Synthesising band limited waveforms using wavetables

Joe Wright – joe@nyrsound.com

17 August 2000

Introduction

When synthesising analogue type waveforms (sawtooth, square, triangle) in the digital domain special care must be taken to ensure the results are bandlimited. The simple method of producing a sawtooth wave is given by:

Analogue: $y_t = A[tf_0 - Int(tf_0)]$

Digital: $y_n = A\left[\frac{nf_0}{f_s} - Int\left(\frac{nf_0}{f_s}\right)\right]$

where *y* is the output, *A* is the amplitude, *t* is time, n is sample number in the discrete digital domain, f_0 is the frequency of the waveform, f_s is the sampling rate and Int(x) is the highest integer less than or equal to x.

The analogue equation produces desirable results because it is working in the continuous domain. However, the digital equation fails because it is sampling a non-bandlimited waveform. Sampling hardware would first feed an input signal through a low pass filter before the ADC. Because direct synthesis in the digital domain of the above formula does not do this, an aliased signal is produce. The aliasing will contaminate the whole spectrum and therefore cannot be filtered. Therefore, a technique is needed for synthesising bandlimited waveforms.

Wavetables

A wavetable is a sample (collection of individual samples) containing one period of a waveform. Synthesis using wavetables simply involves playing back the wavetable as follows:

$$y_n = w_m$$

where $$m = \frac{nlf_0}{fs} \bmod l$$

where *w* is the wavetable and *l* is the length of the wavetable in samples. Most of the time m will contain both integer and fractional parts. To deal with this a large oversampled wavetable can be used with access via Int(m), or a form of interpolation can be used.

Generating a bandlimited wavetable

A bandlimited wavetable contains harmonics whose frequencies are less than the Nyquist frequency (above which aliasing occurs). The frequencies of the harmonics in the wavetable are relative to the frequency at which the wavetable is played. The number of harmonics *h* allowable in a wavetable for a given frequency f_0 obeys the following:

$$h < \frac{f_s}{2f_0}$$

so the highest frequency a wavetable can be played at is given by:

$$f_0 < \frac{f_s}{2h}$$

Given this formula, a series of bandlimited wavetables with varying number of harmonics can be used to play notes across a range of pitches. The wavetables should be normalised to the richest waveform (highest *h*) so the harmonics are kept at constant amplitude.

For absolute coverage the series of wavetables would have h increasing by one each time. The problem with this is that it requires a huge amount of wavetables to cover a good pitch range. Given large oversampled wavetables this would demand a large amount of memory. A solution is to use 128 wavetables corresponding to the midi note range. Each wavetable should have the right number of harmonics for that midi pitch. It is recommended to use wavetables with 4096 samples and to play back using linear interpolation. This gives a good range of wavetables with good sound quality.

If sliding between pitches is required, the correct wavetable to use should be determined for each wave cycle. For this, use a reverse look-up table that specifies which of the 128 wavetables should be used for a given frequency (in fact, integer of frequency).

Generating a bandlimited sawtooth waveform

The sawtooth can be generated using its Fourier series:

$$\sin x + \frac{1}{2}\sin 2x + \frac{1}{3}\sin 3x + ... \qquad 0 \le x < \pi$$

For each wavetable, sum the series up to (1/h) sin hx.

Generating a bandlimited square waveform

This can be done in the same way as before using the following series:

$$\sin x + \frac{1}{3}\sin 3x + \frac{1}{5}\sin 5x...$$

However, a pulse wave (variable width of peak and trough) can be generated in real time by subtracting one sawtooth from another with a different phase.

This can be shown for the standard equal width square wave as follows:

Sawtooth 1 $= \sin x + \frac{1}{2}\sin 2x + \frac{1}{3}\sin 3x + ...$

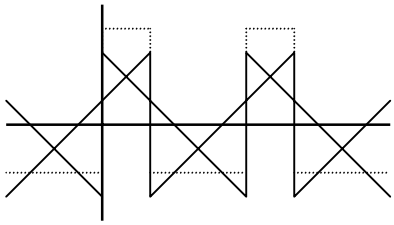
Sawtooth 2 $= \sin(x+\pi) + \frac{1}{2}\sin 2(x+\pi) + \frac{1}{3}\sin 3(x+\pi) + ...$

$= -\sin x + \frac{1}{2}\sin 2x - \frac{1}{3}\sin 3x + ...$

Therefore sawooth1 – sawtooth2 = $2\sin x + 2 \cdot \frac{1}{3}\sin 3x + ...$

which is proportional to the square wave.

The above process can be examined on a graph showing non-bandlimited waveforms.



The sawtooth waves must be properly normalised. Then, for a given phase difference between the first and second (inverted) sawtooth waves to be summed, an offset and scalar has to be applied to produce a properly sized pulse wave.

If 0 < phase < 1 (equivalent of 0 < phase < 2π), and the positive sawtooth is located at 0 then the offset and scalar can be calculated at the negative sawtooth crossover.

Peak = 1 + (2.phase – 1)
Trough = -1 + (2.phase – 1)

The correct peak should be 1 and the correct trough should be –1. By inspection we can see the offset should be (1 - 2.phase) and scalar is not needed.

Creating a slope variable triangle wave

To create a wave that can be adjusted from triangle to sawtooth, a parabola and inverted out of phase parabola can be summed in the same way as the pulse wave was generated.

A parabola wavetable is constructed with the following

$$\frac{\pi^2}{3} - 4\cos x + \cos 2x - \frac{4}{9}\cos 3x + ...(-1)^n \frac{4}{n^2}\cos nx...$$

Which is then centred around 0 and normalised.

e.g.

max=max abs value of wavetable
for each sample
    sample=sample / (max / 2) – 1
next

There is no offset but there is a scalar of:

$$\frac{1}{8(p-p^2)}$$

Gibbs effect

There is a problem with the waveforms generated by their Fourier series. At the transition points, the signal will overshoot. This is because a Fourier series should be infinite (whereby the length in time of the overshoot tends to 0) but the bandlimited case is finite.

To minimise this effect, reduce the amplitude of the higher partials. For example use:

$m = \cos^2((n-1)k)$ where:

$k = \left(\frac{\pi}{2}\right) / \, partials$, n = partial number, and partials = total number of partials

For example, the sawtooth series can be given by:

$$\sum_{1}^{partials} \frac{1}{n} \sin nx \cdot \cos^2((n-1)k)$$

Code example:

```
// An example of generating the sawtooth and parabola wavetables
// for storage to disk.
//
// SPEED=sampling rate, e.g. 44100.0f
// TUNING=pitch of concert A, e.g. 440.0f

////////////////////////////
// Wavetable reverse lookup
// Given a playback rate of the wavetable, what is wavetables index?
//
// rate = f.wavesize/fs e.g. 4096f/44100
// max partials = nyquist/f = wavesize/2rate  e.g. 2048/rate
//
// using max partials we could then do a lookup to find the wavetables index
// in a pre-calculated table
//
// however, we could skip max partials, and lookup a table based on a
// function of f (or rate)
//
// the first few midi notes (0 - 9) differ by < 1 so there are duplicates
// values of (int) f.
// therefore, to get an index to our table (that indexes the wavetables)
// we need 2f
//
// to get 2f from rate we multiply by the constant
// 2f = 2.fs/wavesize  e.g. 88200/4096
//
// our lookup table will have a length>25087 to cover the midi range
// we'll make it 32768 in length for easy processing


        int a,b,n;
        float* data;
        float* sinetable=new float[4096];
        float* datap;
        for(b=0;b<4096;b++)
                sinetable[b]=sin(TWOPI*(float)b/4096.0f);
```

```
int partials;
int partial;
int partialindex,reverseindex,lastnumpartials;
float max,m;
int* reverse;

// sawtooth

	data=new float[128*4096];
	reverse=new int[32768];

	reverseindex=0;
	partialindex=0;
	lastnumpartials=-1;

	for(n=0;n<128;n++)
	{
		partials=(int)((SPEED*0.5f)/float(TUNING*(float)pow(2,(float) (n-69)/12.0f))); //(int) NYQUIST/f
		if(partials!=lastnumpartials)
		{
			datap=&data[partialindex*4096];
			for(b=0;b<4096;b++)
				datap[b]=0.0f;  //blank wavetable
			for(a=0;a<partials;a++)
			{
				partial=a+1;
				m=cos((float)a*HALFPI/(float)partials);  //gibbs
				m*=m;  //gibbs
				m/=(float)partial;
				for(b=0;b<4096;b++)
					datap[b]+=m*sinetable[(b*partial)%4096];
			}
			lastnumpartials=partials;
			a=int(2.0f*TUNING*(float)pow(2,(float) (n-69)/12.0f)); //2f
			for(b=reverseindex;b<=a;b++)
				reverse[b]=partialindex;
			reverseindex=a+1;
			partialindex++;
		}
	}

	for(b=reverseindex;b<32768;b++)
		reverse[b]=partialindex-1;

	ar << (int) partialindex; //number of waveforms
	ar << (int) 4096; //waveform size (in samples)

	max=0.0;
	for(b=0;b<4096;b++)
	{
		if(fabs(*(data+b))>max)  //normalise to richest waveform (0)
			max=(float)fabs(*(data+b));
	}
	for(b=0;b<4096*partialindex;b++)
	{
		*(data+b)/=max;
	}

	//ar.Write(data,4096*partialindex*sizeof(float));
	//ar.Write(reverse,32768*sizeof(int));

	delete [] data;
	delete [] reverse;
}
// end sawtooth

// parabola

	data=new float[128*4096];
	reverse=new int[32768];

	reverseindex=0;
	partialindex=0;
	lastnumpartials=-1;

	float sign;

	for(n=0;n<128;n++)
	{
```

```
                        partials=(int)((SPEED*0.5f)/float(TUNING*(float)pow(2,(float) (n-69)/12.0f)));
                        if(partials!=lastnumpartials)
                        {
                                datap=&data[partialindex*4096];
                                for(b=0;b<4096;b++)
                                        datap[b]=PI*PI/3.0f;
                                sign=-1.0f;
                                for(a=0;a<partials;a++)
                                {
                                        partial=a+1;
                                        m=cos((float)a*HALFPI/(float)partials); //gibbs
                                        m*=m;  //gibbs
                                        m/=(float)(partial*partial);
                                        m*=4.0f*sign;
                                        for(b=0;b<4096;b++)
                                                datap[b]+=m*sinetable[((b*partial)+1024)%4096]; //note, parabola uses cos
                                        sign=-sign;
                                }
                                lastnumpartials=partials;
                                a=int(2.0f*TUNING*(float)pow(2,(float) (n-69)/12.0f)); //2f
                                for(b=reverseindex;b<=a;b++)
                                        reverse[b]=partialindex;
                                reverseindex=a+1;
                                partialindex++;
                        }
                }

                for(b=reverseindex;b<32768;b++)
                        reverse[b]=partialindex-1;

                ar << (int) partialindex; //number of waveforms
                ar << (int) 4096; //waveform size (in samples)

                max=0.0;
                for(b=0;b<4096;b++)
                {
                        if(fabs(*(data+b))>max)  //normalise to richest waveform (0)
                                max=(float)fabs(*(data+b));
                }
                max*=0.5;
                for(b=0;b<4096*partialindex;b++)
                {
                        *(data+b)/=max;
                        *(data+b)-=1.0f;
                }

                //ar.Write(data,4096*partialindex*sizeof(float));
                //ar.Write(reverse,32768*sizeof(int));

                delete [] data;
                delete [] reverse;
        }
        // end parabola


//////////////////////////////////////////////////////////////////////
// An example of playback of a sawtooth wave
// This is not optimised for easy reading
// When optimising you'll need to get this in assembly (especially those
// float to int conversions)
//////////////////////////////////////////////////////////////////////

#define WAVETABLE_SIZE          (1 << 12)
#define WAVETABLE_SIZEF         WAVETABLE_SIZE.0f
#define WAVETABLE_MASK          (WAVETABLE_SIZE - 1)

float index;
float rate;
int wavetableindex;
float ratetofloatfactor;
float* wavetable;

void setupnote(int midinote /*0 - 127*/)
{
        float f=TUNING*(float)pow(2,(float) (midinote-69)/12.0f));
        rate=f*WAVETABLE_SIZEF/SPEED;
        ratetofloatfactor=2.0f*SPEED/WAVETABLE_SIZEF;
        index=0.0f;
        wavetableindex=reverse[(int)(2.0f*f)];
```

```
        wavetable=&sawtoothdata[wavetableindex*WAVETABLE_SIZE];
}

void generatesample(float* buffer,int length)
{
        int currentsample,
        int nextsample;
        float m;
        float temprate;
        while(length--)
        {
                currentsample=(int) index;
                nextsample=(currentsample+1) & WAVETABLE_MASK;
                m=index-(float) currentsample; //fractional part
                *buffer++=(1.0f-m)*wavetable[currentsample]+m*wavetable[nextsample]; //linear interpolation
                rate*=slide; //slide coeffecient if required
                temprate=rate*fm; //frequency modulation if required
                index+=temprate;
                if(index>WAVETABLE_SIZEF)
                {
                        //new cycle, respecify wavetable for sliding
                        wavetableindex=reverse[(int)(ratetofloatfactor*temprate)];
                        wavetable=&sawtoothdata[wavetableindex*WAVETABLE_SIZE];
                        index-=WAVETABLE_SIZEF;
                }
        }
}
```