BLEP and PolyBLEP Applied to Phase Distortion Synthesis with Plug-In Example

Davis Sprague

Phase distortion synthesis is a method originally developed by Casio for its CZ synthesizers in 1984. These products were developed as Casio's first foray into professional-quality musical keyboards and as a direct competitor to Yamaha's FM synthesizers, the DX line. PD Synthesis can produce nearly identical output waveforms to FM synthesis, but the method makes use of hard-synced counters in a wavetable or virtual analog oscillator instead of rapidly modulating the frequency of a signal.

The essential premise of PD Synthesis is relatively simple, and stated quite clearly in Masanori Ishibashi's 1985 patent. The method is based on "the rate of accessing a waveform chang[ing] in one cycle of the waveform." This implies that the original patent was intended for use with a wavetable synthesizer. Our implementation is also wavetable based, but the synthesis method could be implemented in any system that makes use of a counter. This means virtual analog oscillators, sample-based oscillators, and even delays and reverbs (which use circular buffers to achieve their effect) are fair game.

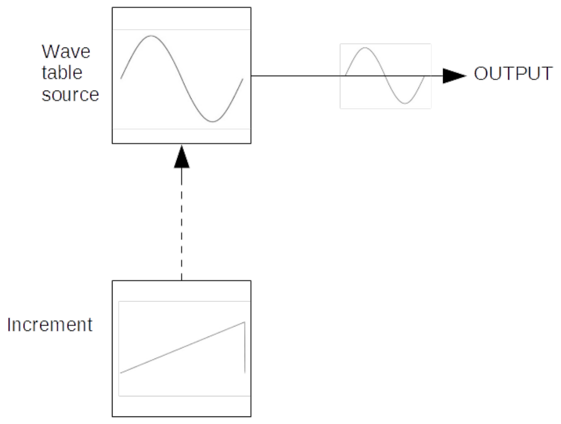

Fig 11.1 : A simple block diagram of a phase distortion synthesis oscillator

It may be a little difficult to completely buy into the effect at first, but I found that a simple example helped to visualize the resultant waveform. Examine figure 11.2 and 11.3, below. To the far left is the source waveform, a 9 point representation of a sine wave. In the middle is the incrementing waveform, a simple single cycle ramp. What this shows us is that every time interval the increment is increased by a consistent amount (in this case one) to produce the waveform in the far right, which is identical to the source.

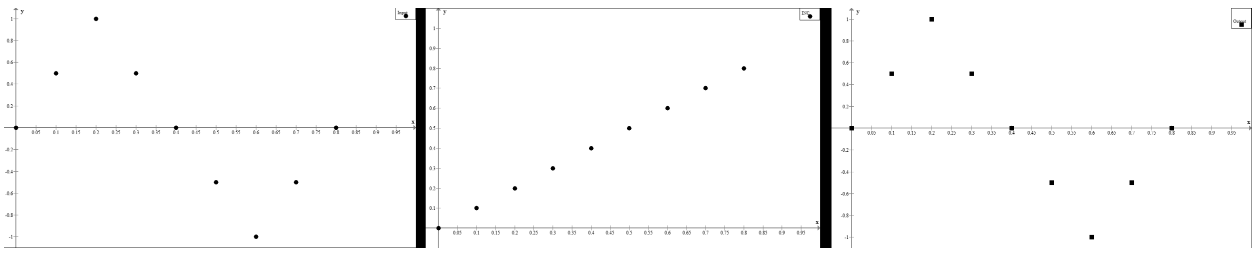

Fig 11.2 : Graphical representations of source, incrementer, and output waveforms

INPUT	INC	OUTPUT
0.0	0	0.0
0.5	1	0.5
1.0	2	1.0
0.5	3	0.5
0.0	4	0.0
-0.5	5	-0.5
-1.0	6	-1.0
-0.5	7	-0.5
0.0	8	0.0

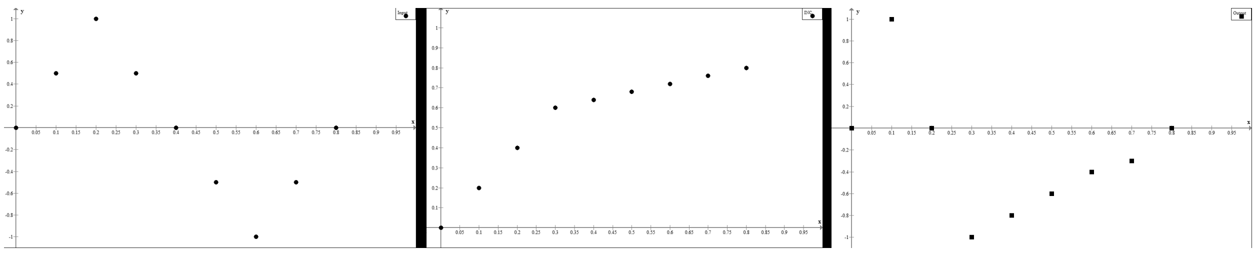

Fig 11.3 : Graphical representations of source, incrementer, and output waveforms

In figure 11.3, a typical example of phase distortion synthesis can be seen. The source, a simple sinusoid, can be seen to the far left. In the center the incrementer is shown again. Note that this time it is not a perfect ramp, but rather a segmented function consisting of two ramp functions with different slopes. It is also important to note that the function still only reaches the size of the wavetable for its maximum value. If the value of the incrementer is allowed to surpass the size of the wavetable, pitch shifting will result. With the segmented incrementer function a distorted waveform, shown to the far right, can be synthesized. The “in between” values in the output were achieved through interpolation.

INPUT	INC	OUTPUT
0.0	0.0	0.0
0.5	2.0	1.0
1.0	4.0	0.0
0.5	6.0	-1.0
0.0	6.4	-0.8
-0.5	6.8	-0.6
-1.0	7.2	-0.4
-0.5	7.6	-0.3
0.0	8.0	0.0

The block diagram shown in figure 11.4 provides another explanation of the logic of PD synthesis. It is worth pointing out again that the two increments – the normal and PD increments – are incremented independently but when the normal increment is reset to zero, the PD increment is forced to zero as well.

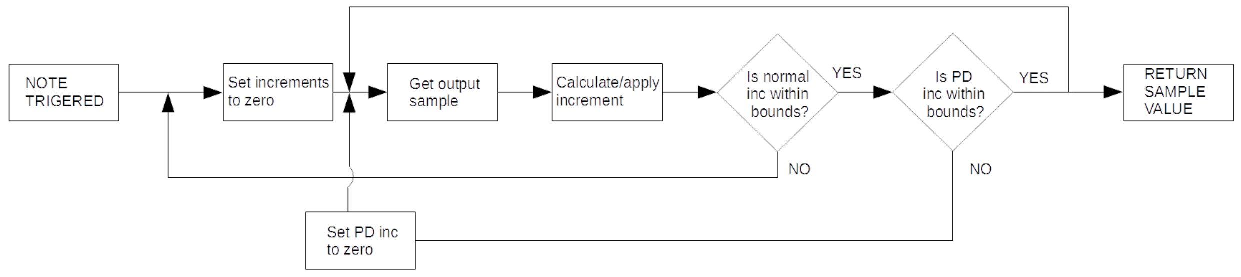

Fig 11.3 : PD flowchart/block diagram

Figure 11.4 shows the spectral content of a sawtooth waveform generated using phase distortion synthesis. Note the extra, unwanted harmonic content below -60dB. The application of BLEP and PolyBlep will help to remedy some of this. Another interesting component is the massive DC offset. This can be remedied with a simple high-pass filter.

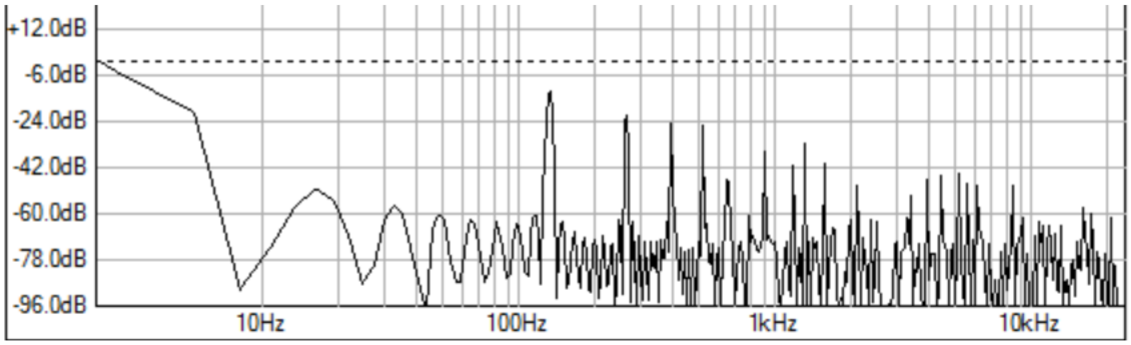

Fig 11.4: Spectrum of uncorrected sawtooth wave

Figure 11.5 shows the waveform that produces the bode plot in figure 11.6. Note that the sawtooth shape retains a slight curve from its source. This can be removed by fiddling with controls, but this shape was chosen to illustrate the wide variety of shapes that can be achieved under the basic heading of a sawtooth using PD synthesis.

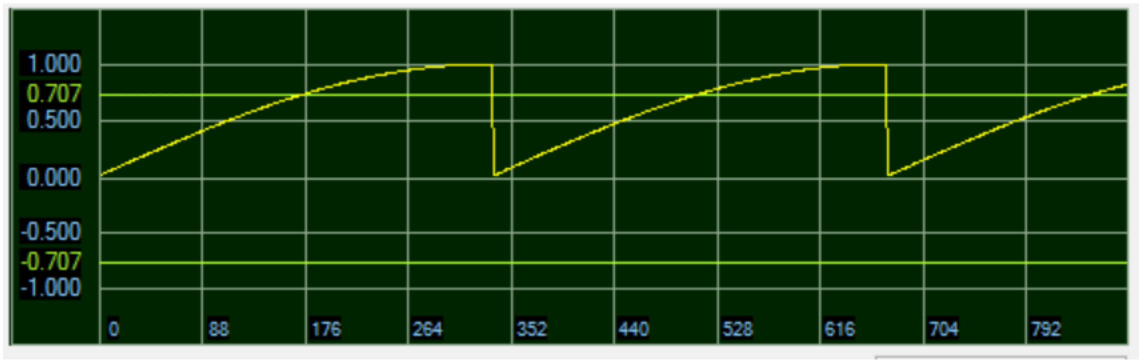

Fig 11.5 : Uncorrected sawtooth waveform

The final subject that needs to be addressed before moving into the application of BLEP to PD synthesis is the use of PD to create a resonant filter sweep. It is possible to emulate a very resonant filter by setting the x component of the incrementer function junction to be less than the y component. Figure 11.6 demonstrates graphically what a function of this type would look like. As you might guess, the incrementer being stuck at the maximum increment value for more than one sample cycle will produce much more high frequency content than otherwise.

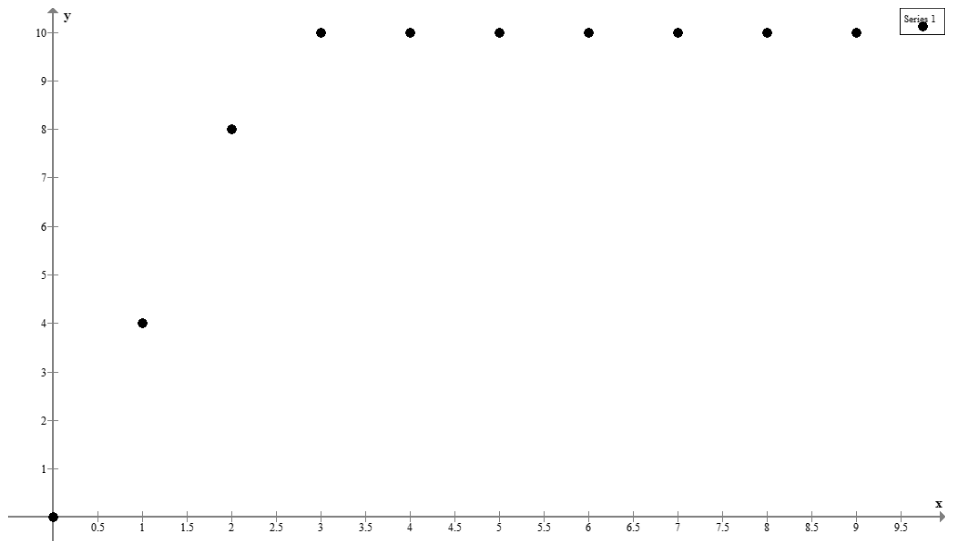

Fig 11.6 : Graphical representation of incrementer wave that will produce resonant peak

By setting the y component of the function junction to its maximum value, we can sweep the entire possible frequency range allowed by this filter emulation. Bode plots displaying the result of this sweep can be seen in figure 11.7. In his patent, Ishibashi refers to this as a case in which the modulation depth is greater than FF (255 in hexadecimal. This is because the x component of the incrementer function appears to have been a fixed value). He then goes on to explain that the waveform is read out multiple times per normal counter cycle, but because the PD increment is always forced to zero when the normal increment is the high frequency content is attenuated into a resonant peak.

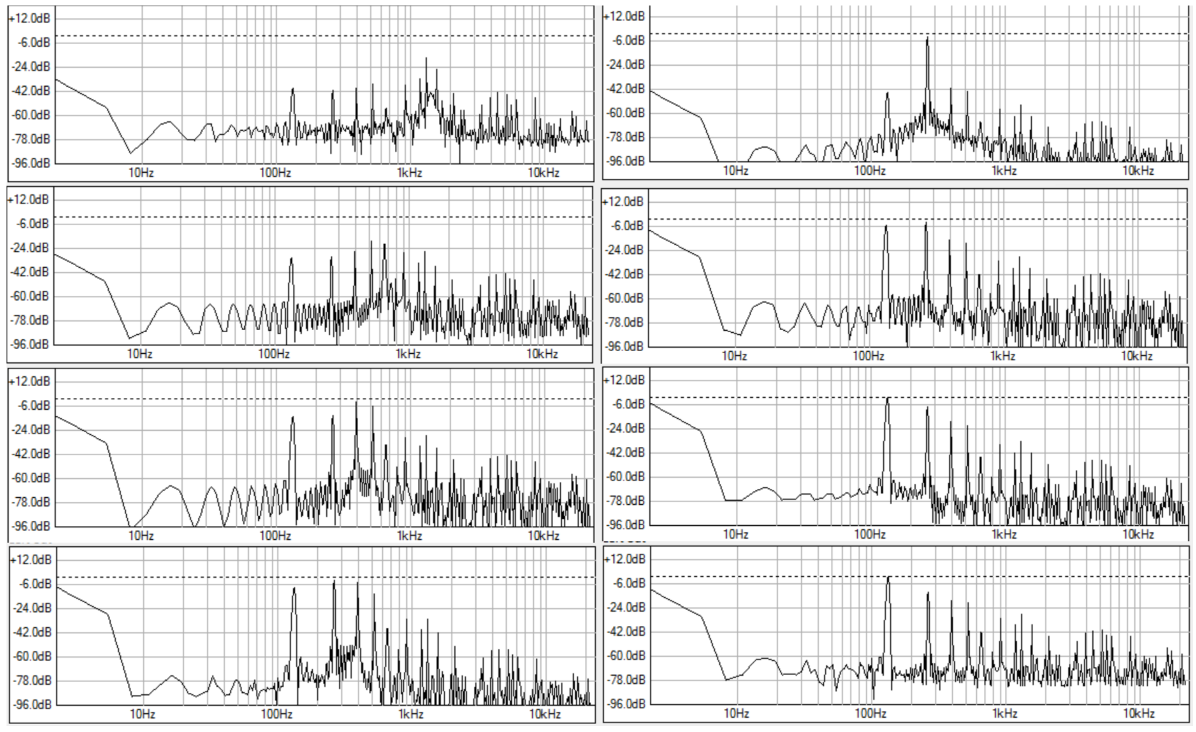

Fig 11.7 : Left side: 80% through 50% X value. Right side: 40% through 10% X value

An inherent problem with the PD Synthesis implementation in this app note is aliasing. Casio does not address their method of anti-aliasing in their patent, so it is left up to us to find a solution to the unwanted spectral content. One method for reducing aliasing is by applying something called BLEP (BandLimited stEP function). This method is usually used on waveforms for which the highest and lowest points are known beforehand, but some simple bookkeeping can help to work around this. The sample plug-in associated with this app note implements both BLEP and a variation called PolyBlep, the theory of which can be found in chapter 5 of *Designing Software Synthesizers in C++*. This app note will assume a basic grasp of the concept of BLEP, but will cover the variations necessary to implement it with PD synthesis.

To deal with the height of the function changing, it is necessary to hold on to the maximum height of the function in a given cycle. This can be accomplished by loading the current output sample into a persistent variable any time the increment wraps around (this includes both the normal increment and the PD increment). This means that there will be one inaccurate cycle at the beginning of any new PD setting, but this is not a perceptible error and provides the flexibility to handle any PD setting.

Because BLEP and PolyBlep are limited to very specific discontinuities, the technique will not reduce all aliasing in a PD synthesis signal. For example, when a signal is synthesized that has crossover-type distortion instead of a vertical discontinuity, the correction tables will be of no real assistance.

Implementing Phase Distortion Synthesis in a Wavetable Oscillator

For this implementation, we will start with the wave table oscillator object CWTOscillator that is detailed in *Designing Audio Effect Plug-Ins in C++* and implemented in pluginobjects.cpp. The parts of the code that we are most interested in are those handling incrementation.

```
class CPhaseDistortionSynthesis : public CPlugIn
{
public:

// - - - - snip - - - - - //
// Add your code here: --------------------------------------------------------- //
        float m_fReadIndex;
        float m_fQuadPhaseReadIndex;
        float m_f_inc;
        float m_fPDInc;
        float m_fPDReadIndex;

// - - - - snip - - - - //

}
```

PhaseDistortionSynthesis.cpp

```
// - - - - snip - - - - //
void CPhaseDistortionSynthesis::cookSlope()
{
        if(m_fPDReadIndex < 1024.0*(m_fRun1/100.0))
        {
                double m = m_fRise1/m_fRun1;
                double dInc = (m*m_fFrequency_Hz)/(float)m_nSampleRate;
                m_fPDInc = 1024.0*dInc;
        }
        else
        {
                double m = (1.0 - m_fRise1)/(1.0 - m_fRun1);
                double dInc = (m*m_fFrequency_Hz)/(float)m_nSampleRate;
                m_fPDInc = 1024.0*dInc;
        }
}
// - - - - snip - - - - //
void CPhaseDistortionSynthesis::doOscillate(float * pYn, float * pYqn)
{
        cookSlope();

        // - - - - snip - - - - //

        // add the increment for next time
        m_fPDReadIndex += m_fPDInc;
        m_fReadIndex += m_f_inc;
        m_fQuadPhaseReadIndex += m_f_inc;

        // check for wrap
        if(m_fReadIndex >= 1024)
        {
                m_fReadIndex = m_fReadIndex – 1024;
                m_fPDReadIndex = m_fReadIndex;
                        m_fStartingHeight = fOutSample;
        }
        if(m_fPDReadIndex >= 1024)
        {
                m_fPDReadIndex = m_fPDReadIndex - 1024;
                m_fStartingHeight = fOutSample;
        }

        if(m_fReadIndex < 0)
        {
                m_fReadIndex = 1023;
                m_fPDReadIndex = m_fReadIndex;
        }
        if(m_fPDReadIndex < 0)
                m_fPDReadIndex = 1024 + m_fPDReadIndex;


        // - - - - snip - - - - //
}
```

As can be seen in the source code above, it is necessary to create separate, persistent variables for the phase distortion read index and the normal read index. The reason can be seen in the conditionals – any time the normal read index reaches its limits, the phase distortion read index must be forced to the same value as the normal read index. If this hard-sync is ignored, the phase distortion will turn into a combination of distortion and pitch shifting.

Also of note is the fact that the cookSlope() function is called every time doOscillate is called. This is important because is allows for users to adjust their incrementer waveform in real time and ensures that the separate phase distortion read index is always accurate.

The last notable chunk of code is doBlep(float). This function is called once per sample period, because the BLEP code that it calls already has logic to hande whether or not to apply a BLEP correction. Essentially all the function does is hand the current sample to the predefined BLEP functions or return the uncorrected sample if no correction has been selected.

```
float doPDBLEP(float fOutSample)
{
	if(m_uBLEP == BLEP4)
	{
		return fOutSample + doBLEP_N(&dBLEPTable_8_BLKHAR[0], /* BLEP table */
		4096, /* BLEP table length */
		m_fReadIndex/1024.0, /* current phase value */
		fabs(m_fPDInc/1024.0), /* abs(dInc) is for FM synthesis with negative frequencies */
		m_fStartingHeight, /* sawtooth edge height = 1.0 */
		false, /* falling edge */
		4, /* 1 point per side */
		false); /* no interpolation */
	}

	if(m_uBLEP == BLEP8)
	{
		return fOutSample + doBLEP_N(&dBLEPTable_8_BLKHAR[0], /* BLEP table */
		4096, /* BLEP table length */
		m_fReadIndex/1024.0, /* current phase value */
		fabs(m_fPDInc/1024.0), /* abs(dInc) is for FM synthesis with negative frequencies */
		m_fStartingHeight, /* sawtooth edge height = 1.0 */
		false, /* falling edge */
		8, /* 1 point per side */
		false); /* no interpolation */
	}

	if(m_uBLEP == PB2)
	{
		return fOutSample + doPolyBLEP_2(m_fReadIndex/1024.0,
		abs(m_fPDInc/1024.0),/* abs(dInc) is for FM synthesis with negative frequencies */
		1.0, /* sawtooth edge = 1.0 */
		false); /* falling edge */
	}

	else
		return fOutSample;
}
```

References:

Ishibashi, Masanori. Electronic Musical Instrument. Casio Computer Co., Ltd., assignee. Patent 4658691. 21 Apr. 1987. Print.

Pirkle, William C. *Designing Software Synthesizer Plug-ins in C++ : For RackAFX, VST3, and Audio Units*. 2014, Focal Press. Print.