CM0340 Tutorial 6: MATLAB Digital Audio Effects

In this tutorial we explain some of the MATLAB code behind some of the digital audio effects we have studied.

Basically this revolves around:

• using filters in MATLAB, and
• modulated filters cut-off frequencies or delay parameters.
• We looked at some key aspects of both these in the last tutorial
The Return of MATLAB Filters

Matlab filter():

FILTER One-dimensional digital filter.

\[ Y = \text{FILTER}(B, A, X) \] filters the data in vector \( X \) with the filter described by vectors \( A \) and \( B \) to create the filtered data \( Y \).

We have two filter banks defined by vectors: \( A = \{a_k\}, B = \{b_k\} \).

We have to specify some values for them.

- We can do this by hand — we could design our own filters
- MATLAB provides standard functions to set up \( A \) and \( B \) for many common filters.

Last Tutorial (and in Lectures) we saw how we hand crafted one set of filters and use \texttt{butter()} or \texttt{buttord()} built-in MATLAB functions to create \( A \) and \( B \).

Now we need slightly more evolved filters for audio effects so we construct some functions to return more filter parameters to use with \texttt{filter()}.
Shelving and Peak Filters

Used in Equalisation, unlike low/high pass etc. filters, enhance/diminish certain frequency bands whilst leaving others unchanged:

**Shelving Filter** — Boost or cut the low or high frequency bands with a cut-off frequency, $F_c$ and gain $G$

**Peak Filter** — Boost or cut mid-frequency bands with a cut-off frequency, $F_c$, a bandwidth, $f_b$ and gain $G$
The following function, `shelving.m` performs a shelving filter:

```matlab
function [b, a] = shelving(G, fc, fs, Q, type)
%
% Derive coefficients for a shelving filter with a given amplitude
% and cutoff frequency. All coefficients are calculated as
%
% Usage: [B,A] = shelving(G, Fc, Fs, Q, type);
% G is the logarithmic gain (in dB)
% FC is the center frequency
% Fs is the sampling rate
% Q adjusts the slope by replacing the sqrt(2) term
% type is a character string defining filter type
% Choices are: 'Base_Shelf' or 'Treble_Shelf'

.... Theory Implemented as in Lecture note/DAFX textbook

% return values
a = [ 1, a1, a2];
b = [ b0, b1, b2];
```
The following script `shelving_eg.m` illustrates how we use the shelving filter function to filter:

```matlab
infile = 'acoustic.wav';

% read in wav sample
[ x, Fs, N ] = wavread(infile);

% set Parameters for Shelving Filter
% Change these to experiment with filter
G = 4; fcb = 300; Q = 3; type = 'Base_Shelf';

[b a] = shelving(G, fcb, Fs, Q, type);
yb = filter(b, a, x);

% write output wav files
wavwrite(yb, Fs, N, 'out_bassshelf.wav');

% plot the original and equalised waveforms
figure(1), hold on;
plot(yb,'b');
plot(x,'r');
title('Bass Shelf Filter Equalised Signal');
```
%Do treble shelf filter
fct = 600; type = 'Treble_Shelf';

[b a] = shelving(G, fct, Fs, Q, type);
yt = filter(b, a, x);

% write output wav files
wavwrite(yt, Fs, N, 'out_treblehelf.wav');

figure(1), hold on;
plot(yb,'g');
plot(x,'r');
title('Treble Shelf Filter Equalised Signal');
Time-varying Filters — Wah-wah Example

The signal flow for a wah-wah is as follows:

\[ y(n) + \times \times BP \text{direct-mix} \times \times \text{wah-mix} \]

where \( BP \) is a time varying frequency bandpass filter.
Time Varying Filter Implementation: State Variable Filter

RECAP (from lecture notes):

In our audio application of time varying filters we now want independent control over the cut-off frequency and damping factor of a filter.

(Borrowed from analog electronics) we can implement a State Variable Filter to solve this problem.

- One further advantage is that we can \textit{simultaneously} get lowpass, bandpass and highpass filter output.
The State Variable Filter

\[ x(n) = \text{input signal} \]
\[ y_l(n) = \text{lowpass signal} \]
\[ y_b(n) = \text{bandpass signal} \]
\[ y_h(n) = \text{highpass signal} \]
The State Variable Filter Algorithm

The algorithm difference equations are given by:

\[ y_l(n) = F_1 y_b(n) + y_l(n - 1) \]
\[ y_b(n) = F_1 y_h(n) + y_b(n - 1) \]
\[ y_h(n) = x(n) - y_l(n - 1) - Q_1 y_b(n - 1) \]

with tuning coefficients \( F_1 \) and \( Q_1 \) related to the cut-off frequency, \( f_c \), and damping, \( d \):

\[ F_1 = 2 \sin(\pi f_c / f_s), \quad \text{and} \quad Q_1 = 2d \]
MATLAB Wah-wah Implementation

We simply implement the State Variable Filter with a variable frequency, $f_c$. The code listing is `wah_wah.m`:

```matlab
% wah_wah.m  state variable band pass
%
% BP filter with narrow pass band, $F_c$ oscillates up and down the spectrum
% Difference equation taken from DAFX chapter 2
%
% changing this from a BP to a BR/BS (notch instead of a bandpass) converts this effect to a phaser
%
% $y_l(n) = F_1*y_b(n) + y_l(n-1)$
% $y_b(n) = F_1*y_h(n) + y_b(n-1)$
% $y_h(n) = x(n) - y_l(n-1) - Q_1*y_b(n-1)$
%
% vary $F_c$ from 500 to 5000 Hz

infile = 'acoustic.wav';

% read in wav sample
[ x, Fs, N ] = wavread(infile);
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% EFFECT COEFFICIENTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% damping factor
% lower the damping factor the smaller the pass band
damp = 0.05;

% min and max centre cutoff frequency of variable bandpass filter
minf=500;
maxf=3000;

% wah frequency, how many Hz per second are cycled through
Fw = 2000;

% change in centre frequency per sample (Hz)
delta = Fw/Fs;

% create triangle wave of centre frequency values
Fc=minf:delta:maxf;
while(length(Fc) < length(x) )
    Fc= [ Fc (maxf:-delta:minf) ];
    Fc= [ Fc (minf:delta:maxf) ];
end

% concatenate tri wave to size of input
Fc = Fc(1:length(x));
% difference equation coefficients
% must be recalculated each time Fc changes
F1 = 2*sin((pi*Fc(1))/Fs);
% this dictates size of the pass bands
Q1 = 2*damp;

yh=zeros(size(x)); % create empty output vectors
yb=zeros(size(x));
yl=zeros(size(x));

% first sample, to avoid referencing of negative signals
yh(1) = x(1);
yb(1) = F1*yh(1);
yl(1) = F1*yb(1);

% apply difference equation to the sample
for n=2:length(x),
    yh(n) = x(n) - yl(n-1) - Q1*yb(n-1);
yb(n) = F1*yh(n) + yb(n-1);
yl(n) = F1*yb(n) + yl(n-1);
    F1 = 2*sin((pi*Fc(n))/Fs);
end

% normalise
maxyb = max(abs(yb));
yb = yb/maxyb;
% write output wav files
wavwrite(yb, Fs, N, 'out_wah.wav');

figure(1)
hold on
plot(x,'r');
plot(yb,'b');
title('Wah-wah and original Signal');

The output from the above code is (red plot is original audio):
Click here to hear: original audio, wah-wah filtered audio.
Wah-wah Code Explained

Three main parts:

• Create a triangle wave to modulate the centre frequency of the bandpass filter.

• Implementation of state variable filter

• Repeated recalculation if centre frequency within the state variable filter loop.
Creation of triangle waveform we have seen last week — see waveforms.m.

- Slight modification of this code here to allow 'frequency values' (Y-axis amplitude) to vary rather than frequency of the triangle waveform — here the frequency of the modulator wave is determined by wah-wah rate, $F_w$, usually a low frequency:

```matlab
% min and max centre cutoff frequency of variable bandpass filter
minf=500; maxf=3000;
% wah frequency, how many Hz per second are cycled through
Fw = 2000;

% change in centre frequency per sample (Hz)
delta = Fw/Fs;
% create triangle wave of centre frequency values
Fc=minf:delta:maxf;
while(length(Fc) < length(x) )
    Fc= [ Fc (maxf:-delta:minf) ];
    Fc= [ Fc (minf:delta:maxf) ];
end
% trim tri wave to size of input
Fc = Fc(1:length(x));
```
Wah-wah Code Explained (Cont.)

**Note:** As the Wah-wah rate is not likely to be in perfect sync with input waveform, \( x \), we must trim it to the same **length** as \( x \)

** Modifications to Wah-wah **

- Adding Multiple Delays with differing centre frequency filters but all modulated by same \( F_c \) gives an **M-fold wah-wah**

- Changing filter to a **notch** filter gives a **phaser**
  
  - **Notch Filter** (or bandreject/bandstop filter (BR/BS)) — attenuate frequencies in a narrow bandwidth (High Q factor) around cut-off frequency, \( u_0 \)

- See Lab worksheet and useful for coursework
Bandreject (BR)/Bandpass(BP) Filters

(Sort of) Seen before (Peak Filter). Here we have, BR/BP:

$$y(n) = A_2(z) \pm BR = + \quad BP = - \quad \frac{1}{2} \times x(n) y_1(n)$$

where $A_2(z)$ (a second order allpass filter) is given by:

$$x(n) \quad T \quad x(n-1) \quad T \quad x(n-2)$$

$$\times -c \quad \times d(1-c) \quad \times 1$$

$$+ \quad + \quad +$$

$$\times c \quad \times -d(1-c)$$

$$T \quad y_1(n-2) \quad T \quad T$$

$$y_1(n-1) \quad y_1(n)$$

$$y(n)$$
Bandreject (BR)/Bandpass (BP) Filters (Cont.)

The difference equation is given by:

\[
\begin{align*}
y_1(n) &= -cx(n) + d(1 - c)x(n - 1) + x(n - 2) \\
&\quad - d(1 - c)y_1(n - 1) + cy_1(n - 2) \\
y(n) &= \frac{1}{2}(x(n) \pm y_1(n))
\end{align*}
\]

where

\[
d = -\cos\left(2\pi \frac{f_c}{f_s}\right)
\]

\[
c = \frac{\tan\left(2\pi \frac{f_c}{f_s}\right) - 1}{\tan\left(2\pi \frac{f_c}{f_s}\right) + 1}
\]

Bandreject = +
Bandpass = −
Delay Based Filters: Vibrato

**Vibrato** - A Simple Delay Based Effect

- **Vibrato** — Varying the time delay periodically
- If we vary the distance between an observer and a sound source *(cf. Doppler effect)* we hear a change in pitch.
- Implementation: A Delay line and a low frequency oscillator (LFO) to vary the delay.
- Only listen to the delay — no forward or backward feed.
- Typical delay time = 5–10 Ms and LFO rate 5–14Hz.
Vibrato MATLAB Code

**vibrato.m** function, Use **vibrato_eg.m** to call function:

```matlab
function y=vibrato(x,SAMPLERATE,Modfreq,Width)

ya_alt=0;
Delay=Width; % basic delay of input sample in sec
DELAY=round(Delay*SAMPLERATE); % basic delay in # samples
WIDTH=round(Width*SAMPLERATE); % modulation width in # samples
if WIDTH>DELAY
    error('delay greater than basic delay !!!');
    return;
end;

MODFREQ=Modfreq/SAMPLERATE; % modulation frequency in # samples
LEN=length(x); % # of samples in WAV-file
L=2+DELAY+WIDTH*2; % length of the entire delay
Delayline=zeros(L,1); % memory allocation for delay
y=zeros(size(x)); % memory allocation for output vector
```
for n=1:(LEN-1)
    M=MODFREQ;
    MOD=sin(M*2*pi*n);
    ZEIGER=1+DELAY+WIDTH*MOD;
    i=floor(ZEIGER);
    frac=ZEIGER-i;
    Delayline=[x(n);Delayline(1:L-1)];
    %---Linear Interpolation-----------------------------
    y(n,1)=Delayline(i+1)*frac+Delayline(i)*(1-frac);
    %---Allpass Interpolation---------------------------
    %y(n,1)=((Delayline(i+1)+(1-frac)*Delayline(i)-(1-frac)*ya_alt);
    %ya_alt=ya(n,1);
end
Vibrato MATLAB Code Explained

Click here to hear: original audio, vibrato audio.

The code should be relatively self explanatory, except for one part:

- We work out the delay (modulated by a sinusoid) at each step, n:

  \[ M = \text{MODFREQ}; \]
  \[ \text{MOD} = \sin(M \times 2 \times \pi \times n); \]
  \[ \text{ZEIGER} = 1 + \text{DELAY} + \text{WIDTH} \times \text{MOD}; \]

- We then work out the nearest sample step: \( i = \text{floor} (\text{ZEIGER}); \)

- The **problem** is that we have a *fractional delay line* value: \( \text{ZEIGER} \)

  \[
  \begin{align*}
  \text{ZEIGER} & = 11.2779 \quad i = 11 \\
  \text{ZEIGER} & = 11.9339 \quad i = 11 \\
  \text{ZEIGER} & = 12.2829 \quad i = 12
  \end{align*}
  \]
Fractional Delay Line - Interpolation

- To improve effect we can use some form of interpolation to compute the output, $y(n)$.
  - Above uses Linear Interpolation
    $$y(n,1) = \text{Delayline}(i+1)*\text{frac} + \text{Delayline}(i)*(1-\text{frac})$$
    
    or:
    $$y(n) = x(n - (M + 1)).\text{frac} + x(n - M).(1 - \text{frac})$$
  - Alternatives (commented in code)
    
    ```
    %---Allpass Interpolation-----------------------------------------------
    %y(n,1)=(Delayline(i+1)+(1-frac)*Delayline(i)-... 
    %      (1-frac)*ya_alt);
    %ya_alt=y(n,1);
    
    or:
    $$y(n) = x(n - (M + 1)).\text{frac} + x(n - M).(1 - \text{frac}) - 
    y(n-1).(1 - \text{frac})$$
    
    or spline based interpolation — see DAFX book p68-69.