An Efficient Phase-Vocoder Algorithm for Pitch Shifting in Haskell

Scott H. W. Snyder
May 10, 2010

Advised by Paul Hudak
Yale University
Department of Computer Science Senior Project
1 Abstract

The Haskell programming language has seen great developments with regard to the world of audio signal generation and music composition with the advent of Euterpea and Haskore, respectively. With the combination of the two, one can write (manually or algorithmically) and hear/output the results using synthesized signals based on a variety of input parameters.

However, one area of Euterpea that has not been fully explored is that of digital signal processing on existing waveforms. With a certain amount of Euterpea expertise, it is possible to build a complex instrument (or set of instruments) using CSound-like parameters. But while the world of digital instrument synthesis suffices for many situations, sampling real instruments is often the only way to get a realistic sounding result from MIDI or Haskore information.

With that in mind, this project set out to test a few different “pitch-shift” algorithms designed to take an input waveform, stretch the waveform to a new length without losing pitch or tonal information, and resample to transpose to the desired pitch.

The major portion of this project focuses on one of the more complicated audio timescale/pitch modification algorithms known as Phase Vocoder Shift. It utilizes a FFT to compute the instantaneous frequency to amplitude relationship of many small signal frames. After applying transformation to these frames, an inverse FFT is taken on each of the blocks, which are then added together and resampled to create a smooth output wave.

2 Introduction

There are several ways to attain a pitch-shifted audio file. The three outlined here include strict resampling, time domain harmonic scaling, and the algorithm implemented below, phase vocoder pitch shift.

Pitch shifting has been around for many years and its roots can be traced back to the slowing down or speeding up of an analog recording. This is known best as the “chipmunk effect” named after the ubiquitous music group Alvin and the Chipmunks. The first step towards understanding how to pitch shift without affecting the clip length is understanding the digital version of this chipmunk effect, also known as resampling.

When a WAVE file is encoded, it maintains a sample rate (often at 44100 or 48000 Hz) that determines how quickly a media player should output the list of integers that make up the waveform data. At 44.1 kHz, this means that a player
will literally output 44100 samples every second. With this we can see how simple it might be to reproduce this chipmunk effect digitally. In fact it can be achieved by simply discarding (or duplicating) a certain number of samples from the file and re-encoding with the original sampling rate. To increase the pitch by a whole octave, simple remove one half of the original samples and export at the same sampling rate (see Figure 1).

![Diagram showing resampling, shift up one octave](http://www.guitarpitchshifter.com/)

**Figure 1: Resampling, shift up one octave**

The code for this process in Haskell is as simple as:

```haskell
dropnth xs n = case drop (n-1) xs of
    (y:ys) -> y : dropnth ys n
    []  -> []

output = dropnth input 2
```

Where `input` and `output` are of type `[Integer]`.

However, removing one half of the samples and playing at the same speed means that the output will also be half as long. It is for this reason that digital resampling, by itself, is not an intelligent enough way to shift pitch significantly, especially for pitched instruments.

---

1 Image taken from http://www.guitarpitchshifter.com/
The next algorithm to try is known as Time Domain Harmonic Scaling, invented by Rabiner and Schafer in 1978. The central idea behind this algorithm is to find the period of a given waveform and fade one period into another.

This project’s exploration of this algorithm found these periods by detecting zero crossings in the input waveform and merging duplicates of the split list together. In other words, functions were defined to find the zero crossings:

```haskell
splitZeroCrossings =
  let f x y =
      ((x >= 0) && (y >= 0)) || ((x < 0) && (y < 0))
  in groupBy f

to repeat split lists:

duplicate n = concatMap (replicate n)

and to merge these all together:

merger n [] = []
merger n ls = let (h,hs) = splitAt n ls
  in (concat h) : m

the combination of these parts looks as follows:

output = concat(duplicate 2
 (merger 20 (splitZeroCrossings orig)))

where orig and output are both of type [Double] ranging from -1.0 to 1.0.

This works with some success. If we can easily find the zero crossings, this does in fact stretch the input sound. Unfortunately, for polyphonic or complex tones, discovering these zero crossings is either impossible or not useful. Even with monophonic sounds, the result is often a sort of stuttering mess not very recognizable when compared to the original.

Assuming we do get a sufficiently good stretched sound, changing the pitch is just a matter of applying the digital resampling discussed above. Now that our waveform is twice as long at the same pitch, applying the resampling will cut its length in half and also increase the pitch by one octave. With this we have arrived at a waveform that is the same length as the original and one octave higher.

3 Analysis of Algorithm
The third and final algorithm explored by this project is far more complex than the previous two. We will build a phase vocoder, named such because it scales the time domain as well as the frequency based on wave phase, where phase represents the percentage of a cycle the wave has traversed.

In doing this, we can elongate small frames of the signal while accounting for discontinuities at frame bounds.

The overarching idea behind the phase vocoder algorithm is as follows:
- Split the input signal into many small frames with a high percentage of overlap
- Apply a Hann window FFT on each frame
- Account for difference in phase from one frame to the next
- Using phase different information, find frame’s frequency deviation and true frequency
- Correct phase for each frame
- Take Hann windowed inverse FFT of each frame
- Compress frames with different overlap percentage

We will take a look at each of these parts in depth.

**Splitting the signal into frames**

![Figure 2: Split into overlapping frames](image)

We split the input waveform into frames of size 1024 samples and with skip size of 256 samples. With this the frames have an overlap of 75%. The code for this is as follows:

```haskell
createFrames :: Array Integer e -> Array Integer [e]
createFrames arr =
  let num = ((snd (bounds arr)) - windowSize) `div` skipSize
      lst = elems arr
  in createFrame i = take (fromIntegral windowSize)
```
$\text{drop} \ (i \times (\text{fromIntegral} \ \text{skipSize})) \ \text{lst}$

in listArray (0,num-1)(map createFrame [0..fromIntegral(num-1)])

With these frames, we can run the Fourier Transform on each.

**FFT**

On each of these frames, we need to take an FFT to discover phase information. But first, we need to create a Hann window and use this function as a window function to select a subset of a series of samples. Then we can perform a Fourier transform or other calculations. The purpose of this is that while a rectangular window has more energy in the side lobes, a Hanning window focuses most of its energy near the center. The result looks more like an impulse, which would lead to a perfect reconstruction of the signal.

![Hann window](image)

**Figure 3: Hann window**

\[ w(n) = 0.5 \left(1 - \cos \left(\frac{2\pi n}{N-1}\right)\right) \]

\[
\begin{align*}
\text{hann} :: & \ \text{Int} \to \text{Array} \ \text{Int} \ \text{Double} \\
\text{hann} = & \ \text{makeArray} \ \text{hann}' \\
\text{hann}' :: & \ \text{Double} \to \text{Double} \to \text{Double} \\
\text{hann}' \ m \ n = & \ 0.5 - 0.5 \times \cos(2 \times \pi \times n / m) \\
\text{makeArray} :: & (\text{Double} \to \text{Double} \to \text{Double}) \to \text{Int} \to \text{Array} \ \text{Int} \ \text{Double} \\
\text{makeArray} \ \text{win} \ m = \\
& \ \text{let} \ \text{md} = \text{fromIntegral} \ m \\
& \ \text{in} \ \text{listArray} \ (0,m) \ $ \\
& \ \text{map} \ (\text{win} \ \text{md} \ \text{fromIntegral}) \ [(0::\text{Int}) \ ..]
\]
We multiply each frame in our signal by this window. Now we can transform each frame with a Fast Fourier Transform.

**Phase difference**

It is unlikely that the frequency of the signal we are analyzing is exactly the same as the window size we chose to use. This is known as phase difference or phase shift from frame to frame. If we splice the frames back together (keep in mind: with a different overlap size) without accounting for this phase difference, we will get artifacts and discontinuities as each break, causing a stuttering effect, much like what we experienced with time domain harmonic scaling.

![Figure 4: Example of phase difference between frames](image)

To prevent this problem, the frequency deviation from the frame is calculated and then wrapped. This amount is added to the frame frequency to obtain the true frequency of the component within the frame. In addition, because the output from the FFT wraps, we must rescale to $-\pi$ to $\pi$. Now we can use this phase information to find the true frequency for each frame.

```haskell
changePh = zipWith (zipWith (-)) framesPhases framesPhasesPrev  -- remove expected phase difference
changePh' = zipWith (zipWith (-))
    (repeat [0, (2*pi*(fromIntegral skipSize))]/(fromIntegral windowSize)\ldots)
    changePh

changePh'Mod1 = map (map (+ pi)) changePh'
changePh'Mod = map (map (fmod(pi*2))) changePh'Mod1

freq1 = repeat (map (* ((2*pi)/(fromIntegral windowSize)))) [0\ldots]
freq2 = map (map (/ (fromIntegral skipSize))) changePh'Mod
freqs = zipWith (zipWith (-)) freq1 freq2
```
The new phase of each bin can then be calculated by adding the phase shift required to avoid discontinuities.

**Inverse FFT, combine frames**

We can now take the inverse of the FFT for each frame. In addition, we will want window with another Hann function, this time to smooth the signal.

At this point, it is time to recombine the frames to make a single waveform.

```haskell
combineFrames' :: (Num a) => [[a]] -> Int -> [a]
combineFrames' lst v =
  let
    l = head lst
    ls = tail lst
    x = head l
    xs = tail l
  in
    if ls == [] then l else
      if (length l) > v
        then
          x:(combineFrames' (xs:ls) v)
        else
          let (l1, l2) = splitAt v (head ls)
            in (zipWith (+) l l1) ++ (combineFrames' (l2:(drop 1 ls)) v)

The output can be converted back to Integers and written to a waveform. This will result in a time scaled output.

If in addition to the above steps we digitally resample as described earlier, we can achieve a shifted pitch. If we are not rescaling to 2x, 3x, etc, we must use a linear interpolation algorithm where alpha represents the pitch scaling factor.

```haskell
out = map (interpolate
  (zip
    (map fromIntegral [0..(outSLen-1)]) outStretched))
  [0, alpha..((fromIntegral outSLen)-1)]

interpolate dat x =
  case M.splitLookup x vals of
    (_, Just y, _) -> y
    (M.maxViewWithKey -> Just ((x1, y1), _),
      Nothing,
      M.minViewWithKey -> Just ((x2, y2), _))
      -> ((x-x1)*y2+(x2-x)*y1)/(x2-x1)
error "Out of range"
where vals = M.fromList dat

4 Results

While I was able to achieve some kind of time scale with each of these algorithms, the output was not as ideal as had hoped. A large amount of strange reverb was added to each output signal. This is known as "phase smearing" and can never be truly eliminated. In addition I had countless problems recombining the frames at the end of the algorithm.

5 Acknowledgements

I would like to thank Paul Hudak for helping me discover the more musical world of computer science and for being so patient and understanding. Thank you.