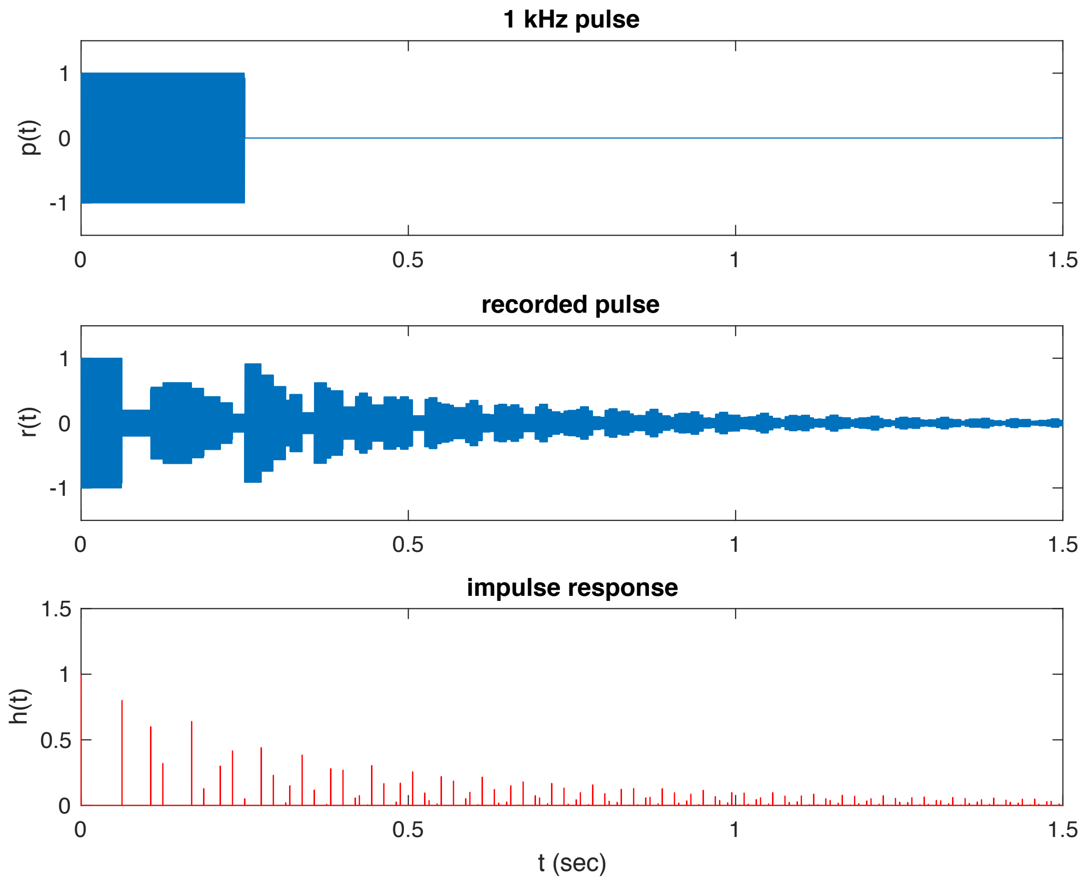
In the z-domain, we have that

(22) $P(z)H(z) = R(z)$

which gives,

(23) $$H(z) = R(z)\frac{1}{P(z)}$$

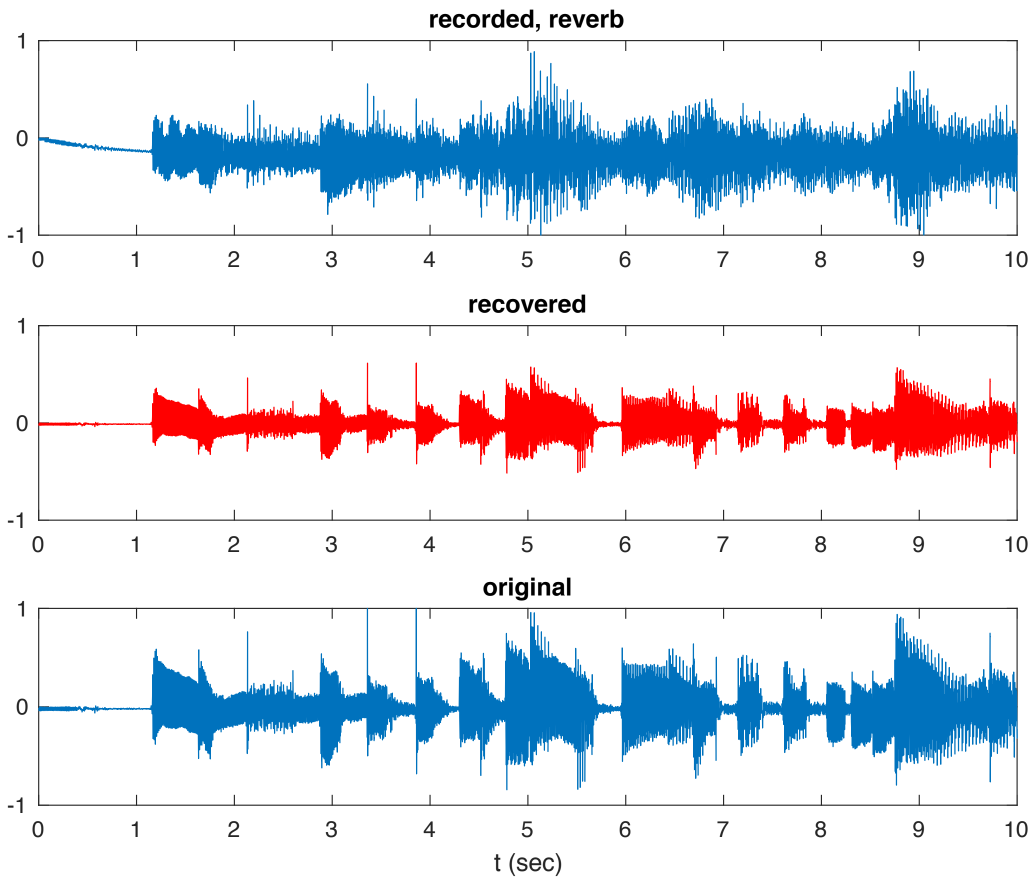
showing that if we run the known signal R(z) through a filter whose transfer function is the inverse of P(z) and take the inverse z transform, we obtain the "impulse" response of the room. Doing this using the filter function in MATLAB we obtain the following plots.



To recover the original signal we send the recording through an inverse filter with this impulse response.

(24) $$Y(z) = \frac{1}{H(z)} X(z)$$

Doing this using the filter function we obtain these results. (See Appendix A.2 for MATLAB code).



Again we can see that the recovered audio is practically the same as the original, By listening to the original and recovered audio, we hear that any small differences are inaudible.

Digital Audio Effects, Comb Filter

Now that we have the recovered recording we'll add a few audio effects. The algorithm used to remove the echo can be modified to be used as a creative effect. Instead of removing an echo from a recording we can add one.

(25) $y(n) = x(n) + ax(n - D)$

where again a represents the strength of the repeated copy and is between zero and one. This filter has an interesting frequency response. Taking the z-transform and solving for H(z) we obtain:

(26) $H(z) = 1 + az^{-D}$

substituting $z = e^{j\omega}$ and taking the magnitude,

(27) $H(e^{j\omega}) = 1 + ae^{-j\omega D}$

(28) $|H(\omega)| = \sqrt{1 + 2acos(\omega D) + a^2}$

which has zeros at $\omega = a^{1/D}e^{\pi j(2k + 1)/D}$ and peaks at $\omega = 2\pi k/D$, making a "comb" of filters alternating between attenuating and increasing amplitude, which mimics constructive and destructive interference at selected frequencies being introduced into the audio signal.

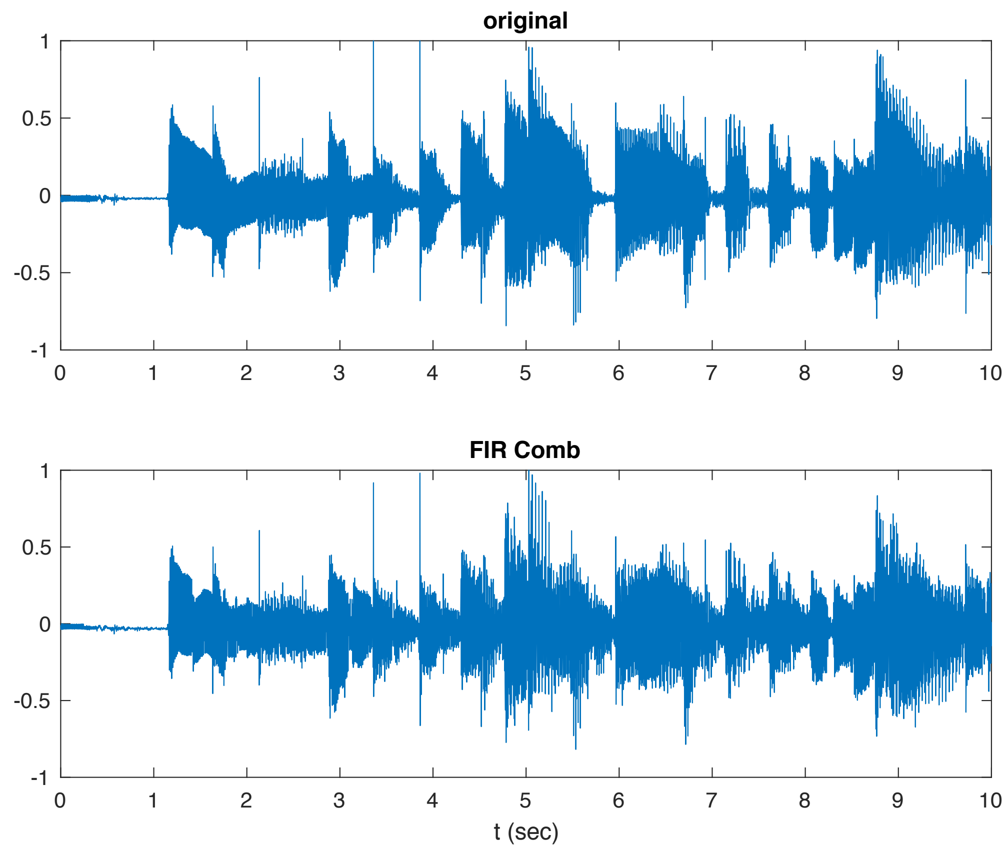
This filter can be further modified a superposition of comb filters to have an even more interesting frequency response by adding additional copies with varying attenuation into the signal.

(29) $y(n) = x(n) + ax(n - D) + a^2x(n - 2D) + a^3x(n - 3D)$

whose transfer function is:

(30)
$$H(z) = 1 + az^{-D} + a^2z^{-2D} + a^3z^{-3D} = \frac{1 - a^4z^{-4D}}{1 - az^{-D}}$$

Using this filter, and a value of a = 0.45, and D = 4000 (Note: $D*f_s$ = 250msec) we can introduce a pleasing characteristic to the sound, a more subtle yet desirable form of reverberation. Appendix A.3 has the MATLAB code. The execution time for the sample-by-sample (circular delay-line buffer) method is 0.083497 seconds, which is about 13.5 times faster than MATLAB's built-in filter function, which is 1.131073 seconds.



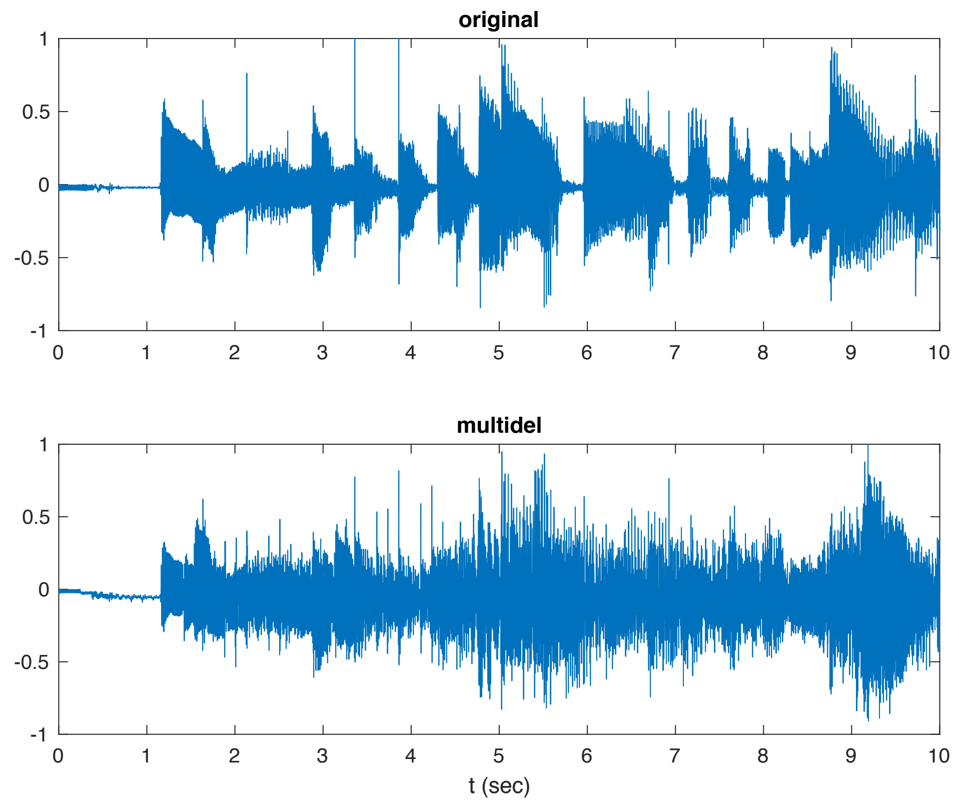
Digital Audio Effects, Multi-Delay Filter

In a real room, the acoustics are characterized by three main classifications of delay times between the sound source and the listener, the direct sound which goes straight from the source to the listener, the early reflections, which reach the listener after bouncing off of one to a few surfaces, and the late reflections, which are the early reflections bouncing off of several more surfaces before reaching the listener. The timing between delays gets progressively smaller, and the sound appears more dense to the listener. By taking the echoes of the previous filter and sending them into another delay unit with a different delay value D_2, we can have these echoes repeated more densely and mimic the more dense portion of the early reflections.

To implement this, will define the following transfer function:

(31)
$$H(z) = b_0 + b_1[\frac{z^{-D_1}}{1 - a_1 z^{-D_1}}] + b_2[\frac{z^{-D_1}}{1 - a_1 z^{-D_1}}][\frac{z^{-D_2}}{1 - a_2 z^{-D_2}}]$$

where a_1 and a_2 are the first and second reflection coefficients. We will let them equal 0.2 and 0.4 respectively, and b_0, b_1, and b_2 also play the role of reflection coefficients, but to scale the clusters of echoes as groups, with b_0 being the strength of the direct sound. For simplicity we let b_0, b_1, and b_2 all equal 1, D_1 = 0.25 msec, and D_2 = 0.125 msec. Appendix A.4 has the algorithm. Again we'll use the circular-buffer implementation and compare it with MATLAB's filter function. The execution time for the circular buffer method is 0.050099 seconds, which is about 50 times faster than MATLAB's built-in filter function, which is 2.495146 seconds. The larger difference is due to the additional data shifts of now using two delay lines.



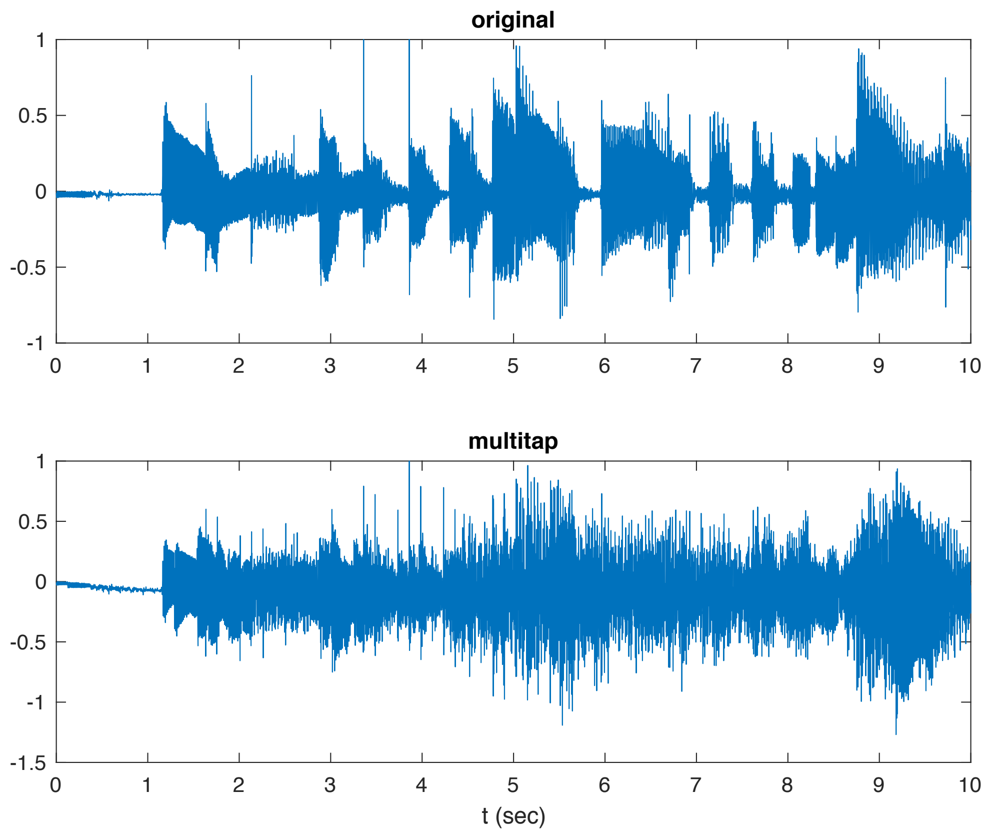
Digital Audio Effects, Multi-Tap Filter

The idea of adding a delayed, and scaled, copy of the input signal can be generalized to include any arbitrary number of delays (within processing time limitations between samples). Here we will illustrate this by creating a filter with two delays fed back into the input, along with these same delays being sent to the output but with different scaling factors, b_1 and b_2. The transfer function will be:

$$H(z) = b_0 + \frac{b_1 z^{-D_1} + b_2 z^{-(D_1+D_2)}}{1 - a_1 z^{-D_1} - a_2 z^{-(D_1+D_2)}} \tag{32}$$

showing that the delays involved will be D_1 and $D_1 + D_2$. Again b_0 here represents the direct sound coefficient. The corresponding sample processing algorithm can be found in Appendix

A.5. The



graphs of the input and output signals are shown below. It should be noted that the value $|a_1| + |a_2|$ should be less than one to maintain stability within the feedback network.

Digital Audio Effects, Flanging

Flanging is an effect that uses the same filter that we used to add a single echo into the signal, only allowing the value of the delay to vary sinusoidally with time. The result is a comb filter that sweeps up and down the frequency spectrum. The comb filter has its peaks at even multiples of f_s/D, and notches at odd multiples of $f_s/2D$. By allowing the value of D to vary sinusoidally with time in the time domain we get a sweeping back and forth along the frequency spectrum in the frequency domain.

The input output relationship is now:

(33) $y(n) = x(n) + ax(n - d(n))$

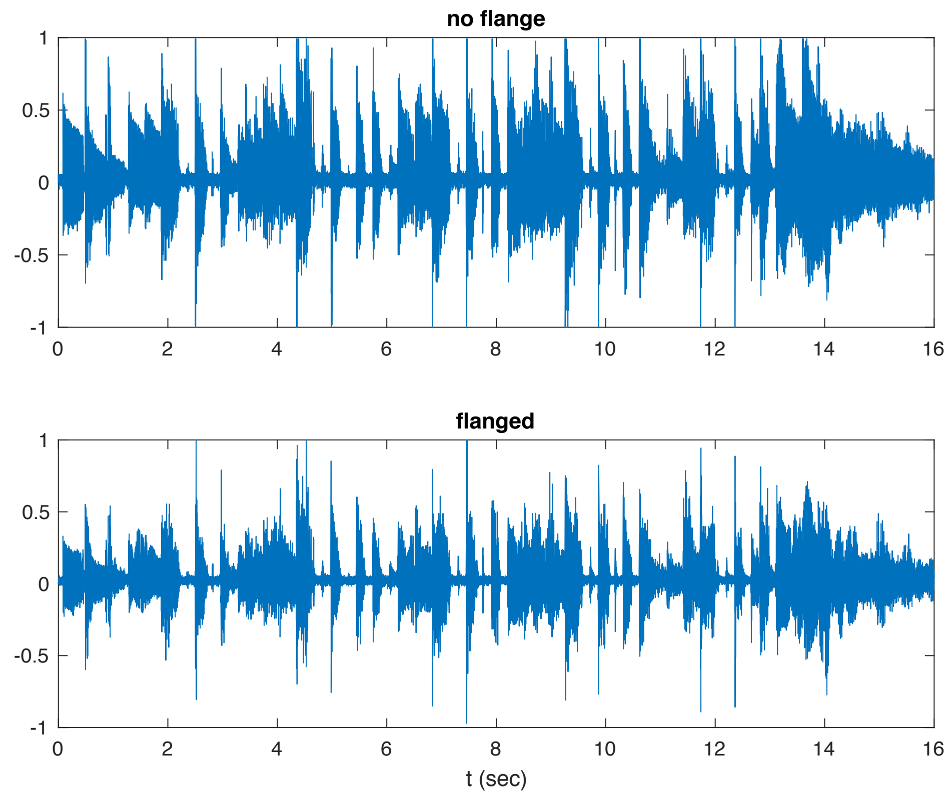
where

(34) $d(n) = (D/2)*[1 - \cos(2\pi F_d n)],\qquad 0 \le d(n) \le D$

and

and F_d is a low-frequency sinusoid, say a few Hz.

Because the value d(n) can take on non-integer values, we need to either truncate, round, or interpolate the values. Interpolation methods are more accurate, but to keep the code more straight-forward we will simply round the values here. For our example we will allow the delay d(n) to range from 0 to 3 msec with a modulating frequency of 2 Hz. The code for the example is shown in Appendix A.6, and the graphs of the input and output signals are shown below.



Appendix

A.1

```
% Loading the file with the echo into a vector x.

[x,fs] = audioread('echo.wav');        % Sampling rate is 16kHz.
[s,fs] = audioread('original.wav');

% Compute the autocorrelation and find Rxx(0)

Rxx = xcorr(x);
[Rxx0, Rxx0_k] = max(Rxx);

% Finding Rxx(D) and D

range = Rxx((Rxx0_k + 8000):(Rxx0_k + 15000));
[RxxD,RxxD_k] = max(range);

D = 8000 + RxxD_k - 1;

% Solving for a
```

```
syms A

quadratic = (A^2)*RxxD - A*Rxx0 + RxxD == 0;

a = double(solve(quadratic, A))

% Taking the first root
a = a(1);

%% Implement the DSP algorithm

[N,k] = size(x);
% Internal delay buffer for y(n)
w = zeros(1, D + 1);
% Delay buffer index variable
q = 1;
% Delay buffer tap
tap = D + 1;
% Loop through input signal
for n = 1:N
   % Read input sample directly into arithmetic unit to save space (for a dedicated DSP
chip,
   % although this can be read into register w(n) if the hardware does not allow it, and
w(n) can
   % be overwritten by y(n) after the input sample is used).
   w(q) = x(n) - a*w(tap);     % y(n) = x(n) - ay(n-D)
   y(n) = w(q);                % output y(n)
   q = q - 1;                  % Backshift index
   if q < 1                    % Circulate index
      q = D + 1;
   end
   tap = tap - 1;              % Backshift tap
   if tap < 1                  % Circulate tap
      tap = D + 1;
   end
end
```

<u>A.2</u>

```
%% Reverberation Removal
[x, fs] = audioread('reverb.wav');          % Recording with reverb
[s, fs] = audioread('original.wav');        % Desired (original) audio

[p, fs] = audioread('pulse.wav');           % Pulse signal
[r, fs] = audioread('pulse_rec.wav');                % Recorded pulse

h = filter(1,p,r);                                   % Obtain the impulse response
```

```
y = filter(1,h,x);                              % Obtain the recovered recording
```

A.3

```
%% (i) FIR Comb Filter

[s,fs] = audioread('original.wav');

D = 0.25*fs;
a = 0.45;


clear y;
[N,k] = size(s);
% Internal delay buffer for y(n)
w = zeros(1, 3*D + 1);
% Delay buffer index variable
q = 1;
% Delay buffer taps
tap1 = D + 1;
tap2 = 2*D + 1;
tap3 = 3*D + 1;
% Loop through input signal
tic
for n = 1:N
   % Read input into w.
   w(q) = s(n);
   % y(n) = s(n) + as(n-D) + a^2s(n-2D) + a^3s(n-3D)
   y(n) = w(q) + a*w(tap1) + a^2*w(tap2) + a^3*w(tap3);
   q = q - 1;                 % Backshift index
   if q < 1                   % Circulate index
      q = 3*D + 1;
   end
   tap1 = tap1 - 1;             % Backshift tap1
   if tap1 < 1                % Circulate tap1
      tap1 = 3*D + 1;
   end
      tap2 = tap2 - 1;        % Backshift tap2
   if tap2 < 1                % Circulate tap2
      tap2 = 3*D + 1;
   end
      tap3 = tap3 - 1;        % Backshift tap3
   if tap3 < 1                % Circulate tap1
      tap3 = 3*D + 1;
   end
end
toc

% Normalize y(n)
ymax = max(y);
y = y/ymax;
```

A.4

```
%% (ii) Multi-Delay
[s,fs] = audioread('original.wav');
D1 = 0.25*fs; D2 = 0.125*fs;
b0 = 1; b1 = 1; b2 = 1;
a1 = 0.2; a2 = 0.4;

clear y;
[N,k] = size(s);
% Internal delay buffer 1 for s(n)
w1 = zeros(1, D1 + 1);
% Internal delay buffer 2 for w1(n)
w2 = zeros(1, D2 + 1);
% Delay buffer 1 index variable
q1 = 1;
% Delay buffer 2 index variable
q2 = 1;
% Delay buffer taps
tap1 = D1 + 1;
tap2 = D2 + 1;
% Loop through input signal
tic
for n = 1:N
   s1 = w1(tap1);
   s2 = w2(tap2);
   y(n) = b0*s(n) + b1*s1 + b2*s2;
   w2(q2) = s1 + a2*s2;
   w1(q1) = s(n) + a1*s1;
   q1 = q1 - 1;              % Backshift index 1
   if q1 < 1                 % Circulate index 1
      q1 = D1 + 1;
   end
   tap1 = tap1 - 1;          % Backshift tap1
   if tap1 < 1               % Circulate tap1
      tap1 = D1 + 1;
   end
      q2 = q2 - 1;           % Backshift index 2
   if q2 < 1                 % Circulate index 2
      q2 = D2 + 1;
   end
      tap2 = tap2 - 1;       % Backshift tap2
   if tap2 < 1               % Circulate tap2
      tap2 = D2 + 1;
   end
end
toc

% Normalize y(n)
ymax = max(y);
y = y/ymax;


sound(y,fs);
```

A.5

```
%% (iii) Multi-tap Delay
[s,fs] = audioread('original.wav');
D1 = 0.125*fs; D2 = 0.25*fs;
b0 = 1; b1 = 1; b2 = 1;
a1 = 0.2; a2 = 0.4;

clear y;
[N,k] = size(s);

% Internal delay buffer for s(n)
w = zeros(1, D1 + D2 + 1);
% Delay buffer index variable
q = 1;
% Delay buffer taps
tap1 = D1 + 1;
tap2 = D1 + D2 + 1;
% Loop through input signal
tic
for n = 1:N
   s1 = w(tap1);
   s2 = w(tap2);
   y(n) = b0*s(n) + b1*s1 + b2*s2;
   w(q) = s(n) + a1*s1 + a2*s2;
      q = q - 1;                % Backshift index 1
   if q < 1                     % Circulate index 1
      q = D1 + D2 + 1;
   end
   tap1 = tap1 - 1;             % Backshift tap1
   if tap1 < 1                  % Circulate tap1
      tap1 = D1 + D2 + 1;
   end
      tap2 = tap2 - 1;          % Backshift tap2
   if tap2 < 1                  % Circulate tap2
      tap2 = D1 + D2 + 1;
   end
end
toc
% Normalize y(n)
ymax = max(y);
y = y/ymax;
```

A.6

```
%% (iv) Flanging

[x,fs] = audioread('noflange.wav');      % original, sampling frequency is 22.05 kHz

D = round(0.003*fs);
F = 2/fs;
% Internal delay buffer for x(n)
w = zeros(1, D + 1);
% Delay buffer index variable
```

```
q = 1;
a = 0.9;
[N,k] = size(x);


clear y;
for n = 1:N
    d = round((D/2)*(1 - cos(2*pi*F*n)));
    tap = q + d;
    if tap < 1
        tap = tap + (D + 1);
    end
    if tap > (D + 1)
        tap = tap - (D + 1);
    end
    y(n) = x(n) + a*w(tap);
    w(q) = x(n);
    q = q - 1;
    if q < 1
        q = D + 1;
    end
end
```