Synthesis of Quasi-Bandlimited Analog Waveforms Using Frequency Modulation /Draft, version 3/

Peter Schoffhauzer
scoofy@inf.elte.hu

Abstract

This paper investigates the possibilities of approximating bandlimited analog waveforms using frequency modulation (FM synthesis), a technique often used in hardware and software synthesizers. The goal was to create a waveform with the same spectral content as a saw or a pulse wave, which have linearly spaced sine partials with a decay rate of 6dB/octave. A method is presented which creates a roughly good approximation of bandlimited saw and pulse waveforms. A C pseudo code implementation is also given.

1 Introduction

Classic FM synthesizers, like the Yamaha DX series allow a waveform to modulate the frequency of itself using a feedback loop. When the frequency of a sine wave is modulated, additional harmonics are created above the fundamental frequency. When the sine wave is modulated by itself, harmonics are created with a rolloff rate of 6dB/octave, similar to the rolloff of the harmonics of a saw wave. After a certain point, the harmonics roll off with a soft knee, similar in effect to applying a four pole filter to a saw wave, as seen in Figure 1.

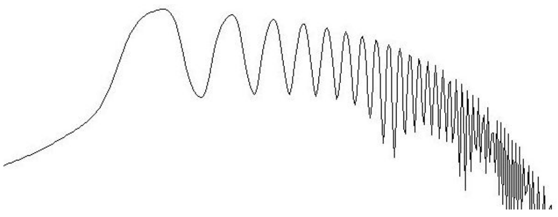

Figure 1. Spectrum of a sine wave modulated by itself

Note that a 6dB/Oct slope was applied to the figure, so a bandlimited saw wave has peaks of equal height. The bandwidth of the FM harmonics can be controlled by scaling the output before feeding back to the frequency. This should be higher for low frequencies, and lower for high frequencies to prevent aliasing.

2 Algorithm

The basic algorithm of the oscillator is the following:

$$x_n = \sin(\pi(\theta + \beta x_{n-1}))$$

where x_n is the output of the oscillator, θ is the phase accumulator ranging from -1 to 1, and β is the scaling function. The previous output sample modulates the phase of the oscillator. The Yamaha DX series use a similar method for frequency modulation. Using empirical methods, polynomial scaling functions were tested. For a given ω normalized frequency, the following equation was found adequate:

$$\beta = 38(0.5 - \omega)^6$$

This scaling function gave a good approximation of a bandlimited saw wave, with a full sound and only a slight amount of aliasing. However, some problems were encountered. One problem was a loud high frequency ringing at Nyquist. Another problem was aliasing that appeared for middle and low frequencies. The third problem was a loud resonant ringing at samplerate/4 for very low frequencies. These problems were attributed to the feedback path. To overcome these problems, the average of the output and the previous output of the oscillator were taken, giving a smoothing effect for the Nyquist ringing.

$$x_n = \frac{x_{n-1} + \sin(\pi(\theta + \beta x_{n-1}))}{2}$$

This eliminated very well all three problems mentioned before. However, the scaling function needed to be adjusted. The following function was found to be good:

$$\beta = 13(0.5 - \omega)^4$$

The spectrogram of a linear frequency sweep using this equation is shown in Figure 2. The range of the spectrogram is 0 to -90 dB.

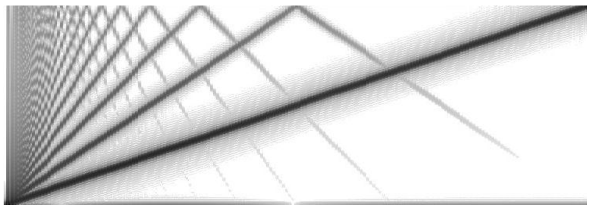

Figure 2. Spectrogram of a linear frequency sweep

For high frequencies, aliasing may be reduced somewhat on the expense of reducing brightness of the wave by using a different scaling function, such as:

$$\beta = 54(0.5 - \omega)^6$$

The spectrogram obtained using this scaling function is shown in Figure 3.

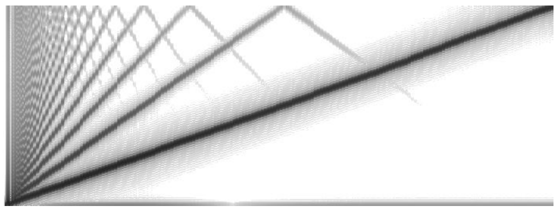

Figure 3. Spectrogram of a linear frequency sweep

The aliasing is reduced, but it is noticeable both on the spectrogram and by listening to the generated sounds that the brightness of higher frequency waves is also reduced.

3 Improving the algorithm

With this method, a bandlimited saw-like wave was created. However, there are some possible improvements. The algorithm described above gives a rough approximation, and the high harmonics are not as bright as in an ideal bandlimited wave, created using inverse DFT for example. There is a high frequency rolloff above around samplerate/10 compared to an ideal waveform. This is shown in Figure 4.

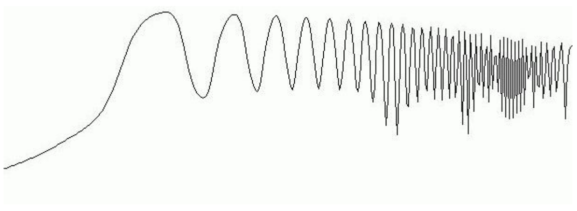

Figure 4. HF rolloff of the basic implementation

To improve this, the idea was to spectrally invert a one pole lowpass filter, to get a highshelf-like filter, which will compensate the rolloff. The inversion can quickly be done in TobyBear's FilterExplorer. The coefficients for a given +3dB frequency are:

$$x = e^{-2\pi\omega}$$
$$a0 = (1 - x)^{-1}$$
$$a1 = -xa0$$

The normalized frequency was set to around 0.0813. This gives coefficients $a0 = 2.5$ and $a1 = -1.5$, thus the transfer function of the compensation filter becomes

$$y_n = 2.5x_n - 1.5x_{n-1}$$

This filter was found nearly ideal to compensate the high frequency rolloff. The frequency spectrum of the improved saw wave is shown in Figure 5. The spectrum is nearly ideal, and remains balanced for most of the frequency range.

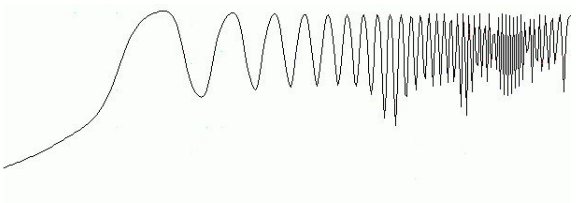

Figure 5. Frequency spectrum of the improved saw wave

Note that the frequency of the HF filter is fixed, which causes increased amplitude when the frequency of the oscillator is high. Scaling the output by

$$\gamma = 1 - 2\omega.$$

compensates this very well. The spectrum of the output after the compensation is very close to an ideal bandlimited waveform created using inverse DFT. It should be noted that the waveform is quite asymmetric, and the HF compensation filter increases the asymmetry even more.

4 DC offset

The algorithm was implemented in floating point mode. In both single and double precision, there is a considerable amount of DC offset. At low and middle frequencies, the DC offset is around -0.376, which converges to zero near Nyquist. Thus adding the following DC compensation seemed to reduce the DC offset:

$$DC = 0.376 - 0.752\omega$$

However, this does not eliminate it completely, because the amount of DC for different frequencies is very irregular. When the frequency is around samplerate/4, the offset may be as much as -20 dB. Although the DC offset after the compensation is around -30 to -40 dB for most frequencies, a one pole highpass filter with a -3dB frequency set to 10-30 Hz may be used for almost completely eliminating the DC offset.

Note that when synthesizing a pulse wave, the DC components cancel out each other, thus no additional adjustment is needed.

5 Results

The algorithm described above gives a fairly good sound with slight aliasing, suitable for using in realtime sound synthesis. The oscillator has good stability, and fairly good sound clarity and brightness down to frequencies as low as 0.01 Hz. However, there are some drawbacks. The waveform is quite asymmetric, which is emphasized more by the HF compensation filter. The asymmetry may reduce the dynamic range of the system, and it may change the perception of the waveform if the system is nonlinear. This is shown in Figure 6.

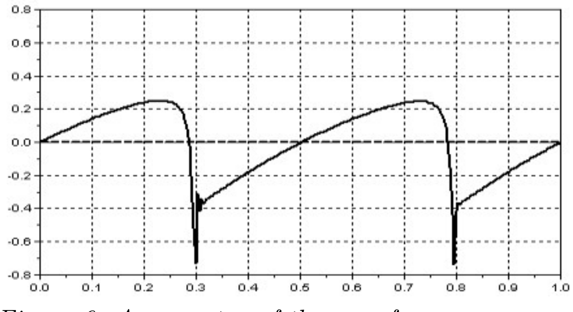

Figure 6. Asymmetry of the waveform

Attempts were made to make the waveform more symmetric and look more like a saw. One method was to apply a waveshaping function, which introduced some additional aliasing. Another method was to subtract a scaled sinewave with the frequency of the fundamental from the output of the oscillator, which does not create any aliasing. With both methods, the cost was quite heavy for such a small visual makeup. The difference between the adjusted waveforms and the original was very subtle. There was no notable change in sound or in the spectrum, the adjustments only made the curve straighter.

In blind A/B tests during comparison with an ideal bandlimited saw, the two waveform were found distinguishable. The FM saw was found to have a slightly 'warmer' tone. Upon closely observing the spectrum, it was found out that the rolloff rate is not exactly 6dB/octave, and the partials near the fundamental frequency have slightly higher amplitude. The ideal bandlimited saw sounded more flat, clean and raw.

The algorithm also produces a slight amount of aliasing. This is due to the fact that the rolloff of the harmonics is a smooth curve, not a sudden step, as would be in an ideally bandlimited waveform. Figure 7. shows a spectrogram of a linear frequency sweep.

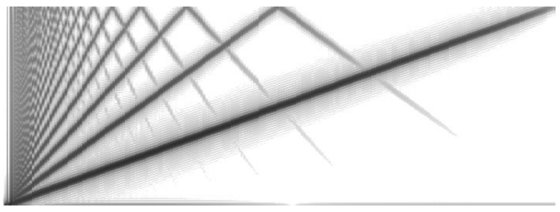

Figure 7. Spectrogram of linear frequency sweep

The scale of the spectrogram is 0 to -90dB. There is no audible aliasing below samplerate/4. This also means that a 2x oversampled oscillator would produce a nearly alias-free waveform.

6 Performance

The cost of calculating one sample is moderate. The following are needed:

1) Update the phase accumulator
2) Apply scaling function
3) Calculate sin() (can be approximated by table lookup or polynomial)
4) Do averaging
5) Apply HF boost
6) Normalize output (compensate HF boost)
7) Apply DC filter

Since 2) and 6) can be precalculated, they mean only one multiplication per sample. If more oscillators are running in parallel, it may be enough to apply the one pole filter(s) to the mixed output of the complete oscillator block, since the filters are linear. Thus the actual cost of the filter(s) may be lower.

An implementation using a 9th order polynomial approximation were found to take about 2x as much CPU on an AMD Sempron procesor compared to a linear interpolated wavetable oscillator with a table size of 2048. One advantage is that no table is needed when polynomial approximation is used, which may be useful on environments with limited memory. The other advantage is that the floating point operations can be parallelized very well on SIMD machines. Thus the actual cost of an oscillator may be very low on machines with SIMD instructions when more oscillators are running.

7 Other waveforms

Probably the best method for generating PWM pulse waveforms is to create two saw waves with a phase offset, and subtract them from each other. The phase offset gives the pulse width of the resulting waveform.

Other ways of approximating a bandlimited pulse wave with a pulse width of 50% were also found using the FM method. It was discovered that by squaring the output of the oscillator before feeding back, a square-like waveform is created with harmonics at $f, 3f, 5f$, etc. However, both the smoothing function and the HF compensation filter needed to be modified. The basic function became:

$$x_n = 0.45x_{n-1} + 0.55\sin(\pi(\theta - \beta x_{n-1}^2))$$

Note that the sign of the feedback was also changed. The HF filter was modified to:

$$y_n = 1.9x_n - 0.9x_{n-1}$$

Using this algorithm, the following waveform is obtained:

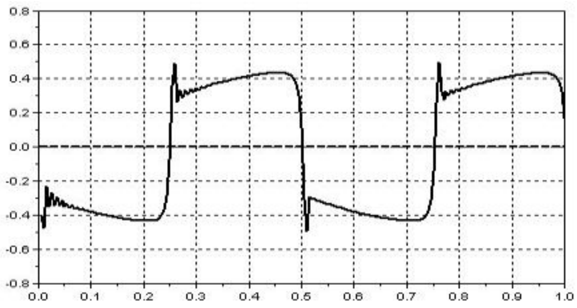

Figure 8. Pulse wave

The spectrum is close to ideal, but the brightness of the highs is somewhat reduced for higher frequencies. This could be compensated by modifying the scaling function, but then aliasing would be increased. However, the brightness and sound quality was found much better when obtaining a square by subtracting two saw waves, so this method was not investigated more. The squaring also introduces more aliasing, which is not present when creating the pulse by subtracting two saw waves with a phase difference. The logarithmic spectrum of the pulse obtained using this method is shown in Figure 9.

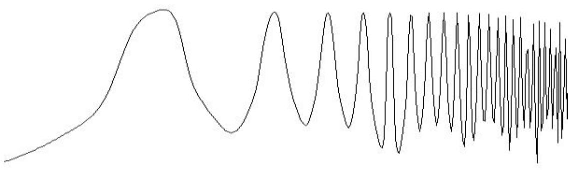

Figure 9. Spectrum of the pulse wave

8 Further possibilities

By multiplying the β scaling function by a value ranging from 0 to 1, it is possible to simulate a gradually opening filter, which creates a morphing from sine to saw or pulse. The effect is similar to a four pole filter, but somewhat different. Using this method, a filter can be simulated easily without actually using a filter. Unfortunately, resonance cannot be simulated, so this feature is of limited usability.

When the modulation index is very high, the aliasing increases. After a certain point, the whole spectrum is dominated by the aliasing, turning the oscillator into a (bit costy) white noise generator.

9 References

Smith, Steven W. (1997). *The Scientist and Engineer's Guide to Digital Signal Processing*

10 Appendix - C pseudo code

```
// variables and constants
float osc; // output of the saw oscillator
float osc2; // output of the saw oscillator 2
float phase; // phase accumulator
float w; // normalized frequency
float scaling;  // scaling amount
float DC; // DC compensation
float *output; // pointer to array of floats
float pw; // pulse width of the pulse, 0..1
float norm; // normalization amount

float const a0 = 2.5f; // precalculated coeffs
float const a1 = -1.5f; // for HF compensation
float in_hist; // delay for the HF filter

// calculate w and scaling
w = freq/samplerate; // normalized frequency
float n = 0.5f-w;
scaling = 13.0f * n*n*n*n; // calculate scaling
DC = 0.376f - w*0.752f; // calculate DC compensation
osc = 0.f; phase = 0.f; // reset oscillator and phase
norm = 1.0f - 2.0f*w; // calculate normalization

// process loop for creating a bandlimited saw wave
while(--sampleFrames >= 0)
{
    // increment accumulator
    phase += 2.0f*w; if (phase >= 1.0f) phase -= 2.0f;
    // calculate next sample
    osc = (osc + sin(2*pi*(phase + osc*scaling)))*0.5f;
    // compensate HF rolloff
    float out = a0*osc + a1*in_hist; in_hist = osc;
    out = out + DC; // compensate DC offset
    *output++ = out*norm; // store normalized result
}

// process loop for creating a bandlimited PWM pulse
while(--sampleFrames >= 0)
{
    // increment accumulator
    phase += 2.0f*w; if (phase >= 1.0f) phase -= 2.0f;
    // calculate saw1
    osc = (osc + sin(2*pi*(phase + osc*scaling)))*0.5f;
    // calculate saw2
    osc2 = (osc2 + sin(2*pi*(phase + osc2*scaling + pw)))
        *0.5f;
    float out = osc-osc2; // subtract two saw waves
    // compensate HF rolloff
    out = a0*out + a1*in_hist; in_hist = out;
    *output++ = out*norm; // store normalized result
}
```