An Exploration of Time-Scale Modification Methods

By Allison Crim

Introduction

Audio signal processing is a subfield of signal processing that has applications in a wide array of spaces including entertainment, defense, and audio effects. One of these audio effects that is widely used in television, music, and media is called Time-Scale Modification (TSM). TSM is a method of speeding up or slowing down an audio signal while maintaining the signals pitch. Without TSM, a sped-up signal would sound squeaky and a slowed down signal would sound deep. This paper will explore the various methods that have been designed to combat these unwanted effects.

Real Life Applications

TSM has a wide variety of applications in media. Between television, music, and video streaming, it is important to understand the benefits and drawbacks of various TSM applications to create the best content.

One of the most easily recognizable applications of TSM is in video playback. From YouTube videos to zoom recordings, users have the option to speed up or slow down a video either to save time or increase comprehension. TSM is the chosen method to implement this feature because it allows the speed of the video to change but doesn't affect the pitch.

Another application for TSM came as a surprise to the people who developed it. In the 1970's a company called Eventide had released the H910, a harmonizer that "Agnello [one of the company's co-founders] considers the first true digital effect unit" (Scarth). Although it was used for its original purpose by many customers, the H910 ended up taking off based on the deregulation of the number of commercials allowed in a hour of TV. At that point in time, "broadcasters had rules on how much commercial time they could have in an hour". When these rules were taken away, late night television reruns of *I Love Lucy* were sped up to fit in more commercials. Viewers began to complain about Lucy's high pitched voice that was a result of speeding up the video without modifying the pitch. According to Agnello, "We started selling 910s to Channel 5 in New York and I couldn't figure out why until someone told me they were using it to pitch-correct Lucille Ball's voice"(Scarth).

A final application of TSM is in DJing or music production. In order to create remixes of existing songs, the producer can use TSM to increase or decrease the speed of a song to sync up with the other song that they are remixing. Although this seems like a simple task, songs can contain complex features like multiple harmonies and changing tempos that make TSM more difficult. Methods for stretching or compressing these complex signals will be explored further in the paper.

Approach

Over the years, engineers have discovered many ways to implement TSM depending on their intended input signal and output behavior. This paper will explore two of these methods and compare the effectiveness of each.

Overlap Add (OLA)

The first approach that was implemented is called the Overlap-Add (OLA) method. This method is implemented in the time domain and works well to keep the high quality of a signal. In order to implement OLA based TSM, the user must do the following steps.

It is important to note that the parameter that the signal will be stretched or compressed by is represented by the term α which is a ratio of synthesis hopsize, H_s over analysis hopsize ,H_a.

The first step in the OLA process is to divide the input signal, x into analysis frames of length N spaced by analysis hopsize Ha. This can be accomplished using the following equation.

$$x_m(r) = \begin{Bmatrix} x(r + mH_a, & if\ r \epsilon [-\frac{N}{2} : \frac{N}{2} - 1 \\ 0, & otherwise \end{Bmatrix} \quad (1)$$

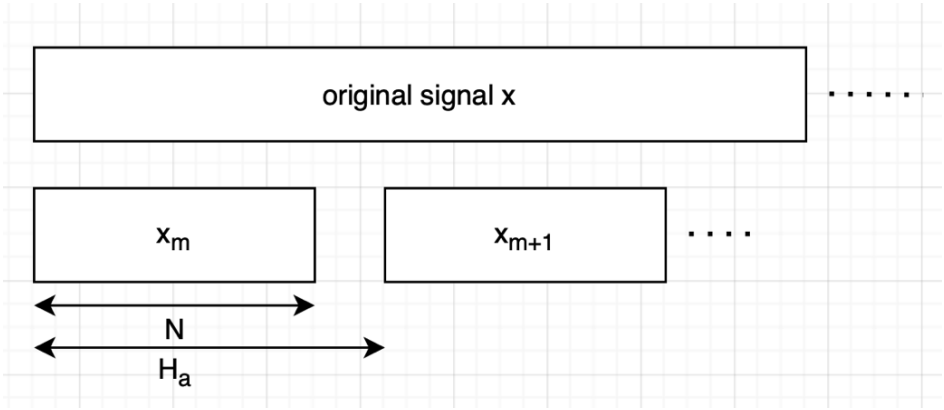

Figure 1: Break Original Signal into Analysis Frames

Next, a Hann window is applied to each analysis frame. Applying this window before summing the analysis frames removes jumps in the reconstructed signal that would be present without the window. The equation for a Hann window is shown below.

$$w(r) = \begin{Bmatrix} 0.5\left(1 - \cos\left(\frac{2\pi\left(r + \frac{N}{2}\right)}{N - 1}\right)\right), if\ r \epsilon \left[-\frac{N}{2} : \frac{N}{2}\right] \\ 0, \quad otherwise \end{Bmatrix} \quad (2)$$

A matlab generated Hann window is shown below.

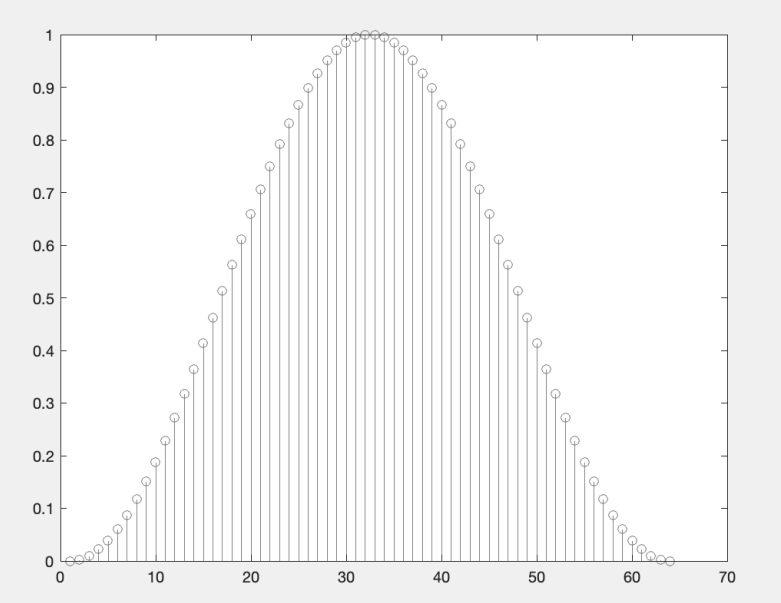

Figure 2: Hann Window Plotted in matlab

Next, the windowed analysis frames are recolated on the time axis a distance of Hs. It is important in choosing the synthesis hopsize to be familiar with the following property of Hann windows. It is encouraged to choose a value of N/2 for the synthesis hopsize to simplify the process of generating a synthesis frame.

$$\sum_{n\epsilon\mathbb{Z}} w\left(r - n\frac{N}{2}\right) = 1 \tag{3}$$

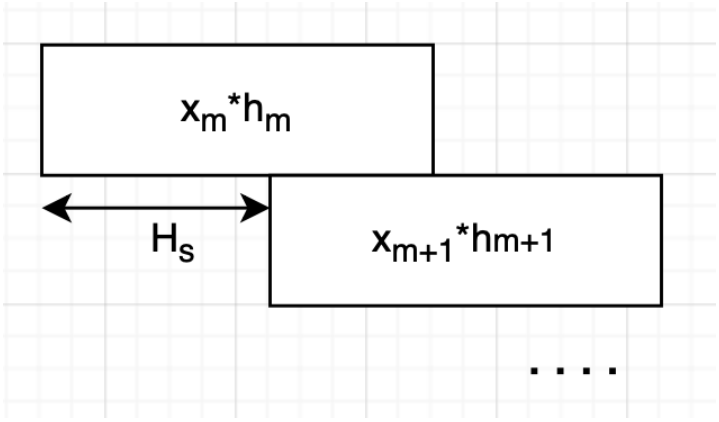

Figure 3: Illustration of Windowing with Synthesis Hopsize H_s

Finally, following equation is used to generate the synthesis frame. Because the windowing has already been performed and the denominator simplifies to 1, the output signal is made simply by adding the analysis frames together.

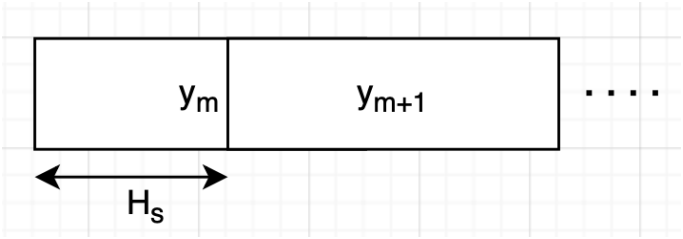

Figure 4: Final Reconstruction of TSM Signal

$$y_m(r) = \frac{w(r) * x_m(r)}{\sum_{n \in Z} w(r - nH_s)} \tag{4}$$

Phase-Vocoder (PV-TSM)

The second implementation of TSM that was implemented is called Phase Vocoder TSM. PV-TSM is performed in the frequency domain and is primarily interested in modifying the phase of each analysis frame to avoid phase mismatch found in the OLA method. In order to implement PV-TSM, the user must do the following steps.

Like the OLA process above, the stretching factor is a ratio of a ratio of synthesis hopsize, H_s over analysis hopsize ,H_a. The first step of the PV-TSM process is also the same as OLA, to divide the input signal, x into analysis frames of length N spaced by analysis hopsize Ha.

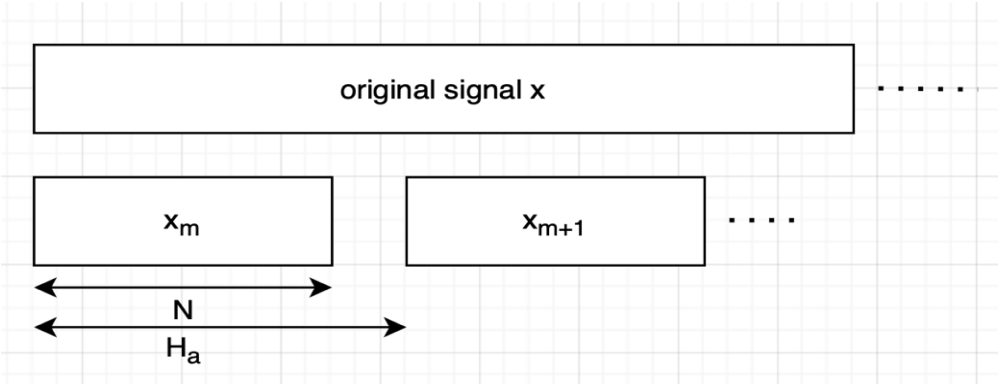

Figure 5: Break Original Signal into Analysis Frames

Next, the Fourier transform of x_m is taken to bring the signal into the frequency domain. Now that the signal is in the frequency domain, the namesake phase vocoder portion of the process begins. The first step is to calculate the magnitude and phase of X and save the phase as the 'initial phase'. This initial phase will be updated each frame. Next, calculate the instantaneous frequency estimate as shown in the equation below. Note that the symbol Ψ means to shift the phase to be between $-\pi$ to π.

$$F^{IF} = \omega + \frac{\Psi(\Phi_2 - (\Phi_1 + \omega\Delta t)}{\Delta t} \quad (5)$$

The instantaneous frequency estimate is used to calculate the modified phase that is shown in the oval in the image below.

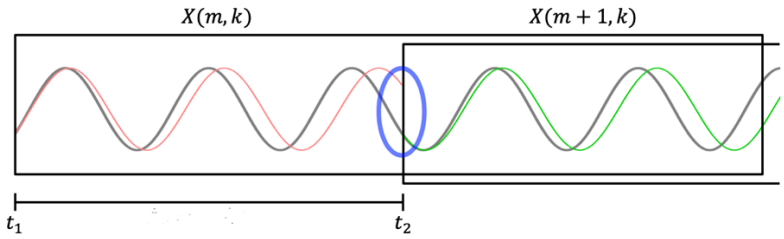

Figure 6: Illustration of Modified Phase (Dreidger)

To calculate this modified phase, the equation below is used which includes the previous modified phase. Note that the modified phase equals the calculated phase of X in the initial analysis frame.

$$\phi_{Mod}(m+1) = \phi_{Mod}(m) + F^{IF} * \frac{Hs}{Ha} \quad (6)$$

After the modified phase is calculated, the equation below is used to find the modified signal in the frequency domain.

$$X^{Mod}(m) = |X(m)| * \exp\left(2\pi i * \phi^{Mod}(m)\right) \quad (7)$$

A Hann window the size of the analysis frame is applied to X^{Mod}. Next, the inverse fft (ifft) is calculated to take the signal back into the time domain. The final step is to reconstruct the signal in the same way as performed in the OLA method. For each frame, the signal was saved in a row in a matrix zero padded by a multiple of the frame length plus the synthesis hopsize H_s. When all of the frames are processed, the matrix is added together to produce the output signal.

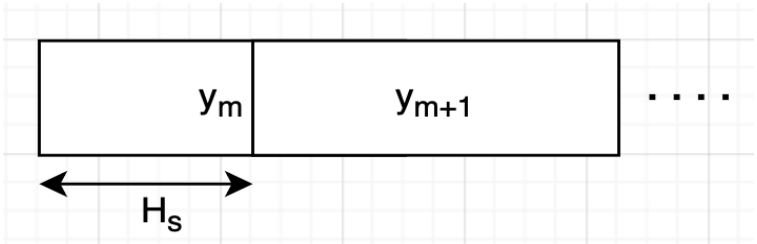

Figure 7: Final Reconstruction of TSM Signal

Harmonic Results

The first portion of the project focused on processing a harmonic signal with both the OLA method and the PV method. The prediction based on the reference material was that the PV method would be better suited for processing the harmonic signal.

OLA was implemented first and was discovered to have limitations processing harmonic signals. Using OLA, a signals periodic structure is not retained well. The paper states that, "While OLA is unsuited for modifying audio signals with harmonic content, it delivers high quality results for purely percussive signals…"(Dreidger). Below is a zoomed in version of a sine wave with its 2x speed and 0.5x speed versions plotted. It is clear that the periodic structure was not preserved for the input sine wave. If the periodic structure had been maintained, the three colored signals would look the same.

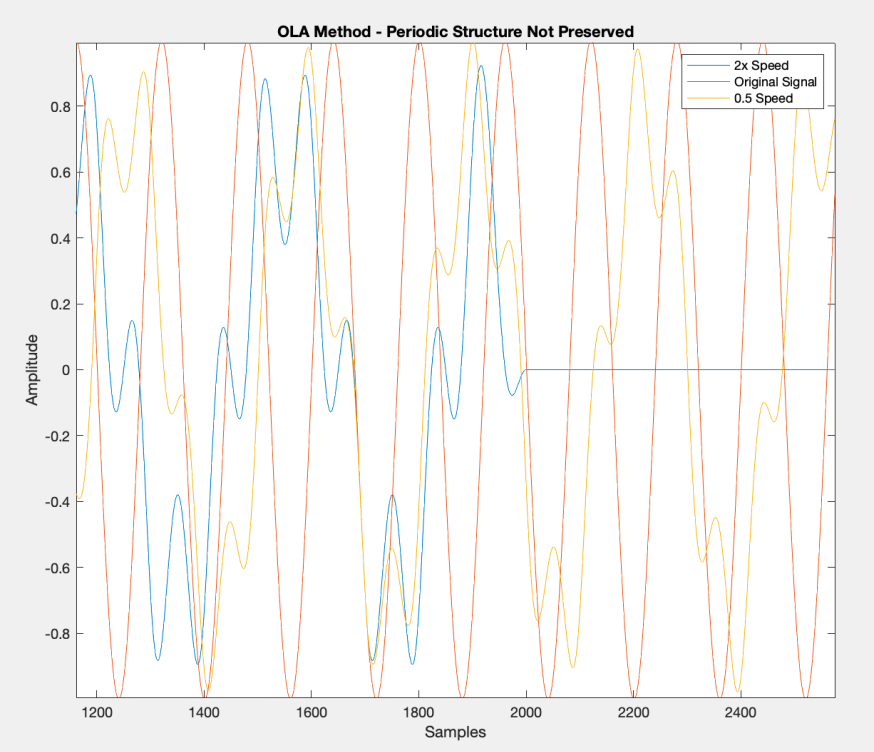

Figure 8: Harmonic Signal with OLA Implementation

The next test involved processing a harmonic signal using the PV method. The same sine wave was inputted into the system and the resulting graph was produced. The PV method introduced in the paper worked to preserve the periodic behavior of the signal but has some clear reverberation. When a speech signal was inputted, the resulting signal sounded echoey. According to the paper, "output signals of PV-TSM commonly have a very distinct sound coloration known as *phasiness,* an artifact caused by the loss of vertical phase coherence." (Dreidger). Although the reverberation is quite serious, the lack of phase jumps makes PV-TSM the obvious choice for harmonic signals.

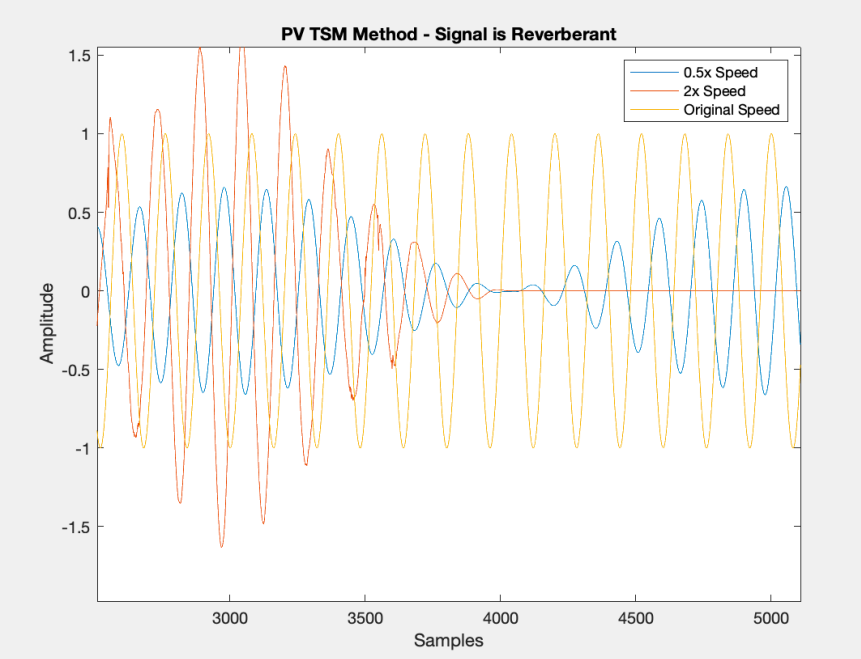

Figure 9: Harmonic Signal with PV Implementation

In a final effort to produce an excellent sounding stretched harmonic signal, a different version of PV-TSM was tested. Dan Ellis of Columbia University implemented the Golden and Dolson phase vocoder algorithm in matlab which produced the following plot. This particular implementation is unique because it does not split the signal up which may be where the phase coherence issues stem from. It is clear that this method was effective at maintaining the periodic behavior of the signal without introducing reverberation.

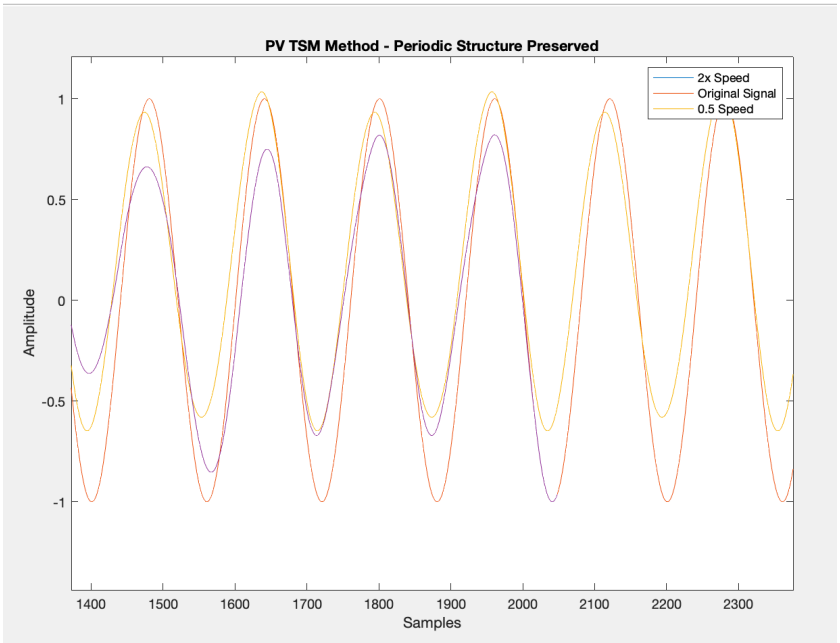

Figure 10: Harmonic Signal with Flanagan & Golden/Dolson Implementation (Ellis)

Percussive Results

Although OLA is not ideal for modifying harmonic signals, it excels at modifying percussive signals. The next image shows the 0.5x speed, original speed, and 2x speed versions of a drum sequence. It is clear from the image below that the OLA method worked extremely well to maintain the beat of the drums, producing clear and crisp drum beats regardless of the signal speed.

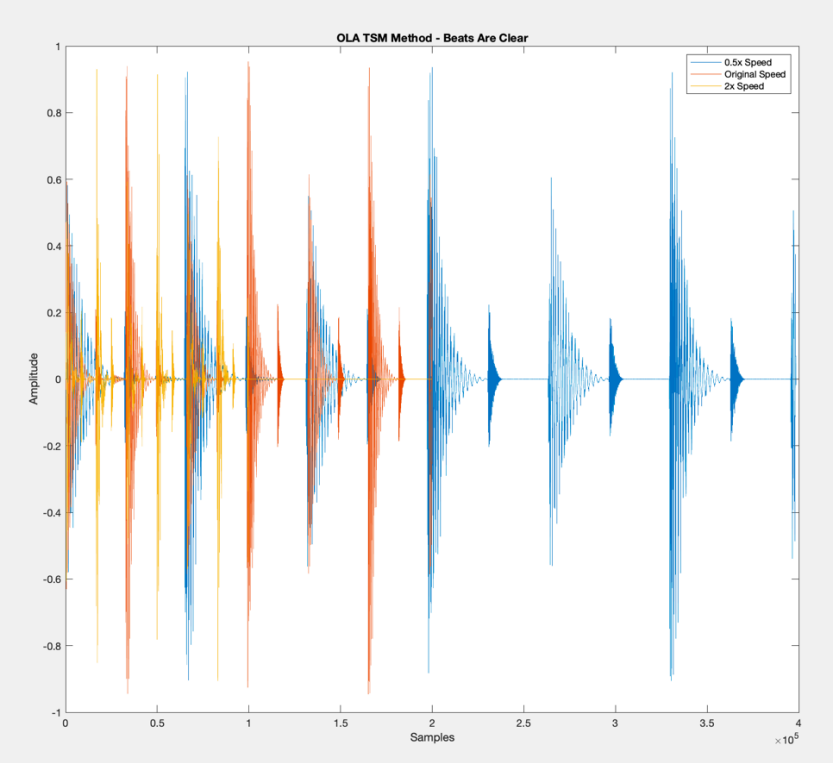

Figure 11: Percussive Signal with OLA Implementation

As discussed above, the PV method of TSM is not ideal for modifying percussive signals . According to Dreidger, “The loss of vertical phase coherence affects the time localization of events such as transients.”. This along with the behavior mentioned above about PV producing phasiness adds up to a subpar result when a percussive signal is processed. In the image below, it is clear that the yellow (0.5x speed) signal is very fuzzy compared to the original (blue) signal. This leads to the drum beat sounding dull and shaky which is not ideal.

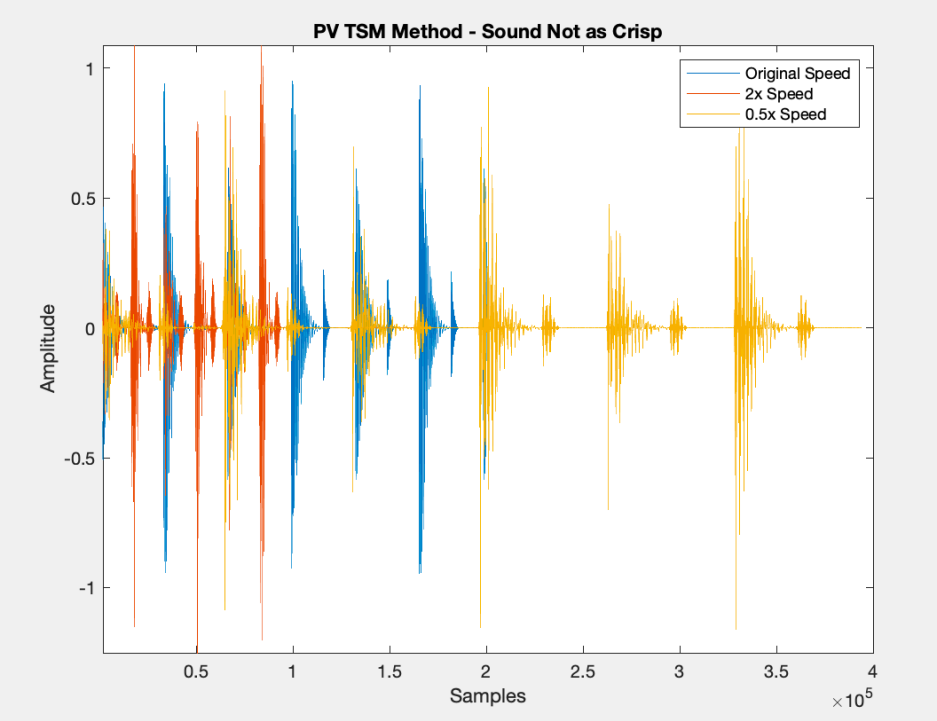

Figure 12: Percussive Signal with PV-TSM Implementation

Conclusions

In this paper, several methods of time-scale modification were presented and their performance was evaluated. It became clear that the OLA method is ideal for performing TSM on percussive signals and that PV-TSM is superior for performing TSM on harmonic signals. While each method for implementing TSM had some drawbacks and some advantages, both have their place in an audio signal processing engineer's toolkit. If more time allowed, an interesting next step for this project would be to implement harmonic-percussive separation on a complex song. That would be a good way to combine the two methods explored above to modify a complicated signal. This project was a good introduction to the field of audio signal processing and a solid jumping off point for future DSP work.

Bibliography

Driedger, Jonathan, and Meinard Müller. “A Review of Time-Scale Modification of Music Signals.” *Applied Sciences*, vol. 6, no. 2, 2016, p. 57., https://doi.org/10.3390/app6020057.

Ellis, Dan. *A Phase Vocoder in Matlab*, https://www.ee.columbia.edu/~dpwe/resources/matlab/pvoc/.

Scarth, Greg. “Untold History: Eventide and the Digital Revolution.” *Attack Magazine*, 15 June 2018, https://www.attackmagazine.com/features/long-read/untold-history-eventide-and-the-digital-revolution/.

Udo Zolzer. *DAFX: Digital Audio Effects*. Wiley, 2002.

Appendix

```
function y_final = TSM(signal_in, alpha)
% TSM(signal_in, alpha) stretches a signal 'signal in' by a factor of
% alpha. This method is based on the OLA-TSM method presented in 'A Review
% of Time-Scale Modification of Music Signals' by Jonathan Dreidger

%Fs for the drum file is 44100


% f=50
% Amp=1
% ts=1/8000;
% T=.5;
% t=0:ts:T;
% ysin=sin(2*pi*f*t);
% plot(t,ysin)

signal_in = signal_in.';

N = 1000;
Hs= N/2;
Ha = Hs/alpha;

[row,col] = size(signal_in);

shift_matrix = zeros(N,col);

for m = 0:((col-N)/Ha)-1
    %separate into analysis frames
    xm = signal_in(:,(m*Ha)+1:N+(m*Ha));
    %window
    w = hann(N).';
    ym = xm.*w;
    %%shift frame based on Hs
    shift_matrix(m+1,m*Hs+1:(m*Hs)+N) = ym;
end

%reconstruct signal
y_final = sum(shift_matrix,1);
```

```
function y_final = TSM_PV(signal_in, alpha)
% TSM_PV(signal_in, alpha) stretches a signal 'signal in' by a factor of
% alpha. This method is based on the PV-TSM method presented in 'A Review
% of Time-Scale Modification of Music Signals' by Jonathan Dreidger

%Fs for the drum file is 44100



%
% f=50
% Amp=1
% ts=1/8000;
% T=1;
% t=0:ts:T;
% ysin=sin(2*pi*f*t);
% ysin = ysin.';



N = 2000;
Hs = N/2;
Ha= Hs/alpha;

[row,col] = size(signal_in);
omega = 2*pi*Ha*[0:N - 1]'/N;
phase_init= zeros(N,1);
phase_mod = zeros(N,1);

shift_matrix = zeros(N,row);

for m = 0:((row-N)/Ha)-1
    %separate into analysis frames
    xm = signal_in((m*Ha)+1:N+(m*Ha),:);
    X = fft(xm);
    Xmag = abs(X);
    Xphase = angle(X);
    %calculate instantaneous frequency estimate
    FIF = omega + ((Xphase-phase_init-omega)-(2*pi*round(Xphase-phase_init-
omega/(2*pi))));
    phase_init = Xphase;
    phase_mod = phase_mod+FIF*alpha;
    %calculate modified FFT
    Xmod = (Xmag.*exp(2*pi*i*phase_mod));
    xMod = real(ifft(Xmod));
    %window
    w = hann(N);
    xMod = xMod.*w;
    %shift frame based on Hs
    shift_matrix(m+1,m*Hs+1:(m*Hs)+N) = xMod;
end

y_final = sum(shift_matrix,1);
```

The following functions were written by Dan Ellis at Columbia.

```
function y = pvoc(x, r, n)
% y = pvoc(x, r, n)  Time-scale a signal to r times faster with phase vocoder
%      x is an input sound. n is the FFT size, defaults to 1024.
%      Calculate the 25%-overlapped STFT, squeeze it by a factor of r,
%      inverse spegram.
% 2000-12-05, 2002-02-13 dpwe@ee.columbia.edu.  Uses pvsample, stft, istft
% $Header: /home/empire6/dpwe/public_html/resources/matlab/pvoc/RCS/pvoc.m,v
1.3 2011/02/08 21:08:39 dpwe Exp $

if nargin < 3
  n = 1024;
end

% With hann windowing on both input and output,
% we need 25% window overlap for smooth reconstruction
hop = n/4;
% Effect of hanns at both ends is a cumulated cos^2 window (for
% r = 1 anyway); need to scale magnitudes by 2/3 for
% identity input/output
%scf = 2/3;
% 2011-02-07: this factor is now included in istft.m
scf = 1.0;

% Calculate the basic STFT, magnitude scaled
X = scf * stft(x', n, n, hop);

% Calculate the new timebase samples
[rows, cols] = size(X);
t = 0:r:(cols-2);
% Have to stay two cols off end because (a) counting from zero, and
% (b) need col n AND col n+1 to interpolate

% Generate the new spectrogram
X2 = pvsample(X, t, hop);

% Invert to a waveform
y = istft(X2, n, n, hop)';
```

```
function D = stft(x, f, w, h, sr)
% D = stft(X, F, W, H, SR)                        Short-time Fourier
transform.
%       Returns some frames of short-term Fourier transform of x.  Each
%       column of the result is one F-point fft (default 256); each
%       successive frame is offset by H points (W/2) until X is exhausted.
%       Data is hann-windowed at W pts (F), or rectangular if W=0, or
%       with W if it is a vector.
%       Without output arguments, will plot like sgram (SR will get
%       axes right, defaults to 8000).
%      See also 'istft.m'.
% dpwe 1994may05.  Uses built-in 'fft'
% $Header: /home/empire6/dpwe/public_html/resources/matlab/pvoc/RCS/stft.m,v
1.4 2010/08/13 16:03:14 dpwe Exp $
```

```
if nargin < 2;  f = 256; end
if nargin < 3;  w = f; end
if nargin < 4;  h = 0; end
if nargin < 5;  sr = 8000; end

% expect x as a row
if size(x,1) > 1
  x = x';
end

s = length(x);

if length(w) == 1
  if w == 0
    % special case: rectangular window
    win = ones(1,f);
  else
    if rem(w, 2) == 0   % force window to be odd-len
      w = w + 1;
    end
    halflen = (w-1)/2;
    halff = f/2;   % midpoint of win
    halfwin = 0.5 * ( 1 + cos( pi * (0:halflen)/halflen));
    win = zeros(1, f);
    acthalflen = min(halff, halflen);
    win((halff+1):(halff+acthalflen)) = halfwin(1:acthalflen);
    win((halff+1):-1:(halff-acthalflen+2)) = halfwin(1:acthalflen);
  end
else
  win = w;
end

w = length(win);
% now can set default hop
if h == 0
  h = floor(w/2);
end

c = 1;

% pre-allocate output array
d = zeros((1+f/2),1+fix((s-f)/h));

for b = 0:h:(s-f)
  u = win.*x((b+1):(b+f));
  t = fft(u);
  d(:,c) = t(1:(1+f/2))';
  c = c+1;
end;

% If no output arguments, plot a spectrogram
if nargout == 0
  tt = [0:size(d,2)]*h/sr;
  ff = [0:size(d,1)]*sr/f;
  imagesc(tt,ff,20*log10(abs(d)));
  axis('xy');
```

```
  xlabel('time / sec');
  ylabel('freq / Hz')
  % leave output variable D undefined
else
  % Otherwise, no plot, but return STFT
  D = d;
end
```

```
function c = pvsample(b, t, hop)
% c = pvsample(b, t, hop)   Interpolate an STFT array according to the 'phase
vocoder'
%     b is an STFT array, of the form generated by 'specgram'.
%     t is a vector of (real) time-samples, which specifies a path through
%     the time-base defined by the columns of b.  For each value of t,
%     the spectral magnitudes in the columns of b are interpolated, and
%     the phase difference between the successive columns of b is
%     calculated; a new column is created in the output array c that
%     preserves this per-step phase advance in each bin.
%     hop is the STFT hop size, defaults to N/2, where N is the FFT size
%     and b has N/2+1 rows.  hop is needed to calculate the 'null' phase
%     advance expected in each bin.
%     Note: t is defined relative to a zero origin, so 0.1 is 90% of
%     the first column of b, plus 10% of the second.
% 2000-12-05 dpwe@ee.columbia.edu
% $Header: /homes/dpwe/public_html/resources/matlab/dtw/../RCS/pvsample.m,v
1.3 2003/04/09 03:17:10 dpwe Exp $

if nargin < 3
  hop = 0;
end

[rows,cols] = size(b);

N = 2*(rows-1);

if hop == 0
  % default value
  hop = N/2;
end

% Empty output array
c = zeros(rows, length(t));

% Expected phase advance in each bin
dphi = zeros(1,N/2+1);
dphi(2:(1 + N/2)) = (2*pi*hop)./(N./(1:(N/2)));

% Phase accumulator
% Preset to phase of first frame for perfect reconstruction
% in case of 1:1 time scaling
ph = angle(b(:,1));

% Append a 'safety' column on to the end of b to avoid problems
```

```
% taking *exactly* the last frame (i.e. 1*b(:,cols)+0*b(:,cols+1))
b = [b,zeros(rows,1)];

ocol = 1;
for tt = t
  % Grab the two columns of b
  bcols = b(:,floor(tt)+[1 2]);
  tf = tt - floor(tt);
  bmag = (1-tf)*abs(bcols(:,1)) + tf*(abs(bcols(:,2)));
  % calculate phase advance
  dp = angle(bcols(:,2)) - angle(bcols(:,1)) - dphi';
  % Reduce to -pi:pi range
  dp = dp - 2 * pi * round(dp/(2*pi));
  % Save the column
  c(:,ocol) = bmag .* exp(j*ph);
  % Cumulate phase, ready for next frame
  ph = ph + dphi' + dp;
  ocol = ocol+1;
end



function x = istft(d, ftsize, w, h)
% X = istft(D, F, W, H)                    Inverse short-time Fourier
transform.
%       Performs overlap-add resynthesis from the short-time Fourier transform
%       data in D.  Each column of D is taken as the result of an F-point
%       fft; each successive frame was offset by H points (default
%       W/2, or F/2 if W==0). Data is hann-windowed at W pts, or
%       W = 0 gives a rectangular window (default);
%       W as a vector uses that as window.
%       This version scales the output so the loop gain is 1.0 for
%       either hann-win an-syn with 25% overlap, or hann-win on
%       analysis and rect-win (W=0) on synthesis with 50% overlap.
% dpwe 1994may24.  Uses built-in 'ifft' etc.
% $Header: /home/empire6/dpwe/public_html/resources/matlab/pvoc/RCS/istft.m,v
1.5 2010/08/12 20:39:42 dpwe Exp $

if nargin < 2; ftsize = 2*(size(d,1)-1); end
if nargin < 3; w = 0; end
if nargin < 4; h = 0; end  % will become winlen/2 later

s = size(d);
if s(1) ~= (ftsize/2)+1
  error('number of rows should be fftsize/2+1')
end
cols = s(2);

if length(w) == 1
  if w == 0
    % special case: rectangular window
    win = ones(1,ftsize);
  else
    if rem(w, 2) == 0   % force window to be odd-len
      w = w + 1;
    end
    halflen = (w-1)/2;
```

```
    halff = ftsize/2;
    halfwin = 0.5 * ( 1 + cos( pi * (0:halflen)/halflen));
    win = zeros(1, ftsize);
    acthalflen = min(halff, halflen);
    win((halff+1):(halff+acthalflen)) = halfwin(1:acthalflen);
    win((halff+1):-1:(halff-acthalflen+2)) = halfwin(1:acthalflen);
    % 2009-01-06: Make stft-istft loop be identity for 25% hop
    win = 2/3*win;
  end
else
  win = w;
end

w = length(win);
% now can set default hop
if h == 0
  h = floor(w/2);
end

xlen = ftsize + (cols-1)*h;
x = zeros(1,xlen);

for b = 0:h:(h*(cols-1))
  ft = d(:,1+b/h)';
  ft = [ft, conj(ft([((ftsize/2)):-1:2]))];
  px = real(ifft(ft));
  x((b+1):(b+ftsize)) = x((b+1):(b+ftsize))+px.*win;
end;
```