MUSIC 420B Project Report

Thomas Walther

June 8, 2013

Contents

1 Introduction

The goal of my 420B project is to create a real-time additive guitar synthesizer. The synthesizer will take the analog guitar signal input and make it sound like a traditional additive synthesizer. This will be done in two steps: alter the waveform of the analog guitar signal, and then mix that waveshaped signal with transposed copies. The project is divided in three parts: create a waveshaping algorithm for monophonic signals, implement a pitch-shifting algorithm for polyphonic signals, and optimize the waveshaping algorithm for polyphonic signals. Since the synthesizer is intended for real-time use, the project always focuses on low latency. This will be especially important selecting a pitch-shifting algorithm.

2 Project Overview

I implemented waveshaping algorithms for square, triangle and 8-bit sounds. I will detail the functionality of different waveshaping algorithms in section 3, focusing on the square waveshaping algorithm. I tested all waveshapers with monophonic and polyphonic sounds, and found that they work well with monophonic sounds but that improvements are needed for polyphonic sounds. I present an approach on waveshaping polyphonic sounds via filterbanks in section 5. I tested the output of the square waveshaper against a reference sound from a traditional synthesizer by copying and pitch shifting the output with an external software, with compelling results[1]. I researched different pitch shifting algorithms, with focus on the Rollers algorithm by Juillerat et al. [4] and the Ocean algorithm by Juillerat and Hirsbrunner [3], as these publications claime to present pitch shifting algorithms that provide appealing results within a latency of 12ms or less. I implemented the Rollers algorithm, but could not reproduce a latency as low as presented in the paper while retaining appealing results. The Rollers algorithm requires a filterbank, which also defines its latency. I present different approaches to designing filterbanks in section 4, and explain how I deal with the larger latency of more accurate filter banks in order to keep the pitch-shifting usable for real-time performances.

3 Waveshaping

Waveshaping is a fairly old technique. The basic idea is to take a sample x and apply to it a nonlinear function f. This is often used in distortion, where a simple distortion algorithm would be $y = \frac{1}{|atan(a+b)|} * atan(a * (x-b))$[2]. One of the major differences between our goal and adding spectral content via distortion is that distortion creates a very harsh sound, whereas we aim for a smooth synthesizer sound.

A typical guitar waveform can be seen in figure 1b. This waveform shows a single tone being played. The tone is a c# staccato tone. Figure 1a shows the pluck of the note, as well as the first few resonances. It is interesting to see that, in contrary to intuition, the pluck is indeed less loud than the resonances that follow. The DFT in 1c shows what can also be guessed from the waveform in figure 1a: the guitar tone consists of the base frequency f0 and several overtones.

A typical synthesizer wave, such as a square, saw or triangle wave, has much more spectral content than the guitar sample above. When transforming the guitar sound to such a synthesizer sound, we want to add that spectral content. However, we do not need to create a perfect square or saw wave. In fact, we don't want to do that at all. Synthesizer sounds from analog synths of companies such as Moog or Waldorf remain popular because the imperfection of the generated waves are musically more interesting than their perfect digital counterparts. The goal is to add the spectral content of a square and saw wave while keeping some

[1] This was presented in the project overview presentation on tuesday, April 30.

[2] This is, in fact, the distortion algorithm I used in 420A.

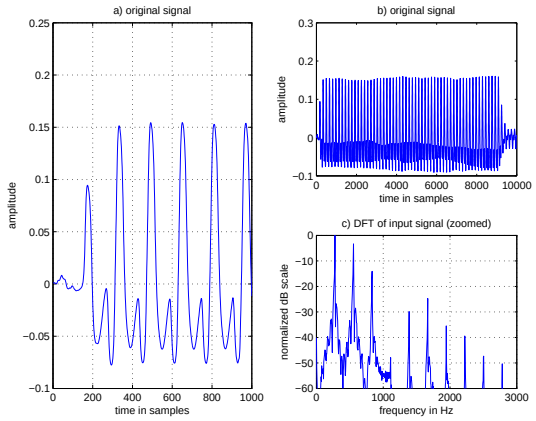

Figure 1: Waveform of typical guitar sample. Sample rate for this and all following plots is 44.1 kHz.

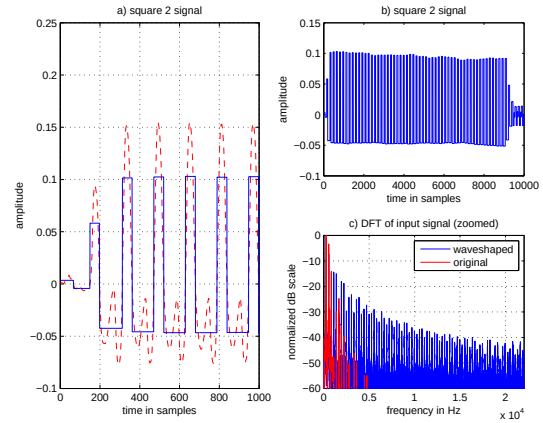

Figure 2: Waveform of square waveshaping algorithm 1.

of the content of the original audio signal. Thus, we don't want to extract too much information of the audio signal - if we converted the signal to MIDI, we would loose all the analog characteristics.

3.1 Square Waveshaping

3.1.1 A First Square Algorithm

The obvious solution to creating a square wave from an input would be to find the base frequency f0 and create a square wave. This, however, would mean that we track the pitch - something that we don't want to do. A simpler technique is to look at where the graph of the samples crosses the x-axis, and draw rectangles between those zero crossings. The height of the rectangle would be chosen so that the area inside the rectangle is equal to the area of the original signal graph - in other words, so that the power of the square wave matches the original signal. Listing 1 shows the Matlab code for that. The resulting signal is displayed in figure 2 a, b and c. This algorithm works fine when the wave is similar to the wave in figure 1a: the zero crossings are only created by the fundamental sinusoid f0, and the overtones do not add other zero crossings. Figure 3 shows a sample of an E7 guitar chord where this is not the case. Here, the result sounds like the original guitar sound mixed with distortion noise.

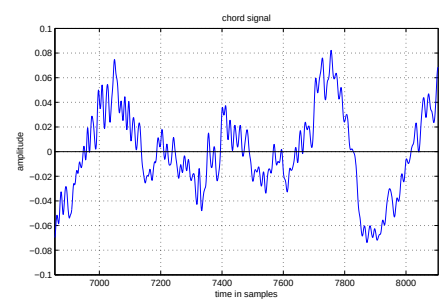

Figure 3: Overtones create additional zero crossings

The presented algorithm sounds fine for monophonic signals. However, its minimum latency depends on the largest zero crossing one will allow, as the amplitude of the square wave depends on the area between two zero crossings. The frequency of the lowest guitar string, an E2, is 82.4Hz, so one period is about 12ms. As we only need one half of the period, a 10ms delay should be enough - remember that our halves are not equal, as seen in figure 2a.

Listing 1 Square Waveshaping

```
function y = square_waveshaping(x)
    y = zeros(size(x));
    N = length(x);
    last_zero = 1;
    power = 0;
    delta_samples = 0;
    for k = 2:N
    % look for zero crossing
        if sign(x(k)) ~= sign(x(k-1)) && delta_samples > 0
      % calculate amplitude
            amplitude = power / delta_samples;
      % draw square curve
            for l = last_zero:k-1
                y(l) = amplitude;
            end
      % reset helper variables
            last_zero = k;
            power = 0;
            delta_samples = 0;
        else
            power = power + x(k);
            delta_samples = delta_samples + 1;
        end
    end
end
```

3.1.2 A Square Algorithm Without Delay

I tried another waveshaping method that needs no delay at all, shown in listing 2. The idea behind this algorithm is that it will look for a large sample x_1 and keep it until a larger sample x_2 arrives. While keeping the value x_1 in a variable m_0, the next samples x_i are compared against a version of x_1 that keeps getting smaller over time: m. Another way to look at this is that there is a forgetting factor $lambda$ that influences the comparison algorithm. The resulting waveform is illustrated in figure 4.

3.1.3 Comparison to Analog Square Waves

While the first square wave algorithm creates a more perfect square wave than the second one, the second one is closer to analog square waves in one regard: It doesn't have extremely steep curves.

In analog circuits, a perfect jump from -1 to 1 is impossible. Instead, a continuous movement from -1 to 1 will occur in an exponential curve (cf. figure 5). As a result, the square wave doesn't sound that harsh in high frequencies. As this affects high frequencies, the first idea is to simply lowpass filter the curve. The result of a Kaiser lowpass filter with passband frequency 6630Hz (0.3 π) and stopband frequency 7735Hz (0.35 π), 1db passband ripple and 60db stopband attenuation is shown in figure 6. The results are obviously not exactly what we intended. We could tweak the lowpass filter parameters, but I instead propose to implement the step method shown in figure 5 in our algorithm of listing 1. The result is shown in listing 3 and figure 7. This square wave sounds much nicer. By changing the parameter $lambda$ in listing 3, one controls the amount of high frequencies in the square wave.

Listing 2 A square-waveshaping algorithm with no delay

```
function y = square_waveshaping(x)
    y = zeros(size(x));
    N = length(x);
    m = 0;
    m0 = 0;
    lambda = 0.98;
    for k=1:N
       if abs(x(k)) > abs(m*lambda)
           m = x(k);
           m0 = m;
       else
           m = m*lambda;
       end
       y(k) = m0;
    end

    if(max(y) > 1)
         y = y / max(y);
    end
end
```

3.2 Triangular Waves

Triangular waves are similar in consideration to square waves. However, no zero-delay version is possible here since we need to know the slope before we can output a triangular wave. For this, we again need about one half of a period of our lowest frequency (one period = 12ms), and since the halves of the triangle again do not need to be equally sized, a 10ms delay should provide a large enough buffer for any size of "half" periods. As the code is very straight forward, no listings will be included here. Because triangular waves contain no unit steps, the discussion in subsection 3.1.3 does not apply.

3.3 Saw Waves

Saw waves need a buffer of the size of a whole wave period, since the slope is constant over the whole period. I have not implemented a saw wave waveshaper, although it should not be too hard, as it is very similar to the implementation of the first square wave algorithm in listing 1, only with linear interpolation over the length of a whole wave period.

3.4 8bit filter

The 8bit filter is a popular waveshaper due to its very simple algorithm. Every sample x is rounded to a small number of possible output values y. The name 8bit implies that the number of output values are limited, although 8bit filters do not necessarily limit themselves to 2^8 values, but often use smaller ranges. The distribution of these values between -1 and 1 is arbitrary. I implemented a simple 8bit filter with linear distribution of 2^4 possible values, to compare the result against the wave shapers I discussed before. The output is illustrated in figure 8. Acoustically, one could describe the 8bit sound as a little "thinner" and "more mean" than the square wave sound of section 3.1.1, which could be described as "thicker" and "more aggressive".

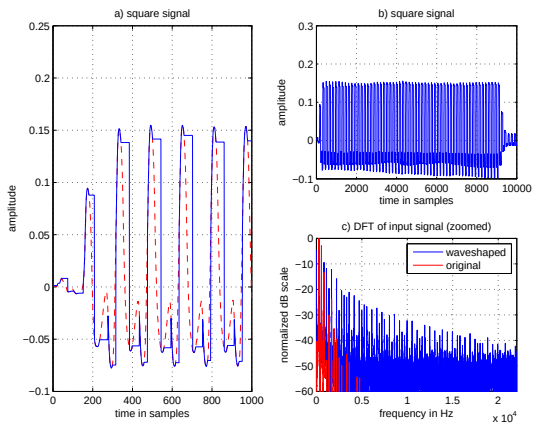

Figure 4: Waveform of square waveshaping algorithm 2.

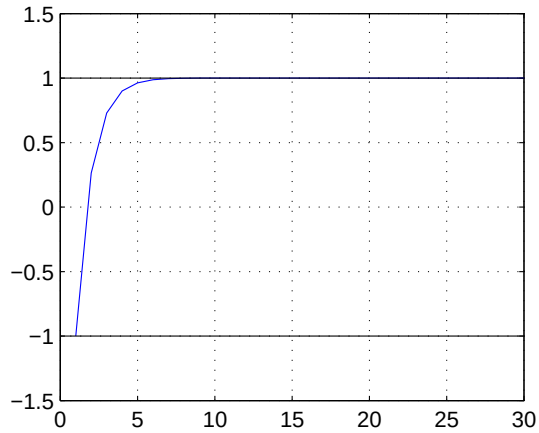

Figure 5: A step from -1 to 1 in an analog circuit.

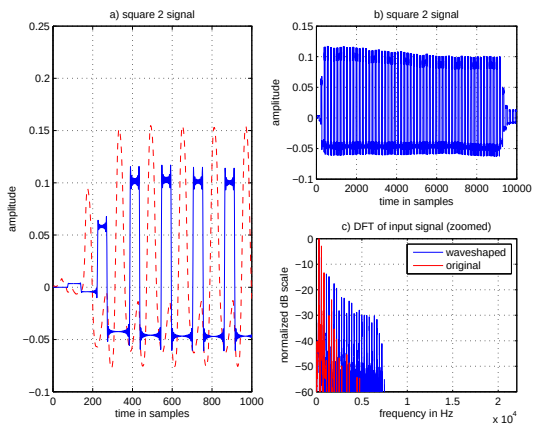

Figure 6: Effects of a lowpass filter applied on the output shown in figure 2

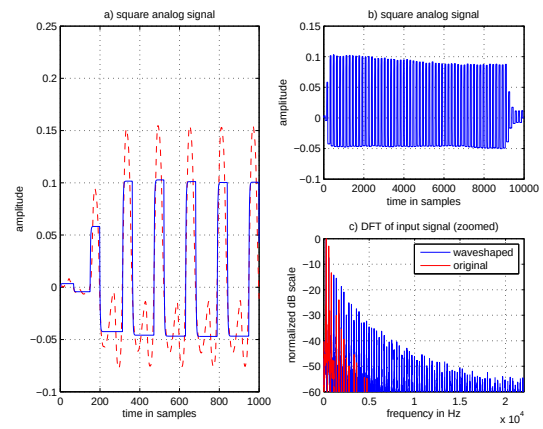

Figure 7: Output of the analog circuit algorithm in listing 3

4 Real-time Pitch Shifting

Real-time pitch shifting algorithms have been explored for a long time. Most pitch shifting algorithms are based on the phase vocoder [2, 9] and on synchronous overlap-add (SOLA) [6, 5, 1]. Two recent papers focus specifically on low latency pitch shifting algorithms. One introduces a time domain based pitch shifting algorithm called "Rollers" by Juillerat, Schubiger-Banz and Arisona [4] that achieves latencies less than 10ms by sacrificing on audio quality. The other one by Juillerat and Hirsbrunner, called "Ocean", uses a STFT based approach with very small window sizes for latencies of 12ms [3].

4.1 The Ocean Algorithm

The STFT based algorithm explained in [3] is computationally less expensive than the Rollers algorithm introduced in section 4.2. It uses a relatively rough approximation for pitch shifting, but has a nice solution for keeping the group delay constant across groups of frequencies, thus eliminating any phasing effects that other

Listing 3 Square waveshaping with the analog step illustrated in figure 5

```
function y = square_waveshaping_analog(x)
    y = zeros(size(x));
    N = length(x);
    last_zero = 1;
    power = 0;
    delta_samples = 0;
    lambda = 0.5; % analog circuit step speed
    for k = 2:N
        if sign(x(k)) ~= sign(x(k-1)) && delta_samples > 0
            amplitude = power / delta_samples;

            if last_zero == 1, start = 0;
            else start = y(last_zero-1); end
            stepsize = amplitude - start;

            timespan = last_zero:k-1;
            y(timespan) = amplitude - stepsize*exp( lambda*(last_zero - timespan) );

            last_zero = k;
            power = 0;
            delta_samples = 0;
        else
            power = power + x(k);
            delta_samples = delta_samples + 1;
        end
    end
end
```

STFT based algorithms suffer from. Audio examples are available on pitchtech.ch/Confs/ICALIP2010/index.html.

4.2 The Rollers Algorithm

Shifting a pitch can be seen as transposing a piece on a piano. To play a piece one tone higher, one just shifts his or her hands two keys to the right. On the music score, one would add two semitones to every note. In the frequency domain, notes are distributed on a log scale along frequencies (actually, $pitch\ log_2(frequency)$) , so shifting the pitch results in multiplying/scaling frequencies. The Rollers algorithm, however, **shifts** frequencies. This results in a distorted pitch shifted signal. To minimize the negative effect of the frequency shifting, the input signal is split into several bands which are individually shifted. Juillerat et al. propose about 200 filters, equally spaced on a logarithmic scale. The results of the algorithm, together with the results of other pitch shifting algorithms, can be heard on www.pitchtech.ch/Confs/ICALIP2008/Rollers.html. Because we are distorting the input audio signal with our waveshaping algorithms anyway, the artifacts of this algorithm are neglectable to a certain extend. Because of the large sice of the fiter bank, the Rollers algorithm is computationally more expensive that the Ocean algorithm. As we will see in section 5 however, the bandpass filter bank of the Rollers algorithm can be reused for waveshaping polyphonic audio signals. Hence, I chose the Rollers algorithm for my guitar synthesizer project.

4.3 Frequency Shifting Using Hilbert Transforms

Frequency shifting can be implemented in the time domain using a Hilbert transform. The basic idea of this technique is to do a frequency modulation (also known as ring modulation) which creates two shifted sidebands. One then only picks one of those shifted sidebands to get the resulting frequency shifted signal.

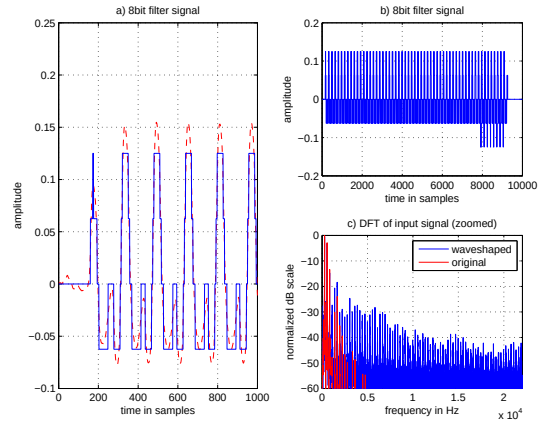

Figure 8: The "8bit filter" waveshaper

This is commonly referred to as single sideband modulation (SSB), and is explained in detail in [12]. The upper sideband can be retrieved via eq 1, where $\hat{x}$ denotes the Hilbert transform of x.

$$x_{SSB}(t) = x(t)\, cos(t\, 2\pi \frac{\delta f}{fs}) - \hat{x}(t)\, sin(t\, 2\pi \frac{\delta f}{fs}) \tag{1}$$

4.3.1 Approximating the Hilbert Transform

The Hilbert transform can be approximated by a linear-phase filter. When using a linear-phase filter in eq 1, one has to make sure that the signal x is delayed by the group delay of the filtered signal $\hat{x}$. The quality of the approximation depends on the length of the filter. I have found a filter length of N=100 and a passband range of 0.01 to provide excellent results using a least squares filter design (c.f. figure 9). A Hilbert filter of length N=50 also yields very good results, and can be more practical if performance is an issue - the Hilbert filter will have to be applied to every single band in our bandpass filter bank.

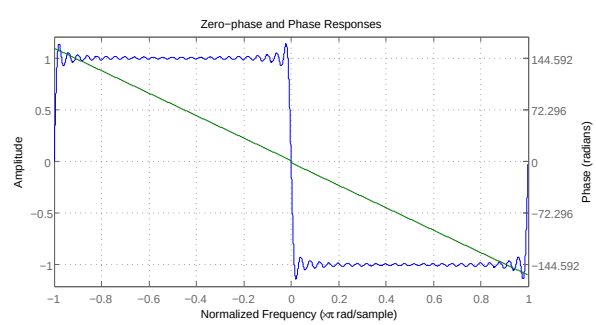

Figure 9: Hilbert allpass filter with N=100, transition bandwidth 0.01 and least squares filter design.

4.3.2 Audio Effects using the Hilbert Transform

When not delaying the signal x in eq. 1 to match the group delay of the Hilbert filter, high frequencies are attenuated. This results in a more aggressive and bassy sound. At the same time, the sound is also cleaner, because less frequencies are going to be shaped by our waveshaper. Changing the group delay compensation for the signal x can thus be a nice parameter for the audio effect.

4.4 Generating a Filter Bank

The Rollers pitch shifting algorithm requires a filter bank. The filter bank should be spaced in logarithmic intervals, with narrow filters at low frequencies and incrementally wider filters at high frequencies. The complexity of the filter bank is thus constrained by the design of the lowest bandpass filter of the bank. In order to make sure that the combined output of the fitlered bands has the same amplitude in all frequencies as the original signal, the transition band needs to overlap with the transition band of the next filter. This simple design rule works because transition bands have a nearly linear slope, as explained in [11]. To minimize filter complexity, the transition bands should go from the center frequency of the bandpass filter i to the center frequencies of the bandpass filters $i-1$ and $i+1$, so that there is nearly no passband region. I have tried both IIR and FIR filterbanks and have found, in contrast to the Juillerat et al. [4], that FIR filter banks sound much better.

4.4.1 IIR Filter Bank

IIR filters need much less CPU power when filtering the audio signal than FIR filters. Therefore it would be desirable to create an IIR filter bank, as suggested by Juillerat et al. Figure 10 shows the magnitude and group delay response of an IIR filter bank with 12 filters per octave, the first band-pass filter being centered on 82.41Hz[3]. One can clearly see that the bands are well separated. Figure 11 shows the group delay of the same filter bank. It is easily seen that the group delay has a peak at the center frequency of every bandpass. Because it also has an extremely sharp roll off, the group delay for the frequencies within each bandpass filter will vary greatly. This results in a sound with weird phasing issues that generally doesn't sound very well. The code for generating an IIR filter bank is the same as for FIR filter banks, and can be seen in listing 4. I chose the Butterworth design method for designing the band-pass filters. I also tried an elliptic design, but that resulted in peak group delays of 500ms for the same parameters, making it a much worse result than the Butterworth design.

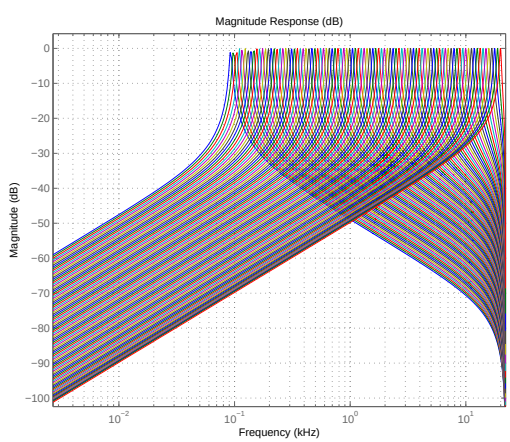

Figure 10: Magnitude response of IIR filter bank

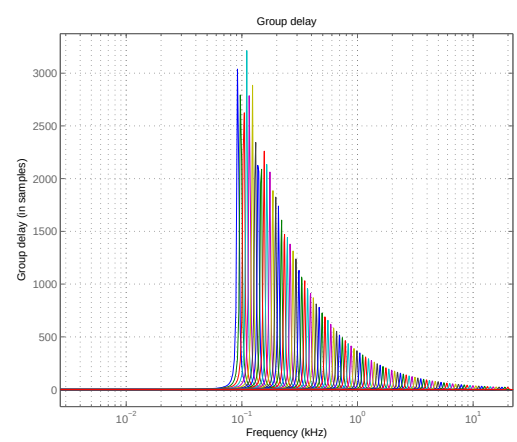

Figure 11: Group delay of IIR filter bank

[3] 82.41Hz is the frequency of the lower E string of a guitar standard tuning.

4.4.2 FIR Filter Bank

Using linear-phase filters, I could solve the phase distortion I encountered with the IIR filter bank. However, the filtering of an audio sample is a lot more expensive via FIR filters. This is an important drawback since with the above parameters and a sampling rate of $fs = 44.1\,kHz$, we have 94 bandpass filters. Figure 12 and 13 show the magnitude response and group delay of the filter bank. The average group delay of all bandpass filters is 22ms. This means that every sample requires about 184.000 multiplications and/or additions, or an 8Ghz clocked CPU.

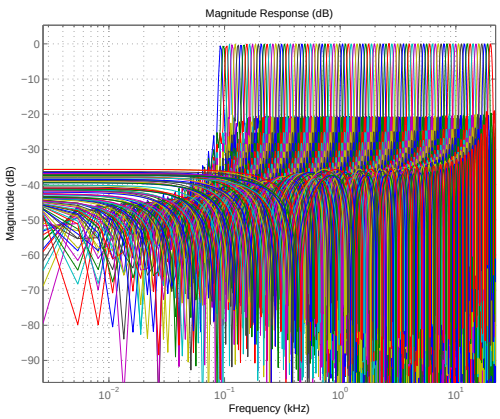

Figure 12: Magnitude response of IIR filter bank

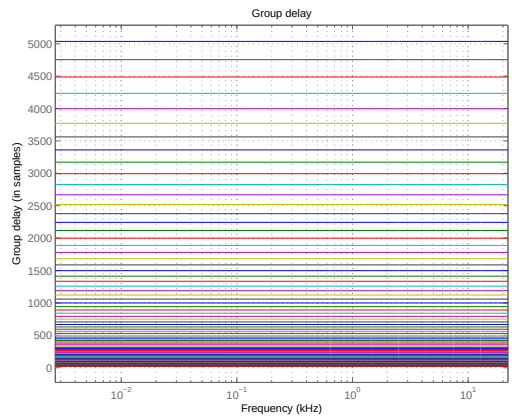

Figure 13: Group delay of IIR filter bank

While the FIR filter bank is computationally very expensive, it produces very good results. Looking at the magnitude in figure 12, this might seem a little surprising. To keep the order of the filters low, I use a Kaiser window filter design with a stopband attenuation of only 10dB. This produces an acceptable attenuation at higher frequencies, as can be seen in figure 14. Figure 15 shows an equiripple design with -30dB stopband attenuation. This filter has a much higher order than the kaiser window. The code for designing the filter bank is shown in listing 4.

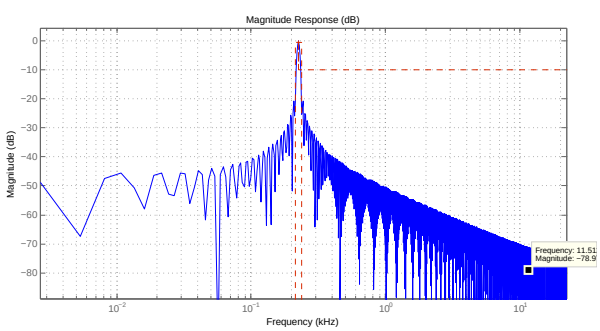

Figure 14: One Kaiser bandpass filter with -10dB stopband attenuation.

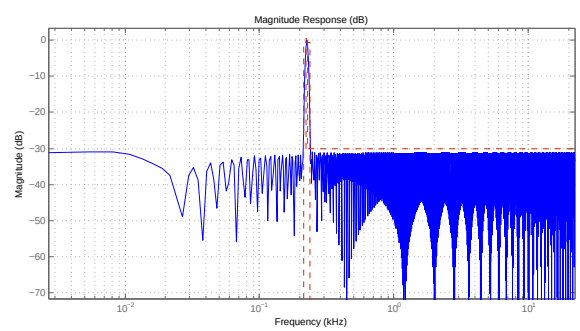

Figure 15: One bandpass filter with equiripple design and -30dB stopband attenuation.

Listing 4 Generation of an FIR filter bank in Matlab. To create an IIR filter bank, one only needs to change the parameter $'kaiserwin'$ in the last line to $'butter'$ for Butterworth filters.

```
function [Hd, nBands, centerFrequencies] = design_filterbank_fir(fBase, fs,
    BandsPerOctave, filterDistribution)

  % Derivation for nBands:
  %
  % fBase * 2.^(i/12) < fs/2
  % log_2(fBase) + i/12 < log2(fs/2)
  % i < 12*log2(fs/2) - log2(fBase)*12

  if strcmpi(filterDistribution, 'lin')
    nBands = floor((fs/2)/fBase);
    centerFrequencies = fBase*(1:nBands);
  else
    nBands = floor(BandsPerOctave*log2(fs/2) - log2(fBase)*BandsPerOctave);
    centerFrequencies = fBase * 2.^( (1:nBands) /BandsPerOctave);
  end

  normLFreq = centerFrequencies(1:end-2) / (fs/2); % center frequencies of the
      previous (left) band
  normCFreq = centerFrequencies(2:end-1) / (fs/2); % center frequencies of the
      current band
  normRFreq = centerFrequencies(3:end) / (fs/2); % center frequencies of the next (
      right) band

  centerFrequencies = centerFrequencies(2:nBands-1);
  nBands = nBands - 2;

  Hd(nBands) = fdesign.bandpass;
  parfor i=1:nBands
    fs1 = normLFreq(i);
    fs2 = normRFreq(i);
    df1 = normCFreq(i) - fs1;
    df2 = fs2 - normCFreq(i) ;
    fp1 = normCFreq(i)-df1/100;
    fp2 = normCFreq(i)+df2/100;
    d = fdesign.bandpass(fs1, fp1, fp2, fs2, 10, 1, 10);
    Hd(i) = design(d,'kaiserwin');
  end
end
```

4.4.3 A Note on Matlabs fdesign.octave Method

Matlab has an integrated filter bank generation method for IIR filters called $fdesign.octave()$. I found that the filters that come out of this filter bank have a considerably higher group delay than the IIR filters generated by listing 4. Furthermore, Matlab does not allow arbitrary center frequencies, but forces to choose from certain center frequencies. $fdesign.octave()$ does not support creating FIR filter banks.

4.5 Compensating the Group Delay

As we can see in figures 11 and 13, the group delay of the lowest filters can be as high as 3000 samples (68ms) for the IIR or 5000 samples (113ms) for the FIR filter. The average group delay, however, is much shorter, in case of the FIR filter it is 22ms. Preis suggests in [10] that the group delay at low frequencies is not as easily perceived as at high frequencies, which coincides with our results. Even though Preis talks

about very small ranges of 3ms for low frequencies, we found that just compensating group delays up to 15ms creates a very nice sound.

5 Polyphonic Audio Signals

All pitch shifting algorithms that were discussed are suitable and intended for polyphonic audio. Hence, we only need to look at the waveshaping algorithms for polyphonic audio. Surprisingly, the waveshaping algorithms work with polyphonic audio. More precisely, the pitch of the input signal is completely preserved. However, the more polyphonic a signal becomes, the more it will sound like the original guitar sound plus some added distortion. This is because a complex waveform of a polyphonic signal will likely have many zero crossings. With an increasing number of zero crossings per period, a waveshape algorithm for squaring sounds as discussed in section 3.1.1 will increasingly just replicate the original sine wave.

5.1 Separate Waveshaping of Each Band

By reusing the filter bank of the pitch-shifter in 4.2, we can waveshape every single note in a chord separately. This increases computation cost, since we are going to apply the wave-shaper to 94 bands instead of just one. Fortunately, the algorithms presented in 3 are computationally very cheap. We will later see that most of the computation is indeed spent on applying the filter bank to the input signal.

5.2 Effects of Filter Bank on Waveshaping Monophonic Audio

As discussed in section 5, waveshaping bands separately and then merging the waveshaped bands to one output makes our system work for polyphonic signals. However, the result of this process sounds different than simply waveshaping the whole signal. One could describe the resulting effect as smooth or muddy and less clear. To minimize this effect, one can group the bandpass filters by octaves. Listing 5 shows the algorithm. The resulting sound still doesn't sound like a waveshaper on the original signal, but sounds considerably less smooth, and clearer and more aggressive than a waveshaper that filters every band separately.

Listing 5 Merging bands

```
nSemitones = length(config.pitchshift);
% size of bpX: length(input_x) * numChannels(input_x) * numBandpassFilters *
    nSemitones;
bpY = zeros([size(input_x, 1), size(input_x, 2), config.BandsPerOctave, nSemitones]);
for i=1:filterbank.nBands
  k = mod(i-1, config.BandsPerOctave)+1;
  bpY(:,:,k,:) = bpY(:,:,k,:) + bpX(:,:,i,:);
end
```

6 Simulating Synthesizer Features

Right now we have a waveshaper and a pitch shifter. We can use these tools to create a fake additive synthesizer by creating several copies of the input signal, shifting them to different pitches, waveshaping them separately and then summing them together. The result will sound like an additive synthesizer. We could then go on and include features like chorus, delay, reverb, ring modulation, etc., but they are not

necessary since stomp-boxes exist for these effects. The combination of pitch-shifter and waveshaper can be seen as an equivalent to an oscillator.

7 Putting It All Together

Now that we know the single components of our algorithm, let's look at how all parts come together. Figure 16 shows the architecture of the whole filter. In short, the input signal is filtered by a filter bank, which is of size 94 in our example. The outputs of the filters are shifted and then grouped by semitones, resulting in 12 signals that are individually waveshaped. The sum of the 12 waveshaped signals forms the output signal. The frequency shifts and waveshapers are grouped in a gray box. This gray box is analogous to one oscillator on an additive synthesizer, as described in section 6. By running multiple copies of this box, we can generate multiple waves of different pitch with our guitar input signal.

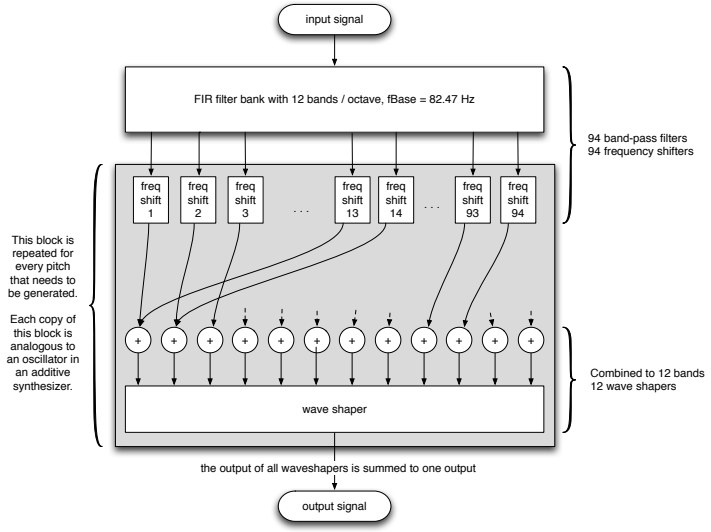

Figure 16: Architecture with 12 filters per octave, a filter bank base frequency of 82.47 Hz and a sampling rate of 44.1kHz, resulting in 94 filters.

8 Complexity and Latency of the Algorithm

We will look at the complexity and latency of each part of the algorithm step by step. To provide a feeling for the computational complexity, runtimes for processing a mono signal of $7.5s$ length at $44.1kHz$ will be included. The exact runtimes will of course differ on every system, but their relative proportions will remain the same. Therefore, the runtimes are meant to only illustrate which part of the algorithms are most computationally expensive. The runtimes were measured on a single-threaded system.

8.1 FIR Filter Bank

The FIR filter bank is the dominating part in this algorithm. The group delay of the filter bank depends on the size and quality of the filter bank. We found in section 4.5 that not every band has to be equally compensated in terms of group delay, and that a total compensation of 15ms is sufficient. Thus, we will summarize that the filter bank only introduces a group delay of 15ms. The computational complexity of the filter bank however poses a problem. The FIR filter bank presented in section 4.4.2 takes about $146s$ to

process $7.5s$ of audio using the Matlab script. This is somewhat to be expected, since the average group delay of the 94 FIR filters is 22ms, resulting in about 184.000 computations per sample (as already described in section 4.4.2). Because of the low amount of memory needed to be transferred, this can be calculated really quickly on a GPU. The narrowest band-pass filter would need 20kB of local memory, which is an amount that would for example fit into the maximum 48kB of local memory[4] in CUDA 1.2 architectures (c.f. [8]). Of course, it would be beneficial if this filter bank could be implemented with less computations.

8.2 IIR Filter Bank

The IIR filter bank presented in section 4.4.1 unfortunately didn't yield any results that sounded as good as the ones from the FIR filter bank. In general, the group delay of the IIR filter bank is shorter, with a maximum delay of $73ms$ and an average group delay of $16ms$. Also, the computational cost is a lot lower. The $7.5s$ long input signal is processed in Matlab within $1.30s$. It would therefore be highly preferable to get good-sounding results from the IIR filter bank to reduce computation time.

8.3 Frequency Shifting

The complexity of the frequency shift depends on the number of pitches, the number of bands in the filter bank and the order $N_{Hilbert}$ of the Hilbert linear-phase filter. Setting $N_{Hilbert} = 80$ the time for computing the Hilbert transform of all 94 bands is about $6.9s$. Applying eq. 1 three times for three pitch-shifted versions (in addition to the original key) takes another $2.9s$. The group delay of the Hilbert filter is $N_{Hilbert}/2$, which for $N_{Hilbert} = 80$ results in a group delay of less than $1ms$.

8.4 Wave Shaping

Latency of different wave shaping algorithms was already discussed in section 3. In short, we can have a latency between 1 sample and 10ms. The complexity of the waveshaper depends on the implementation but is generally okay, even when running it $n_{pitches} * 12$ times (where $n_{pitches}$ describes the number of copies of the gray system in figure 16). For $n_{pitches} = 4$, the wave shaping algorithm of section 3.1.2 took only 0.56 seconds. The wave shaping algorithm of section 3.1.3 was more expensive because it uses an exponential function, which is rather expensive in Matlab. With the same configuration, it took 2.75 seconds to compute. Note that these times can be dramatically improved if one deviced to sum the bands before waveshaping, therefore assuming that the input signal is monophonic (the different pitches will still be wave-shaped separately).

8.5 Latency of the combined system

Using the square waveshaper in section 3.1.3 and any filter bank with group delay compensation described in section 4.5,the total delay of this system is

$$delay_{waveshaper} + compensated\_delay_{filterbank} + delay_{Hilbert} = 10ms + 15ms + 1ms = 26ms$$

9 Summary and Future Work

The method presented in this work provides a nice-sounding way to convert a guitar to an additive synthesizer. Different wave-shapers were presented to simulate different oscillators, and their computational complexity

[4] Local memory on CUDA is L1 cache, the fastest cache in the architecture.

is very low. The wave-shapers presented are by far not the only ones possible, and this work is only intended to provide the groundwork to explore more possibilities in making a guitar sound like a synthesizer. The pitch-shifting method that was used in this work sounds great and has a theoretically low latency of less than 30ms. The computational complexity of the algorithm however needs to be reduced for implementation in commercial products. Further research needs to be done to replace the FIR filter bank by an IIR filter bank with equally good sounding results. It would be interesting to see how Juillerat et al. defined their IIR filter bank, but they did not publish any source code for their implementation. Also, the time needed for the computation of the Hilbert transform in the frequency shifting method seems pretty long. A good starting point for further research would be the work by Yang, Liu and Lim [13]. In their work, they present a filter-bank of linear-phase bandpass filters with very sharp transition bands that have a very low computational cost. Lim and Yo also applied this technique to designing very good Hilbert transformers [7]. The summary of their results looks very promising and could make the audio effect presented in this work very efficient.

References

[1] D. Dorran, E. Coyle, and R. Lawlor. Audio time-scale modification using a hybrid time-frequency domain approach, 2005.

[2] J. L. Flanagan and R. M. Golden. Phase Vocoder. *Bell System Technical Journal*, 38(5):1493–1509, 1966.

[3] N. Juillerat and B. Hirsbrunner. Low latency audio pitch shifting in the frequency domain, 2010.

[4] N. Juillerat, S. Schubiger-Banz, and S. M. Arisona. Low latency audio pitch shifting in the time domain, 2008.

[5] A. V. D. Knesebeck and P. Ziraksaz. High Quality Time-Domain Pitch Shifting Using PSOLA and Transient Preservation. In *Proc 129th Audio Eng Soc*, pages 1–10, 2010.

[6] J. Laroche. Autocorrelation method for high-quality time/pitch-scaling, 1993.

[7] Y. C. Lim and Y. J. Yu. Synthesis of very sharp Hilbert transformer using the frequency-response masking technique, 2005.

[8] C. Nvidia. NVIDIA CUDA C Programming Guide. *Changes*, (350):173, 2011.

[9] S. Park. Real-time Pitch (frequency) Shifting Techniques, 1991.

[10] D. Preis. Measures and perception of phase distortion in electroacoustical systems, 1980.

[11] R. W. Schafer, L. R. Rabiner, and O. Herrmann. FIR Digital Filter Banks for Speech Analysis, 1975.

[12] S. Wardle. A Hilbert-Transformer Frequency Shifter for Audio.

[13] R. Yang, B. Liu, and Y. C. Lim. A new structure of sharp transition FIR filters using frequency-response masking, 1988.

A Reproducing the Results in Matlab

The source code of all Matlab scripts and all input audio files for this work can be downloaded at ccrma.stanford.edu/∼thon
This appendix is aimed at giving an introduction to the structure of the Matlab project.

A.1 Structure of the Matlab Project

There are three files that are meant to be run directly. The other files are functions that are called by other files.

waveshaper_overview.m This file generates one plot for presentation purposes that combines the zero-delay square, the 8bit and the triangle filter in a single window (c.f. figure 17).

waveshaper_test.m This file generates multiple figures for testing the different waveshaping algorithms (c.f. figure 18).. When clicking on the left part of the figure, the waveshaped audio signal will be played back. This test suite is excellent for developing new waveshaping algorithms.

run_test_suite.m This is the test suite for running the whole filter architecture. We will discuss this file shortly.

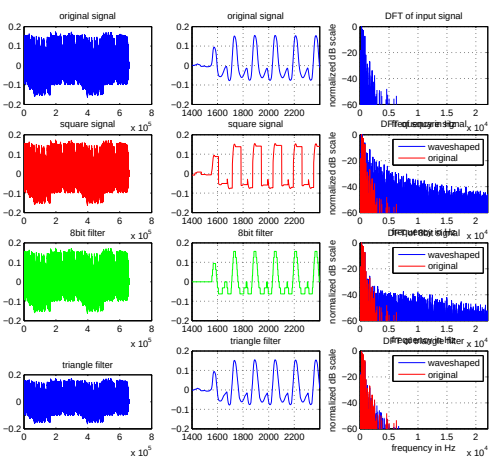

Figure 17: Plot from *waveshaper_overview.m*

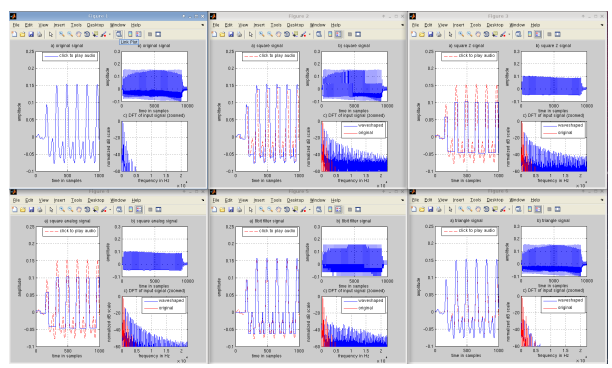

Figure 18: Plots created by *waveshaper_test.m*

Input .wav files for the test suites are saved inside the *input* folder. The output folders *waveshaper_overview* and *waveshaper_test* contain the output files of their corresponding Matlab files. The output of the *run_test_suite.m* script is saved in the folder named *output*.

Here's a list of all other Matlab files in the project:

apply_filterbank.m Applies a filterbank provided by an input argument to the input signal and returns the bandpassed output signals.

calcGroupDelay.m Calculates the group delay for every bandpass filter in Hd at the frequency defined in the vector centerFrequencies and given sampling rate fs.

design_filterbank_fir.m Designs an FIR filterbank.

design_filterbank_iir.m Designs an IIR filterbank.

design_filterbank_iir_octave.m Designs an IIR filterbank using Matlab's fdesign.octave command.

filter8bit.m An 8bit-filter waveshaper explained in section 3.4.

filterbank_waveshaping.m Applies a waveshaping algorithm to every band in bandpassed input array.

frequency_shift.m Frequency-shifts a single input signal x by one or multiple different semitones. Returns an array of all shifted copies.

guitar_test_suite.m This file runs the whole system algorithm. It contains benchmarks for each section, and writes the output resulting file in the *output* directory.

iir_lpf.m A simple IIR low-pass filter of order 4 with cutoff frequency 0.06π for preprocessing an input signal to get rid of noise.

plot_dft.m, plot_guitar.m Helper functions used by the *waveshaper_test.m* suite to display the time-domain plots and DFTs of the waveshaped signals.

shiftd.m, shiftu.m, shift/* Helper functions to shift multidimensional arrays in a single dimension. Used to delay signals when compensating group delay in *apply_filterbank.m* and *frequency_shift.m*. License and readme files can be found in the *shift* folder.

sola_pitch_shift.m This function takes a set of bandpassed input signals bpX and frequency shifts them separately.

square_waveshaping.m, square_waveshaping_2.m, square_waveshaping_analog.m Waveshaping algorithms of sections 3.1.1, 3.1.2 and 3.1.3 respectively.

triangle_waveshaping.m A waveshaping algorithm of section 3.2.

In addition to these files, an Ableton Live Project can be downloaded from the website. The project includes a demo of this effect being used with a kickdrum and a french-house effect (a compressor on the filtered guitar signal, sidechained to the kickdrum).

A.2 Running run_test_suite

run_test_suite contains parameters to configure many ways of the whole effect system. Each parameter is commented in detail. It will call *guitar_test_suite*, which will then run the actual algorithm. The output of the test suite will be saved to the output/ folder, and the filename will include date and time, name of the input audio file, and a couple of parameter settings so you know from the filename which settings produced this result.