CM3106 Chapter 7: Digital Audio Effects

Prof David Marshall
dave.marshall@cs.cardiff.ac.uk
and
Dr Kirill Sidorov
K.Sidorov@cs.cf.ac.uk
www.facebook.com/kirill.sidorov

School of Computer Science & Informatics
Cardiff University, UK

Digital Audio Effects

Having learned to make basic sounds from basic waveforms and more advanced synthesis methods lets see how we can at some digital audio effects.
These may be applied:

- As part of the audio creation/synthesis stage — to be subsequently filtered, (re)synthesised
- At the end of the *audio chain* — as part of the production/mastering phase.
- Effects can be applied in different orders and sometimes in a *parallel* audio chain.
- The order of applying the same effects can have drastic differences in the output audio.
- Selection of effects and the ordering is a matter for the sound you wish to create. There is no absolute rule for the ordering.

FX Pipeline

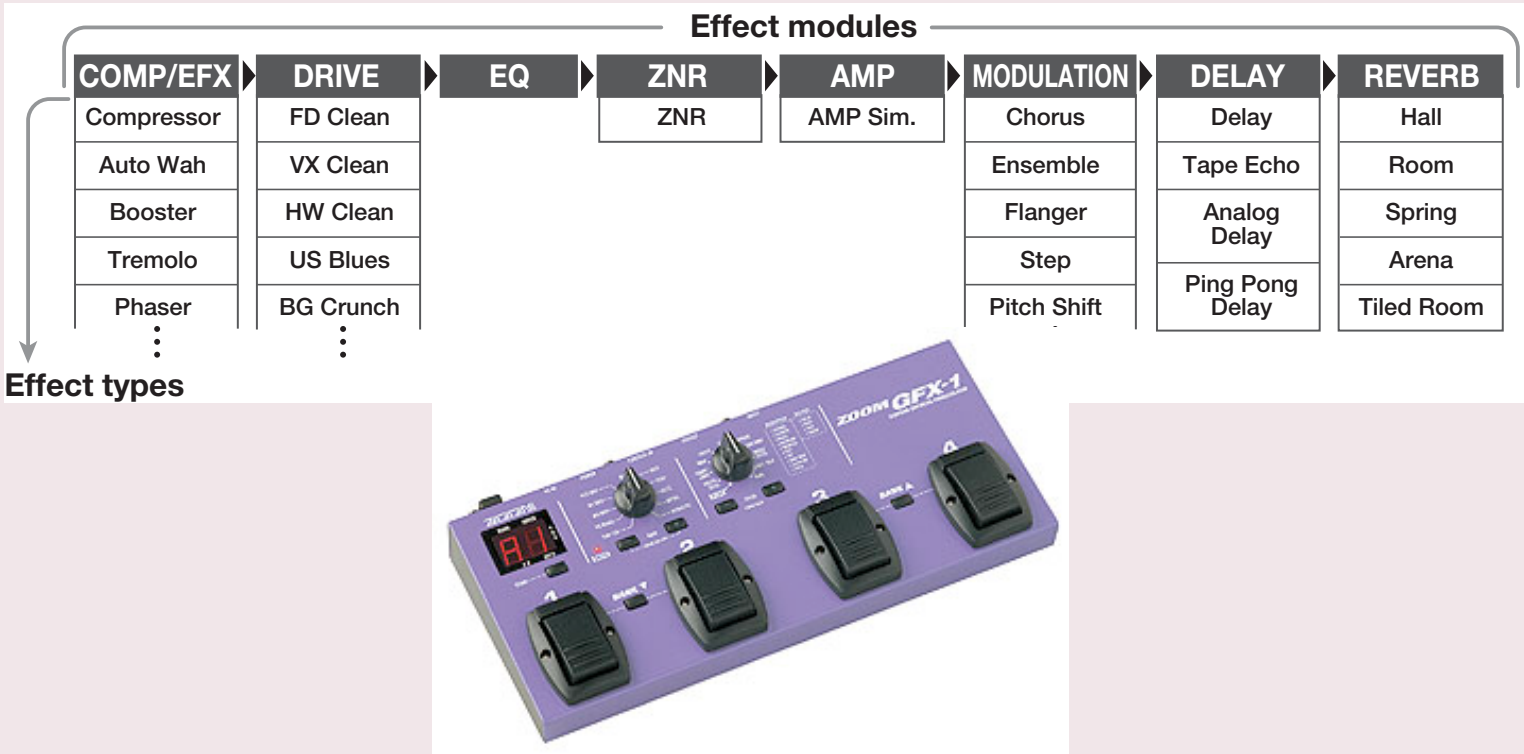
Apply effects in which order?

Some ordering is *standard* for some audio processing, *E.g*:

Compression → Distortion → EQ → Noise Redux → Amp Sim → Modulation → Delay → Reverb

Can also be configurable.

Common for order guitar (and other sources) effects pedal:



Effects Types

Audio effects can be classified by the way process signals:

Basic Filtering:	Lowpass, Highpass filter etc., Equaliser
Time Varying Filters:	Wah-wah, Phaser
Delays:	Vibrato, Flanger, Chorus, Echo
Modulators:	Ring modulation, Tremolo, Vibrato
Non-linear Processing:	Compression, Limiters, Distortion, Exciters/Enhancers
Spacial Effects:	Panning, Reverb, Surround Sound

Basic Digital Audio Filtering Effects: Equalisers

Filtering:

Filters by definition **remove/attenuate** audio from the spectrum above or below some cut-off frequency.

- For many audio applications this a little too restrictive

Equalisation:

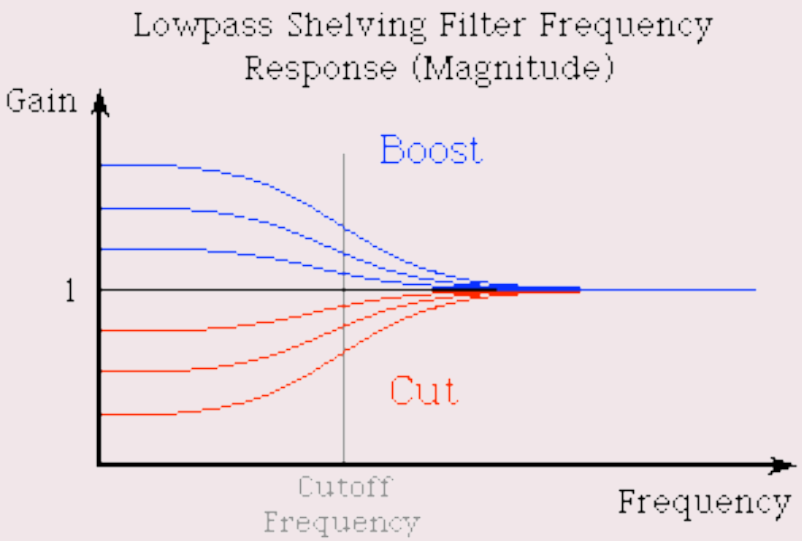
Equalisers, by contrast, **enhance/diminish** certain frequency bands whilst leaving others **unchanged**:

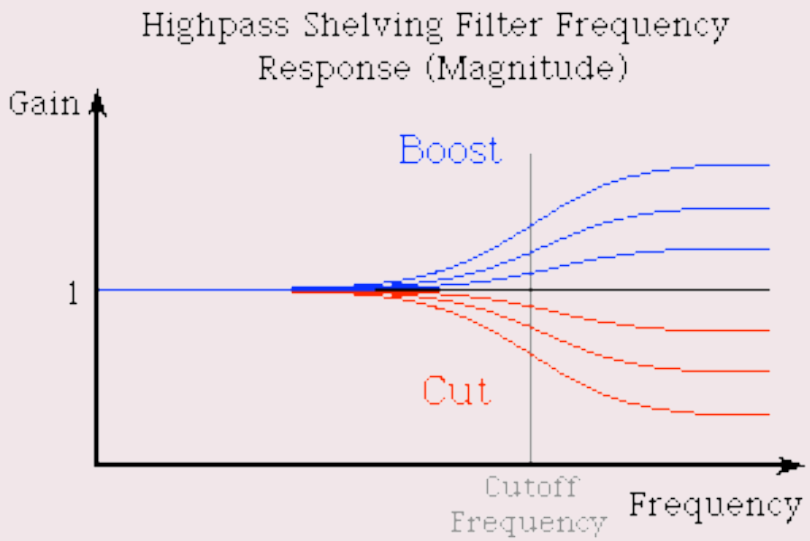
- Built using a series of *shelving* and *peak* filters
- First or second-order filters usually employed.

Shelving and Peak Filters

Shelving Filter:

Boost or **cut** the **low** or **high frequency bands** with a **cut-off frequency**, F_c and gain G:

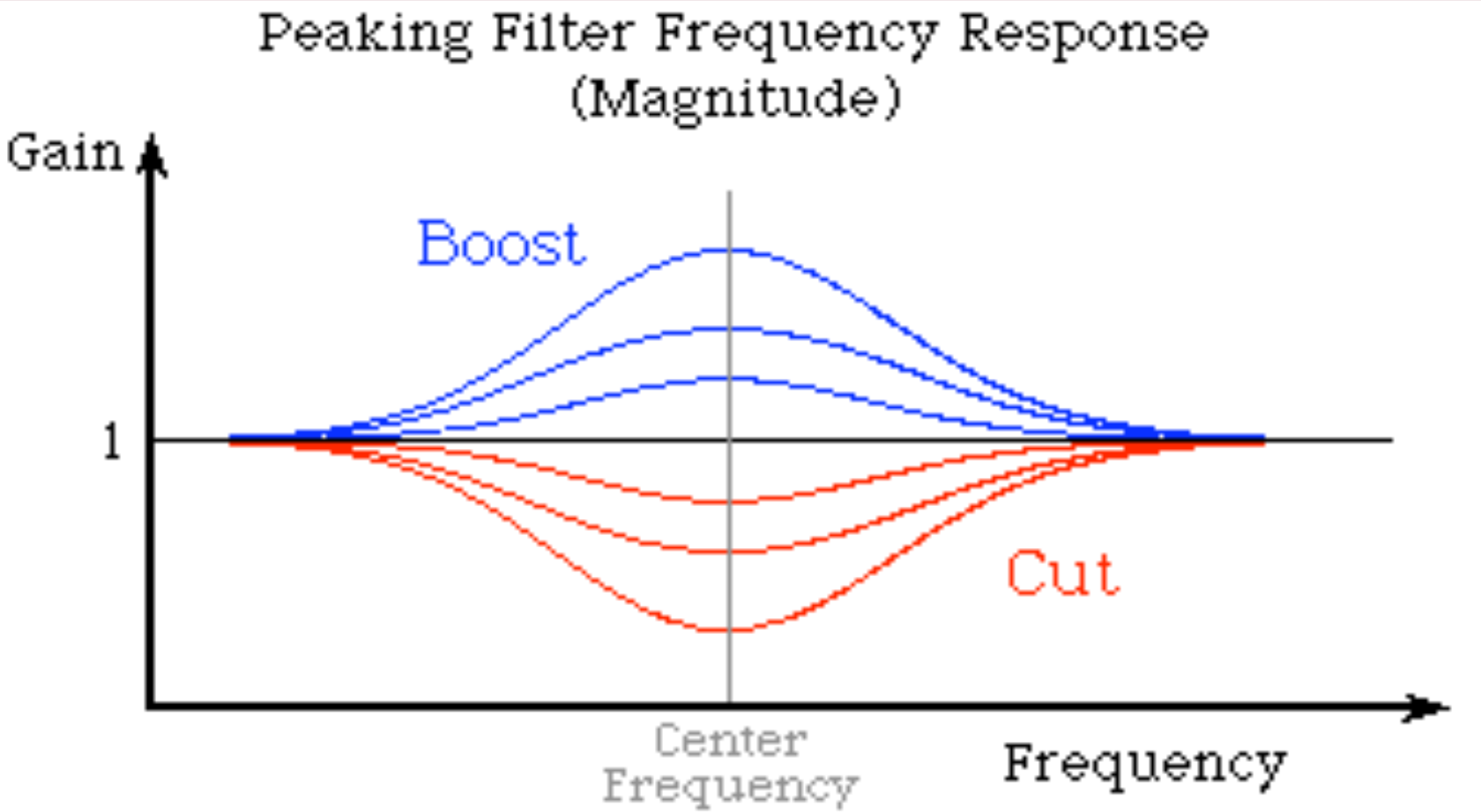




Shelving and Peak Filters (Cont.)

Peak Filter:

Boost or **cut** **mid-frequency** bands with a **cut-off frequency**, F_c, a **bandwidth**, f_b and **gain** G:



Shelving Filters

A First-order Shelving Filter:

Transfer function:

$$H(z) = 1 + \frac{H_0}{2}(1 \pm A(z)) \quad \textbf{where } LF/HF + /-$$

where $A(z)$ is a first-order **allpass** filter — passes all frequencies but modifies phase:

$$A(z) = \frac{z^{-1} + a_{B/C}}{1 + a_{B/C}z^{-1}} \quad \textbf{B=Boost, C=Cut}$$

which leads the following **algorithm**/**difference equation**:

$$\begin{aligned} y_1(n) &= a_{B/C}x(n) + x(n-1) - a_{B/C}y_1(n-1) \\ y(n) &= \frac{H_0}{2}(x(n) \pm y1(n)) + x(n) \end{aligned}$$

Shelving Filters (Cont.)

Shelving Filter Parameters:

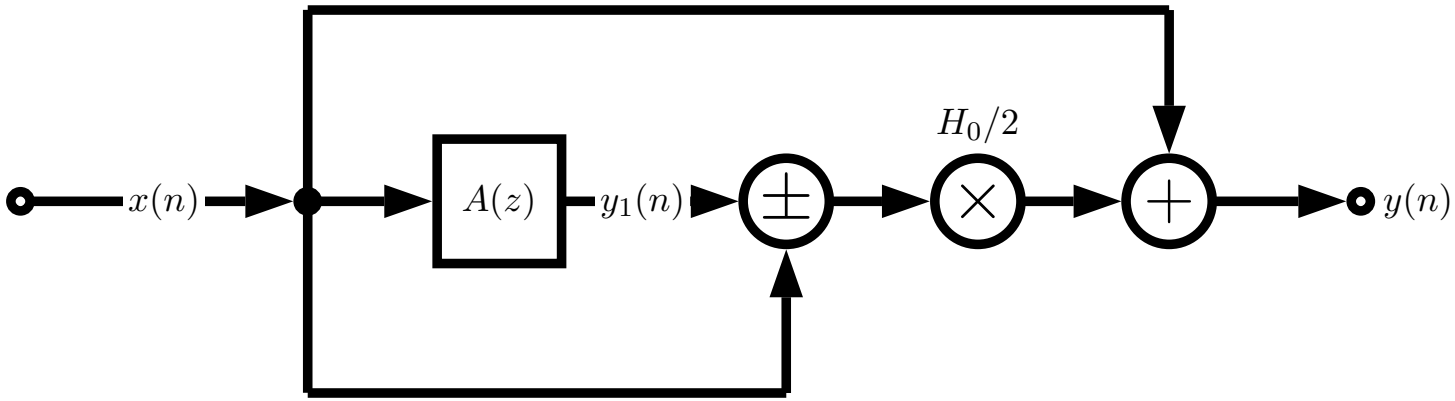
The **gain**, G, in dB can be adjusted accordingly:

$$H_0 = V_0 - 1 \quad \textbf{where } V_0 = 10^{G/20}$$

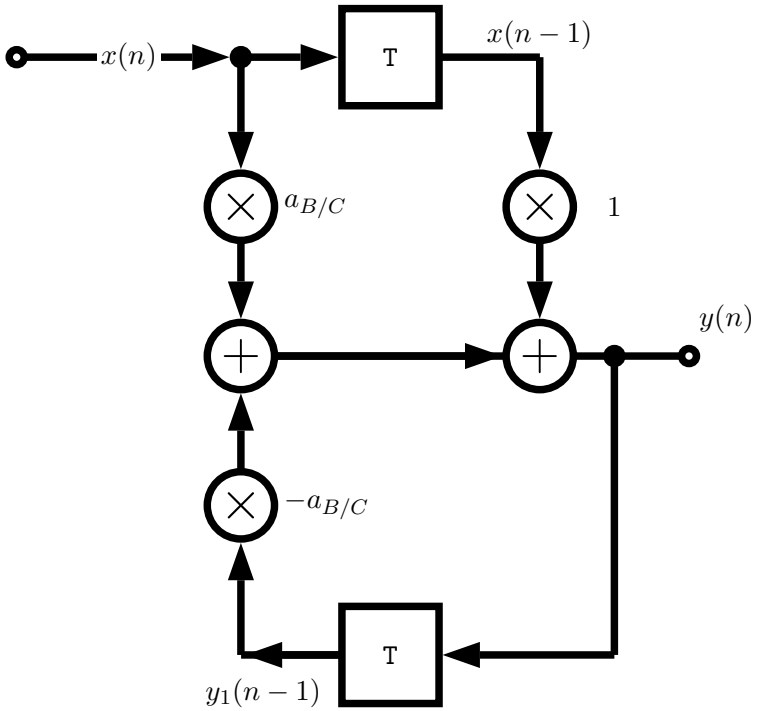
and the cut-off frequency for **boost**, a_B, or **cut**, a_C are given by:

$$\begin{aligned} a_B &= \frac{tan(2\pi f_c/f_s) - 1}{tan(2\pi f_c/f_s) + 1} \\ a_C &= \frac{tan(2\pi f_c/f_s) - V_0}{tan(2\pi f_c/f_s) + V_0} \end{aligned}$$

Shelving Filters Signal Flow Graph



where $A(z)$ is given by:



Peak Filters

A 2nd-order Peak Filter

Transfer function:

$$H(z) = 1 + \frac{H_0}{2}(1 - A_2(z))$$

where $A_2(z)$ is a **second-order allpass** filter:

$$A(z) = \frac{-a_B + (d - da_B)z^{-1} + z^{-2}}{1 + (d - da_B)z^{-1} + a_B z^{-2}}$$

which leads the following algorithm/difference equation:

$$\begin{aligned} y_1(n) &= 1a_{B/C}x(n) + d(1 - a_{B/C})x(n-1) + x(n-2) \\ &\quad -d(1 - a_{B/C})y_1(n-1) + a_{B/C}y_1(n-2) \\ y(n) &= \frac{H_0}{2}(x(n) - y_1(n)) + x(n) \end{aligned}$$

Peak Filters (Cont.)

Peak Filter Parameters:

The **center/cut-off frequency**, d, is given by:

$$d = -cos(2\pi f_c/f_s)$$

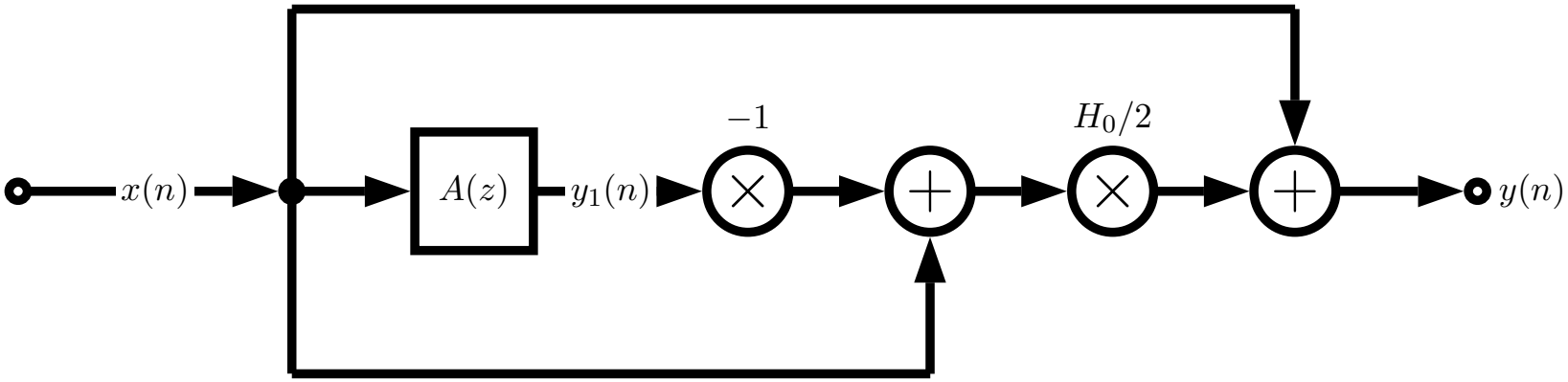
The H_0 by relation to the gain, G, as before:

$$H_0 = V_0 - 1 \quad \textbf{where } V_0 = 10^{G/20}$$

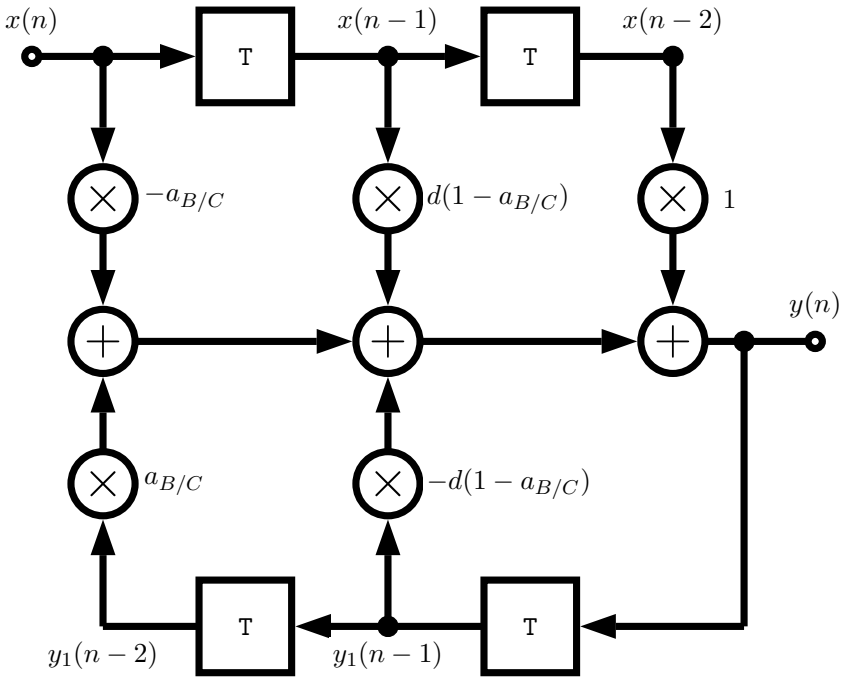
and the bandwidth, f_b is given by the limits for **boost**, a_B, or **cut**, a_C are given by:

$$\begin{aligned} a_B &= \frac{tan(2\pi f_b/f_s) - 1}{tan(2\pi f_b/f_s) + 1} \\ a_C &= \frac{tan(2\pi f_b/f_s) - V_0}{tan(2\pi f_b/f_s) + V_0} \end{aligned}$$

Peak Filters Signal Flow Graph



where $A(z)$ is given by:



Shelving Filter EQ MATLAB Example (1)

shelving.m

```
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
% described in Zolzer's DAFX book (p. 50 -55).
%
% Usage:      [B,A] = shelving(G, Fc, Fs, Q, type);
%
%             G is the logrithmic gain (in dB)
%             FC is the center frequency
%             Fs is the sampling rate
%             Q adjusts the slope be replacing the sqrt(2) term
%             type is a character string defining filter type
%             Choices are: 'Base_Shelf' or 'Treble_Shelf'

% Error Check
if((strcmp(type,'Base_Shelf') ~= 1) && ...
          (strcmp(type,'Treble_Shelf') ~= 1))
    error(['Unsupported Filter Type: ' type]);
end
```

Shelving Filter EQ MATLAB Example (2)

shelving.m cont.

```
K = tan((pi * fc)/fs);
V0 = 10^(G/20);
root2 = 1/Q;

% Invert gain if a cut
if (V0 < 1)
    V0 = 1/V0;
end

%%%%%%%%%%%%%%%%%%%%
%    BASE BOOST
%%%%%%%%%%%%%%%%%%%%
if(( G > 0 ) & (strcmp(type,'Base_Shelf')))

    b0 = (1 + sqrt(V0)*root2*K + V0*K^2) / (1 + root2*K + K^2);
    b1 = (2 * (V0*K^2 - 1) ) / (1 + root2*K + K^2);
    b2 = (1 - sqrt(V0)*root2*K + V0*K^2) / (1 + root2*K + K^2);
    a1 =  (2 * (K^2 - 1) ) / (1 + root2*K + K^2);
    a2 =  (1 - root2*K + K^2) / (1 + root2*K + K^2);
```

Shelving Filter EQ MATLAB Example (3)

shelving.m cont.

```
%%%%%%%%%%%%%%%%%%%%
%     BASE CUT
%%%%%%%%%%%%%%%%%%%%
elseif (( G < 0 ) & (strcmp(type,'Base_Shelf')))

    b0 = (1 + root2*K + K^2) / (1 + root2*sqrt(V0)*K + V0*K^2);
    b1 =  (2 * (K^2 - 1) ) / (1 + root2*sqrt(V0)*K + V0*K^2);
    b2 =  (1 - root2*K + K^2) / (1 + root2*sqrt(V0)*K + V0*K^2);
    a1 =  (2 * (V0*K^2 - 1) ) / (1 + root2*sqrt(V0)*K + V0*K^2);
    a2 = (1 - root2*sqrt(V0)*K + V0*K^2) / ...
              (1 + root2*sqrt(V0)*K + V0*K^2);
```

Shelving Filter EQ MATLAB Example (3)

shelving.m cont.

```
%%%%%%%%%%%%%%%%%%%%
%    TREBLE BOOST
%%%%%%%%%%%%%%%%%%%%
elseif (( G > 0 ) & (strcmp(type,'Treble_Shelf')))

    b0 = (V0 + root2*sqrt(V0)*K + K^2) / (1 + root2*K + K^2);
    b1 =  (2 * (K^2 - V0) ) / (1 + root2*K + K^2);
    b2 = (V0 - root2*sqrt(V0)*K + K^2) / (1 + root2*K + K^2);
    a1 =  (2 * (K^2 - 1) ) / (1 + root2*K + K^2);
    a2 =  (1 - root2*K + K^2) / (1 + root2*K + K^2);
```

Shelving Filter EQ MATLAB Example (4)

shelving.m cont.

```
%%%%%%%%%%%%%%%%%%%%
%   TREBLE CUT
%%%%%%%%%%%%%%%%%%%%

elseif (( G < 0 ) & (strcmp(type,'Treble_Shelf')))

    b0 =   (1 + root2*K + K^2) / (V0 + root2*sqrt(V0)*K + K^2);
    b1 =    (2 * (K^2 - 1) ) / (V0 + root2*sqrt(V0)*K + K^2);
    b2 =    (1 - root2*K + K^2) / (V0 + root2*sqrt(V0)*K + K^2);
    a1 =    (2 * ((K^2)/V0 - 1) ) / (1 + root2/sqrt(V0)*K ...
                + (K^2)/V0);
    a2 = (1 - root2/sqrt(V0)*K + (K^2)/V0) / ....
             (1 + root2/sqrt(V0)*K + (K^2)/V0);

%%%%%%%%%%%%%%%%%%%%
%   All-Pass
%%%%%%%%%%%%%%%%%%%%
else
    b0 = V0;
    b1 = 0; b2 = 0; a1 = 0; a2 = 0;
end

%return values
a = [  1, a1, a2];
b = [ b0, b1, b2];
```

Shelving Filter EQ MATLAB Example (5)

Example use: shelving_eg.m

```
infile = 'acoustic.wav';

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

% 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
audiowrite('out_bassshelf.wav', yb, Fs);

% Plot the original and equalised waveforms
figure(1), hold on;
plot(yb,'b');
plot(x,'r');
title('Bass Shelf Filter Equalised Signal');
```

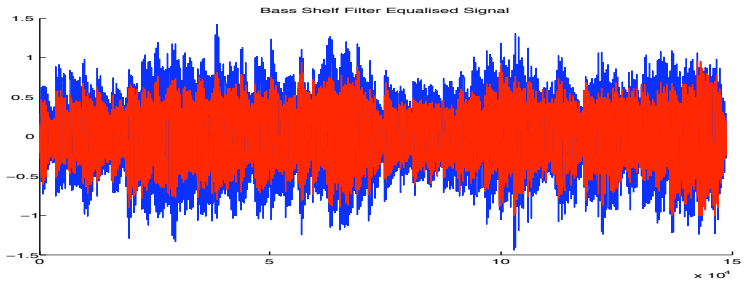
Shelving Filter EQ MATLAB Example (6)

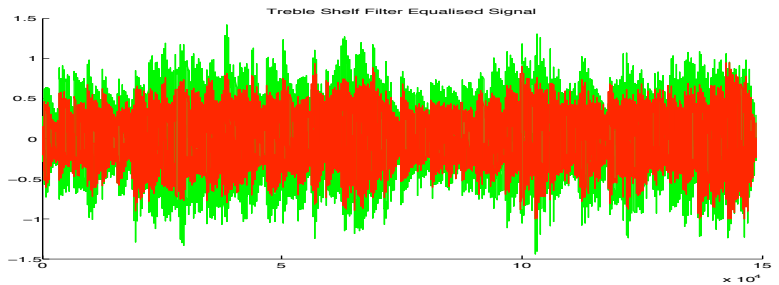
shelving_eg.m cont.

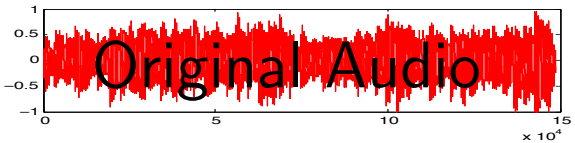
```
% Do treble shelf filter
fct = 600; type = 'Treble_Shelf';

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

% Write output wav files
audiowrite('out_treblehelf.wav', yt, Fs);

figure(1), hold on;
plot(yb,'g');
plot(x,'r');
title('Treble Shelf Filter Equalised Signal');
```

Shelving Filter EQ MATLAB Example Output

The output from the above code is (red plot is original audio):







Click on above images or here to hear: original audio, bass shelf filtered audio, treble shelf filtered audio.

Time-varying Filters

Time-varying Filter Effects

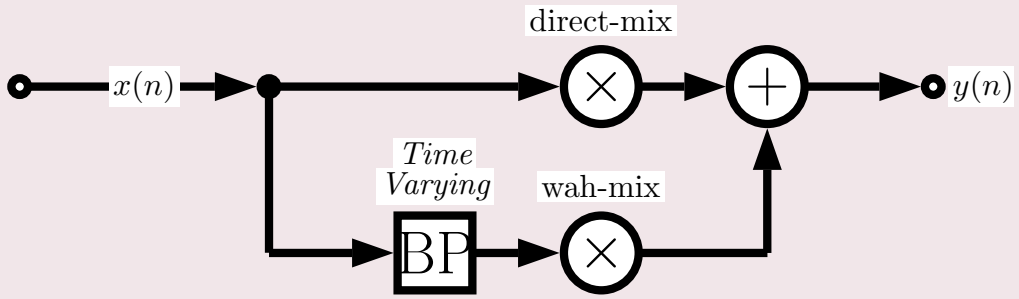
Some common effects are realised by simply time varying a filter in a couple of different ways:

Wah-wah: A bandpass filter with a (**modulated**) time varying centre (resonant) frequency and a small bandwidth. Filtered signal mixed with direct signal.

Phasing: A notch filter, that can be realised as set of cascading IIR filters, again mixed with direct signal.

Wah-wah Example

Wah-wah, Signal flow diagram:



where **BP** is a **time-varying frequency bandpass filter**.

Wah-wah Variations

- A *phaser* is similarly implemented with a **notch filter replacing** the **bandpass filter**.
- A variation is the *M*-**fold wah-wah** filter where M tap delay bandpass filters spread over the entire spectrum change their centre frequencies simultaneously.
  - A **bell effect** can be achieved with around a **hundred** *M* **tap delays** and **narrow bandwidth filters**

Time Varying Filter Implementation: State Variable Filter

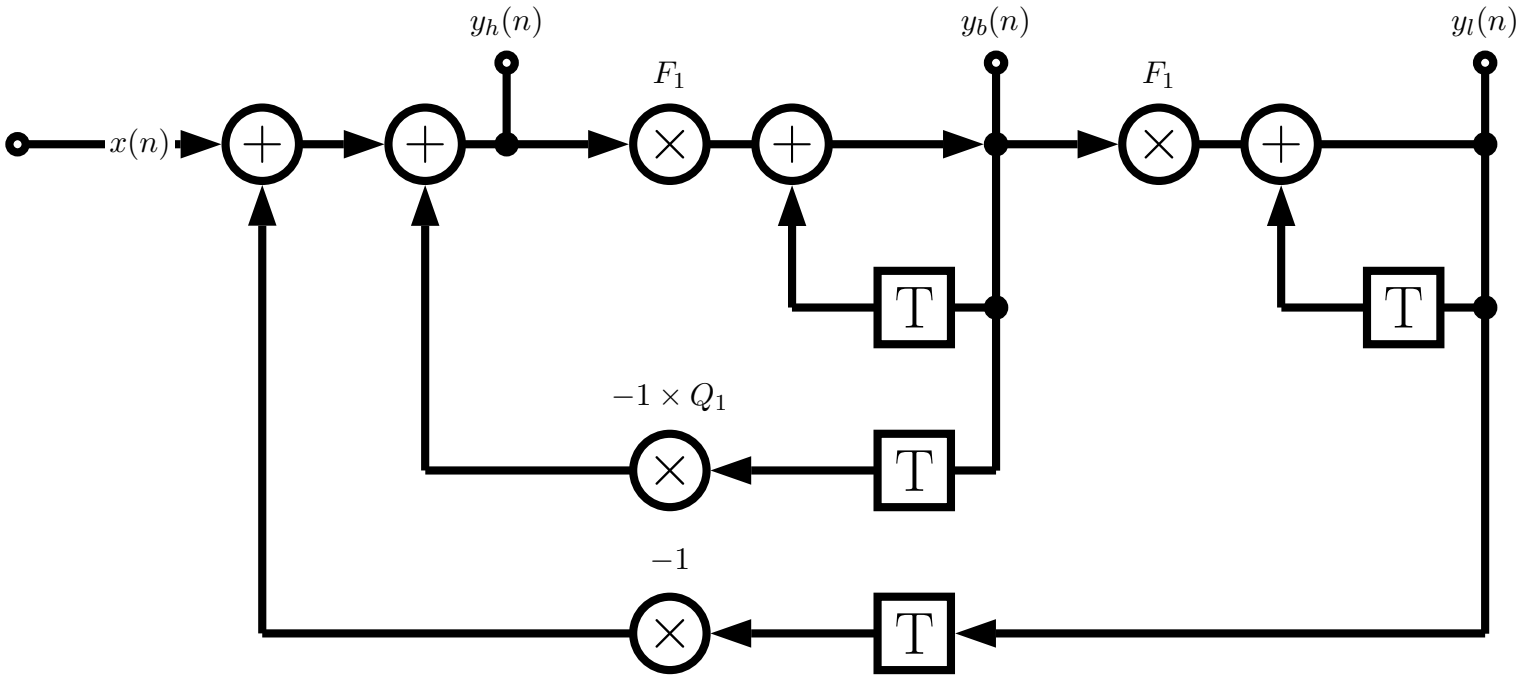
The Practical State Variable Filter

In 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 **simultaneously** get **lowpass**, **bandpass** and **highpass** filter output.

The State Variable Filter



where:

$$x(n) = \textbf{input signal}$$
$$y_l(n) = \textbf{lowpass signal}$$
$$y_b(n) = \textbf{bandpass signal}$$
$$y_h(n) = \textbf{highpass signal}$$

The State Variable Filter Algorithm

State Variable Filter difference equations are given by:

$$\begin{aligned} y_l(n) &= F_1 y_b(n) + y_l(n-1) \\ y_b(n) &= F_1 y_h(n) + y_b(n-1) \\ y_h(n) &= x(n) - y_l(n-1) - Q_1 y_b(n-1) \end{aligned}$$

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

$$F_1 = 2\sin(\pi f_c/f_s), \qquad \textbf{and} \quad Q_1 = 2d$$

MATLAB Wah-wah Implementation

Making a Wah-wah

We simply implement the State Variable Filter with a Sinusoid **Modulated** (**variable**) **frequency**, f_c.

wah_wah.m:

```
% wah_wah.m   state variable band pass
%
% BP filter with narrow pass band, Fc 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
%
% yl(n) = F1*yb(n) + yl(n-1)
% yb(n) = F1*yh(n) + yb(n-1)
% yh(n) = x(n) - yl(n-1) - Q1*yb(n-1)
%
% vary Fc from 500 to 5000 Hz
```

Wah-wah Implementation

wah_wah.m (Cont.):

```
infile = 'acoustic.wav';

% read in wav sample
[ x, Fs] = audioread(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;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Wah-wah Implementation

wah_wah.m (Cont.):

```
% 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));

% 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;
```

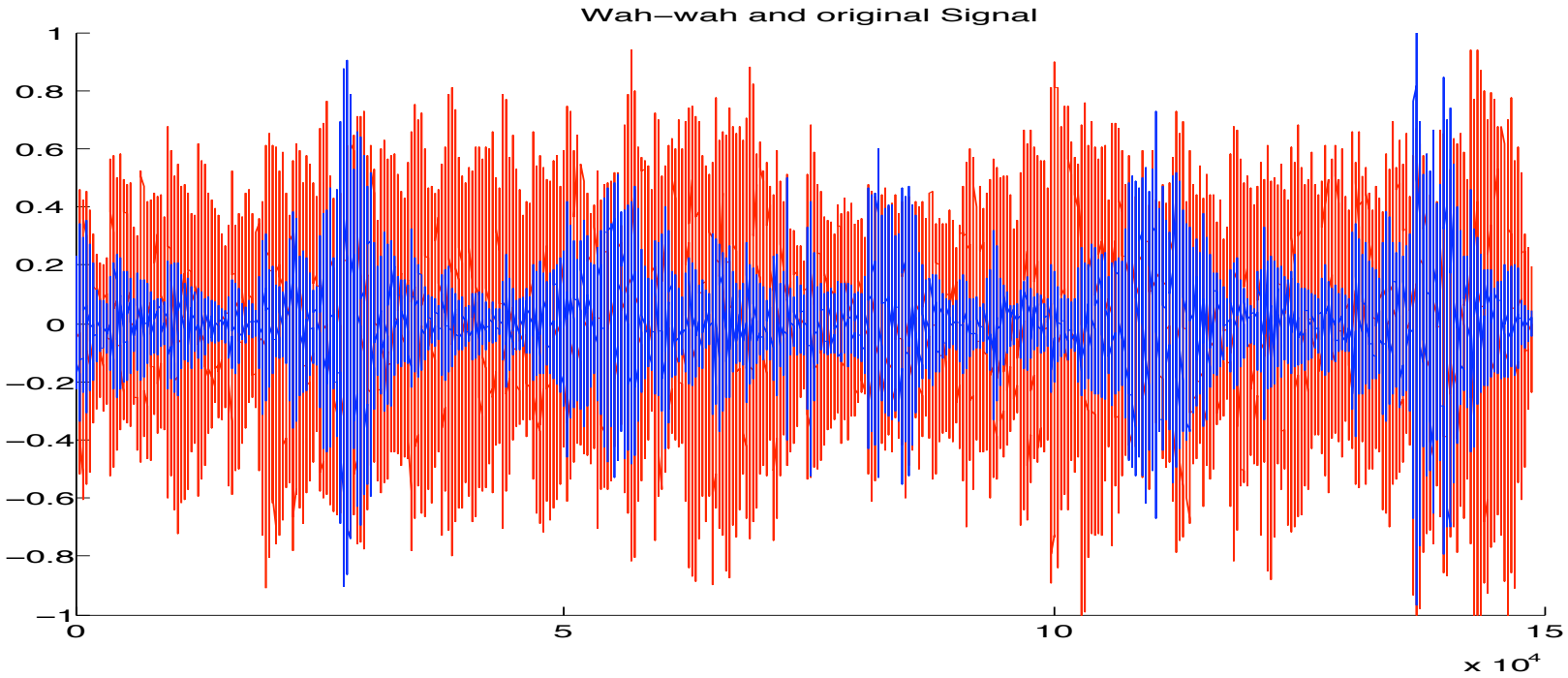
Wah-wah Implementation

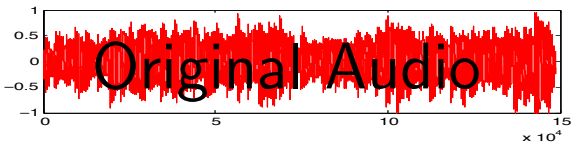
wah_wah.m (Cont.):

```
yh=zeros(size(x));          % create emptly out 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  and  Output .........
```

Wah-wah MATLAB Example (Cont.)

The output from the above code is (red plot is original audio):





Click on images or here to hear: original audio, wah-wah audio.

Delay Based Effects

Many useful audio effects can be implemented using a **delay structure**:

- Sounds reflected off walls
  - In a cave or large room we hear an echo and also **reverberation** takes place – this is a different effect — **see later**
  - If walls are closer together repeated reflections can appear as parallel boundaries and we hear a modification of sound colour instead.
- **Vibrato**, **Flanging**, **Chorus** and **Echo** are examples of delay effects

Basic Delay Structure

The Return of IIR and FIR filters:

We build basic delay structures out of some very basic **IIR** and **FIR** filters:

- We use *FIR* and *IIR comb filters*
- Combination of FIR and IIR gives the **Universal Comb Filter**

FIR Comb Filter

FIR Comb Filter: A single delay

This simulates a **single delay**:

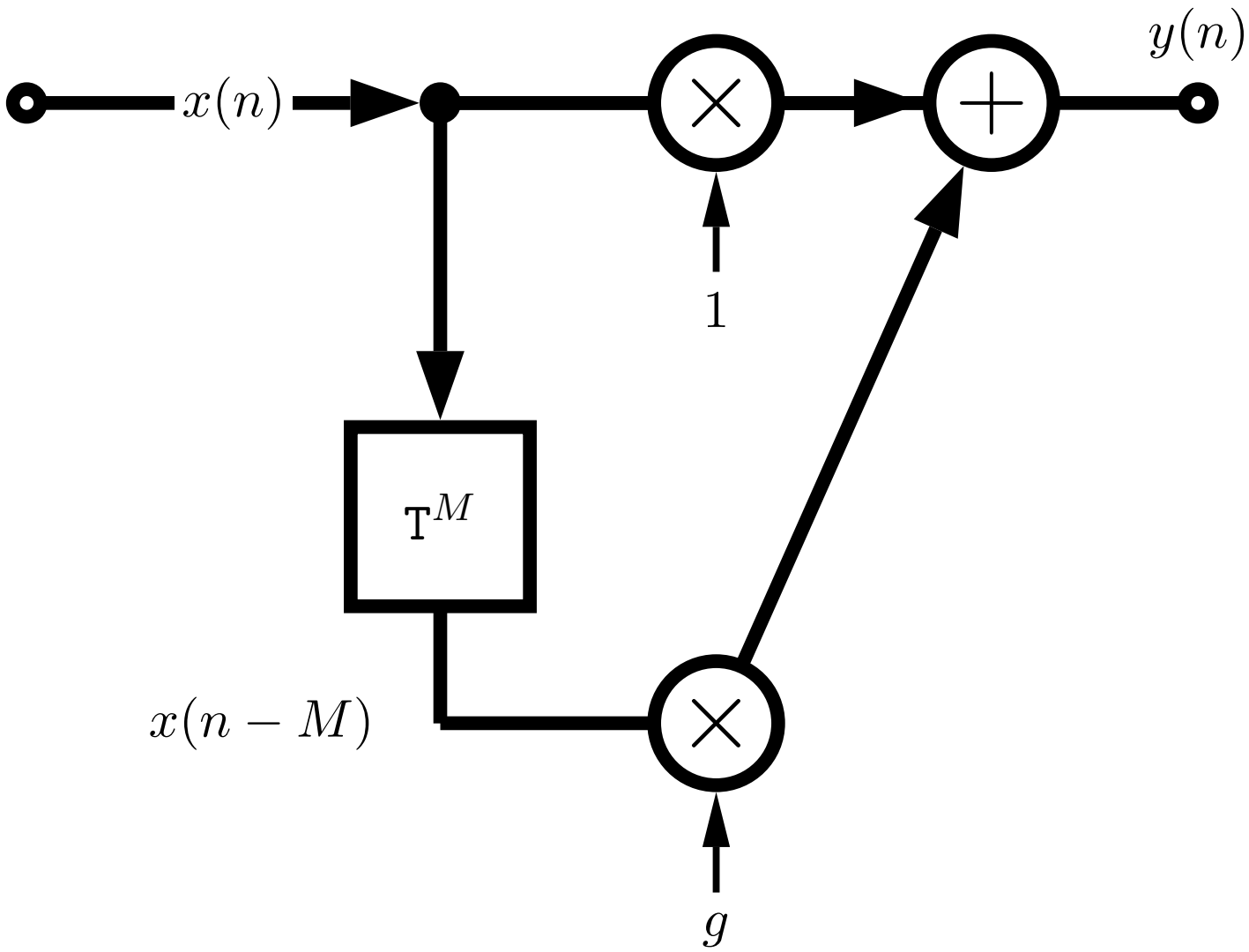
- The input signal is delayed by a given time duration, τ.
- The delayed (processed) signal is added to the input signal some amplitude gain, g
- The difference equation is simply:

$$y(n) = x(n) + gx(n - M) \qquad \text{with} \quad M = \tau / f_s$$

- The transfer function is:

$$H(z) = 1 + gz^{-M}$$

FIR Comb Filter Signal Flow Diagram



FIR Comb Filter MATLAB Code

fircomb.m:

```
x=zeros(100,1);x(1)=1; % unit impulse signal of length 100

g=0.5; %Example gain

Delayline=zeros(10,1); % memory allocation for length 10

for n=1:length(x);
  y(n)=x(n)+g*Delayline(10);
  Delayline=[x(n);Delayline(1:10-1)];
end;
```

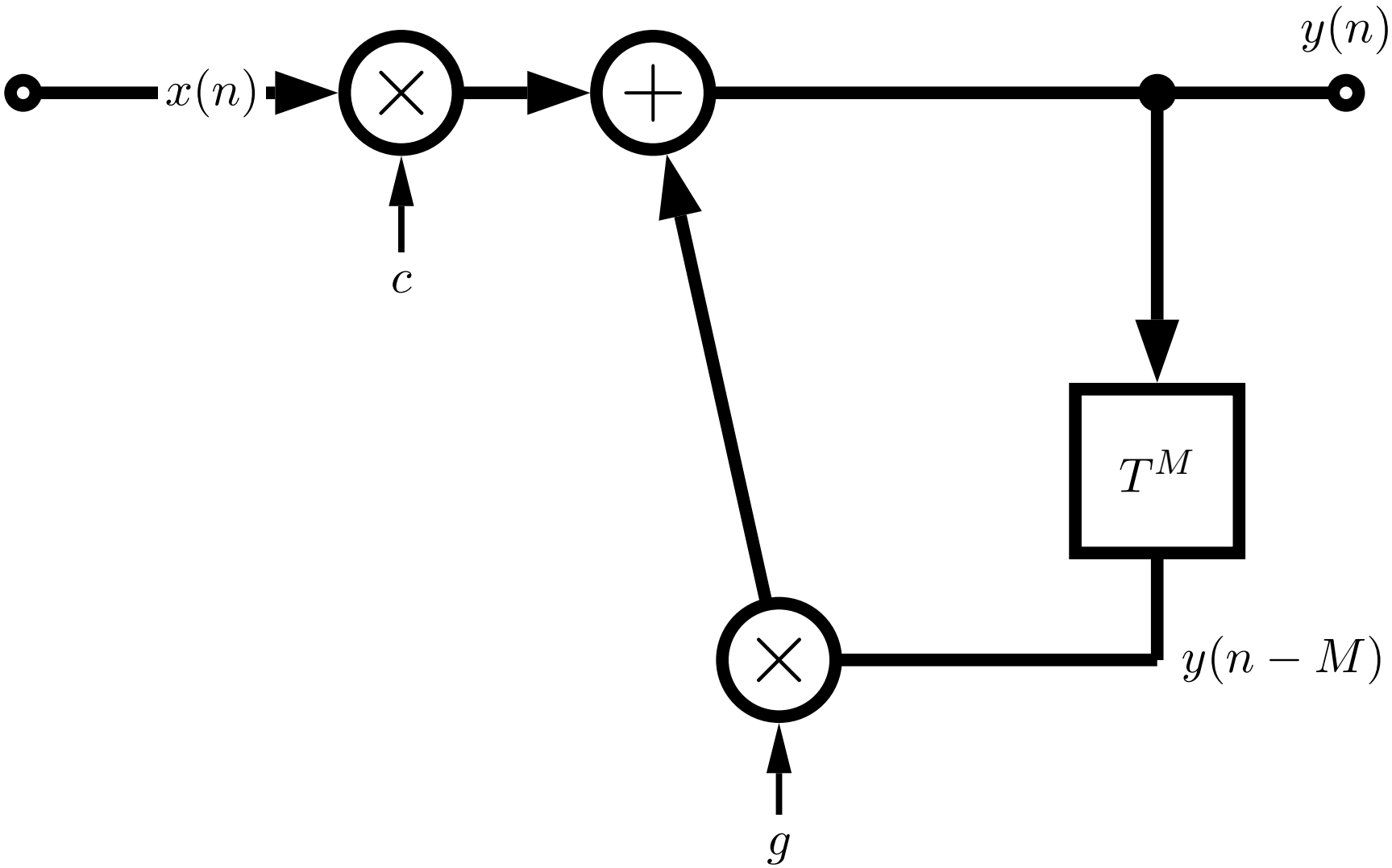
IIR Comb Filter

IIR Comb Filter

- Simulates *endless reflections* at both ends of cylinder.
- We get an endless series of responses, $y(n)$ to input, $x(n)$.
- The input signal circulates in delay line (delay time τ) that is fed back to the input.
- Each time it is fed back it is attenuated by g.
- Input sometime scaled by c to **compensate** for high amplification of the structure.
- The difference equation is simply:

$$y(n) = Cx(n) + gy(n - M) \qquad \textbf{with} \quad M = \tau / f_s$$

IIR Comb Filter Signal Flow Diagram



IIR Comb Filter MATLAB Code

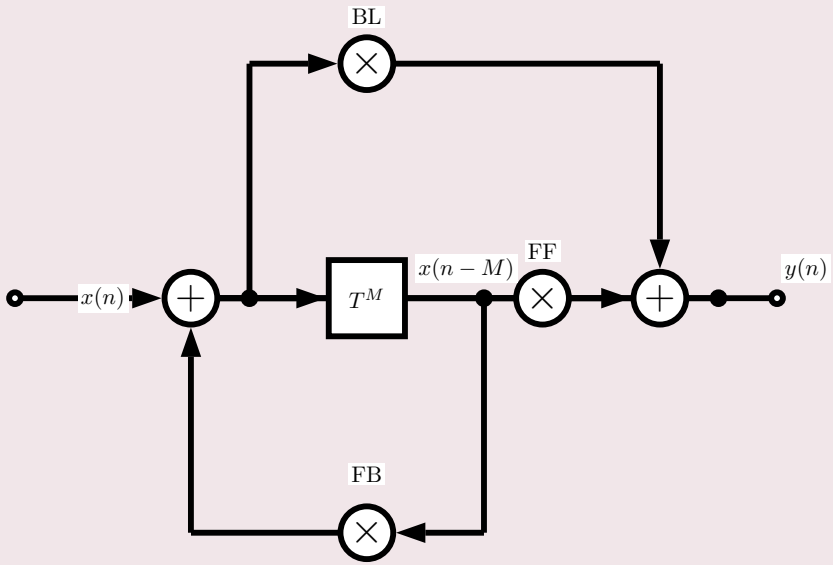
iircomb.m:

```
x=zeros(100,1);x(1)=1; % unit impulse signal of length 100

g=0.5;

Delayline=zeros(10,1); % memory allocation for length 10

for n=1:length(x);
   y(n)=x(n)+g*Delayline(10);
   Delayline=[y(n);Delayline(1:10-1)];
end;
```

Universal Comb Filter

Universal Comb Filter

- Combination of the FIR and IIR comb filters.
- Basically this is an **allpass filter** with an **M** sample delay operator and an additional multiplier, **FF**.



- Parameters:
  FF = **feedforward**, **FB** = **feedbackward**, **BL** = **blend**

Universal Comb Filter Parameters

Why is "Universal"?

- **Universal** in that we can form any **comb** filter, an **allpass** or a delay filter:

	BL	FB	FF
FIR Comb	1	0	g
IIR Comb	1	g	0
Allpass	a	$-a$	1
delay	0	0	1

Universal Comb Filter MATLAB Code

unicomb.m:

```
x=zeros(100,1);x(1)=1; % unit impulse signal of length 100

BL=0.5;
FB=-0.5;
FF=1;
M=10;

Delayline=zeros(M,1); % memory allocation for length 10

for n=1:length(x);
  xh=x(n)+FB*Delayline(M);
  y(n)=FF*Delayline(M)+BL*xh;
  Delayline=[xh;Delayline(1:M-1)];
end;
```

Vibrato - A Simple Delay Based Effect

Vibrato:

- **Vibrato** — **Varying** (**modulating**) 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 = **5–14**Hz.

Vibrato MATLAB Code

vibrato.m function:

- See vibrato_eg.m for sample call this function

```
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
```

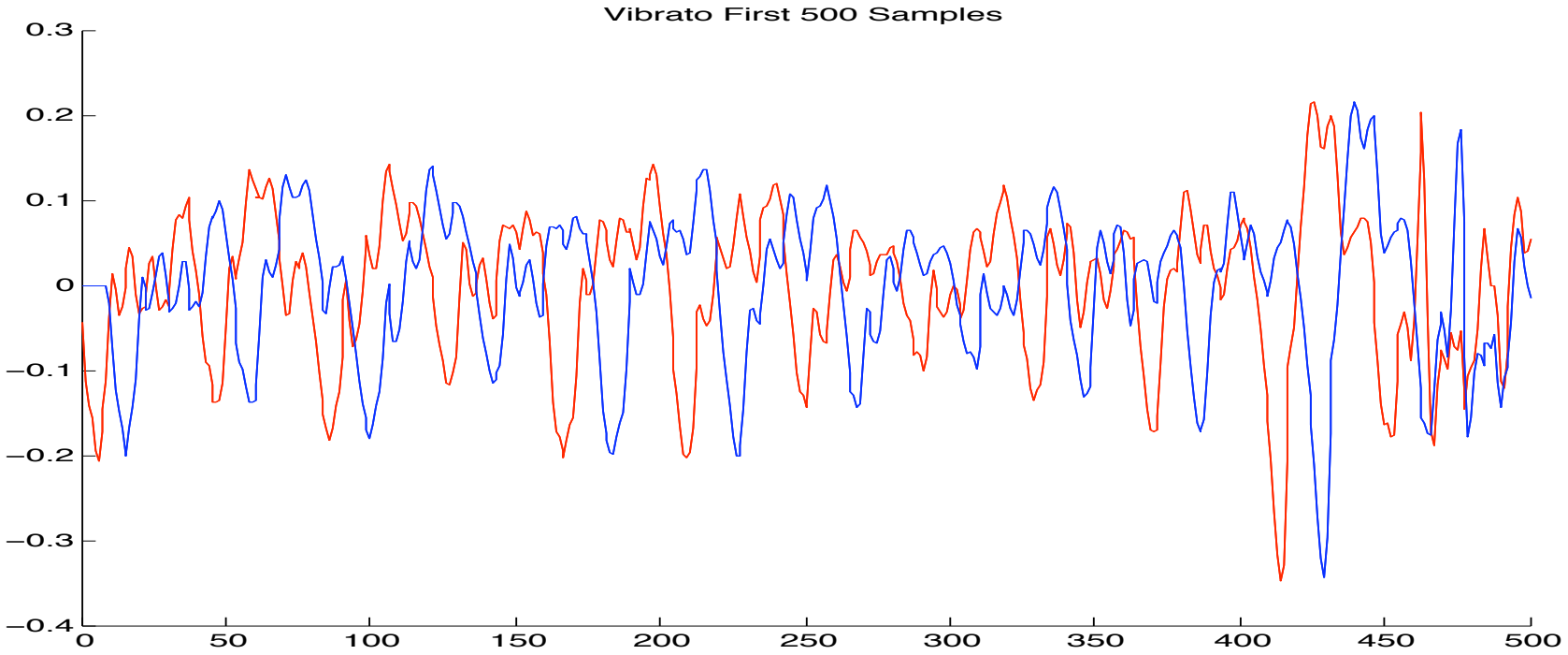
Vibrato MATLAB Code (Cont.)

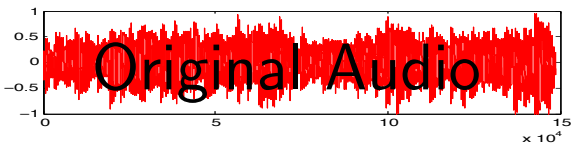
vibrato.m (Cont.)

```
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 Example (Cont.)

The output from the above code is (red plot is original audio):





Click image or here to hear: original audio, vibrato audio.

Comb Filter Delay Effects: Flanger, Chorus, Slapback, Echo

- A few other popular effects can be made with a comb filter (FIR or IIR) and some modulation.
- Flanger, Chorus, Slapback, Echo same basic approach but *different sound* outputs:

Effect	Delay Range (ms)	Modulation
Resonator	$0 \ldots 20$	None
Flanger	$0 \ldots 15$	Sinusoidal (≈ 1 Hz)
Chorus	$10 \ldots 25$	Random
Slapback	$25 \ldots 50$	None
Echo	> 50	None

- **Slapback** (or doubling) — quick repetition of the sound,
  Flanging — continuously varying LFO of delay,
  Chorus — **multiple copies** of sound delayed by small random delays

Flanger MATLAB Code

flanger.m:

```
% Creates a single FIR delay  with the delay time oscillating from
%  Either 0-3 ms or 0-15 ms at 0.1 - 5 Hz

infile='acoustic.wav';
outfile='out_flanger.wav';

% read the sample waveform
[x,Fs] = audioread(infile);

% parameters to vary the effect %
max_time_delay=0.003; % 3ms max delay in seconds
rate=1; %rate of flange in Hz

index=1:length(x);

% sin reference to create oscillating delay
sin_ref = (sin(2*pi*index*(rate/Fs)))';

%convert delay in ms to max delay in samples
max_samp_delay=round(max_time_delay*Fs);
```

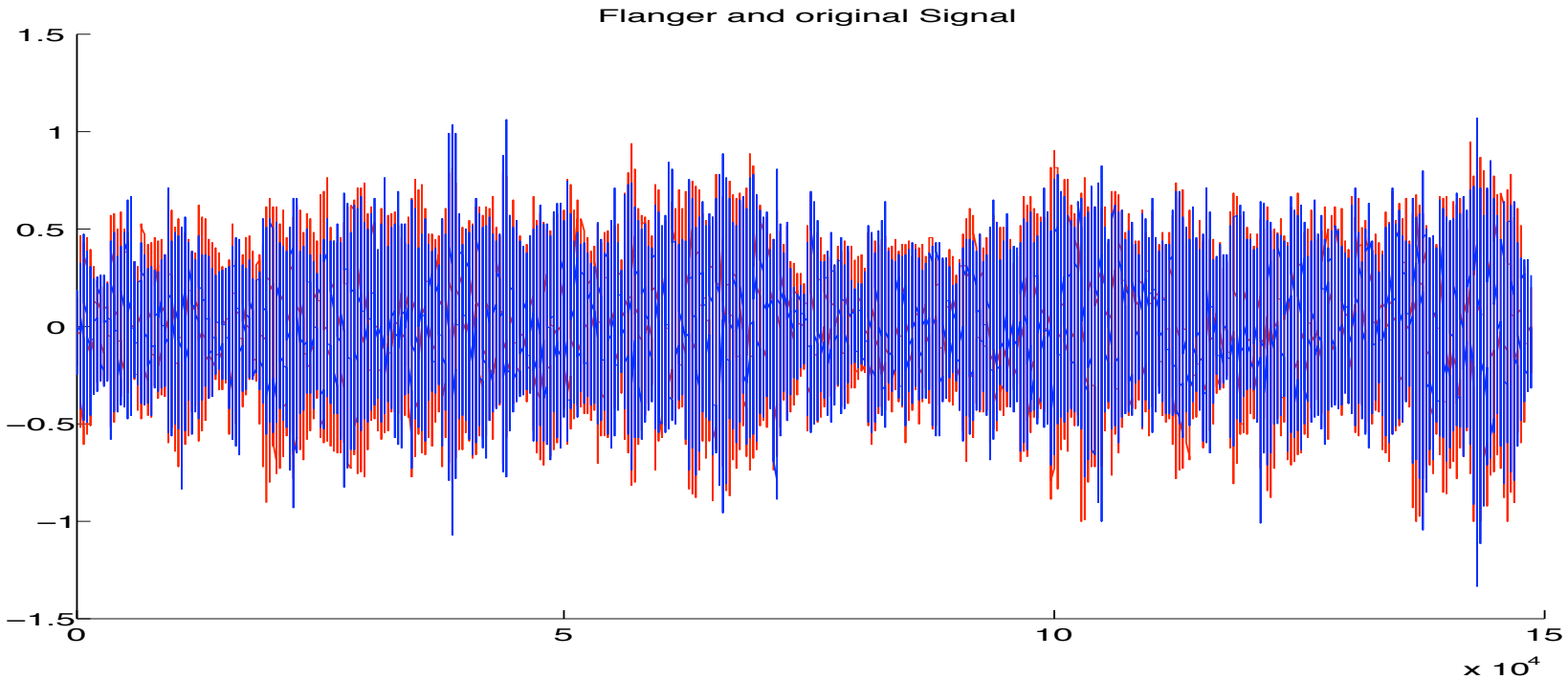
Flanger MATLAB Code (Cont.)

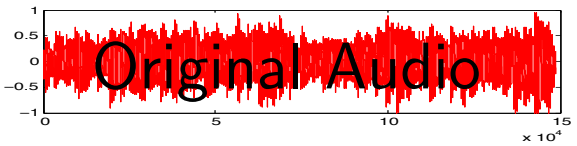
flanger.m (Cont.):

```
% create empty out vector
y = zeros(length(x),1);

% to avoid referencing of negative samples
y(1:max_samp_delay)=x(1:max_samp_delay);

% set amp suggested coefficient from page 71 DAFX
amp=0.7;

% for each sample
for i = (max_samp_delay+1):length(x),
  cur_sin=abs(sin_ref(i));    %abs of current sin val 0-1
  % generate delay from 1-max_samp_delay and ensure whole number
  cur_delay=ceil(cur_sin*max_samp_delay);
  % add delayed sample
  y(i) = (amp*x(i)) + amp*(x(i-cur_delay));
end

% write output
audiowrite(outfile, y, Fs);
```

Flanger MATLAB Example (Cont.)

The output from the above code is (red plot is original audio):





Click here to hear: original audio, flanged audio.

Modulation

Modulation:

The process where parameters of a sinusoidal signal (amplitude, frequency and phase) are modified or varied by an audio signal.

We have met some example effects that could be considered as a class of modulation already:

Amplitude Modulation: Wah-wah, Phaser

Frequency Modulation: Audio synthesis technique

Phase Modulation: Vibrato, Chorus, Flanger

We will now look at some other Modulation effects.

Ring Modulation

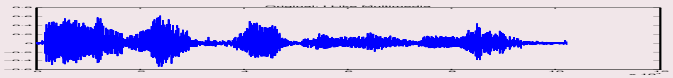
Ring modulation (RM)

RM is where the audio *modulator* signal, $x(n)$ is **multiplied** by a sine wave, $m(n)$, with a *carrier* frequency, f_c.

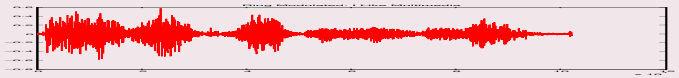
- This is very simple to implement digitally:

$$y(n) = x(n).m(n)$$

- Although audible result is easy to comprehend for simple signals things get more complicated for signals having numerous partials
- If the **modulator** is also a sine wave with frequency, f_x then one hears the sum and difference frequencies: $f_c + f_x$ and $f_c - f_x$, for example.
- When the input is *periodic* with at a **fundamental** frequency, f_0, then a spectrum with amplitude lines at frequencies $|kf_0 \pm f_c|$.
- Used to create robotic speech effects on old sci-fi movies and can create some odd almost non-musical effects if not used with care. (Original speech). *ring_modllikeMM.m code here*



Original Signal



Ring Modulated Signal (Robotic)

MATLAB Ring Modulation

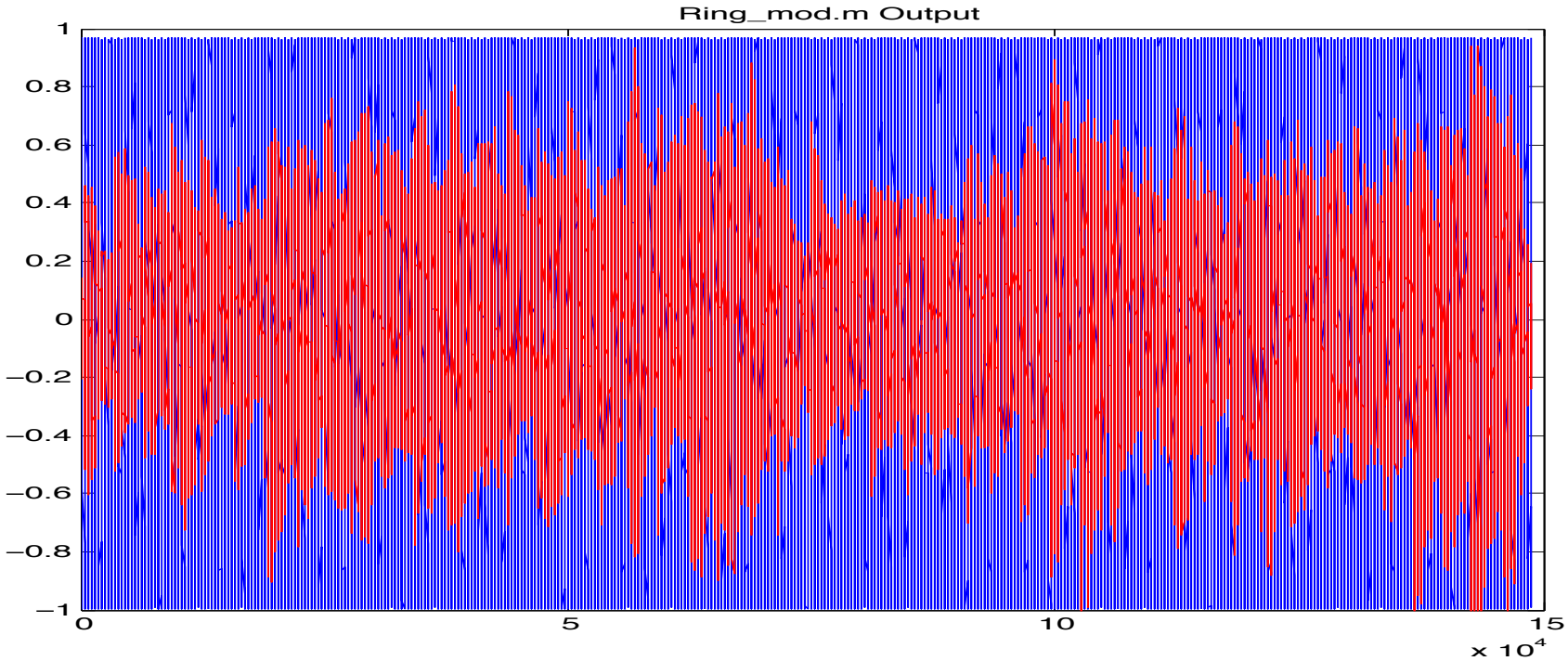
Two examples

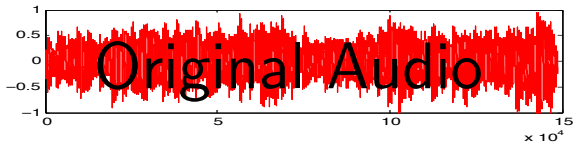
An audio sample and a sine wave being modulated by a sine wave.

Example 1: Audio RM, ring_mod.m

```
% read the sample waveform
[x,Fs] = audioread('acoustic.wav');

index = 1:length(x);

% Ring Modulate with a sine wave frequency Fc
Fc = 440;
carrier= sin(2*pi*index*(Fc/Fs))';

% Do Ring Modulation
y = x.*carrier;

% write output
audiowrite('out_ringmod.wav', y,Fs);
```

Example 1: Audio RM Output





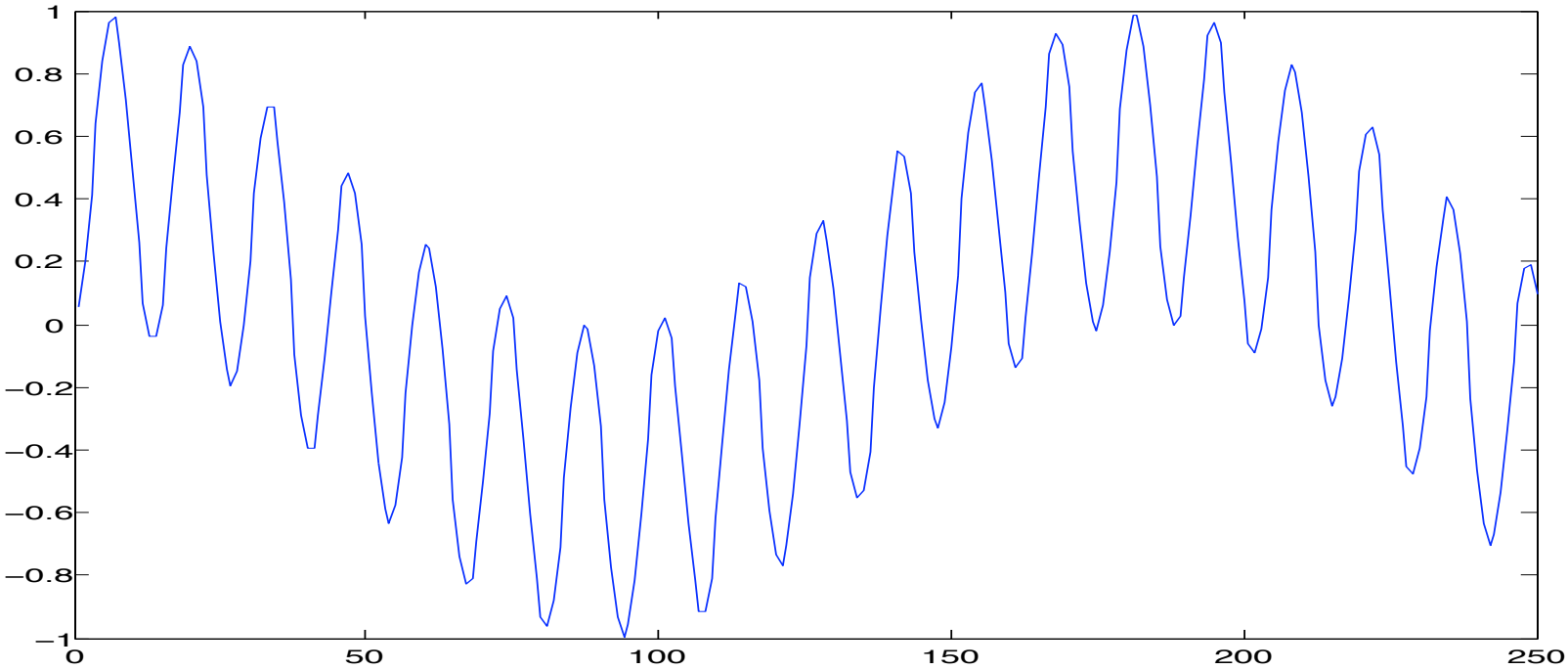
Click image or here to hear: original audio, ring modulated audio.

MATLAB Ring Modulation: Two sine waves

Example 2: Two sine waves RM ring_mod_2sine.m

```
% Ring Modulate with a sine wave frequency Fc
Fc = 440;
carrier= sin(2*pi*index*(Fc/Fs))';

%create a modulator sine wave frequency Fx
Fx = 200;
modulator = sin(2*pi*index*(Fx/Fs))';

% Ring Modulate with sine wave, freq. Fc
y = modulator.*carrier;

% write output
audiowrite('twosine_ringmod.wav', y,Fs);
```

Example 2: Two Sine RM Output



Output of Two sine wave ring modulation ($f_c = 440$, $f_x = 380$)

Click image or here to hear:

Two RM sine waves ($f_c = 440$, $f_x = 200$)

Amplitude Modulation

Amplitude Modulation (AM)

AM is defined by:

$$y(n) = (1 + \alpha m(n)).x(n)$$

- Normalise the peak amplitude of $m(n)$ to 1.
- α is *depth of modulation*

  $\alpha = 1$ gives maximum modulation
  $\alpha = 0$ tuns off modulation
- $x(n)$ is the audio **carrier** signal
- $m(n)$ **is** a low-frequency oscillator **modulator**.
- When $x(n)$ and $m(n)$ **both** sine waves with frequencies f_c and f_x respectively we here **three** frequencies: carrier, difference and sum: $f_c, f_c - f_x, f_c + f_x$.

Amplitude Modulation: Tremolo

AM Example: tremolo

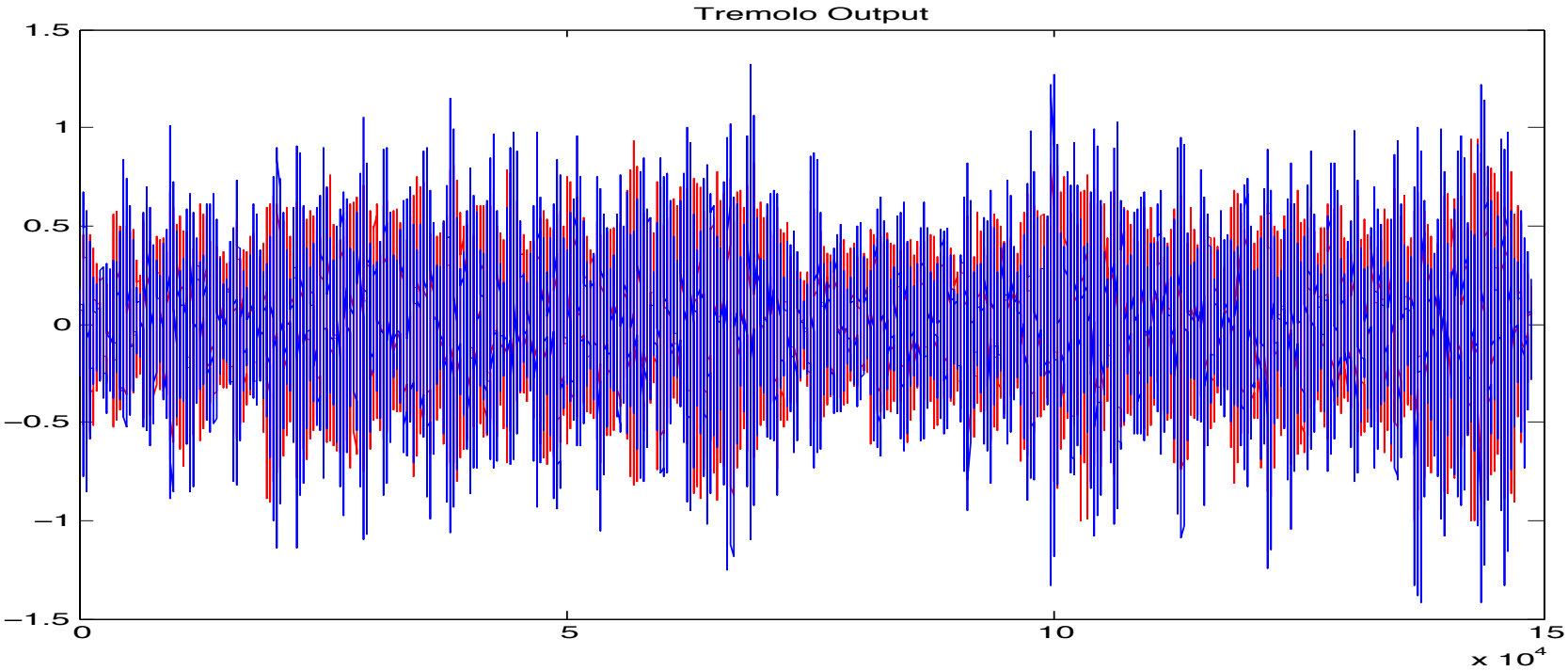
Modulate the amplitude:

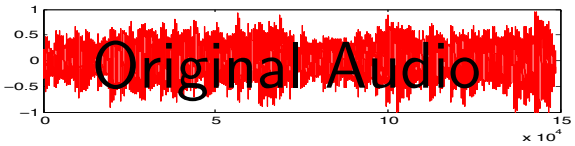
- Set modulation frequency of a sine wave to below 20Hz.

tremolo1.m

```
filename='acoustic.wav';% read the sample waveform
[x,Fs] = audioread(filename);

index = 1:length(x);

Fc = 5;
alpha = 0.5;

trem=(1+ alpha*sin(2*pi*index*(Fc/Fs)))';
y = trem.*x;

% write output
audiowrite('out_tremolo1.wav', y,Fs);
```

Amplitude Modulation: Tremolo Output





Click image or here to hear: original audio, AM tremolo audio.

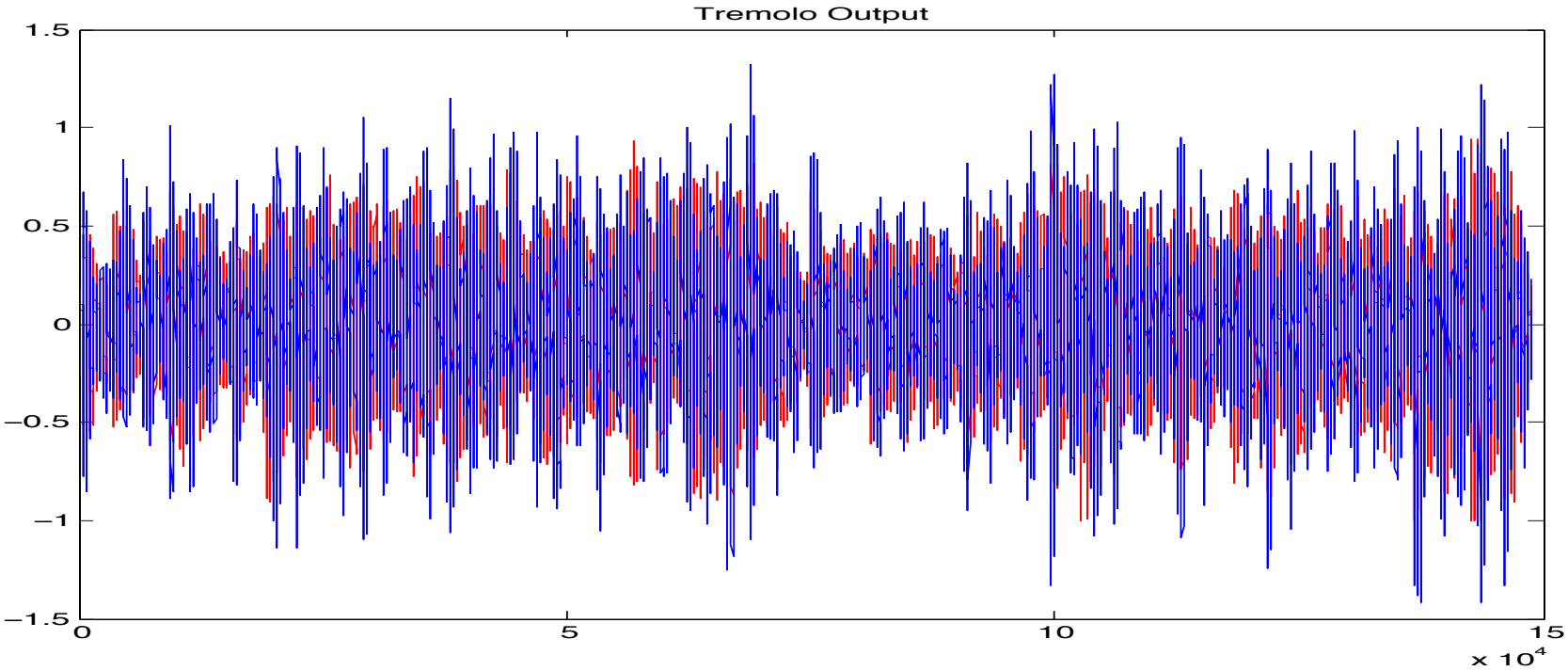
Tremolo via Ring Modulation

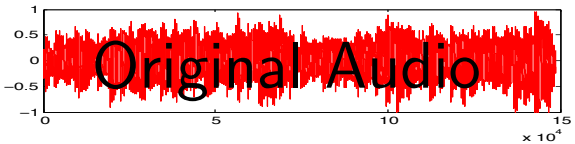
tremolo2.m

If you ring modulate with a **triangular wave** (or try another waveform) you can get **tremolo via RM**.

```
% read the sample waveform
filename='acoustic.wav';
[x,Fs] = audioread(filename);

% create triangular wave LFO
delta=5e-4;
minf=-0.5;
maxf=0.5;

trem=minf:delta:maxf;
while(length(trem) < length(x) )
   trem=[trem (maxf:-delta:minf)];
   trem=[trem (minf:delta:maxf)];
end

%trim trem
trem = trem(1:length(x))';
```

Tremolo via Ring Modulation Output





Click here to hear: original audio, RM tremolo audio.

Non-linear Processing

Non-linear Processors:

Characterised by the fact that they create (intentional or unintentional) harmonic and inharmonic frequency components **not present** in the original signal.

Three major categories of non-linear processing:

Dynamic Processing: control of signal envelope — aim to minimise harmonic distortion. Examples: **Compressors**, **Limiters**

Intentional non-linear harmonic processing: Aim to introduce strong harmonic distortion. Examples: Many electric guitar effects such as **distortion**

Exciters/Enhancers: add additional harmonics for subtle sound improvement.

Limiter

Limiter:

A device that **controls high peaks** in a signal but aims to change the dynamics of the main signal as little as possible:

- A limiter makes use of a peak level measurement and aims to react very quickly to **scale** the level if it is above some threshold.
- By lowering peaks the **overall signal** can be **boosted**.
- Limiting is used **not only** on single instrument but on **final** (multichannel) audio for CD mastering, radio broadcast *etc.*

MATLAB Limiter Example

limiter.m:

- The following code creates a modulated sine wave and then limits the amplitude when it exceeds some threshold.

```
% Create a sine wave with amplitude
% reduced for half its duration

anzahl=220;
for n=1:anzahl,
   x(n)=0.2*sin(n/5);
end;
for n=anzahl+1:2*anzahl;
   x(n)=sin(n/5);
end;
```

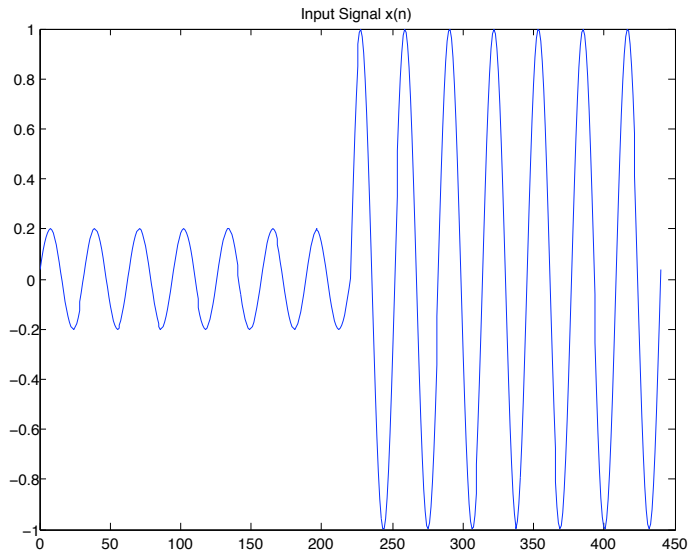
MATLAB Limiter Example (Cont.)

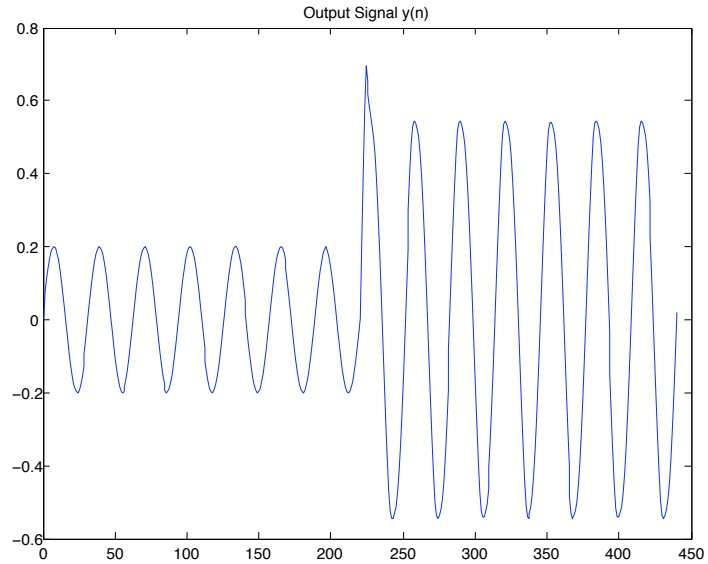
limiter.m (Cont.) :

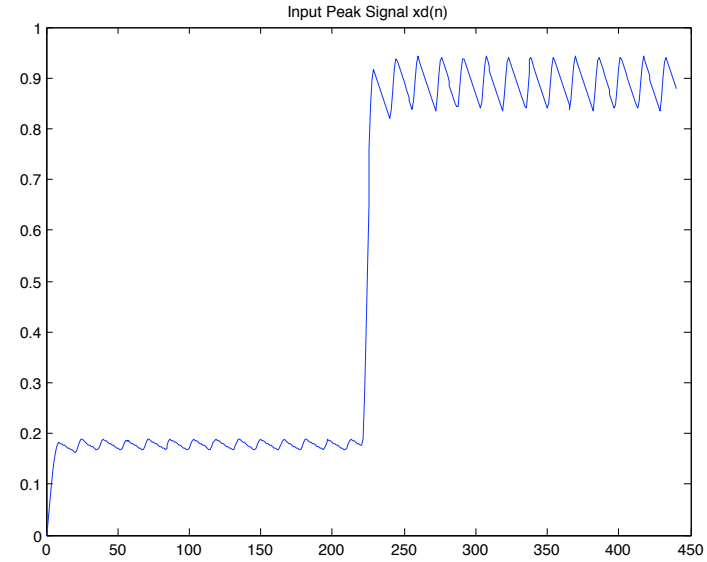
```
% do Limiter

slope=1;
tresh=0.5;
rt=0.01;
at=0.4;

xd(1)=0; % Records Peaks in x
for n=2:2*anzahl;
  a=abs(x(n))-xd(n-1);
  if a<0, a=0; end;
    xd(n)=xd(n-1)*(1-rt)+at*a;
    if xd(n)>tresh,
      f(n)=10^(-slope*(log10(xd(n))-log10(tresh)));
      % linear calculation of f=10^(-LS*(X-LT))
    else f(n)=1;
  end;
  y(n)=x(n)*f(n);
end;
```

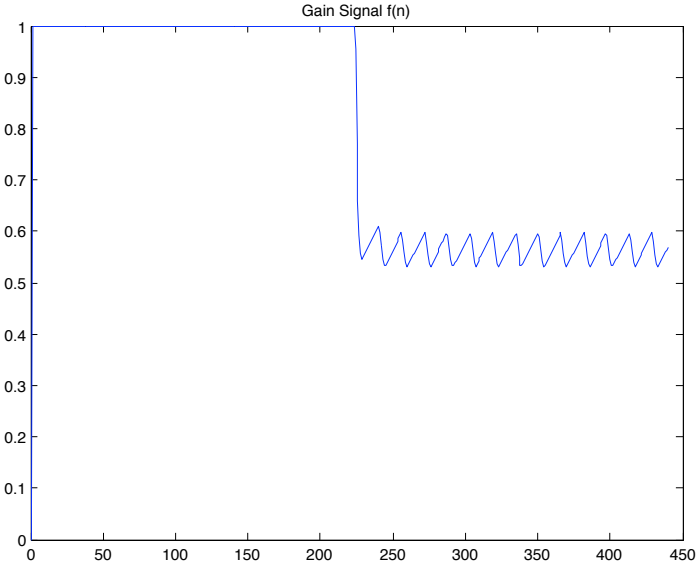
MATLAB Limiter Example Output

Display of the signals from the above limiter example:









Compressors/Expanders

Compressors:

Devices used to reduce the dynamics of the input signal:

- Quiet parts are **modified**.
- Loud parts are reduced according to some static curve.
- A bit like a limiter and used again to boost overall signals in mastering or other applications.
- Often, used on vocals and guitar effects.

Expanders:

Devices that operate on **low signal levels** and **boost** the dynamics in these signals.

- Used to create a more **lively** sound characteristic.

MATLAB Compressor/Expander

compexp.m:

```
function y=compexp(x,comp,release,attack,a,Fs)
% Compressor/expander
% comp          - compression: 0>comp>-1, expansion: 0<comp<1
% a                       - filter parameter <1
h=filter([(1-a)^2],[1.0000 -2*a a^2],abs(x));
h=h/max(h);
h=h.^comp;
y=x.*h;
y=y*max(abs(x))/max(abs(y));
```

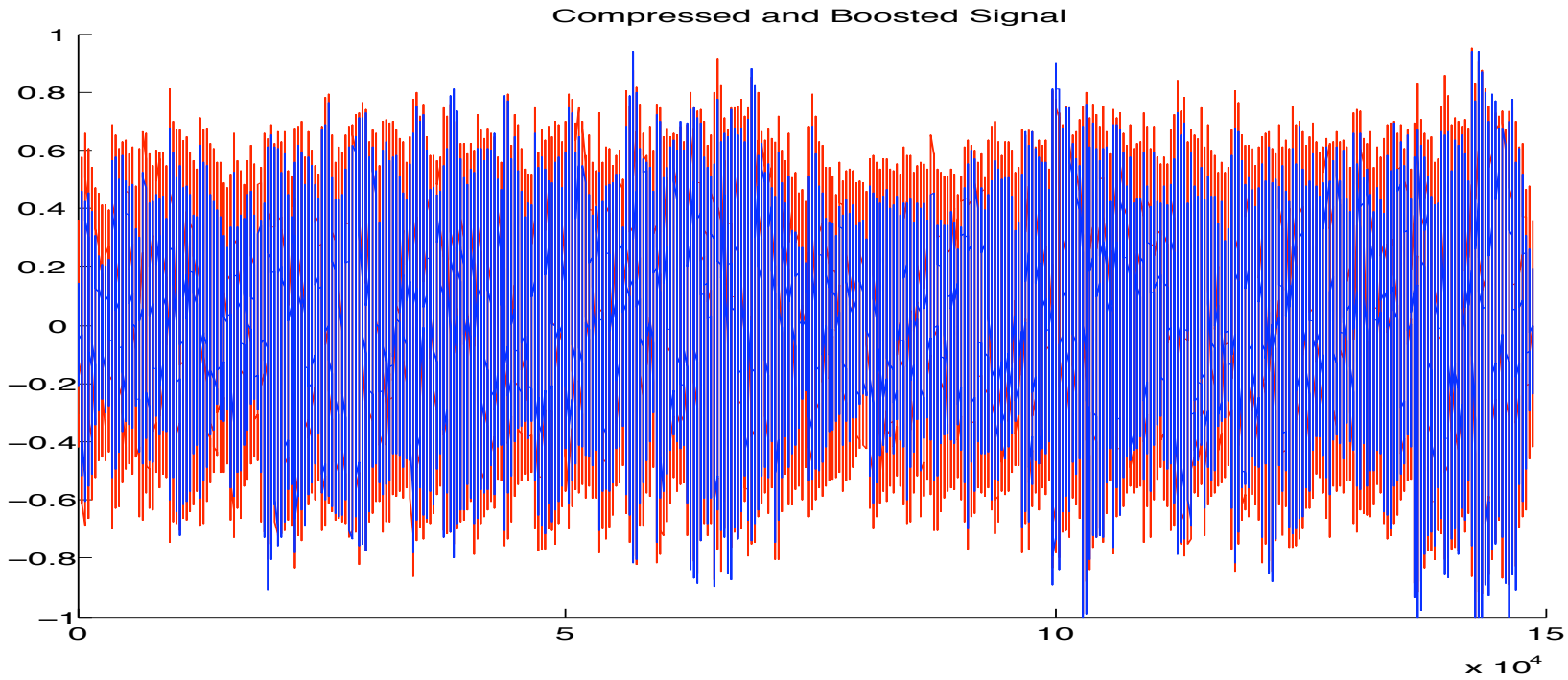
MATLAB Compressor/Expander (Cont.)

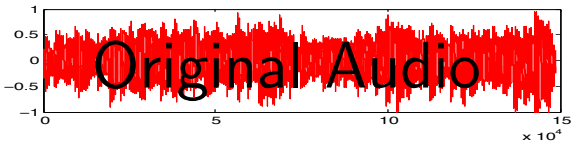
Example call: compression_eg.m:

```
% read the sample waveform
filename='acoustic.wav';
[x,Fs] = audioread(filename);

comp = -0.5; %set compressor
a = 0.5;
y = compexp(x,comp,a,Fs);

% write output
audiowrite('out_compression.wav', y,Fs,bits);

figure(1);
hold on
plot(y,'r');
plot(x,'b');
title('Compressed and Boosted Signal');
```

MATLAB Compressor Output

A **compressed signal** looks and sounds like this:





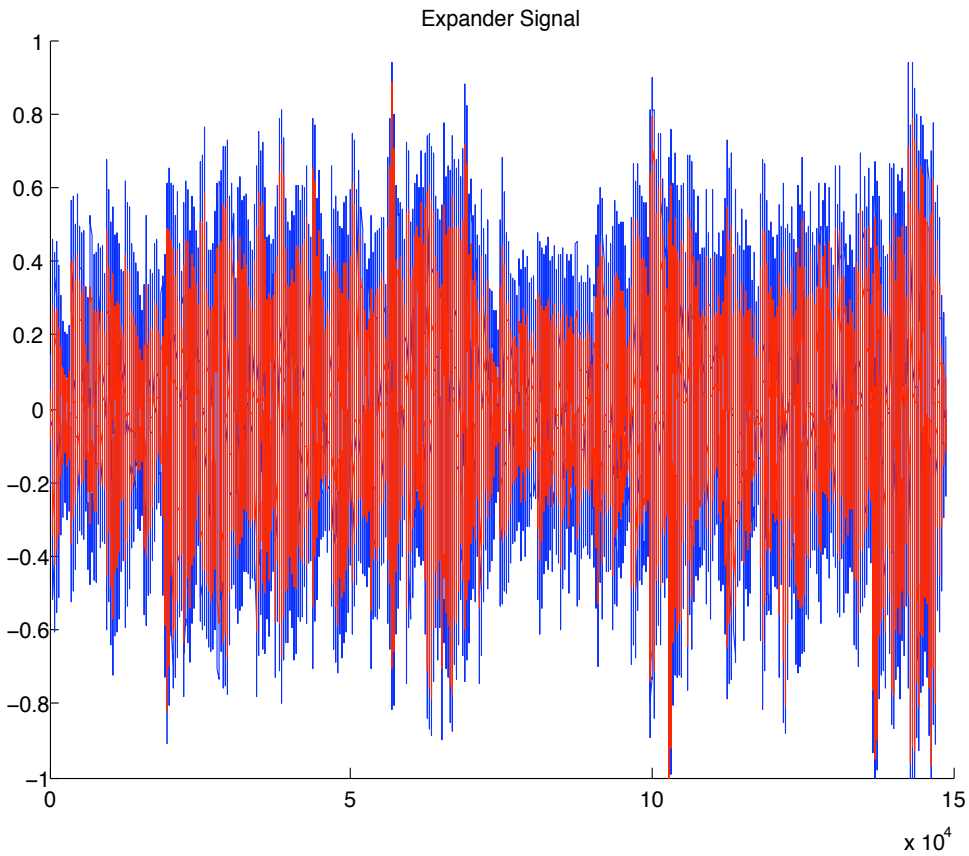
Click Image or here to hear: original audio, compressed audio.

MATLAB Expander Output

An **expanded signal** looks like this:

expander_eg.m:

```
% read the sample waveform
filename='acoustic.wav';
[x,Fs] = audioread(filename);

comp = 0.5; %set expander
a = 0.5;
y = compexp(x,comp,a,Fs);

% write output
audiowrite('out_expander.wav', y,Fs);

figure(1);
hold on
plot(y,'r');
plot(x,'b');
title('Expander Signal');
```



Click image or here to hear: original audio, expander audio.

Overdrive, Distortion and Fuzz

Distortion:

- plays an important part in electric guitar music, especially rock music and its variants.
- can be applied as an effect to other instruments including vocals.

Three broad classes of distortion:

Overdrive — Audio at a low input level is driven by higher input levels in a non-linear curve characteristic

Distortion — a wider tonal area than overdrive operating at a higher non-linear region of a curve

Fuzz — complete non-linear behaviour, harder/harsher than distortion

Overdrive

Achieving Overdrive:

- **Symmetrical soft clipping** of input values is performed.
- A simple three layer *non-linear soft saturation* scheme may be:

$$f(x) = \begin{cases} 2x & \textbf{for} \quad 0 \leq x < 1/3 \\ \frac{3-(2-3x)^2}{3} & \textbf{for} \quad 1/3 \leq x < 2/3 \\ 1 & \textbf{for} \quad 2/3 \leq x \leq 1 \end{cases}$$

  - In the lower third the output is liner — multiplied by 2.
  - In the middle third there is a non-linear (quadratic) output response
  - Above 2/3 the output is set to 1.

MATLAB Overdrive Example

p

Symmetrical soft clipping, symclip.m:

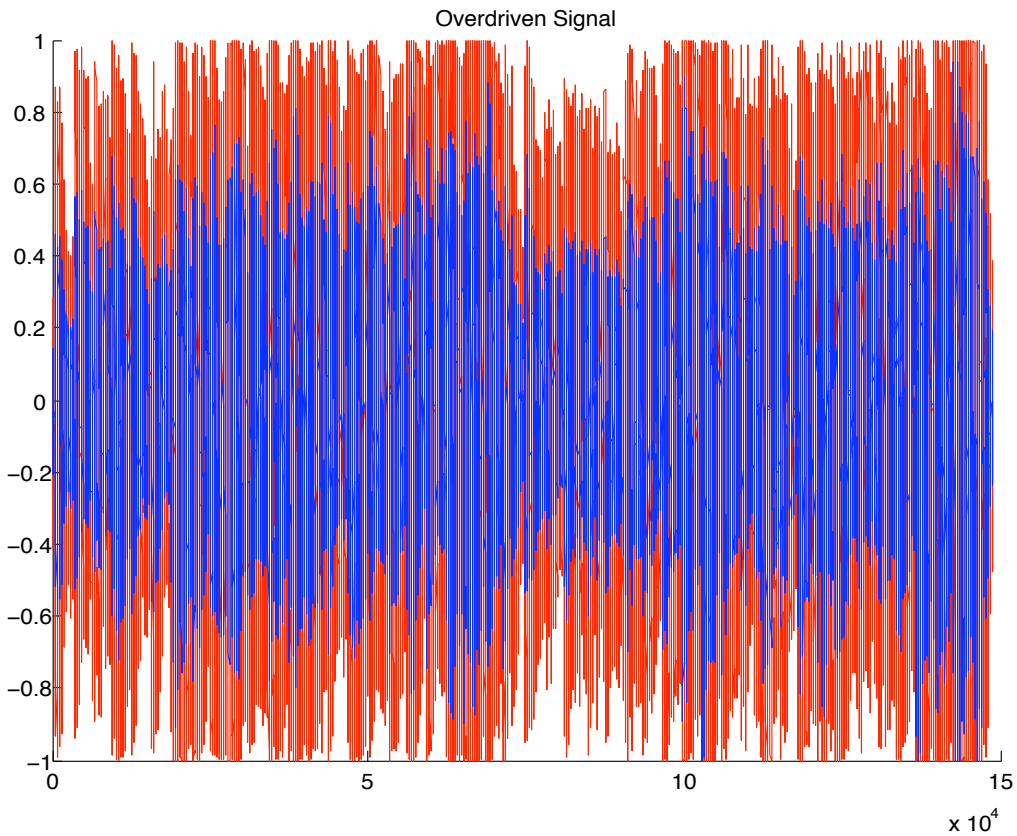
```
function y=symclip(x)

N=length(x);
y=zeros(1,N); % Preallocate y
th=1/3; % threshold for symmetrical soft clipping
        % by Schetzen Formula
for i=1:1:N,
   if abs(x(i))< th, y(i)=2*x(i);end;
   if abs(x(i))>=th,
     if x(i)> 0, y(i)=(3-(2-x(i)*3).^2)/3; end;
     if x(i)< 0, y(i)=-(3-(2-abs(x(i))*3).^2)/3; end;
   end;
   if abs(x(i))>2*th,
     if x(i)> 0, y(i)=1;end;
     if x(i)< 0, y(i)=-1;end;
   end;
end;
```

MATLAB Overdrive Example (Cont.)

An **overdriven signal** looks and sounds like this :

overdrive_eg.m:

```
% read the sample waveform
filename='acoustic.wav';
[x,Fs] = audioread(filename);

% call symmetrical soft clipping
% function
y = symclip(x);

% write output
audiowrite('out_overdrive.wav', y,Fs);

figure(1); hold on;
plot(y,'r');
plot(x,'b');
title('Overdriven Signal');
```



Click image or here to hear: original audio, overdriven audio.

Distortion/Fuzz

Distortion/Fuzz implementation:

- Apply non-linear **amplification function**.
- A non-linear function commonly used to simulate **distortion/fuzz** is given by:

$$f(x) = \frac{x}{|x|}(1 - e^{\alpha x^2/|x|})$$

  - This a non-linear exponential function:
  - The gain, α, controls **level** of distortion/fuzz.
  - Common to **mix** part of the **distorted signal** with **original signal** for **output**.

MATLAB Fuzz Example

fuzzexp.m:

```
function y=fuzzexp(x, gain, mix)
% y=fuzzexp(x, gain, mix)
% Distortion based on an exponential function
% x    - input
% gain - amount of distortion, >0->
% mix  - mix of original and distorted sound, 1=only distorted

q=x*gain/max(abs(x));
z=sign(-q).*(1-exp(sign(-q).*q));
y=mix*z*max(abs(x))/max(abs(z))+(1-mix)*x;
y=y*max(abs(x))/max(abs(y));
```

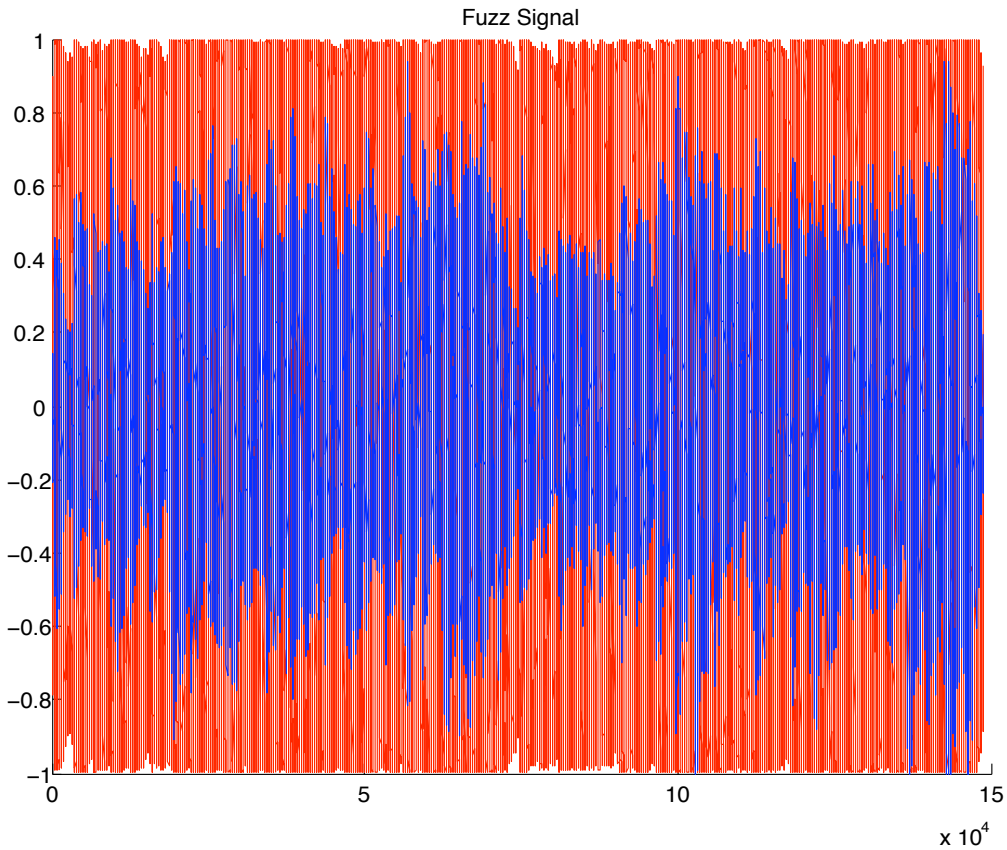
Note: **function allows to `mix` input and fuzz signals at output**

MATLAB Fuzz Example (Cont.)

An **fuzzed up signal** looks and sounds like this:

fuzz_eg.m:

```
filename='acoustic.wav';

% read the sample waveform
[x,Fs] = audioread(filename);

% Call fuzzexp
gain = 11; % Spinal Tap it
mix = 1; % Hear only fuzz
y = fuzzexp(x,gain,mix);

% write output
audiowrite('out_fuzz.wav', y,Fs);
```



Click image or here to hear: original audio, Fuzz audio.

Exciter and Enhancers

Exciter:

A signal processor that emphasises or de-emphasises certain frequencies in order to change a signal's timbre. It can bring extra brightness without necessarily adding in equalisation.

- Frequently used Fourier domain.

Enhancers:

Combine equalisation with non-linear processing.

- introduce a small amount ('just noticeable') of distortion.

Exciters

Achieving Excitation:

- Basic signal processing is achieved by subtle amounts of high frequency distortion and possible phase shifting.
- Performed using the **Short-Time** (Windowed) **Fourier Transform** (STFM) (see **Phase Vocoder**)
- **Compression** often **employed** to non-linear frequency processed element before mixing with the original signal
- Effect can bring *more presence* and clarity to a single instrument in a mix
- Can add natural brightness to a stereo signal
- Can aid intelligibility to speech and vocals.
- Best applied to signals which lack high frequency content unless some odd special effects are required.

Enhancers

Achieving Enhancment:

- Enhancers comprise of a filter network and harmonic generator.
- At least a three band filter is used and an equaliser will boost or cut the frequencies in these bands — independently therefore **non-linearly**.
- Input signal usually mixed with enhanced signal to form output.
- Used in place of equalisers on some mixing consoles.
- Stereo enhancement for radio broadcast and sound reinforcement are also common applications.

Reverb/Spatial Effects

Spatial Effects

The final set of effects we look at are effects that change to spatial localisation of sound.

There a many examples of this type of processing we will study two briefly:

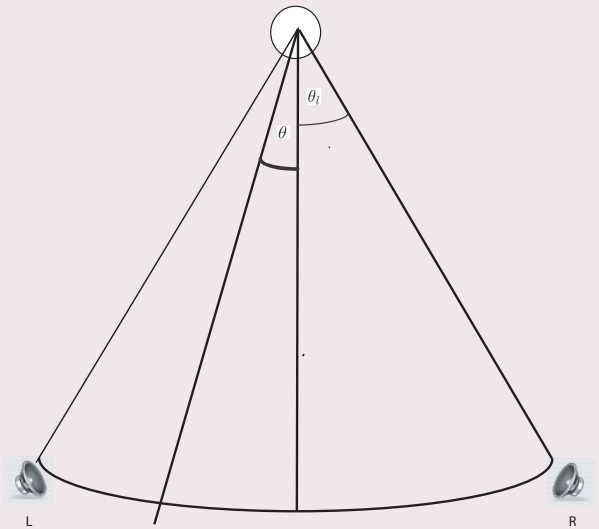
- Panning: in stereo audio
- Reverb: a small selection of reverb algorithms

Panning

What is Panning?

Mapping a **monophonic** sound source across a stereo audio image such that the sound starts in one speaker (**R**) and is moved to the other speaker (**L**) in n time steps.

- We assume that we listening in a central position so that the angle between two speakers is the same, i.e. we subtend an angle $2\theta_l$ between **2** speakers.
  We assume for simplicity, in this case that $\theta_l = 45°$.



Panning Geometry

Simple applications of basic rotation geometry:

- We seek to obtain to signals one for each Left (**L**) and Right (**R**) channel, the gains of which, g_L and g_R, are applied to steer the sound across the stereo audio image.
- This can be achieved by simple 2D rotation, where the angle we sweep is θ:

$$\mathbf{A}_\theta = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

and

$$\begin{bmatrix} g_L \\ g_R \end{bmatrix} = \mathbf{A}_\theta . \mathbf{x}$$

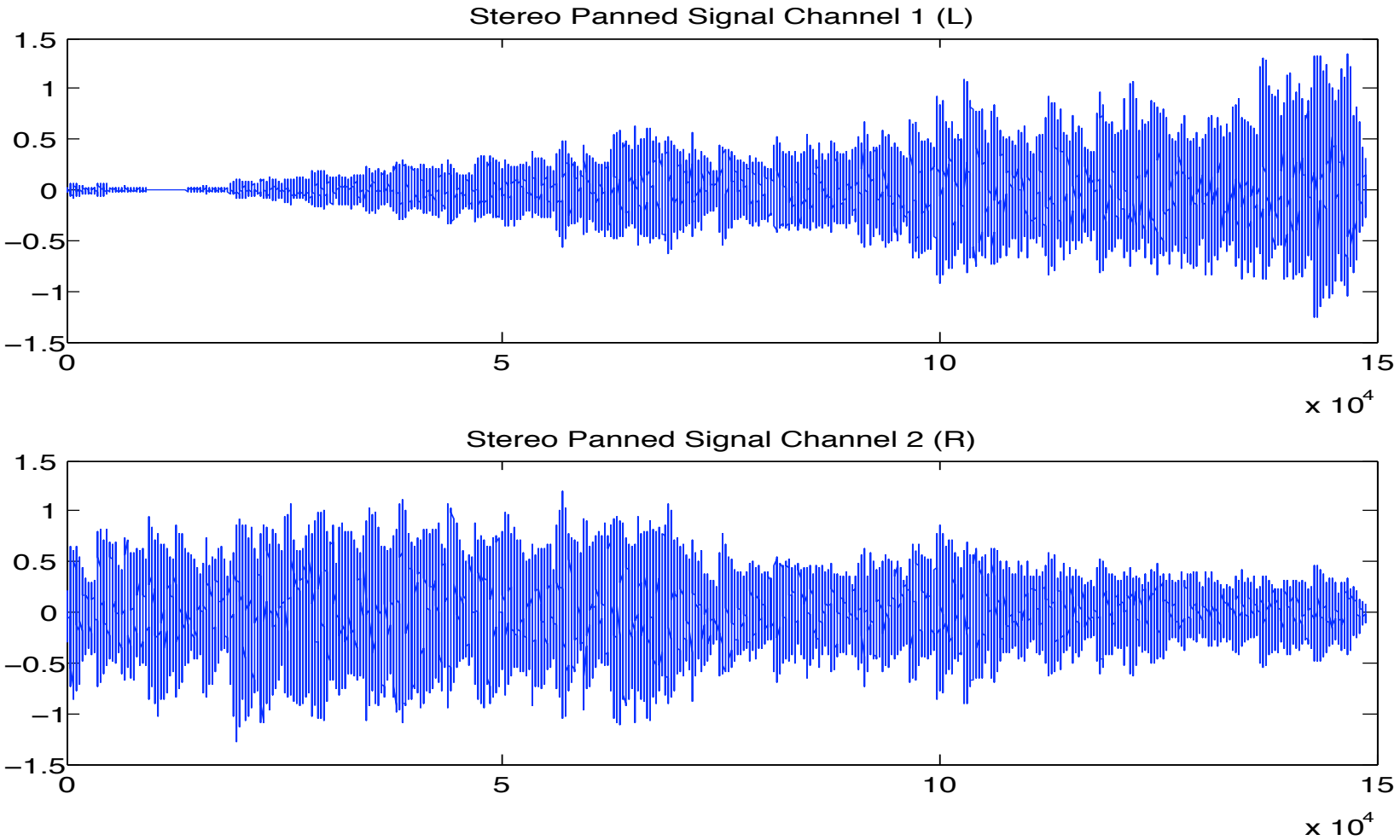
where **x** is a segment of mono audio

MATLAB Panning Example

matpan.m:

```
% read the sample waveform
filename='acoustic.wav';
[monox,Fs] = audioread(filename);

initial_angle = -40; %in degrees
final_angle = 40;    %in degrees
segments = 32;
angle_increment = (initial_angle - final_angle)/segments * pi / 180;
lenseg = floor(length(monox)/segments) - 1;
pointer = 1;
angle = initial_angle * pi / 180; %in radians

y=[[];[]]; % Preallocate
for i=1:segments
 A =[cos(angle), sin(angle); -sin(angle), cos(angle)];
 stereox =
  [monox(pointer:pointer+lenseg)'; monox(pointer:pointer+lenseg)'];
 y = [y, A * stereox];
 angle = angle + angle_increment; pointer = pointer + lenseg;
end;
% write output .......
```

MATLAB Panning Example Output



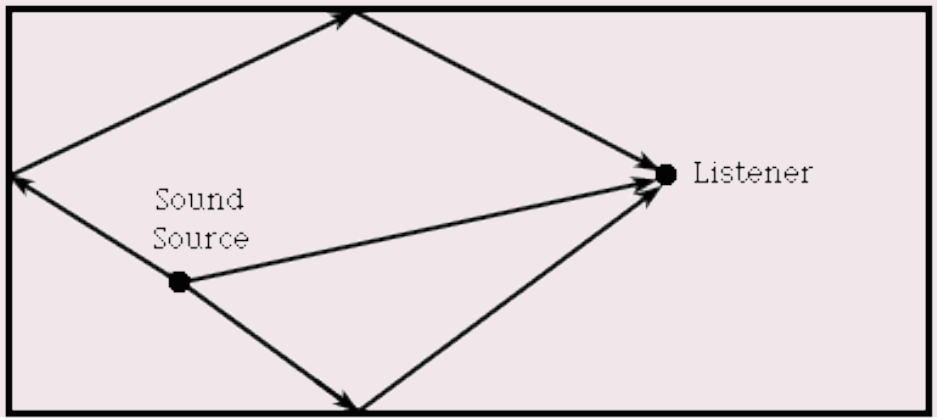
Click image or here to hear: original audio, stereo panned audio.

Reverb

Reverberation

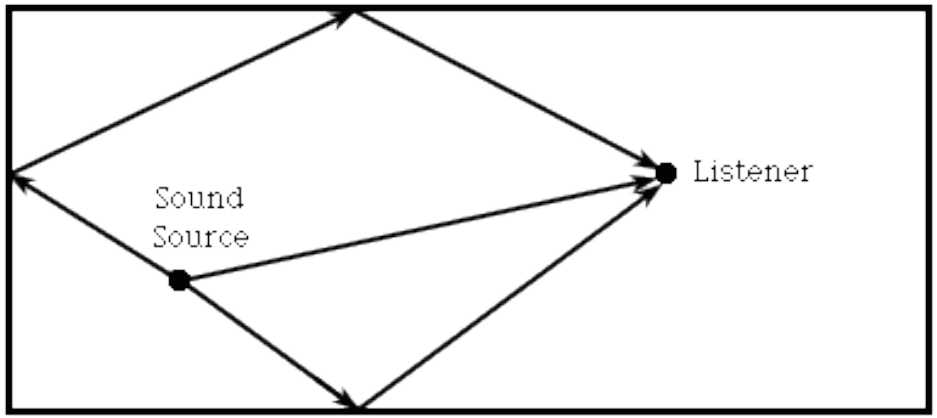
Reverb (for short) is probably one of the most heavily used effects in audio.

- *Reverberation* is the result of the many reflections of a sound that occur in a room.
  - From any sound source, say a speaker of your stereo, there is a direct path that the sounds covers to reach our ears.
  - Sound waves can also take a slightly longer path by reflecting off a wall or the ceiling, before arriving at your ears.



The Spaciousness of a Room

- A reflected sound wave like this will arrive **a little later** than the direct sound, since it travels a longer distance, and is generally a little weaker, as the walls and other surfaces in the room will absorb some of the sound energy.
- Reflected waves can again bounce off another wall before arriving at your ears, and so on.
- This series of delayed and attenuated sound waves is what we call **reverb**, and this is what creates the *spaciousness* sound of a room.
- Clearly large rooms such as concert halls/cathedrals will have a much more spaciousness reverb than a living room or bathroom.



Reverb v. Echo

Is reverb just a series of echoes?

Echo — implies a distinct, delayed version of a sound,

- *E.g.* as you would hear with a delay more than one or two-tenths of a second.

Reverb — each delayed sound wave arrives in such a short period of time that we do not perceive each reflection as a copy of the original sound.

- Even though we can't discern every reflection, we still hear the effect that the entire series of reflections has.

Reverb v. Delay

Can a simple delay device with feedback produce reverberation?

Delay: can produce a similar effect **but** there is one very important feature that a simple delay unit will **not** produce:

- The rate of arriving reflections **changes** over **time**.
- Delay can only simulate reflections with a fixed time interval.

Reverb: for a short period after the direct sound, there is generally a set of well defined directional reflections that are directly related to the shape and size of the room, and the position of the source and listener in the room.

- These are the *early reflections*
- After the early reflections, the rate of the arriving reflections increases greatly are more random and difficult to relate to the physical characteristics of the room.
  This is called the *diffuse reverberation*, or the *late reflections*.
- Diffuse reverberation is the **primary factor** establishing a room's 'spaciousness' — it decays exponentially in good concert halls.

Reverb Simulation

There are many ways to simulate reverb.

Two Broad classes of approach studied here (there are others):

- **Filter Bank**/**Delay Line** methods
- **Convolution**/**Impulse Response** methods

Schroeder's Reverberator

Schroeder's model of Reverb (1961)

- Early digital reverberation algorithms tried to mimic the a rooms reverberation by using primarily consisted of **two types** of infinite impulse response (IIR) filters with the **aim** to make the output **gradually decay**.

  Comb filter: usually in parallel banks

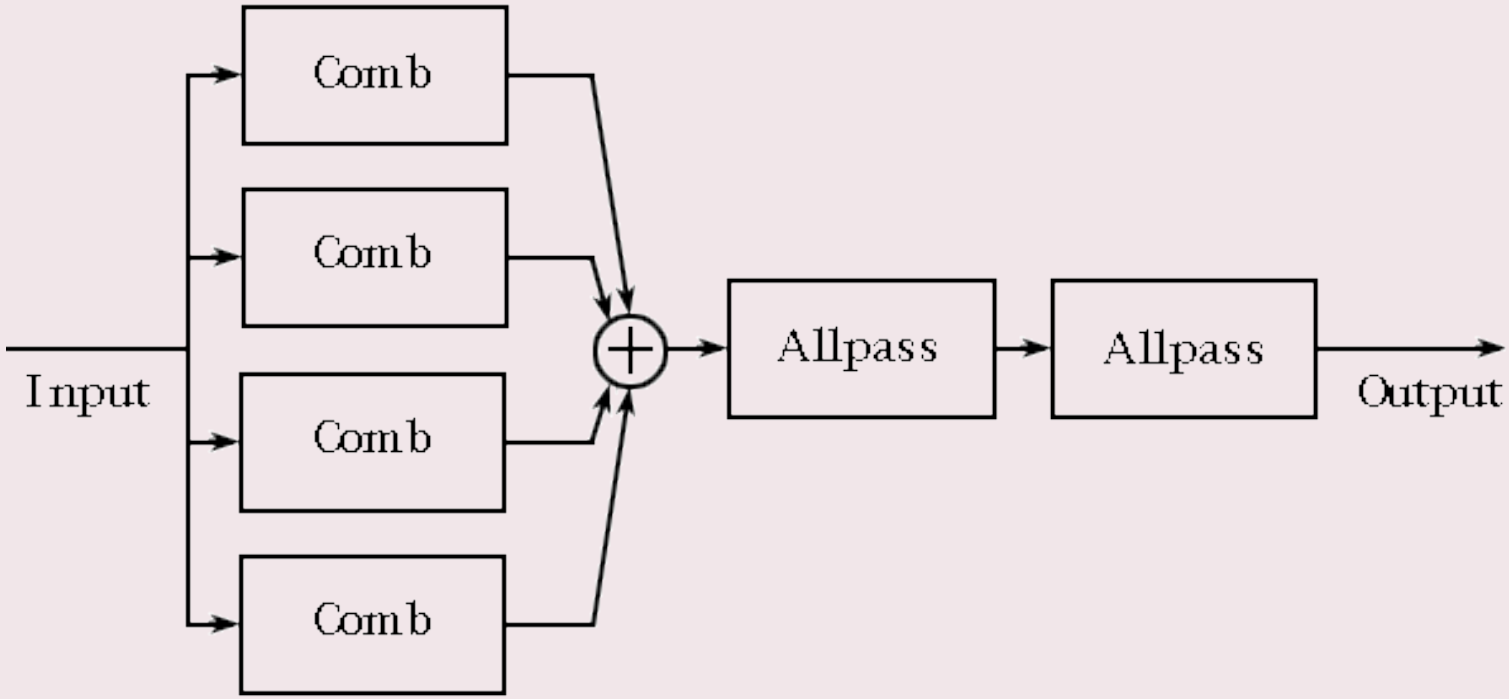
  Allpass filter: usually sequentially after comb filter banks

Much of the early work on digital reverb was performed by Schroeder.

Schroeder's Reverberator (Cont.)

Schroeder's reverberator example (one of a few variations):

This particular design uses **four comb filters** and **two allpass filters**:



Note:This design does not create the increasing arrival rate of reflections, and is rather primitive when compared to current algorithms.

MATLAB Schroeder Reverb Example

Simple Schroeder: schroeder1.m:

- *n* **allpass** filters in series. (**No Comb Filters yet**)

```
function [y,b,a]=schroeder1(x,n,g,d,k)
% This is a reverberator based on Schroeder's design which consists of n
% allpass filters in series.
%
% The structure is:  [y,b,a] = schroeder1(x,n,g,d,k)
%
% where x = the input signal
%       n = the number of allpass filters
%       g = the gain of the allpass filters
%           (should be less than 1 for stability)
%       d = a vector which contains the delay length of each allpass filter
%       k = the gain factor of the direct signal
%       y = the output signal
%       b = the numerator coefficients of the transfer function
%       a = the denominator coefficients of the transfer function
%
% note: Make sure that d is the same length as n.
%
```

MATLAB Schroeder Reverb Example (Cont.)

schroeder1.m (Cont.):

```
% send the input signal through the first allpass filter
[y,b,a] = allpass(x,g,d(1));

% send the output of each allpass filter to the input of
% the next allpass filter
for i = 2:n,
   [y,b1,a1] = allpass(y,g,d(i));
   [b,a] = seriescoefficients(b1,a1,b,a);
end

% add the scaled direct signal
y = y + k*x;

% normalize the output signal
y = y/max(y);
```

MATLAB Schroeder Reverb Example (Cont.)

The support files to do the filtering (for following reverb methods also) are here:

- delay.m,
- seriescoefficients.m,
- parallelcoefficients.m,
- fbcomb.m,
- ffcomb.m,
- allpass.m

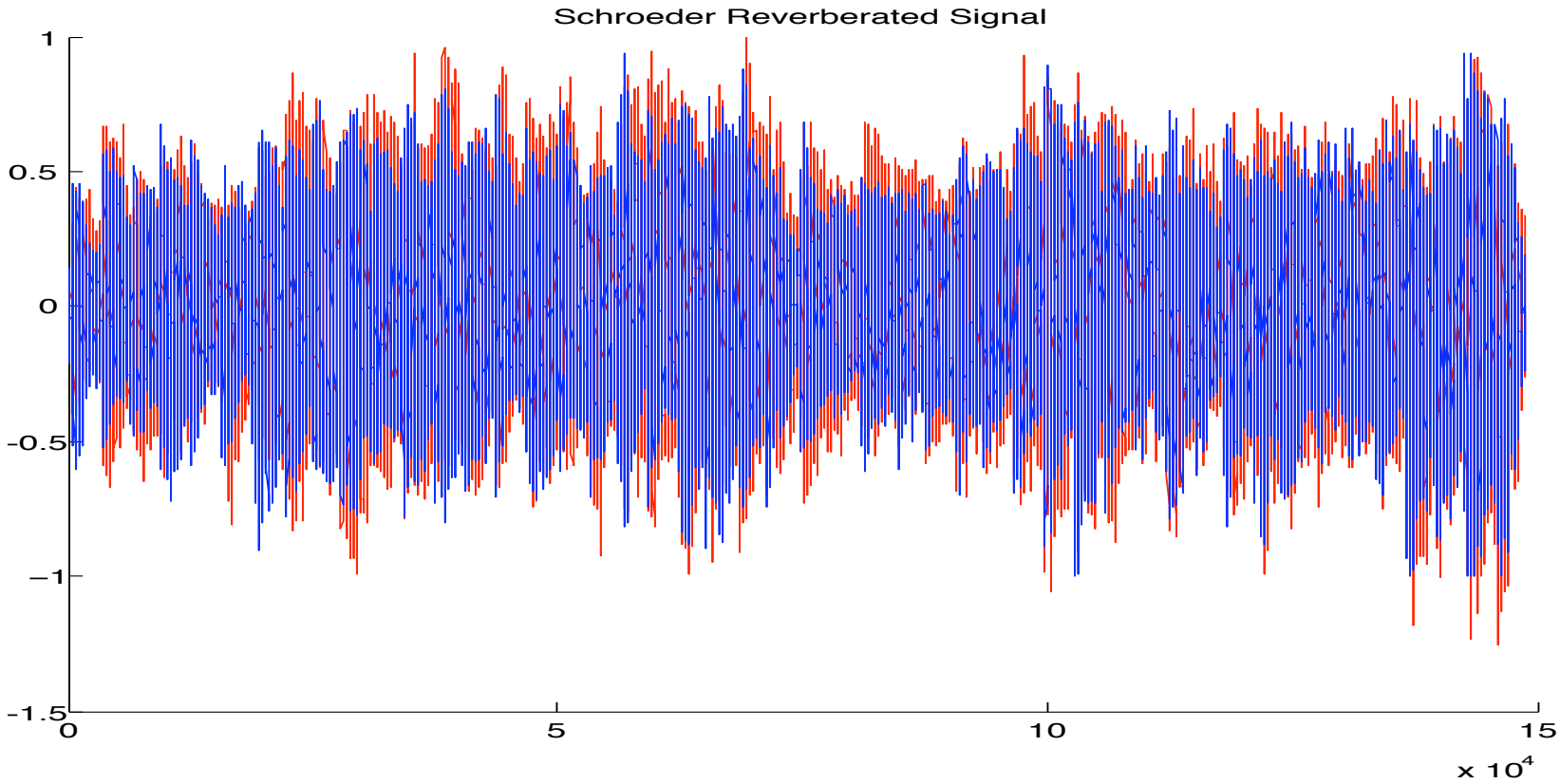
MATLAB Schroeder Reverb (Cont.)

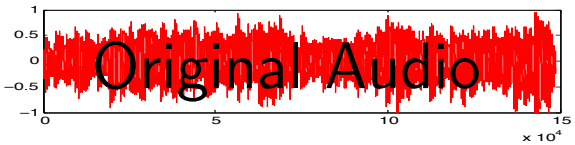
Example call, reverb_schroeder_eg.m:

```
filename='acoustic.wav'; % Read the waveform
[x,Fs] = audioread(filename);

% Set the number of allpass filters
n = 6;
% Set the gain of the allpass filters
g = 0.9;
% Set delay of each allpass filter in number of samples
% Compute a random set of milliseconds and use sample rate
rand('state',sum(100*clock))
d = floor(0.05*rand([1,n])*Fs);
%set gain of direct signal
k= 0.2;

[y b a] = schroeder1(x,n,g,d,k);

% write output
audiowrite('out_schroederreverb.wav', y,Fs);
```

MATLAB Schroeder Reverb (Cont.)

The input signal (blue) and reverberated signal (red):





Click images or here to hear: original audio, reverberated audio.

MATLAB *Classic* Schroeder Reverb Example

Classic Schroeder Reverb, schroeder2.m:

- **4 comb** and **2 allpass** filters.

```
function [y,b,a]=schroeder2(x,cg,cd,ag,ad,k)
% This is a reverberator based on Schroeder's design which consists of 4
% parallel feedback comb filters in series with 2  allpass filters.
%
% The structure is:  [y,b,a] = schroeder2(x,cg,cd,ag,ad,k)
% where x = the input signal
%       cg = a vector of length 4 which contains the gain of each of the
%                comb filters (should be less than 1)
%       cd = a vector of length 4 which contains the delay of each of the
%              comb filters
%       ag = the gain of the allpass filters (should be less than 1)
%       ad = a vector of length 2 which contains the delay of each of the
%                allpass filters
%       k = the gain factor of the direct signal
%       y = the output signal
%       b = the numerator coefficients of the transfer function
%       a = the denominator coefficients of the transfer function
```

MATLAB *Classic* Schroeder Reverb Example (Cont.)

Classic Schroeder Reverb, schroeder2.m (Cont.):

```
% send the input to each of the 4 comb filters separately
[outcomb1,b1,a1] = fbcomb(x,cg(1),cd(1));
[outcomb2,b2,a2] = fbcomb(x,cg(2),cd(2));
[outcomb3,b3,a3] = fbcomb(x,cg(3),cd(3));
[outcomb4,b4,a4] = fbcomb(x,cg(4),cd(4));


% sum the ouptut of the 4 comb filters
apinput = outcomb1 + outcomb2 + outcomb3 + outcomb4;

%find the combined filter coefficients of the the comb filters
[b,a]=parallelcoefficients(b1,a1,b2,a2);
[b,a]=parallelcoefficients(b,a,b3,a3);
[b,a]=parallelcoefficients(b,a,b4,a4);
```

MATLAB *Classic* Schroeder Reverb Example (Cont.)

Classic Schroeder Reverb, schroeder2.m (Cont.):

```
% send the output of the comb filters to the allpass filters
[y,b5,a5] = allpass(apinput,ag,ad(1));
[y,b6,a6] = allpass(y,ag,ad(2));

%find the combined filter coefficients of the the comb filters in
% series with the allpass filters
[b,a]=seriescoefficients(b,a,b5,a5);
[b,a]=seriescoefficients(b,a,b6,a6);

% add the scaled direct signal
y = y + k*x;

% normalize the output signal
y = y/max(y);
```

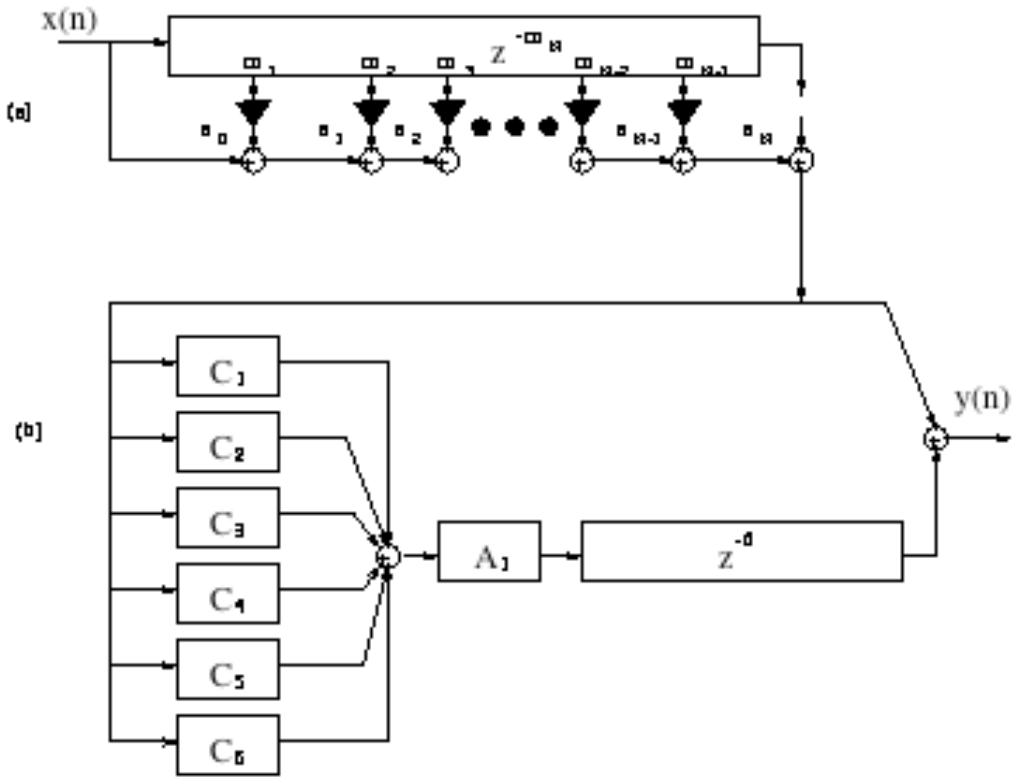
See forthcoming Lab Class for examples of this effect and extensions.

Moorer's Reverberator

Moorer's reverberator (1976): Build's on Schroeder

- **Parallel comb filters** with **different** delay lengths are used to simulate modes of a room, and sound reflecting between parallel walls
- **Allpass** filters to **increase** the reflection density (diffusion).
- **Lowpass** filters **inserted** in the feedback loops to alter the reverberation time as a **function of frequency**
  - **Shorter** reverberation time at **higher** frequencies is caused by air absorption and reflectivity characteristics of wall).
  - Implement a **DC-attenuation**, and a frequency dependent attenuation.
  - Encode a difference in each comb filter because their coefficients **depend** on the **delay line length**.

Moorer's Reverberator



(a) Tapped delay lines simulate *early reflections* — forwarded to (b)

(b) Parallel comb filters which are then **allpass** filtered and delayed before being added back to early reflections — simulates **diffuse reverberation**

MATLAB Moorer Reverb

moorer.m:

```
function [y,b,a]=moorer(x,cg,cg1,cd,ag,ad,k)
% This is a reverberator based on Moorer's design which consists of 6
% parallel feedback comb filters  (each with a low pass filter in the
% feedback loop) in series with an all pass filter.
%
% The structure is:  [y,b,a] = moorer(x,cg,cg1,cd,ag,ad,k)
%
% where x = the input signal
%       cg = a vector of length 6 which contains g2/(1-g1) (this should be less
%               than 1 for stability), where g2 is the feedback gain of each of the
%               comb filters and g1 is from the following parameter
%       cg1 = a vector of length 6 which contains the gain of the low pass
%               filters in the feedback loop of each of the comb filters (should be
%               less than 1 for stability)
%       cd = a vector of length 6 which contains the delay of each of comb filter
%       ag = the gain of the allpass filter (should be less than 1 for stability)
%       ad = the delay of the allpass filter
%       k = the gain factor of the direct signal
%       y = the output signal
%       b = the numerator coefficients of the transfer function
%       a = the denominator coefficients of the transfer function
%
```

MATLAB Moorer Reverb (Cont.)

moorer.m (Cont.):

```
% send the input to each of the 6 comb filters separately
[outcomb1,b1,a1] = lpcomb(x,cg(1),cg1(1),cd(1));
[outcomb2,b2,a2] = lpcomb(x,cg(2),cg1(2),cd(2));
[outcomb3,b3,a3] = lpcomb(x,cg(3),cg1(3),cd(3));
[outcomb4,b4,a4] = lpcomb(x,cg(4),cg1(4),cd(4));
[outcomb5,b5,a5] = lpcomb(x,cg(5),cg1(5),cd(5));
[outcomb6,b6,a6] = lpcomb(x,cg(6),cg1(6),cd(6));

% sum the ouptut of the 6 comb filters
apinput = outcomb1 + outcomb2 + outcomb3 + outcomb4 + outcomb5 + outcomb6;

%find the combined filter coefficients of the the comb filters
[b,a]=parallelcoefficients(b1,a1,b2,a2);
[b,a]=parallelcoefficients(b,a,b3,a3);
[b,a]=parallelcoefficients(b,a,b4,a4);
[b,a]=parallelcoefficients(b,a,b5,a5);
[b,a]=parallelcoefficients(b,a,b6,a6);
```

MATLAB Moorer Reverb (Cont.)

moorer.m (Cont.):

```
% send the output of the comb filters to the allpass filter
[y,b7,a7] = allpass(apinput,ag,ad);

%find the combined filter coefficients of the the comb filters in series
% with the allpass filters
[b,a]=seriescoefficients(b,a,b7,a7);

% add the scaled direct signal
y = y + k*x;

% normalize the output signal
y = y/max(y);
```

MATLAB Moorer Reverb (Cont.)

Example call, reverb_moorer_eg.m:

```
% reverb_moorer_eg.m
% Script to call the Moorer Reverb Algoritm

% read the sample waveform
filename='../acoustic.wav';
[x,Fs] = audioread(filename);

% Call moorer reverb
%set delay of each comb filter
%set delay of each allpass filter in number of samples
%Compute a random set of milliseconds and use sample rate
rand('state',sum(100*clock))
cd = floor(0.05*rand([1,6])*Fs);

% set gains of 6 comb pass filters
g1 = 0.5*ones(1,6);
%set feedback of each comb filter
g2 = 0.5*ones(1,6);
```

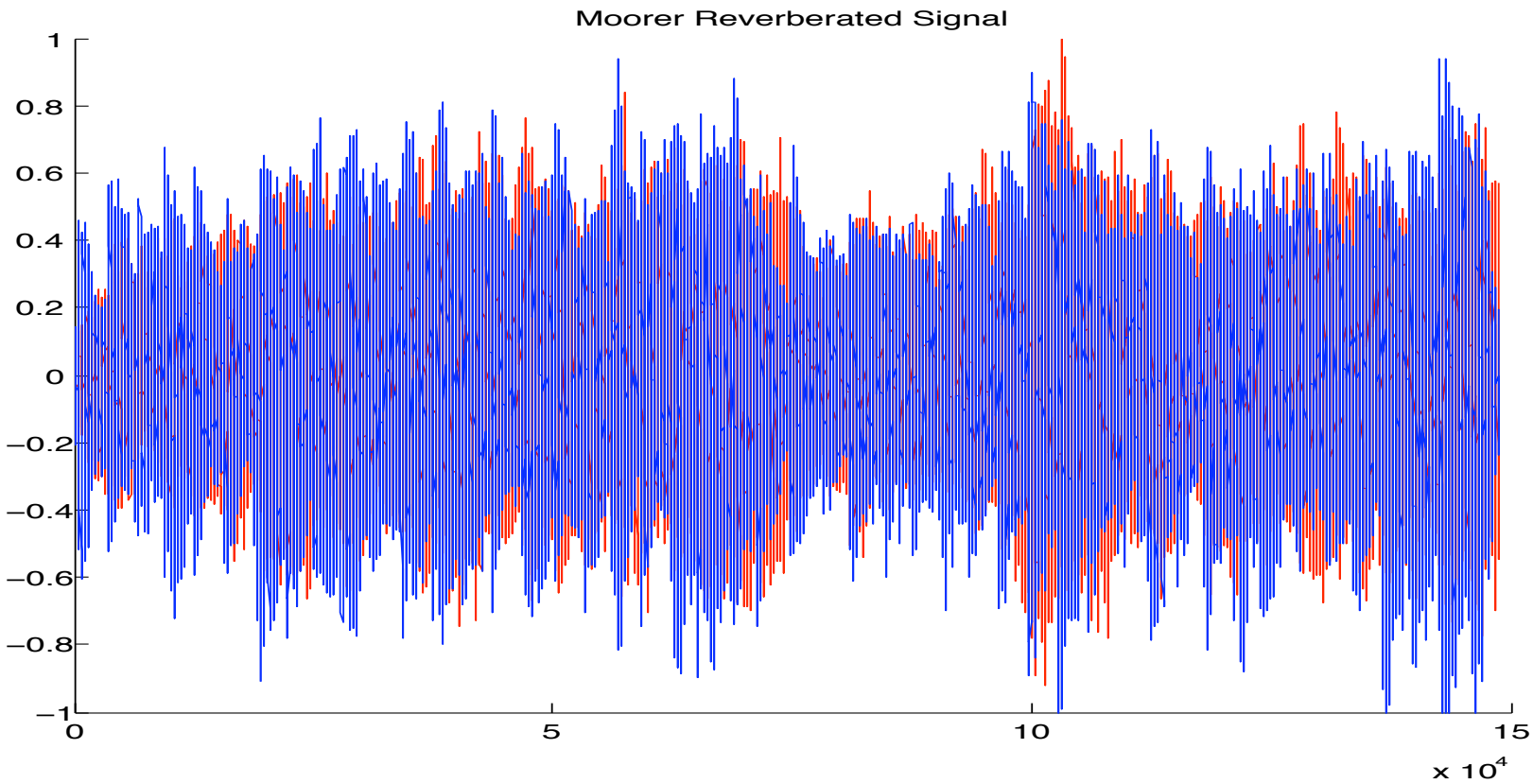
MATLAB Moorer Reverb (Cont.)

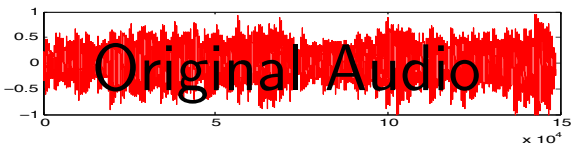
reverb_moorer_eg.m:

```
% set input cg and cg1 for moorer function see help moorer
cg = g2./(1-g1);
cg1 = g1;

%set gain of allpass filter
ag = 0.7;
%set delay of allpass filter
ad = 0.08*Fs;
%set direct signal gain
k = 0.5;

[y b a] = moorer(x,cg,cg1,cd,ag,ad,k);

% write output
audiowrite('out_moorerreverb.wav', y,Fs);
```

MATLAB Moorer Reverb (Cont.)

The input signal (blue) and reverberated signal (red):





Click here to hear: original audio, Moorer reverberated audio.

Convolution Reverb

Convolution Reverb: Basic Idea

If the impulse response of the room is known then the most faithful reverberation method would be to **convolve** it with the input signal.

- Due to the usual length of the target response it is not feasible to implement this with filters — several hundreds of taps in the filters would be required.
- However, **convolution readily implemented** using **FFT**:
  - Recall: The **discrete convolution** formula:

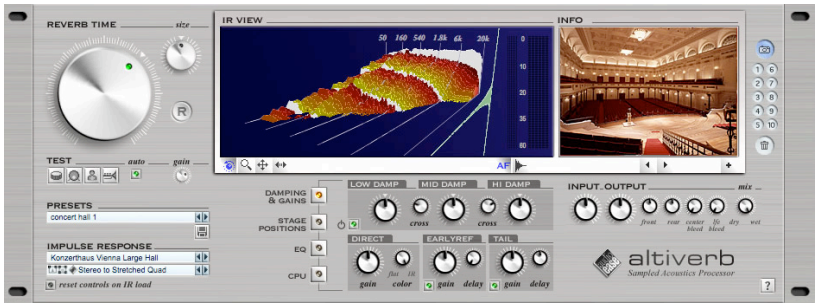
$$y(n) = \sum_{k=-\infty}^{\infty} x(k).h(n-k) = x(n) * h(n)$$

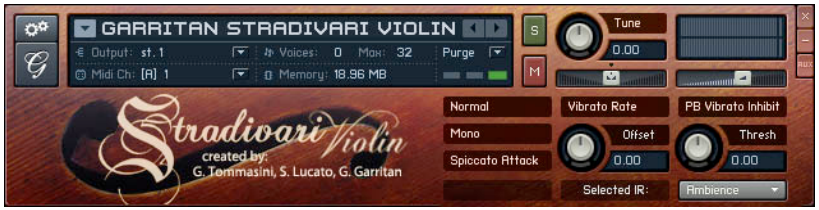
  - Recall: The **convolution theorem** which states that: *If $f(x)$ and $g(x)$ are two functions with Fourier transforms $F(u)$ and $G(u)$, then the Fourier transform of the* **convolution** *$f(x) * g(x)$ is simply the product of the Fourier transforms of the two functions, $F(u)G(u)$.*

Commercial Convolution Reverbs

Commercial Convolution Reverbs

- Altiverb — one of the first mainstream convolution reverb effects units
- Most sample based synthesisers (E.g. Kontakt, Intakt) provide some convolution reverb effect
- Dedicated sample-based software instruments such as Garritan Violin and PianoTeq Piano use convolution not only for reverb simulation but also to simulate key responses of the instruments body vibration.



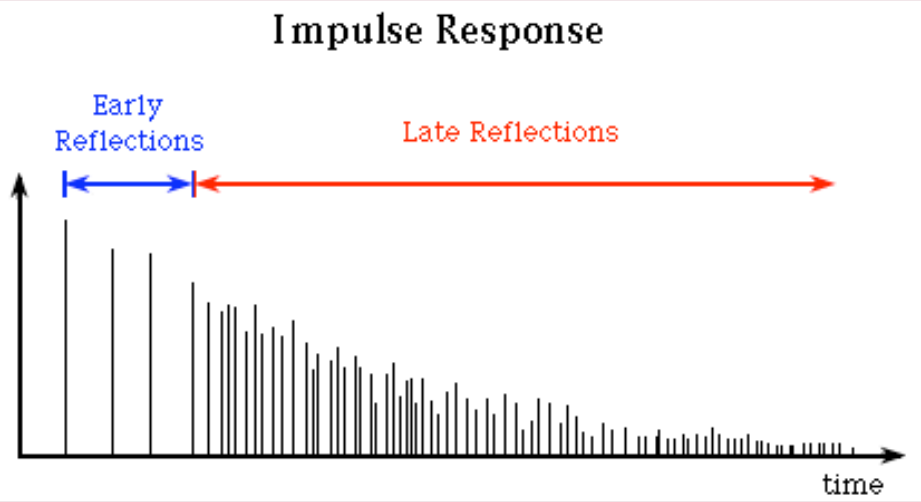


Room Impulse Responses

Record a Room Impulse

Apart from providing a high (professional) quality recording of a room's impulse response, the process of using an impulse response is quite straightforward:

- Record a short impulse (gun shot,drum hit, hand clap) in the room.
- Room impulse responses can be simulated in software also.
- The impulse encodes the rooms reverb characteristics:



MATLAB Convolution Reverb (1)

Let's develop a fast convolution routine: fconv.m

```
function [y]=fconv(x, h)
%   FCONV Fast Convolution
%   [y] = FCONV(x, h) convolves x and h,
%       and normalizes the output   to +-1.
%      x = input vector
%      h = input vector
%

Ly=length(x)+length(h)-1;  %
Ly2=pow2(nextpow2(Ly));    % Find smallest power of 2
                           % that is > Ly
X=fft(x, Ly2);                   % Fast Fourier transform
H=fft(h, Ly2);                   % Fast Fourier transform
Y=X.*H;                           % DO CONVOLUTION
y=real(ifft(Y, Ly2));      % Inverse fast Fourier transform
y=y(1:1:Ly);               % Take just the first N elements
y=y/max(abs(y));           % Normalize the output
```

See also: MATLAB built in function `conv()`

MATLAB Convolution Reverb (2)

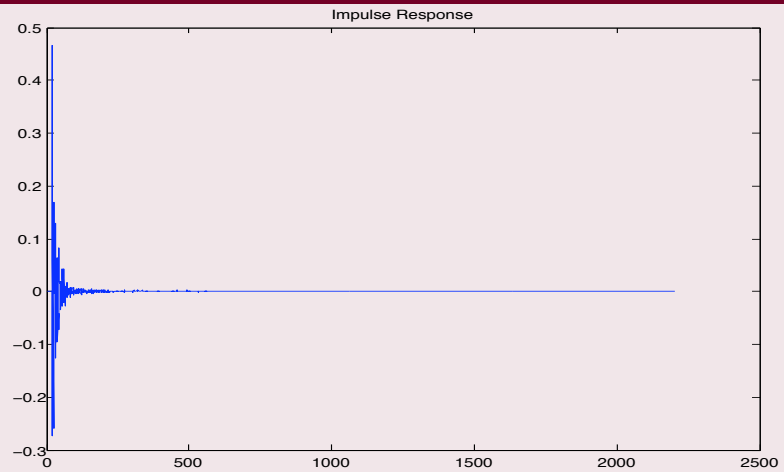
reverb_convolution_eg.m

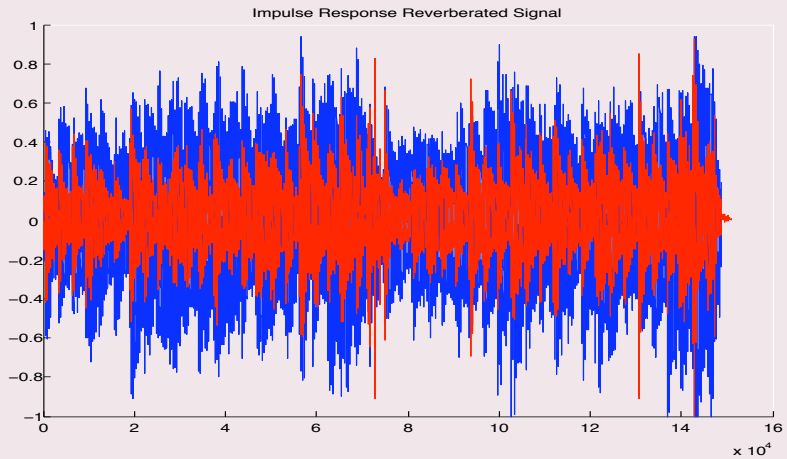
```
% reverb_convolution_eg.m
% Script to call implement Convolution Reverb

% read the sample waveform
filename='../acoustic.wav';
[x,Fs] = audioread(filename);

% read the impulse response waveform
filename='impulse_room.wav';
[imp,Fsimp] = audioread(filename);

% Do convolution with FFT
y = fconv(x,imp);

% write output
audiowrite('out_IRreverb.wav', y,Fs);
```

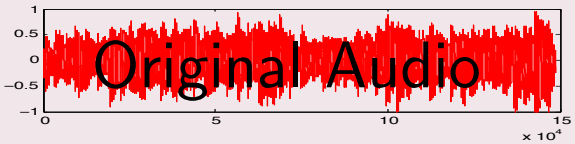
MATLAB Convolution Reverb (3)

Some example results:

Living Room Impulse Response Convolution Reverb:



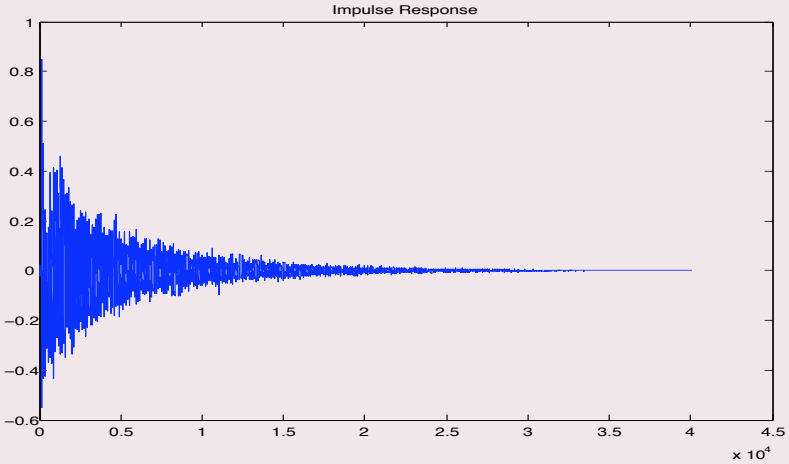


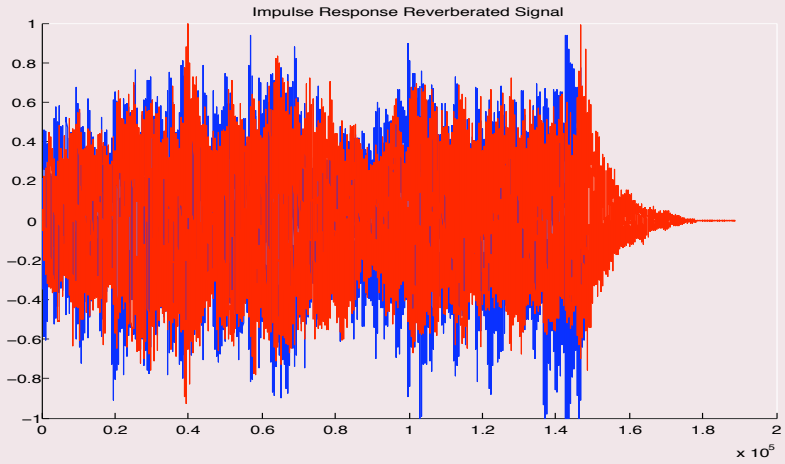


Click on above images or here to hear: original audio, room impulse response audio, room impulse reverberated audio.

MATLAB Convolution Reverb (4)

Cathedral Impulse Response Convolution Reverb:



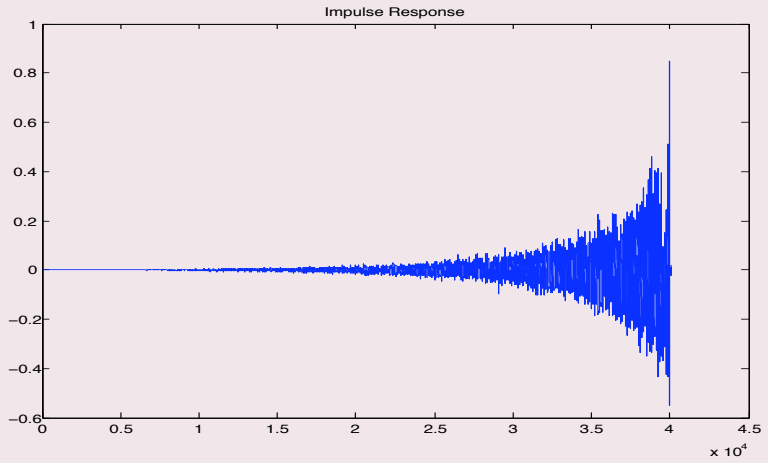


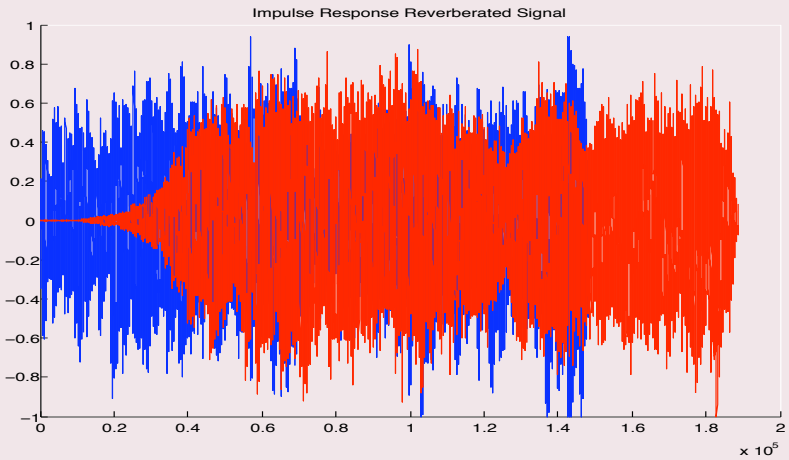
Click on above images or here to hear: original audio, cathedral impulse response audio, cathedral reverberated audio.

MATLAB Convolution Reverb (5)

It is easy to implement some **other (odd?) effects** also

Reverse Cathedral Impulse Response Convolution Reverb:



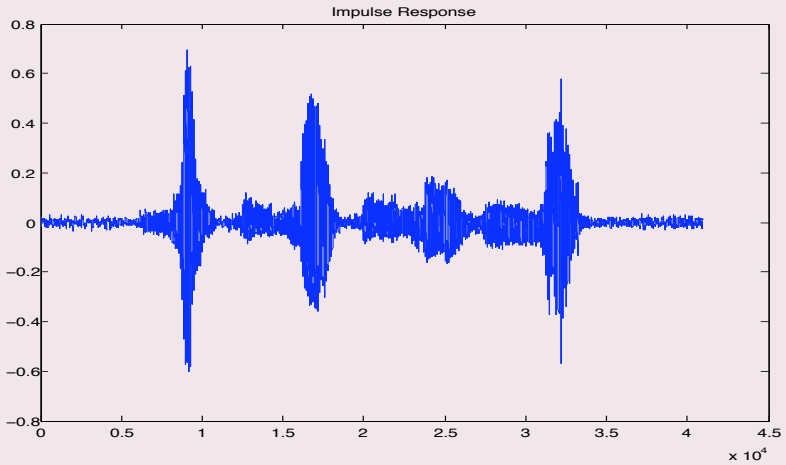


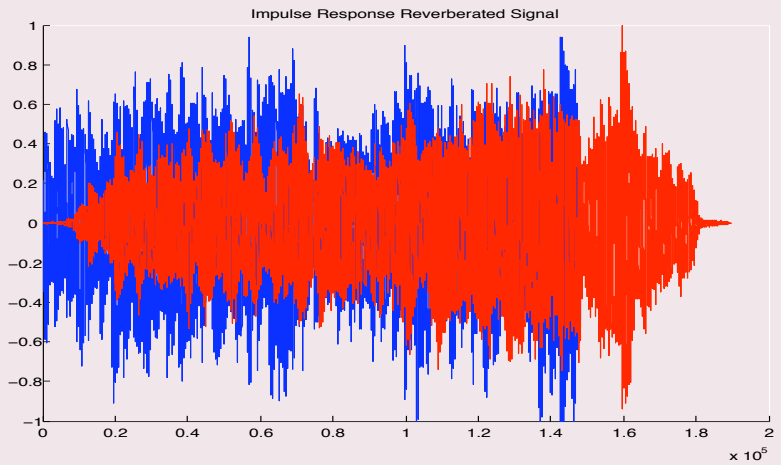
Click on above images or here to hear: original audio, reverse cathedral impulse response audio, reverse cathedral reverberated audio.

MATLAB Convolution Reverb (6)

You can basically convolve with anything.

Speech Impulse Response Convolution Reverb!:





Click on above images or here to hear: original audio, speech *'impulse response'* audio, speech impulse reverberated audio.