Virtual Analog (VA) 2nd Order Moog Half-Ladder Filter
Will Pirkle
September 19, 2013

Background

This brief App Note derives the Virtual Analog (VA) implementation for an interesting 2nd order Moog half-ladder filter based design. The ordinary Moog ladder filter is 4th order and reduces filter gain reduction as the Q of the filter is increased; just before self oscillation the filter gain reduction is about -14.1 dB. This 2nd Order version has a 2nd order response and only about -9 dB of passband reduction.

Figure 8.1 shows the block diagram of the ordinary Moog Ladder filter. There are four 1st order lowpass filters (LPFs) in series inside a delay-less feedback loop with loop gain -K. This creates resonance as the phase inversion of four synchronously tuned filters adds up to -180 degrees. It also reduces passband gain as shown in Figure 8.2. This is covered in detail in [Zavalishin], my other App Note *AN-4 Virtual Analog Filters* as well as many other sources.

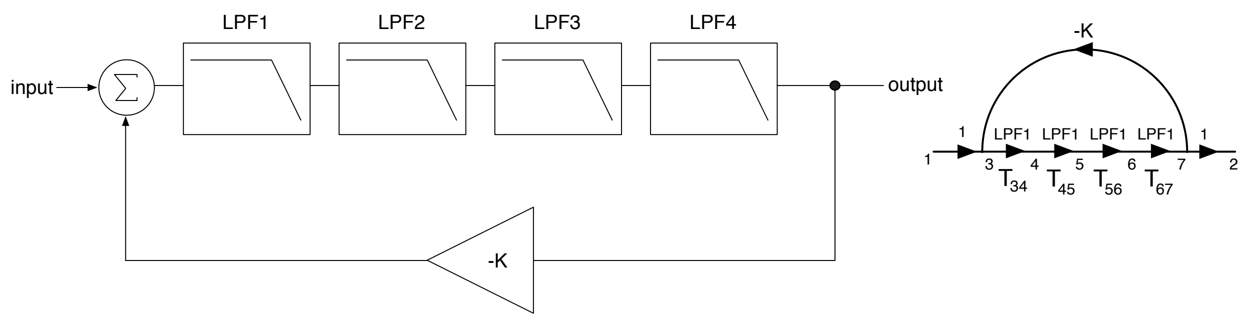

Figure 8.1: block diagram and signal flow graph of the ordinary Moog ladder filter

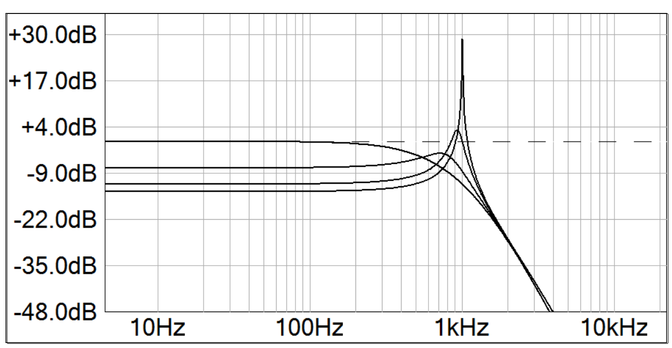

Figure 8.2: response of the 4th order Moog ladder filter with f_c = 1kHz and K = 0 (no peaking) to K = 3.99 (just before self-oscillation)

VA Moog Ladder Model

Figure 8.3 shows the ordinary Moog Ladder Filter block diagram. This is explained in App Note AN-4 Virtual Analog filters.

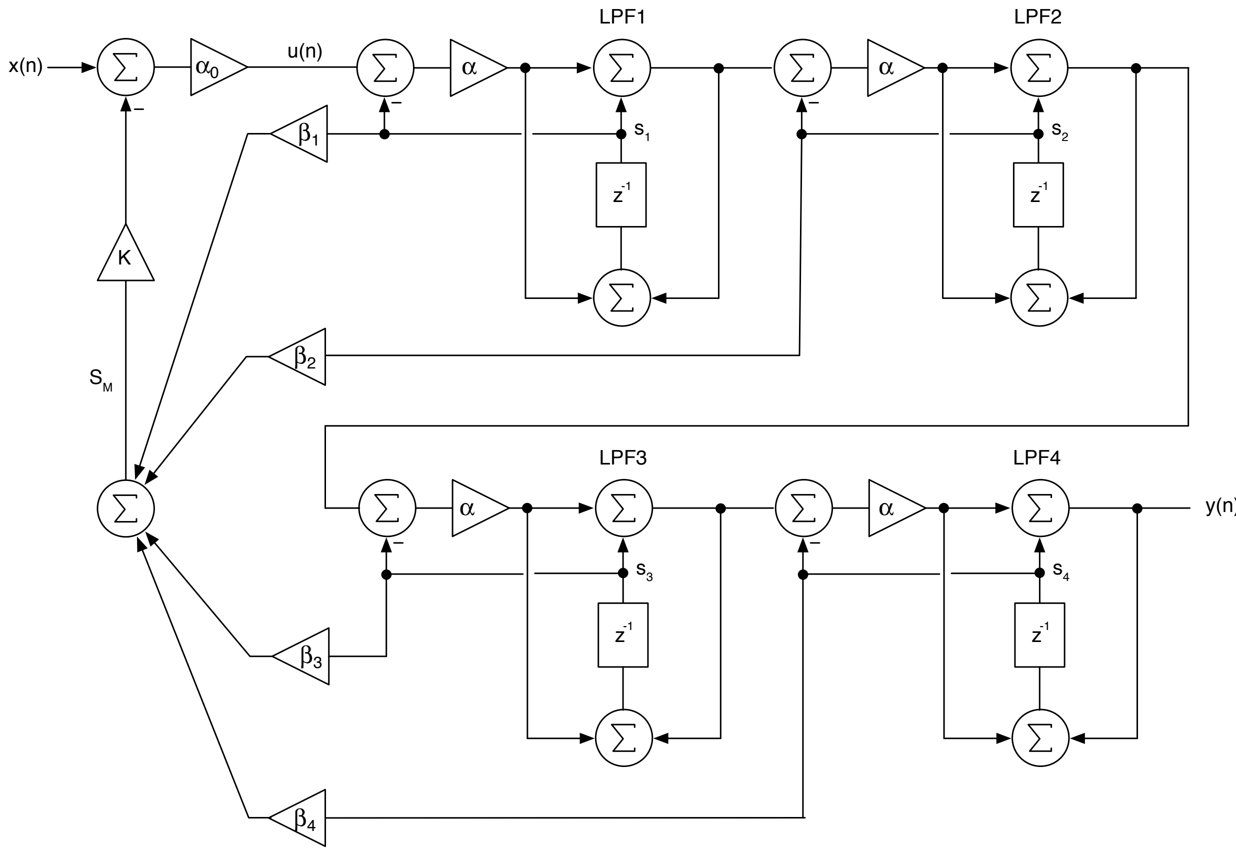

Figure 8.3: VA Moog Ladder Filter

One strategy is to find the value for the node u(n) that feeds the loop (note you can also solve for y vs. x directly and formulate the filter that way, but it will result in a slightly more complex filter). Finding u(n) can be done in two different ways; one way is to put the whole cascade of 1st order LPFs in the form $y = G_M u + S_M$ and then resolve the loop. The other way is to solve for u and resolve the loop at the same time. Since Zavalishin's original version used the first method for finding u(n), we'll use it here. The full derivation is in App Note 4 and the final results are shown here.

$$y = G_M u + S_M \qquad G_M = G^4 \qquad S_M = G^3 S1 + G^2 S2 + GS3 + S4$$

where

$$G = \frac{g}{1+g} = \alpha$$

$$S1 = \frac{s_1}{1+g} \quad S2 = \frac{s_2}{1+g} \quad S3 = \frac{s_3}{1+g} \quad S4 = \frac{s_4}{1+g}$$

and

$$S_M = \beta_1 s_1 + \beta_2 s_2 + \beta_3 s_3 + \beta_4 s_4$$

$$\beta_1 = \frac{G^3}{1+g} \quad \beta_2 = \frac{G^2}{1+g} \quad \beta_3 = \frac{G}{1+g} \quad \beta_4 = \frac{1}{1+g}$$

Then find u(n), the input to the loop.

$$u(n) = \frac{x(n) - KS_M}{1+KG_M}$$

$$= \alpha_0 \left[x(n) - KS_M \right]$$

$$\alpha_0 = \frac{1}{1+KG_M}$$

VA Moog Half-Ladder Model

In order to use the same topology but reduce the filter order, a first order All Pass Filter (APF) is used to replace two of the LPF blocks. The APF provides the missing -90 degrees of phase shift but does not alter the frequency response and therefore passband gain. Figure 8.4 shows the block digram of the new 2nd order Moog ladder-based filter. The filter has the typical 12dB/octave roll-off but with only about -9 dB of passband attenuation as shown in Figure 8.5. Because there are only two LPFs absorbing energy from the loop, the K value is reduced from -4 to -2 for self oscillation.

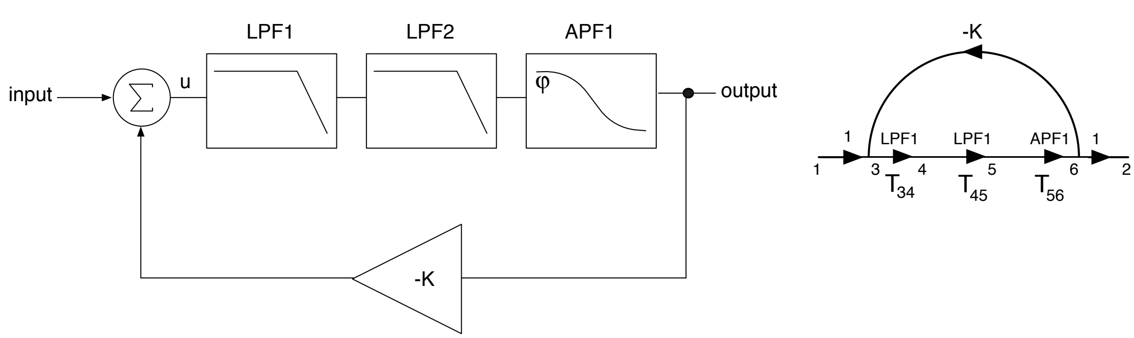

Figure 8.4: block diagram and signal flow graph of the 2nd order ladder filter

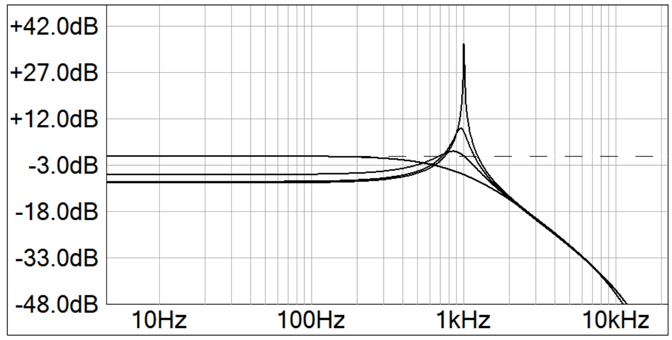

Figure 8.5: frequency response of the 2nd order ladder filter with f_c = 1kHz and K = 0, 1.0, 1.6 and 2.0

Figure 8.6 shows a non-linear model placing the Non Linear Processing block (tanh) in the feed-forward path using Zavalishin's "cheap" implementation.

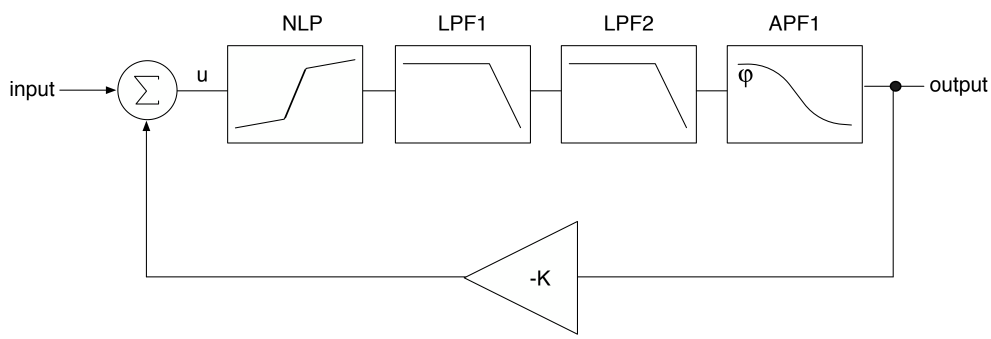

Figure 8.6: the nonlinear model uses the cheap implementation placing the nonlinearity at the entrance to the loop

The Virtual Analog APF is shown in Figure 8.7. It is the one-pole VA filter with the outputs subtracted y_{AP} = y_{LP} - y_{HP}. As with my other VA implementations, I have modified the original structure with a feedback coefficient β to produce the output βs(n) which allows simplification of the block diagram and implementation. This single building block is used to implement the three filters in the design.

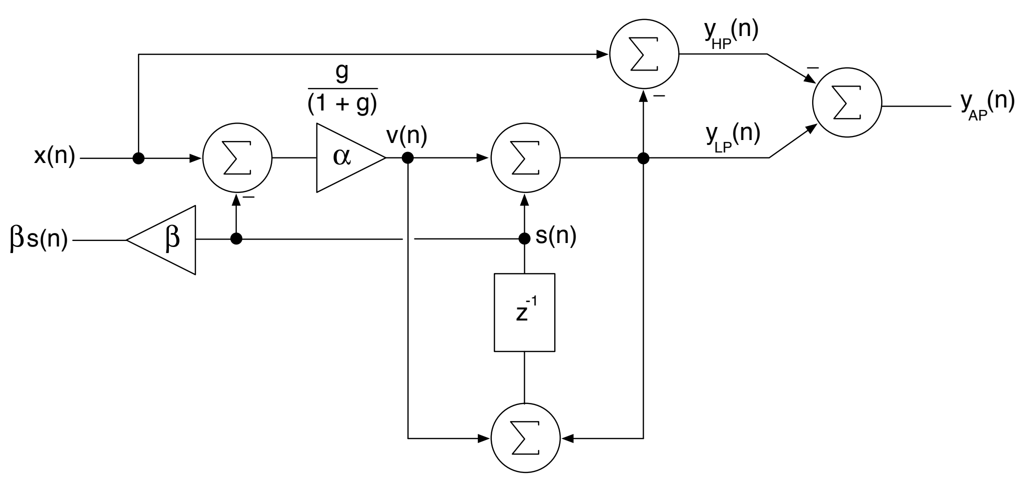

Figure 8.7: APF building block in virtual analog form

VA Equations

First, let's look at the VA equation for the APF:

$$
\begin{aligned}
y_{LP} &= Gx + S \\
y_{HP} &= x - Gx - S \\
y_{AP} &= (2G-1)x + 2S \\
&= G_A x + S_A \\
G_A &= 2G - 1 \\
S_A &= 2S
\end{aligned}
$$

Solving for *u*, the input to the first LPF in the feedback loop (and ignoring the NLP tanh() block) we start with the equation relating *u* and the output *y*:

$$
\begin{aligned}
y_{LP1} &= Gx + S1 \\
y_{LP2} &= Gx + S2 \\
y_{AP1} &= G_A x + S_A \\
G &= \frac{g}{1+g} \\
S1 &= \frac{s_1}{1+g} \\
S2 &= \frac{s_2}{1+g}
\end{aligned}
$$

$$
\begin{aligned}
y &= G_A G^2 u + G_A G S1 + G_A S2 + S_A \\
&= G_M u + S_M \\
G_M &= G_A G^2 \\
S_M &= G_A G S1 + G_A S2 + S_A
\end{aligned}
$$

Now rearrange and find *u*:

$$u = \frac{x - KS_M}{1 + KG_M}$$

let

$$\alpha_0 = \frac{1}{1 + KG_M}$$

$$\beta_1 = \frac{G_A G}{1 + g}$$

$$\beta_2 = \frac{G_A}{1 + g}$$

$$\beta_3 = \frac{2}{1 + g}$$

then

$$\begin{aligned} u &= \alpha_0 (x - KS_M) \\ &= \alpha_0 (x - K(\beta_1 s_1 + \beta_2 s_2 + \beta_3 s_3)) \end{aligned}$$

Using the result for u, we can construct a block diagram of the filter, shown in Figure 8.8.

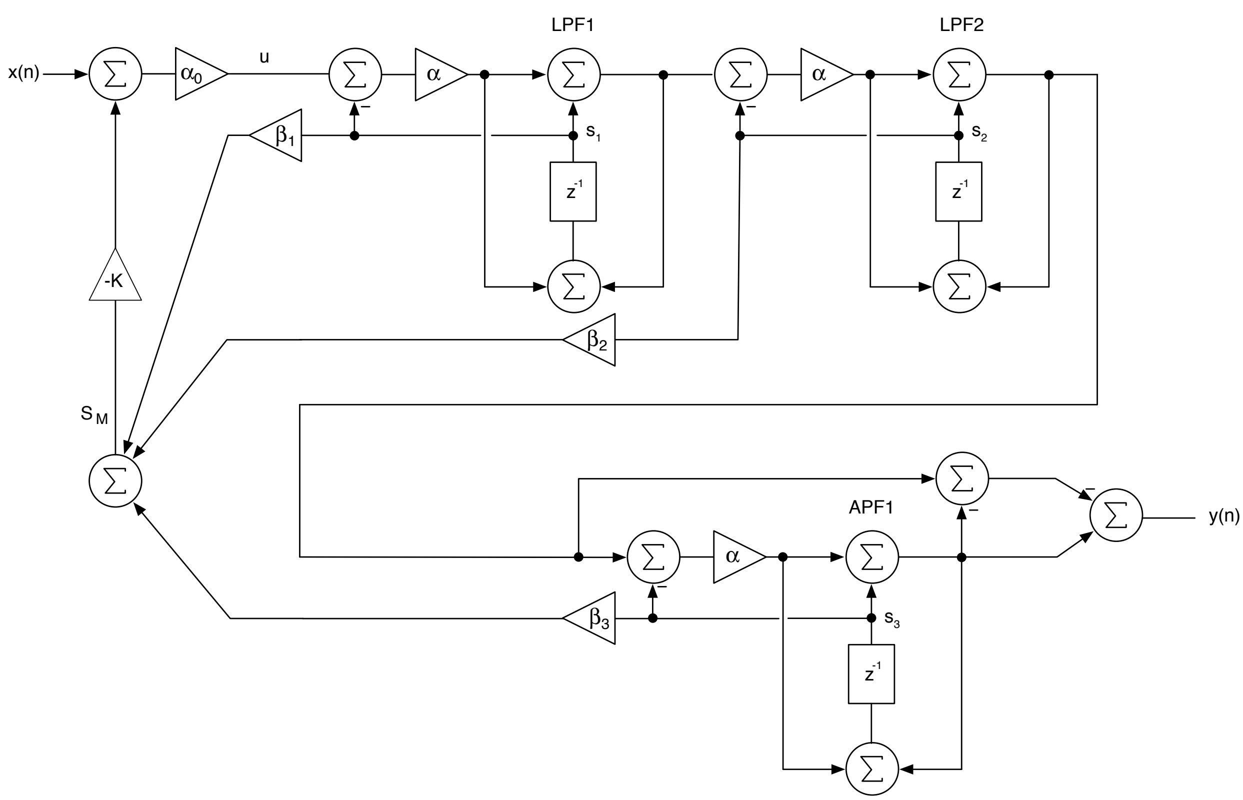

Figure 8.8: the 2nd order ladder filter block VA realization; S_M is the sum of feedback values from each filter

Sample Code

The design was rapidly prototyped and implemented using the RackAFX software. The VA one pole filters are implemented in the CVAOnePoleFilter object while the rest of the plug-in is implemented in CMoogLadderFilter. This plug-in implements both the (full) 4th order and (half) 2nd order versions so you get both filters in one chunk of code. The full code can be found at my website www.willpirkle.com but the interesting bits are here:

In CVAOnePoleFilter you can see the formation of the LP, HP and AP outputs:

```
// do the filter
float CVAOnePoleFilter::doFilter(float xn)
{
	// calculate v(n)
	float vn = (xn - m_fZ1)*m_fAlpha;

	// form LP output
	float lpf = vn + m_fZ1;

	// update memory
	m_fZ1 = vn + lpf;

	// do the HPF
	float hpf = xn - lpf;
	float apf = lpf - hpf;

	if(m_uFilterType == LPF1)
		return lpf;
	else if(m_uFilterType == HPF1)
		return hpf;
	else if(m_uFilterType == APF1)
		return apf;

	// default
	return lpf;
}
```

In MoogLadderFilter.h we declare the member objects and a helper function to update the filter:

```
	// Add your code here: --------------------------------------------------------- //
	CVAOnePoleFilter m_LPF1;
	CVAOnePoleFilter m_LPF2;
	CVAOnePoleFilter m_LPF3;
	CVAOnePoleFilter m_LPF4;

	// for 2nd order half-ladder
	CVAOnePoleFilter m_APF1;

	// for loop scalar
	float m_fAlpha0;

	// for UI changes
	void updateFilters();
```

```
    // to tell the filters what they are
    enum{LPF1,HPF1,APF1}; /* one short string for each */
    // END OF USER CODE ---------------------------------------------------------- //
```

In MoogLadderFilter.cpp:

prepareForPlay()

```
bool __stdcall CMoogLadderFilter::prepareForPlay()
{
    // Add your code here:
    m_LPF1.m_uFilterType = LPF1;
    m_LPF2.m_uFilterType = LPF1;
    m_LPF3.m_uFilterType = LPF1;
    m_LPF4.m_uFilterType = LPF1;
    m_APF1.m_uFilterType = APF1;

    m_LPF1.m_fSampleRate = (float)m_nSampleRate;
    m_LPF2.m_fSampleRate = (float)m_nSampleRate;
    m_LPF3.m_fSampleRate = (float)m_nSampleRate;
    m_LPF4.m_fSampleRate = (float)m_nSampleRate;
    m_APF1.m_fSampleRate = (float)m_nSampleRate;

    m_LPF1.reset();
    m_LPF2.reset();
    m_LPF3.reset();
    m_LPF4.reset();
    m_APF1.reset();

    updateFilters();

    return true;
}
```

updateFilters()

```
void CMoogLadderFilter::updateFilters()
{
    // prewarp for BZT
    double wd = 2*pi*m_dFc;
    double T  = 1/(double)m_nSampleRate;
    double wa = (2/T)*tan(wd*T/2);
    double g  = wa*T/2;

    // G - the feedforward coeff in the VA One Pole
    //     named alpha in my block diagrams
    float G = g/(1.0 + g);

    if(m_uModel == HALF)
    {
        // the allpass G value
        float GA = 2.0*G-1;
```

```
            // set alphas
            m_LPF1.m_fAlpha = G;
            m_LPF2.m_fAlpha = G;
            m_APF1.m_fAlpha = G;

            m_LPF1.m_fBeta = GA*G/(1.0+g);
            m_LPF2.m_fBeta = GA/(1.0+g);
            m_APF1.m_fBeta = 2.0/(1.0+g);

            // calculate alpha0
            // for 2nd order, K = 2 is max so limit it there
            float K = m_fK;
            if(m_uModel == HALF && K > 2.0)
                K = 2.0;

            m_fAlpha0 = 1.0/(1.0 + K*GA*G*G);
        }
        if(m_uModel == FULL)
        {
            // set alphas
            m_LPF1.m_fAlpha = G;
            m_LPF2.m_fAlpha = G;
            m_LPF3.m_fAlpha = G;
            m_LPF4.m_fAlpha = G;

            // set beta feedback values
            m_LPF1.m_fBeta = G*G*G/(1.0+g);
            m_LPF2.m_fBeta = G*G/(1.0+g);
            m_LPF3.m_fBeta = G/(1.0+g);
            m_LPF4.m_fBeta = 1.0/(1.0+g);

            // calculate alpha0
            // Gm = G^4
            m_fAlpha0 = 1.0/(1.0 + m_fK*G*G*G*G);
        }
}
```

processAudioFrame() - note the calls to getFeedbackOutput() - this returns the βs(n) for each filter

```
bool __stdcall CMoogLadderFilter::processAudioFrame(float* pInputBuffer, float* pOutputBuffer,
                                                    UINT uNumInputChannels,
                                                    UINT uNumOutputChannels)
{
        // MONO plugin!
        float SM = 0;
        float y = 0;
        if(m_uModel == HALF)
        {
            SM = m_LPF1.getFeedbackOutput() + m_LPF2.getFeedbackOutput() +
                m_APF1.getFeedbackOutput();
        }
        else if(m_uModel == FULL)
        {
            SM = m_LPF1.getFeedbackOutput() + m_LPF2.getFeedbackOutput() +
```

```
                m_LPF3.getFeedbackOutput() + m_LPF4.getFeedbackOutput();
    }

    float K = m_fK;
    if(m_uModel == HALF && K > 2.0)
        K = 2.0;

    float u = m_fAlpha0*(pInputBuffer[0] - K*SM);

    // saturate?
    if(m_uNLP)
        u = tanh(u);

    // push u through the series
    if(m_uModel == HALF)
        y = m_APF1.doFilter(m_LPF2.doFilter(m_LPF1.doFilter(u)));
    if(m_uModel == FULL)
        y = m_LPF4.doFilter(m_LPF3.doFilter(m_LPF2.doFilter(m_LPF1.doFilter(u))));

    // NOTE this is a mono filter!
    pOutputBuffer[0] = y;

    // Mono-In, Stereo-Out (AUX Effect)
    if(uNumInputChannels == 1 && uNumOutputChannels == 2)
        pOutputBuffer[1] = y;

    // Stereo-In, Stereo-Out (INSERT Effect)
    if(uNumInputChannels == 2 && uNumOutputChannels == 2)
        pOutputBuffer[1] = y;


    return true;
}
```

Revision History:

1.0: Initial Release, *September 26, 2013*

References:

Pirkle, Will. 2012. *Designing Audio Effect Plug-Ins in C++*, Burlington: Focal Press.

Zavalishin, Vadim. 2012. *The Art of VA Filter Design*, http://www.native-instruments.com/fileadmin/ni_media/downloads/pdf/VAFilterDesign_1.0.3.pdf