It's morphin' time

Jennifer Hsu

Music 421b - Spring 2012

Introduction

It has been my intention, this quarter, to explore various implementations of audio morphing in the spectral domain. It seems that many types of manipulations of sound could be considered audio morphing (convolution, cross-synthesis, cross-fading amplitudes) but what matters with audio morphing is that the sound we make sounds like it is "in-between" the two original source sounds. This paper will take you through my journey and the ideas that I came up with throughout the quarter. Please see `http://ccrma.stanford.edu/~jhsu/421b`, for accompanying sound examples.

Automatic Audio Morphing

I first pored over Malcolm Slaney's paper on automatic audio morphing [6]. The basic idea is to take the MFCC of the two original source sounds. You can use the MFCC to temporally match parts of the sound that should be matched (ie. rhythms or beats for rap music or melody for classical and pop music). The MFCC representations are also inverted by taking the discrete cosine transform, undoing the logarithmic scaling, and interpolating to get back to a smooth spectrogram [5]. This gives a representation of the overall spectral shape without the pitch information. To get the pitch information as a spectrogram, the original FFT spectrogram can be divided by this smooth spectrogram. The smooth spectrogram and the pitch spectrogram are the necessary representations for performing audio morphs.

For a smooth morph, we must match the features in our representations. Malcolm argues that It is more important to match the features in our pitch spectrogram than it is to match the features in our smooth spectrogram. Secrest and Doddington showed a way of finding the broad, overall pitch of the sound [4]. We can use this technique to find the overall pitch of the content and then crossfade between the amplitudes of the different frequencies to get a smooth morph.

There are also considerations for when we interpolate across time. Suppose we have two signals that we are morphing between: s1 and s2. We also have two times that we are morphing between: t1, t2. We can use a simple weighted some to find the curve, s at time t and morph level lambda, that lies between these curves:
$s(\lambda, t) = (1 - \lambda)s1(t1) + (\lambda)s2(t2)$

Although I can understand the theory behind this method, I have not been able to complete its implementation, though I have borrowed ideas from this paper and come up with implementations of my own.

Simple Morph

The simplest morph I can think of is a cross-fade of amplitudes in the time domain. This is not spectral audio morphing, but I think that it is nice to compare all the other implementations to this simple morph. This just involves fading out the amplitude of the first sound and fading in the amplitude of the second sound and adding/overlapping these fadeouts so that we transition from the first sound to the second sound. (audio sample located here: `http://ccrma.stanford.edu/~jhsu/421b/simple_morph.wav`

Cross-synthesis

I began by working on an implementation of cross-synthesis in Matlab. Cross-synthesis involves impressing the spectral envelope of one sound onto the flattened spectrum of another envelope. For me, the hardest part with cross-synthesis was getting the smooth spectral envelopes. In the following example, I used the cepstrum to get the smooth spectral envelope. The code here is based on code found at `http://www.cic.unb.br/~lamar/te073/Aulas/mfcc.pdf`.

```
% read in wav file
% modulator
[y fs] = wavread('../data/dropitonmeMono.wav');
[siz temp] = size(y);
% carrier (white noise generated to length of modulator)
x = 0.7*(2*rand(siz, temp)-1);

% parameters
frame_duration = 256;
number_of_frames = floor(siz / frame_duration);
number_of_features = 12;

Nfft = 512;
window = blackman(Nfft);
Y = specgram(y, Nfft, fs, window, round(3/4*length(window)));
```

```
% accumulators
mod = zeros(length(y), 1);
car = zeros(length(x), 1);
xs = zeros(length(y), 1);
sumcepM = zeros(1, 500);
sumcepC = zeros(1, 500);

%%%%%%%%%% where the real work happens %%%%%%%%%%%%%
for i=1:number_of_frames

    % take one frame
    begin_sample = 1+(i-1)*frame_duration;
    end_sample = i*frame_duration;
    % modulator
    s = y(begin_sample:end_sample);
    s = s(:);
    % carrier
    c = x(begin_sample:end_sample);
    c = c(:);

    % calculate spectra
    % windowing
    sw = s.*hamming(frame_duration);
    cw = c.*hamming(frame_duration);

    % STEP ONE: stft calculations of each time frame
    frame_fftM = fft( sw );
    mod(begin_sample:end_sample) = mod(begin_sample:end_sample) + ifft(frame_fftM);
    frame_fftC = fft( cw );
    car(begin_sample:end_sample) = car(begin_sample:end_sample) + ifft(frame_fftC);

    % STEP TWO: cepstra calculations for getting spectral envelope
    dft_rc_tempM = log( abs( fft( sw ) ) );
    dft_rcM = real( ifft( dft_rc_tempM ) );
    % take out negative quefrencies and just take the features
    dft_rcM = dft_rcM(1:floor(frame_duration/2 + 1 ));
    dft_rcM = dft_rcM(1:number_of_features);
    dftcepstraM(i,:) = dft_rcM';
    % same for carrier
    dft_rc_tempC = log( abs( fft( cw ) ) );
    dft_rcC = real( ifft( dft_rc_tempC ) );
    dft_rcC = dft_rcC(1:floor(frame_duration/2 + 1 ));
    dft_rcC = dft_rcC(1:number_of_features);
    dftcepstraC(i,:) = dft_rcC';
```

```
    % reconstruct the spectrum
    cepM = dftcepstraM(i,:);
    Nfft = frame_duration;
    lengthOfPositivePart = floor(Nfft/2 + 1);
    Npad = lengthOfPositivePart - length(cepM);
    cepM = [cepM zeros(1, Npad)];
    cepM = fft(cepM);
    cepM = exp(real(cepM));
    % same for carrier
    cepC = dftcepstraC(i,:);
    cepC = [cepC zeros(1, Npad)];
    cepC = fft(cepC);
    cepC = exp(real(cepC));

    % we have envelopes for carrier and modulator
    % now we need to interpolate to correct lengths
    smoothM = interp(cepM, ceil(length(frame_fftM)/length(cepM)));
    smoothC = interp(cepC, ceil(length(frame_fftC)/length(cepC)));

    % STEP 3: divide spectrum of each carrier frame by
    % envelope to flatten it
    flat_frameC = (frame_fftC)'./smoothC(1:length(frame_fftC));
    %plot(20*log10(abs(flat_frameC)/max(abs(flat_frameC))));
    %pause(0.01);

    % STEP 4: multiply the flattened spectral frame by env of
    % corresponding modulator frame, replacing carrier's env
    % by modulator's envelope
    XS = flat_frameC.*smoothM(1:length(flat_frameC));
    %plot( 20*log10(abs(xs)/max(abs(x))));
    %pause(0.01);
    %plot(real(ifft(xs))); pause(0.01);
    xs(begin_sample:end_sample) = xs(begin_sample:end_sample) + real(ifft(XS))';

    % sum (this isn't really necessary, just here to prove that
    % indexing is correct
    sumcepM(1:length(cepM)) = sumcepM(1:length(cepM)) + cepM;
    sumcepC(1:length(cepC)) = sumcepC(1:length(cepC)) + cepC;
end

soundsc(xs, fs)
```

Here is a figure I made to get a better representation of what is going on with the smoothing:

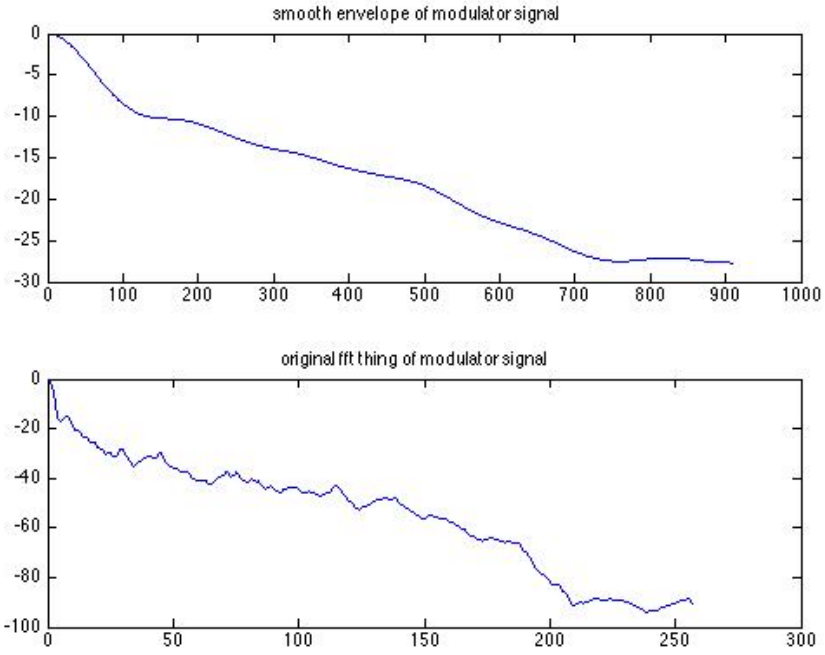

Figure 1: comparison of smooth spectral envelope to original spectral envelope

If you listen to the sound file at `http://ccrma.stanford.edu/~jhsu/421b/xs.wav`, you will most likely notice that it sounds choppy and uneven. I think that may be caused by the interpolation that I am doing. If I continue with this, I would hope to smooth it out and do some finer interpolation. Even though the sound is not perfect, writing the code in Matlab reinforced the method and really helped me learn about playing with sound in the spectral domain.

I proceeded to implement this in SuperCollider. In order to keep track of the different buffers (fft or cepstrum), I made many global variables. I probably could have done this in a more elegant way, but keeping it like this kind of makes it more like the Matlab code.

```
// load things
(
b = Buffer.read(s, "/Users/home/Documents/Jennifer/421b/data/dropitonmeMono.wav");
c = Bus.new('control', 0, 13);
d = Buffer.read(s, "/Users/home/Documents/Jennifer/421b/data/rainm.wav");
~fftbufc = Buffer.alloc(s, 2048);
~fftbufm = Buffer.alloc(s, 2048);
~cepbufc = Buffer.alloc(s, 1024);
~cepbufm = Buffer.alloc(s, 1024);
~envc = Buffer.alloc(s, 2048);
~envm = Buffer.alloc(s, 2048);
)

(
SynthDef("morphintime", {

        |out = 0, bufnum = 0, bufnum2 = 1|
        var in, in2, chain, chain2, chain3, cepsch, cepsch2, fftsize;
        fftsize = 2048;

        bufnum = b.bufnum;
        bufnum2 = d.bufnum;

        // 1. STFT of signal
        // 2. smooth spectral envelope
        // get cepstrum of modulating signal
        in = PlayBuf.ar(1, bufnum, BufRateScale.kr(bufnum), 1, 0, 1);
        chain = FFT(~fftbufm, in);
        cepsch = Cepstrum(~cepbufm, chain);

        // get cepstrum of carrier signal
        in2 = PlayBuf.ar(1, bufnum2, BufRateScale.kr(bufnum2), 1, 0, 1);
        chain2 = FFT(~fftbufc, in2);
        cepsch2 = Cepstrum(~cepbufc, chain2);
```

```
        // PV_BrickWall can act as a low-pass filter, or here, as a wol-pass lifter...
        // ...in practical terms, produces a smoothed version of the spectrum
        // get smooth version of modulator
        cepsch = PV_BrickWall(cepsch, -0.95);
        ICepstrum(cepsch, ~envm);

        // get smoothed version of carrier
        cepsch2 = PV_BrickWall(cepsch2, -0.95);
        ICepstrum(cepsch2, ~envc);

        // 3. divide spectrum of each carrier frame by
        // smooth spectral envelope (to flatten)
        chain2 = chain2.pvcalc2(~envc, fftsize, {|mags, phases, mags2, phases2|
                [mags / mags2, phases - phases2]
        }, frombin: 0, tobin: 125, zeroothers: 0);

        // 4. multiply flattened spectral carrier frame with smooth spectral envelope
        // of modulator
        chain2 = chain2.pvcalc2(~envm, fftsize, {|mags, phases, mags2, phases2|
                [mags * mags2, phases + phases2]
                }, frombin: 0, tobin: 125, zeroothers: 0);

        Out.ar( out, Pan2.ar(IFFT(chain2)) );

}).send(s);
)

x = Synth.new("morphintime");
```

The good news with this SuperCollider implementation is that it sounds better than the Matlab one. Here's the link: `http://ccrma.stanford.edu/~jhsu/421b/xsynth1.wav`.

Bin Manipulation Morphing

I began reading about Laroche and Dolson's extention of IFFT synthesis to see if I could apply that idea to my morphs. As I understand it, the method involves finding spectral peaks and defining "regions of influence" around these peaks and then moving these regions to different areas along the spectrum (for frequency-shifting). Instead of modeling the spectral peak off of some other function, the original spectral portion around each peak is preserved [1, 2].

I think that I could apply this to morphs by replacing regions of interest in

one signal with regions of interest from another signal. I did not get to implement this, so I do not know what it would sound like, but I think it would be an interesting experiment.

I did borrow from this bin manipulation idea and decided to replace bins in one spectrum with bins from another spectrum. Luckily, SuperCollider has a PV unit generator that does this in a random order: PV_RandWipe. In the following code, you can manipulate the morph factor by moving the mouse along the x-axis. Moving the mouse along the y-axis resets the random replacement. A sound example can be found here: `http://ccrma.stanford.edu/~jhsu/randFade.wav`.

```
(
c = Buffer.read(s, "/Users/home/Documents/Jennifer/421b/data/blueMono.wav");
d = Buffer.read(s, "/Users/home/Documents/Jennifer/421b/data/pianoMono.wav");
)

(
SynthDef("randFade", { |out = 0, sndbufnum1, sndbufnum2|
        var in, chain, in2, chain2, fftsize;
        fftsize = 1024;
        in = PlayBuf.ar(1, sndbufnum1, BufRateScale.kr(sndbufnum1), loop:1);
        in2 = PlayBuf.ar(1, sndbufnum2, BufRateScale.kr(sndbufnum2), loop:1);
        chain = FFT(LocalBuf(fftsize), in);
        chain2 = FFT(LocalBuf(fftsize), in2);
        chain = PV_RandWipe(chain, chain2, MouseX.kr, MouseY.kr > 0.5);
        Out.ar(out, 0.5*IFFT(chain).dup);
}).send(s);
)

z = Synth("randFade", [\out, 0, \sndbufnum1, c.bufnum, \sndbufnum2, d.bufnum]);
```

Continuing with this idea of bin replacement, I played around with other PV UGens to replace bins in a non-random order. This first one morphs by cross-fading from bins with lower magnitudes to bins with higher magnitudes. Moving the mouse along the x-axis affects the morph. Here is a sound example: `http://ccrma.stanford.edu/~jhsu/421b/mbelow.wav`

```
(
SynthDef("mbelow", { |out = 0, sndbufnum1, sndbufnum2|
        var in, chain, in2, chain2,fftsize;
        fftsize = 1024;
        in = PlayBuf.ar(1, sndbufnum1, BufRateScale.kr(sndbufnum1), loop:1);
        in2 = PlayBuf.ar(1, sndbufnum2, BufRateScale.kr(sndbufnum2), loop:1);
        chain = FFT(LocalBuf(fftsize), in);
        chain2 = FFT(LocalBuf(fftsize), in2);
        chain = PV_MagBelow(chain, MouseX.kr(1,20));
```

```
        chain2 = PV_MagBelow(chain2, 20-MouseX.kr(1,20));
        chain = chain.pvcalc2(chain2, fftsize, { |mags, phases, mags2, phases2|
                [mags*mags2, phases+phases2]
        }, frombin: 0, tobin: 1024, zeroothers: 0);
        Out.ar(out, IFFT(chain).dup);
}).send(s);
)
```

This example uses PV_BinWipe which combines the low and high frequency bins from two inputs. Moving the mouse to the left replaces bins from the second input from the high bins down to the low bins, and moving the mouse to the right replaces the from the low bins up to the high bins. A sound example can be found here: `http://ccrma.stanford.edu/~jhsu/421b/mbinwipe.wav`

```
(
SynthDef("mbinwipe", { |out = 0, sndbufnum1, sndbufnum2|
        var in, chain, in2, chain2, fftsize;
        fftsize = 1024;
        in = PlayBuf.ar(1, sndbufnum1, BufRateScale.kr(sndbufnum1), loop:1);
        in2 = PlayBuf.ar(1, sndbufnum2, BufRateScale.kr(sndbufnum2), loop:1);
        chain = FFT(LocalBuf(fftsize), in);
        chain2 = FFT(LocalBuf(fftsize), in2);
        chain = PV_BinWipe(chain, chain2, MouseX.kr(-1,1));
        Out.ar(out, 0.5*IFFT(chain).dup);
}).send(s);
)
```

Morphin' Time

I really liked the way I had a control over some type of morph factor with these bin manipulation morphs. That is, I could move my mouse and control the morph factor. I decided to add this to the cross-synthesis SuperCollider code that I was working with in the beginning. I wanted to put in a morph factor λ where when $\lambda = 0$ we would be completely at one sound and at $\lambda = 1$ we would be completely at the other sound. Values for λ in between 0 and 1 would be morphs that are "in-between"

I first tried to "de-emphasize" a signal by taking down it's magnitude, but that does not work when you need to multiply magnitudes to get the sound. We would end up with no sound since we would multiply by 0 at the ends when $\lambda = 0$ and $\lambda = 1$. Lauchlan Casey sent me his solution to this problem which is the "Ghetto-Morph"'. His solution uses three buffers of sound. Two buffers hold the original sound and the third buffer holds the convolution of the two signals. We cross-fade between the three buffers to achieve a morph. It is not the smoothest morph, so I tried to improve on the idea by adding my bin replacement scheme

from above. This did seem to make a smoother morph. Here is a sound example: http://ccrma.stanford.edu/~jhsu/421b/g_morph_below.wav.

While working on improving the "Ghetto-Morph," I came up with a solution to my problem involving morphing by a factor with cross-synthesis. I looked at convolution, since they are similar. Since an impulse is identity for convolution in the time domain, we can look at it in the frequency domain. The Fourier transform of an impulse is all ones in the magnitude, and I believe the phases are all 0 (I plotted it in Matlab and that is what it looked like). I figured that as the morphing factor tends toward 0 or 1, I can interpolate the magnitudes toward 1 and the phases toward 0 as necessary. This turned out to work quite well and the code for convolution morphing over time is below.

```
(
c = Buffer.read(s,"/Users/home/Documents/Jennifer/421b/data/blueMono.wav");
d = Buffer.read(s, "/Users/home/Documents/Jennifer/421b/data/pianoMono.wav");
)

(
SynthDef("whiten", { |out=0, sndbufnum, sndbufnum2|

        var in, chain, in2, chain2, fftsize;
        fftsize = 1024;
        in = PlayBuf.ar(1, sndbufnum, BufRateScale.kr(sndbufnum), loop: 1);
        in2 = PlayBuf.ar(1, sndbufnum2, BufRateScale.kr(sndbufnum2), loop: 1);
        chain = FFT(LocalBuf(fftsize), in);
        chain2 = FFT(LocalBuf(fftsize), in2);

        chain = chain.pvcalc2(chain2, fftsize, {
                |mags, phases, mags2, phases2|
                // interpolate magnitudes up to 1
                mags = MouseY.kr(0,1) * (1-mags) + mags;
                mags2 = MouseY.kr(1,0) * (1-mags2) + mags2;
                // interpolate phases down to 0
                phases = phases - (phases*MouseY.kr(0,1));
                phases2 = phases2 - (phases2*MouseY.kr(1,0));
                [mags * mags2, phases + phases2]
        }, frombin:0, tobin:1024, zeroothers:0);

        Out.ar(out, 0.5 * IFFT(chain).dup);
}).send(s);
)

x = Synth("whiten", [\out, 0, \sndbufnum, c.bufnum, \sndbufnum2, d.bufnum]);
```

I also added this to my cross-synthesis code which only involves modifying the last step in the SynthDef:

```
// multiply flattened spectral carrier frame with smooth spectral envelope
// of modulator
 chain2 = chain2.pvcalc2(~envm, fftsize, {|mags, phases, mags2, phases2|
        [((MouseY.kr(1,0)*(1-mags))+mags) * (MouseY.kr(0,1)*(1-mags2))+mags2),
          (phases-(phases*MouseY.kr(0,1))) + (phases2-(phases2*MouseY.kr(1,0)))]
}, frombin: 0, tobin: fftsize, zeroothers: 0);
```

The two sound examples can be found here: `http://ccrma.stanford.edu/~jhsu/421b/conv_morph.wav` and `http://ccrma.stanford.edu/jhsu/421b/ceps_morph.wav`. In my opinion, the convolution morph sounds better.

End

The bulk of my interest with audio morphing has been centered around creating new timbres, interesting sounds, and moving between them. I still hope to implement the Laroche and Dolson method to explore the other types of sound that can be created. I also hope to add GUIs to my SuperCollider implementations. Using the mouse is a quick test tool, but having a GUI would be good for providing feedback for morphing control.

Sources

A list of papers that influenced my thinking and also links to sites that were of great help to me:

[1] Dolson, M., Laroche, J. (1999) "Improved phase vocoder time-scale modification of audio," IEEE Transactions on Speech and Audio Processing, vol. 7, no. 3, pp. 323332.

[2] Laroche, J., Dolson, M. (1999) "New phase-vocoder techniquesfor pitch shifting, harmonizing and other exotic effects," Proc. of IEEE Workshop on Applications of Signal Processingto Audio and Acoustics (WASPAA), pp. 9194.

[3] Olivero, A., Depalle, P., Torr, B., Kronland-Martinet, R. (2012). "Sound Morphing Strategies Based on Alterations of Time-Frequency Representations by Gabor Multipliers," AES Conference: Intl. Conf. on Applications of Time-Frequency Processing in Audio.

[4] Secrest, B., Doddington, G. (1983) "An integrated pitch tracking algorithm for speech systems," Proc. of IEEE Intl. Conf. on Acous., Speech and Sig. Proc., vol.8, pp. 1352- 1355.

[5] Slaney, M., Naar, D., Lyon, R. F. (1994). "Auditory model inversion for sound separation," Proc. of IEEE Intl. Conf. on Acous., Speech and Sig. Proc., Sydney, vol. II, pp. 77-80.

[6] Slaney, M., Covell, M. Lassiter, B. (1996). "Automatic Audio Morphing", Proc. of IEEE Intl. Conf. on Acous., Speech and Sig. Proc., Atlanta, pp. 14.

[7] http://www.ee.columbia.edu/~dpwe/resources/matlab/rastamat/: a resource of matlab files for pattern recognition

[8] http://www.cic.unb.br/~lamar/te073/Aulas/mfcc.pdf: explanation of cepstra and MFCC along with example code of which I based my code

[9] http://music.columbia.edu/cmc/musicandcomputers/chapter5/05_06.php: simple, overall description of audio morphing