Virtual Analog (VA) Diode Ladder Filter

Will Pirkle
July 19, 2013

This App Note derives the Virtual Analog (VA) equations for the Diode Ladder Filter. The VA derivation and Topology Preserving Transform (TPT) filters are done using the methods and topologies in Zavalishin's *The Art of VA Filter Design*. You will need to be familiar with this book to understand the derivation (see the References for a link to a download). The Zavalishin book lays the groundwork and derives the global topology and global equations for the Diode Ladder Filter but it does not provide a final block diagram or local filter I/O equations. This App Note derives those equations and synthesizes a block diagram from them. You definitely need to refer to the Zavalishin book when using this App Note - there is much information (such as the derivation of the frequency response and pole locations) that I won't repeat here; it is already well documented in the VA book.

Background

The Diode Ladder Filter first appeared in the EMS VCS3 Monophonic Synth designed by David Cockerell in 1969. It is (more famously) incorporated in the Roland TB-303 BassLine monophonic bass synth from 1982. It is based on the Moog Ladder Filter (see App Note 4) but incorporates multiple feedback paths between sections. The effect of the feedback paths on the signal is two-fold: like the Moog Ladder, it reduces overall filter gain as the resonance increases but the reduction is more extreme (by about 12dB) and secondly, as the resonance increases, the resonant frequency migrates upwards, but never makes it to the cutoff frequency (which does not occur in the Moog Ladder). Like the Moog Ladder it also self oscillates. At the point of self-oscillation, the poles (and therefore the resonant peak) will have drifted up to fc/sqrt(2).

Block Diagram

Like the Moog Ladder Filter, the Diode Ladder incorporates four synchronously tuned first order lowpass filters (LPFs) in series embedded in a global feedback loop. The negative feedback loop has a gain *k* which creates positive feedback only at the cutoff frequency due to the fact that a fourth order filter produces 180 degrees of phase shift at the cutoff frequency, f_c. See App Note 4 for more details. However, unlike the Moog Ladder, the Diode Ladder incorporates multiple feedback loops around each section (LPF2 feeds back into LPF1, LPF3 feeds back into LPF2, LPF4 feeds back into LPF3) as shown in Figure 6.1.

The filter contains multiple nonlinearities typically modeled by nonlinear processing (NLP) blocks consisting of the tanh() function. The analysis with these NLP blocks is difficult as it generally results in transcendental equations that can not be solved directly. Therefore it is easy to start with the linear version first, then alter it to add nonlinearities. Figure 6.1 shows the block diagram of the *linear* version. The nonlinear version has NLP blocks in each local feedback path. In addition to the feedback paths, there are also attenuators on the inputs to the last three LPFs, without which the filter becomes unstable. The attenuators are labeled a0 in my analysis. In Figure 6.1 I have labeled the intermediate nodes; the input is x and the output is y_4 while the node labeled u is the input into the first summer. We will be interested in finding this value u (see Zavalishin for details on why). Each LPF output is labeled y_1 through y_4.

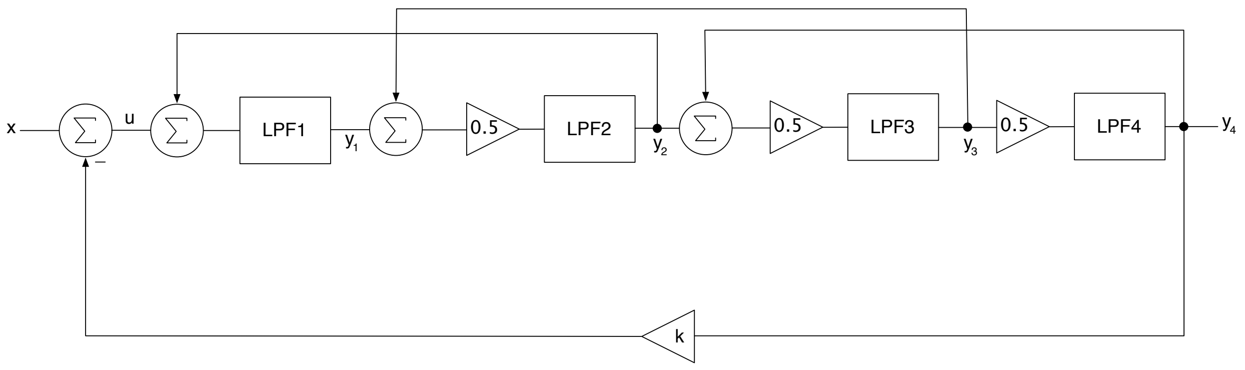


Figure 6.1: block diagram of the linearized Diode Ladder Filter.

Derivation of the Filter Equations

The basic idea here is to solve for the value of the node u, the input to the first summer. You can see that:

$$u = x - ky_4$$

so we need to find a relationship between y₄ and x in order to solve for u. This follows a repeated process outlined and fully explained in the Zavalishin book.

The process begins by isolating just the diode ladder, independent of the global feedback loop. The ladder is shown in Figure 6.2.

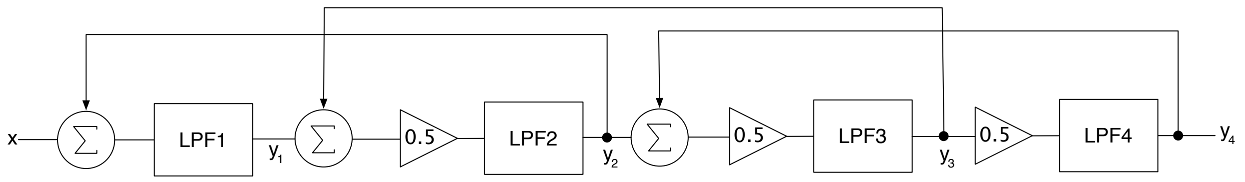


Figure 6.2: the diode ladder isolated; we want to form a relationship between y_4 and x

My derivation follows three steps:

1. each LPF is a first order TPT module with its own local values for G and S where each Nth filter has [Zavalishin Eq. 3.12]

$$y_N = Gx_N + S$$

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

$$S = \frac{s_N}{1+g}$$

2. form equations that relate inputs to outputs with feedback and attenuators; i.e. solve [Zavalishin Eq. 3.12] for each filter with summed inputs (input + feedback); this results in a new set of G and S variables related to the feedback paths around filter pairs

3. form equations that solve the single global feedback path around the final structure in Figure 6.1

Knowing that step 1 holds true for each LPF (if you don't understand this, stop and read Zavalishin Chapters 1-3) we can move to step 2 and begin solving for the input + feedback combinations. This is accomplished by starting with LPF4 and working backwards through the ladder. These exploit equations 3.12 - 3.14 in the Zavalishin book. If you are following along with the Zavalishin derivation (pp 73 - 74) you will notice my equations are different. This is because my equations relate the filter output y_N to each filter input (y_{N-1} + feedback) whereas the Zavalishin equations relate the output to ξ which is (y-x) for each filter. This isn't incorrect, but ultimately you have to resolve the issue. I am resolving it right from the beginning.

LPF4
This filter has an input attenuator a0 = 0.5 and it has no feedback into it; it is the easiest to derive, starting with [Zavalishin Equation 3.12]:

$$\begin{aligned} y_4 &= g(x-y_4)+s_4 \\ &= g(0.5y_3-y_4)+s_4 \\ y_4+gy_4 &= 0.5gy_3+s_4 \\ y_4(1+g) &= 0.5gy_3+s_4 \\ y_4 &= \frac{0.5gy_3+s_4}{1+g} \end{aligned}$$

or

$$y_4 = G4y_3+S4 \qquad G4=\frac{0.5g}{1+g} \qquad S4=\frac{s_4}{1+g} \qquad \text{[Eq. 6.1]}$$

LPF3
This filter has an input attenuator a0 = 0.5 and we have feedback from y4. Thus the input to the filter is $0.5y_2 + 0.5y_4$. Here, we use the results in Eq. 6.1 to fashion an equation that removes the dependence on the y_4 input. This is a common theme in the rest of the derivation. Note that Eq 6.2 below does not have y_4 in it; it is only dependent on the "normal" input y_2.

$$\begin{aligned} y_3 &= g(x-y_3)+s_3 \\ &= g(0.5y_2+0.5y_4-y_3)+s_3 \\ &= g(0.5y_2+0.5(G4y_3+S4)-y_3)+s_3 \\ &= 0.5gy_2+0.5gG4y_3+0.5gS4-gy_3+s_3 \\ y_3+gy_3-0.5gG4y_3 &= 0.5gy_2+0.5gS4+s_3 \\ y_3(1+g-0.5gG4) &= 0.5gy_2+0.5gS4+s_3 \\ y_3 &= \frac{0.5gy_2+0.5gS4+s_3}{1+g-0.5gG4} \end{aligned}$$

or

$$y_3 = G3y_2 + S3 \qquad G3 = \frac{0.5g}{1+g-0.5gG4} \qquad S3 = \frac{0.5gS4+s_3}{1+g-0.5gG4} \quad \text{[Eq. 6.2]}$$

LPF2
This filter has an identical topology as LPF3 so the analysis is essentially the same except the summed input values are different. Again, note that the final result in Eq. 6.3 is independent of the feedback term y_3.

$$\begin{aligned}
y_2 &= g(x-y_2)+s_2 \\
&= g(0.5y_1+0.5y_3-y_2)+s_2 \\
&= g(0.5y_1+0.5(G3y_2+S3)-y_2)+s_2 \\
&= 0.5gy_1+0.5gG3y_2+0.5gS3-gy_2+s_2 \\
y_2+gy_2-0.5gG3y_2 &= 0.5gy_1+0.5gS3+s_2 \\
y_2(1+g-0.5gG3) &= 0.5gy_1+0.5gS3+s_2 \\
y_2 &= \frac{0.5gy_1+0.5gS3+s_2}{1+g-0.5gG3}
\end{aligned}$$

or

$$y_2 = G2y_1 + S2 \qquad G2 = \frac{0.5g}{1+g-0.5gG3} \qquad S2 = \frac{0.5gS3+s_2}{1+g-0.5gG3} \quad \text{[Eq. 6.3]}$$

LPF1
This filter has no input attenuator (a0 = 1.0) but we still have feedback from y2 so the analysis is practically the same as for the preceding two filters.

$$\begin{aligned}
y_1 &= g(x-y_1)+s_1 \\
&= g(x+y_2-y_1)+s_1 \\
&= g(x+G2y_1+S2-y_1)+s_1 \\
&= gx+gG2y_1+gS2-gy_1+s_1 \\
y_1+gy_1-gG2y_1 &= gx+gS2+s_1 \\
y_1(1+g-gG2) &= gx+gS2+s_1 \\
y_1 &= \frac{gx+gS2+s_1}{1+g-gG2}
\end{aligned}$$

or

$$y_1 = G1x + S1 \qquad G1 = \frac{g}{1+g-gG2} \qquad S1 = \frac{gS2+s_1}{1+g-gG2} \quad \text{[Eq. 6.4]}$$

Now we have two sets of equations for the y_n values; the "usual" ones are local to each LPF (Zavalishin Eq. 3.13 exactly) so nothing changes there plus our newly derived equations. The equations we just derived tell us the exact I/O relationship for each filter (with feedback) but based only on the y_{N-1} value as an input. It is important not to try to mix the local and global versions of the equations and this is a potential area of confusion because of the reuse of the G and S terminology for the variables; this is why Eq. 6.1 - 6.4 explicitly state them as GN and SN where N is the filter number.

Block Diagram Synthesis

Now we turn our attention to the individual filters themselves, starting with the last one and working backwards to the first. We will need to create the SN feedback variables within each filter stage to be fed back to the previous one. We also need to place attenuators of 0.5 in front of the last 3 stages.

LPF4

This is the simplest - it has no feedback into the input. It only adds an attenuator a0 value. I modify the normal TPT stage to allow for a feedback output (and a feedback input for the other 3 stages), otherwise it resembles the typical 1st order TPT stage.

LPFs 3,2,1

These all have a local delay-less feedback path from the filter that follows mixed in with the input signal. These need a modified structure to accommodate the feedback signals. Lets look at LPF3 as an example; Figure 6.3 isolates the LPF3/LPF4 combination.

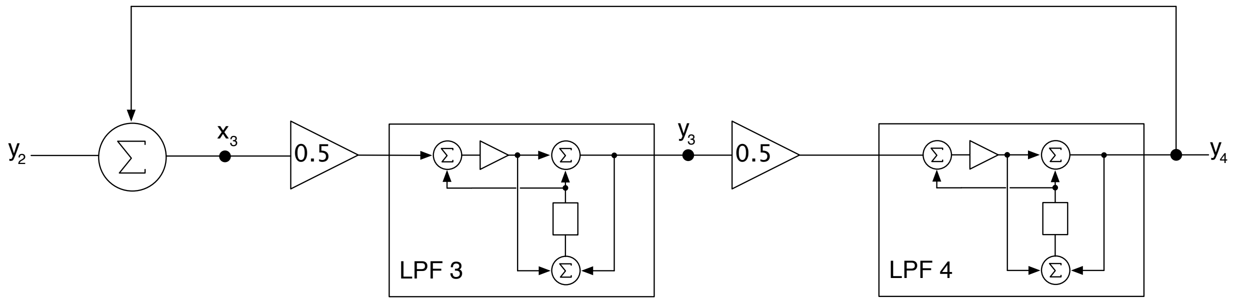


Figure 6.4: the LPF3/LPF4 filter pair in isolation

We have another delay-less feedback loop to deal with around LPF3 and LPF4. But [Eq. 6.2] describes this loop the same way that the (local TPT) delay-less loop equation does, taking into account the source of feedback and formulating an equation based only on only the non-feedback input - it's the same process. So most of the work is done in Eq 6.1 - 6.4. What we need to do is find x_N for LPF N where x_N is the input into the filter (or filter with attenuator a0) *after* the feedback summer, or x_3 above. Since three of the four LPFs have the 0.5 attenuator, we can roll that coefficient into the filter block diagram (and C++ object). Solving for x_3:

$$\begin{aligned} x_3 &= y_2 + y_4 \\ &= y_2 + G4y_3 + S4 \\ &= y_2 + G4(G3y_2 + S3) + S4 \\ &= y_2 + G3G4y_2 + G4S3 + S4 \\ &= y_2(1 + G3G4) + G4S3 + S4 \end{aligned}$$

The G4 and S4 values represent the feedback components. We need to realize this structure in a block diagram. Isolating just this portion we could implement it as in Figure 6.5.

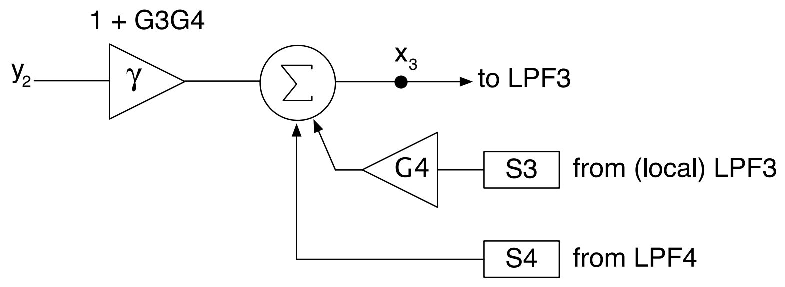


Figure 6.5: one way to get the summed input and feedback signals

Verify for yourself that Figure 6.5 does indeed calculate x_3 as per the equation above:

$$x_3 = y_2(1 + G3G4) + G4S3 + S4$$

The value S4 is going to come from LPF 4, so it is the feedback. The G4 value is based on LPF4, but is pre-calculated and only needs to change when fc changes. The value S3 is local feedback from LPF3.

$$G4 = \frac{0.5g}{1+g} \qquad S4 = \frac{s_4}{1+g}$$

The S3 value comes from LPF3 and is more complicated and it also contains the S4 component.

$$S3 = \frac{0.5gS4 + s_3}{1 + g - 0.5gG4}$$

We need to modify the structure of the diode ladder filter stage to include the input attenuator, the feed back in and feed back out nodes and the coefficients to create the S values. The generalized block here has both the original 1st order stage and the additional components to generate the rest of what we need. Figure 6.6 shows a modified first order TPT lowpass filter stage - you can see the “usual” circuit at the right, with a bunch of new coefficients and a summer added to the left. This altered block diagram is encapsulated in a C++ object I call *CVAOnePoleFilterEx* since I already use CVAOnePoleFilter in other App Notes and projects. The extended (Ex) version adds the new coefficients and summer. This module allows us to inject a feedback a signal into the block (FB IN) as well as extract a feedback signal out of it (FB OUT).

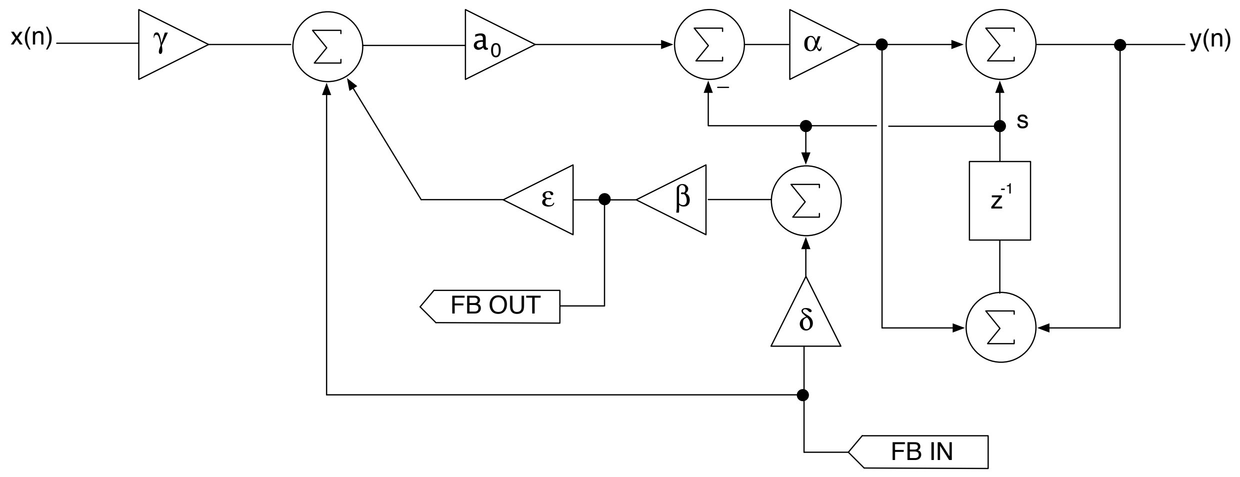


Figure 6.6: the modified TPT LPF for use in the Diode Ladder Filter will allow both feedback input and feedback output

Using LPF3/LPF4 as and example again, we need to implement [Eq. 6.1 & 6.2] in the module. Here's what it looks like:

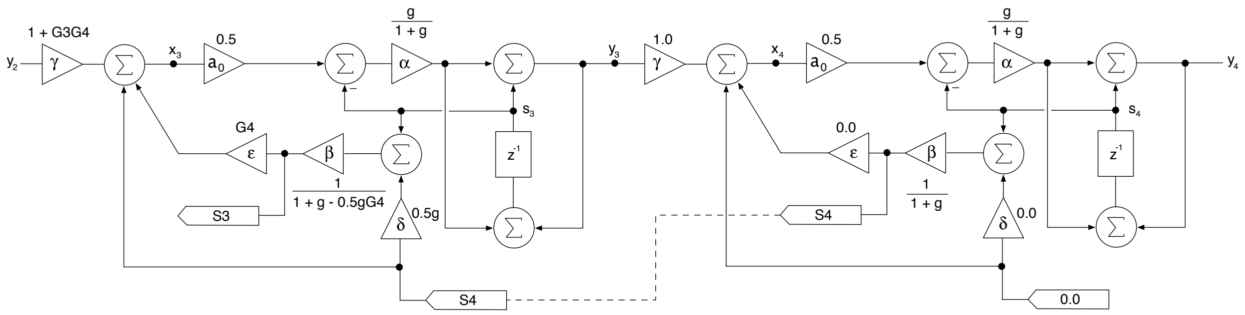


Figure 6.7: the LPF3/LPF4 combination with all feedback paths in place

The feedback output port of LPF4 is S4; there is an implied connection between output ports and input ports with the same label so the dotted line shows the connection between LPF4 back into LPF3. Similarly, S3 will be fed back into LPF2. Look at the diagram and verify the following equations by tracing the signal through the blocks. If you want to understand the block diagram synthesis it is important that you do not continue until you prove to yourself that these values are correct.

$$x_4 = y_3 \qquad S4 = \frac{s_4}{1+g}$$

$$x_3 = y_2(1+G3G4)+G4S3+S4 \qquad S3 = \frac{0.5gS4+s_3}{1+g-0.5gG4}$$

The design calls for four LPFs in series with delay-less feedback through k. We can assemble the four LPFs first by implementing equations [Eq. 6.2 & 6.3] in the same manner as above and connecting them. This yields the Diode Ladder in isolation shown in Figure 6.8. Notice how each section feeds back into the preceding one via SN (i.e. S4 feeds back into LPF3, S3 into LPF2 and so on).

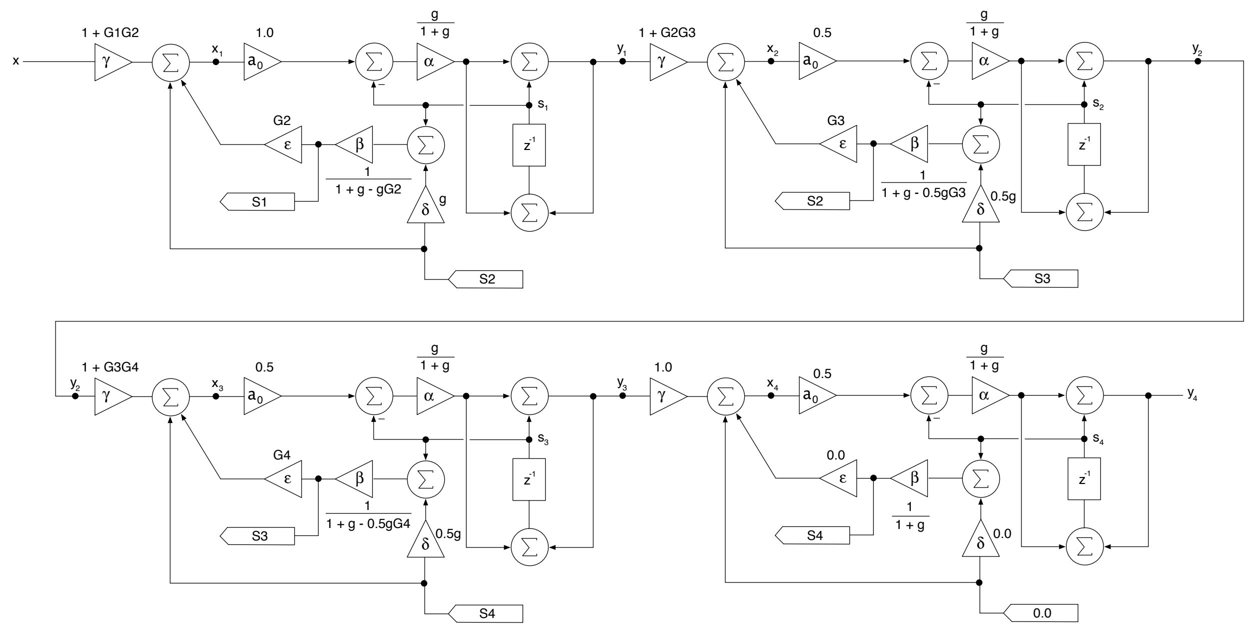


Figure 6.8: the complete isolated Diode Ladder; note that there is an implied connection between output ports and input ports with the same label

To insert this series into the feedback loop, we need to use what I'm calling the "TPT Delay-less k-Loop Equation" which says that if you have an element composed of TPT blocks in a delay-less feedback loop with feedback value k <u>and</u> you can describe the output of the element in the form

$$y = Gx + S$$

then the input u into the element (a sum or difference of the feedback path) will be

$$u = \begin{cases} \dfrac{x + kS}{1 - kG} & \text{positive feedback} \\ \dfrac{x - kS}{1 + kG} & \text{negative feedback} \end{cases}$$

This is the beauty of the TPT blocks and the reason you can solve delay-less feedback loops: the difference equation can be written as a linear equation y = ax + b.

To find the final output y_4 of the series connection, you solve equations [Eq. 6.1 - 6.4] by substituting one into the other.

$$y_4 = G4y_3 + S4 = G4(G3y_2 + S3) + S4 = G4(G3(G2y_1 + S2) + S3) + S4 \text{ ... etc}$$

Ultimately this yields the same equation as [Zavalishin p.74]

$$
\begin{aligned}
y_4 &= G4G3G2G1x + G4G3G2S1 + G4G3S2 + G4S3 + S4 \\
&= \Gamma x + \Sigma \\
\Gamma &= G4G3G2G1 \\
\Sigma &= G4G3G2S1 + G4G3S2 + G4S3 + S4 \\
&\quad and \\
u &= \frac{x - k\Sigma}{1 + k\Gamma}
\end{aligned}
$$

I changed the notation to capital Greek letters to avoid confusion with all the other G and S terms. We can implement u by observing that 1/1+kΓ is a scalar value so the input will be the difference of x and kΣ multiplied by 1/1+kΓ where Σ is the sum of S's.

The final block diagram is shown in Figure 6.9.

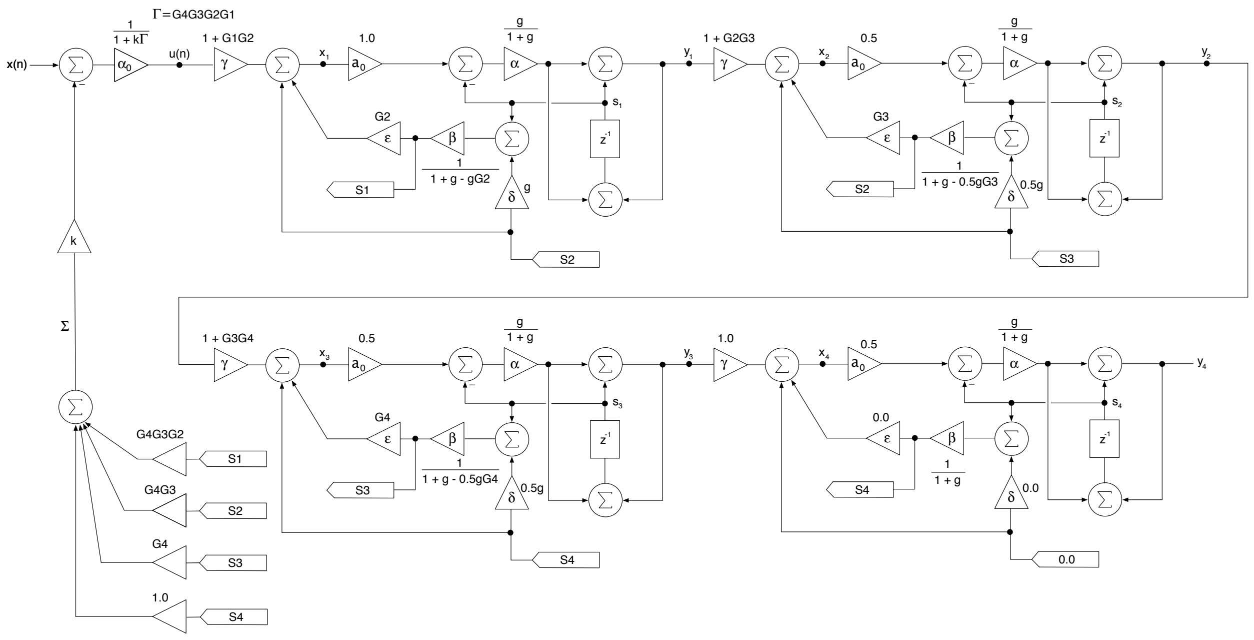
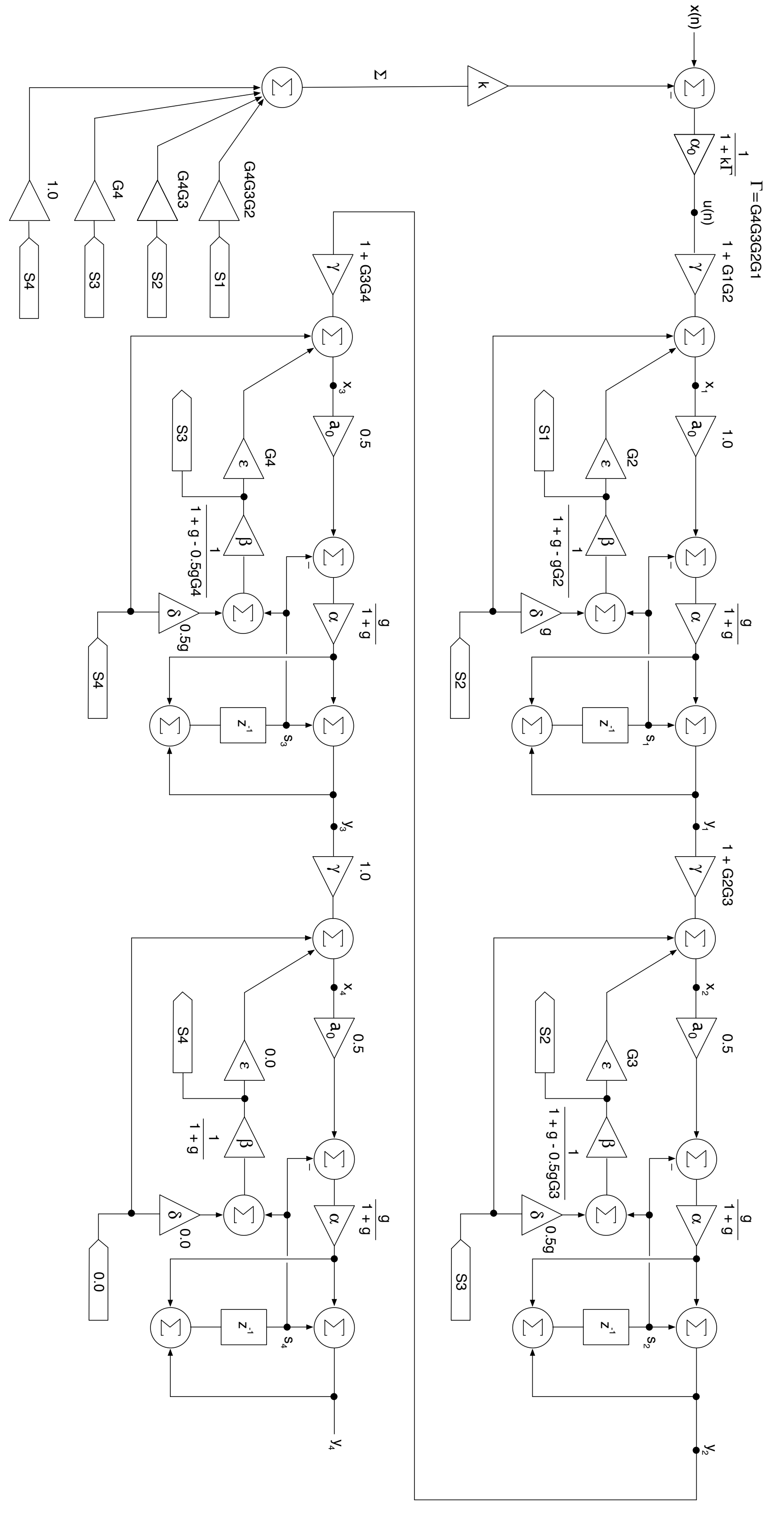


Figure 6.9: the final block diagram of the completed Diode Ladder Filter

A landscape version is provided on the next page for easier readability (print it out or rotate in Acrobat)


x(n)
Σ
k
α_0
$\frac{1}{1 + k\Gamma}$
Γ=G4G3G2G1
u(n)
γ
1 + G1G2
x_1
a_0
1.0
S1
ε
G2
β
$\frac{1}{1 + g - gG2}$
δ
g
α
$\frac{g}{1 + g}$
S2
z^{-1}
s_1
y_1
1 + G2G3
x_2
0.5
S2
G3
$\frac{1}{1 + g - 0.5gG3}$
0.5g
S3
s_2
y_2
1.0
G4
G4G3
G4G3G2
S4
S3
S2
S1
1 + G3G4
x_3
0.5
S3
G4
$\frac{1}{1 + g - 0.5gG4}$
0.5g
S4
s_3
y_3
1.0
x_4
0.5
S4
0.0
$\frac{1}{1 + g}$
0.0
0.0
s_4
y_4

Verify that the input to the series u(n) is

$$u(n) = \frac{x(n) - k\Sigma}{1 + k\Gamma}$$

NOTE: this is only one of several ways to synthesize the block diagram; there may be ways to combine coefficients or otherwise simplify the block diagram however I feel that this one is the easiest to understand since it implements Eq. 6.1 - 6.4 in *direct form* which means you can signal trace your way through the circuit and prove to yourself that each node and feedback output is correct.

Nonlinear Model

The nonlinear version of the filter is difficult to synthesize. It has Non Linear Processing (NLP) blocks in local feedback paths around each filter stage in addition to a NLP block at the input into the ladder. See [Zavalishin pp.69 - 70] for more details on the “advanced” and “cheap” versions. Implementing the either nonlinear model will be a challenge (if not impossible to process) [Zavalishin p.66, 75] and the cheap version will result in errors at high frequencies, but we can easily implement a “budget” version by ignoring the NLP blocks in the feedback paths and only applying an NLP at the input (outside the feedback loop). Figure 6.9 shows the budget NLP implementation in the Diode Ladder Filter. The results will not be sonically identical to the advanced or cheap versions, but the ease of implementation warrants investigation. As always, you are encouraged to try your own variations on the NLP type and location.

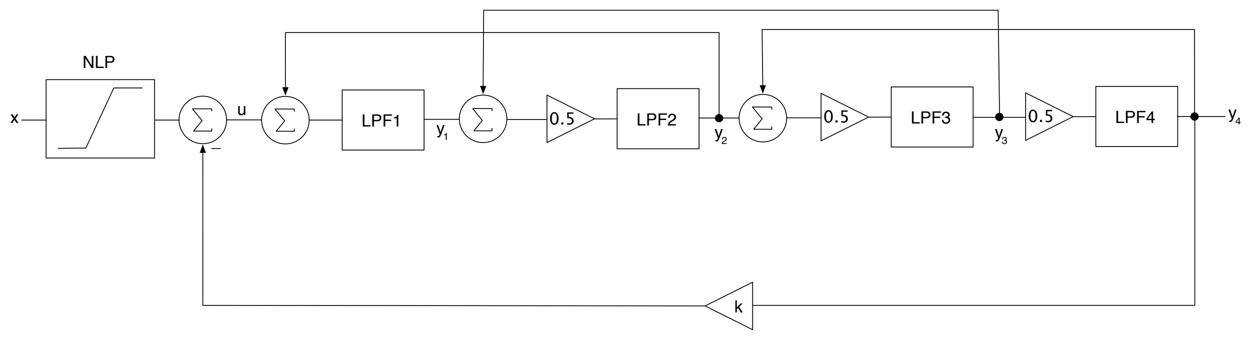


Figure 6.9: the budget NLP implementation only uses the NLP block at the input

As stated previously, tanh() is used as the nonlinear function. With the tanh() function, on the range of x = [-1..+1], tanh(x) outputs a value y that is less than [-1..+1] as shown in Figure 6.10.

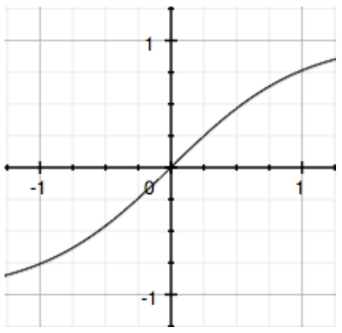


Figure 6.10: tanh(x) produces about 0.8 when x = 1

If you wish, you may normalize this so that when x = +/-1, y = +/-1 as follows:

$$y = \frac{1}{\tanh(1)} \tanh(x)$$

which produces Figure 6.113 (notice that this changes the overall shape considerably, but since this is still an approximation we can experiment with it):

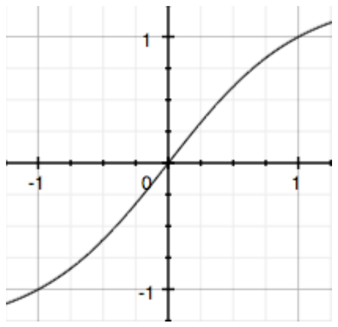


Figure 6.11: The normalized tanh() function transfer curve

For more experimentation, you can add a saturation variable *sat* to the equation which controls the steepness of the sigmoid:

$$y = \frac{1}{\tanh(sat)} \tanh((sat)x)$$

Figure 6.12 shows this new function with *sat* = 3.

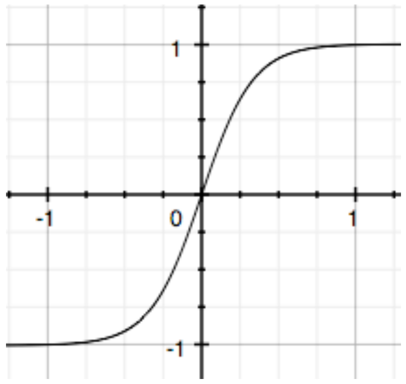


Figure 6.12: A steeper curve is obtained with *sat* > 1, here it is 3

NOTE: as with any nonlinear waveshaping function, aliasing can and will occur. This is mitigated by oversampling techniques; since oversampling is covered in several other App Notes, I am omitting it from the sample project for clarity. You are encouraged to experiment with oversampling to produce higher fidelity results.

C++ Implementation in RackAFX

The sample code is a typical RackAFX project that implements a monophonic Diode Ladder Filter. I created a class called *CVAOnePoleFilterEx* to implement the standard TPT lowpass filter but with the added coefficients to handle the feedback paths in and out of each section. The project is called *VADiodeLadderFilter*. The main C++ Plug-In object has four member objects of type CVAOnePoleFilterEx to handle the four LPFs. The object updates the filters as the UI controls change. It implements the main block diagram shown in Figure 6.9.

Refer to the theoretical amplitude response for the Diode Ladder Filter in [Zavalishin Figure 4.16] which shows the frequency response for values of k between 0 and 16 (I'm guessing this was done in MATLAB). Using the RackAFX realtime analyzer (which takes the impulse response of your filter, then applies the FFT for the frequency response) this Diode Ladder Filter produces the curves in Figure 6.13. *Due to pixel truncation the peak amplitude (k = 16) is somewhat greater than shown.*

Compare Figure 6.13 with [Zavalishin Figure 4.16] and note the agreement (both plots are log(f)). The dotted line is unity gain.

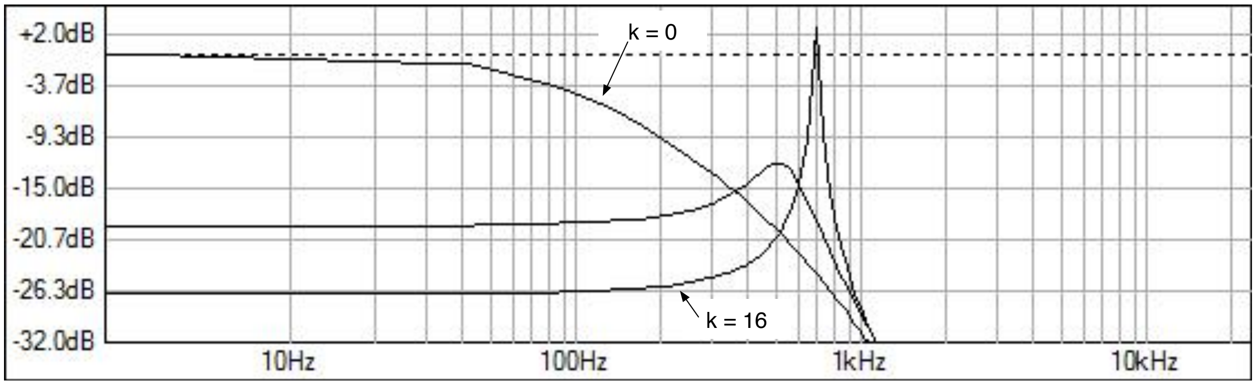


Figure 6.13: the measured frequency response of the Diode Ladder Filter with k = 0 (no resonance), k = 8 and k = 16 (most peaking) with fc = 1 kHz; NLP is OFF.

You will notice that the resonant peak is not at 1kHz; this is what we expect [Zavalishin, p 73]. At k = 17, the filter self-oscillates; the peak frequency is fc/sqrt(2) thus for fc = 1kHz, the peak will occur at 707 Hz. The plot for k = 17 is almost identical to k = 16; I am using k = 16 in the above plot to show how well this matches with the theoretical in both the overall filter gain (about -26dB with k = 16) as well as the peak frequency location. The low resolution "choppiness" at low frequencies for k = 0 is due to resolution issues with the FFT at low frequencies.

At k = 17 the filter self-oscillates. While remaining just on the razor's edge of stability, it never crosses and remains stable. However, depending on your taste, you may want to limit k to 16.999 if you find the self-oscillation irritating. Generally, this depends on the source material and the listener/application. The filter performs well across the spectrum as shown in Figure 6.14.

NOTE: With NLP engaged, the overall gain of the filter can rise above unity since the NLP block can also be a gain block. If you use NLP you may need to add a final gain stage (outside the loop) to compensate for this. Experimentation and tweaking will be necessary depending on your final decision regarding NLP type and saturation level.

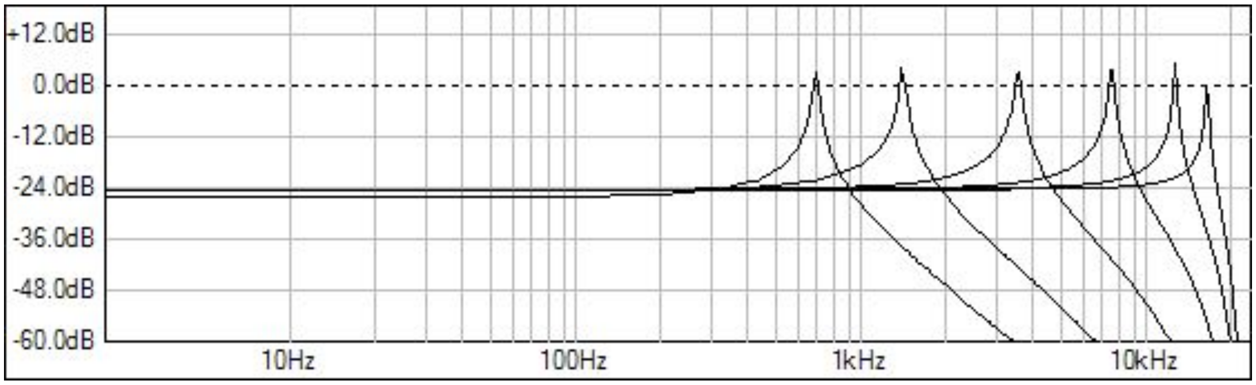


Figure 6.14: the Ladder Diode Filter with k = 16 and fc = 1kHz, 2kHz, 5kHz, 10kHz, 15kHz and 18kHz. The peak locations are off by the expected factor fc/sqrt(2)

CVAOnePoleFilterEx

.h file
- declare the coefficients & storage registers
- provide methods for getting and setting the feedback input/output

```
class CVAOnePoleFilterEx
{
public:
       CVAOnePoleFilterEx(void);
       ~CVAOnePoleFilterEx(void);

       // common variables
       double m_dSampleRate;      /* sample rate*/
       double m_dFc;              /* cutoff frequency */

       // Trapezoidal Integrator Components
       // variables
       double m_dAlpha;           // Feed Forward coeff
       double m_dBeta;            // Feed Back coeff from s + FB_IN

       // extended functionality variables
       double m_dGamma;           // Pre-Gain
       double m_dDelta;           // FB_IN Coeff
       double m_dEpsilon;         // extra factor for local FB
       double m_da0;              // filter gain
       double m_dFeedback;        // Feed Back storage register (not a delay register)

       // provide access to our feedback output
       double getFeedbackOutput();

       // provide access to set our feedback input
       void setFeedback(double fb){m_dFeedback = fb;}

       // for s_N only; not used in the Diode Ladder
       double getStorageValue(){return m_dZ1;}

       // flush buffer
       void reset(){m_dZ1 = 0;}
```

```
	// do the filter
	double doFilter(double xn);

protected:
	double m_dZ1;		// our z-1 storage location
};
```

CVADiodeLadderFilter

.h file
- declare our member LPFs
- declare our coefficients including GAMMA and the S variables
- declare methods for resetting, updating and implementing the filter

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

	double m_dGAMMA; // Gamma see App Note

	// our feedback S values (global)
	double m_dSG1;
	double m_dSG2;
	double m_dSG3;
	double m_dSG4;

	void reset()
	{
		m_LPF1.reset(); m_LPF2.reset();
		m_LPF3.reset(); m_LPF4.reset();

		m_LPF1.setFeedback(0.0); m_LPF2.setFeedback(0.0);
		m_LPF3.setFeedback(0.0); m_LPF4.setFeedback(0.0);
	}

	// recalc the coeffs
	void updateFilter();

	// do the filter
	double doFilter(double xn);
	// END OF USER CODE --------------------------------------------------- //
```

.cpp file

constructor()
- set the coefficients on the member LPFs that are constant
- clear our variables as needed

```
CVADiodeLadderFilter::CVADiodeLadderFilter()
{
	// Added by RackAFX - DO NOT REMOVE
	//
	// initUI() for GUI controls: this must be called before initializing/using any
	   GUI variables
	initUI();
	// END initUI()

	<SNIP SNIP SNIP>

	// Finish initializations here
	m_dGAMMA = 0.0;
	m_dK = 0.0;

	// our feedback S values (global)
	m_dSG1 = 0.0;
	m_dSG2 = 0.0;
	m_dSG3 = 0.0;
	m_dSG4 = 0.0;

	// Filter coeffs that are constant
	// set a0s
	m_LPF1.m_da0 = 1.0;
	m_LPF2.m_da0 = 0.5;
	m_LPF3.m_da0 = 0.5;
	m_LPF4.m_da0 = 0.5;

	// last LPF has no feedback path
	m_LPF4.m_dGamma = 1.0;
	m_LPF4.m_dDelta = 0.0;
	m_LPF4.m_dEpsilon = 0.0;
	m_LPF4.setFeedback(0.0);

}
```

prepareForPlay()
- reset the filters
- update the filters for the initial state (fc, Q)

```
bool __stdcall CVADiodeLadderFilter::prepareForPlay()
{
	// this flushes all storage registers in filters including feedback register
	reset();

	// set the initial coeffs
	updateFilter();

	return true;
}
```

updateFilters()
- calculate our GAMMA coefficient
- calculate and set all the necessary coefficients on the member LPFs

```
void CVADiodeLadderFilter::updateFilter()
{
	// calculate alphas
	double wd = 2*pi*m_dFc;
	double T  = 1/(float)m_nSampleRate;
	double wa = (2/T)*tan(wd*T/2);
	double g = wa*T/2;

	// Big G's
	double G1, G2, G3, G4;

	G4 = 0.5*g/(1.0 + g);
	G3 = 0.5*g/(1.0 + g - 0.5*g*G4);
	G2 = 0.5*g/(1.0 + g - 0.5*g*G3);
	G1 = g/(1.0 + g - g*G2);

	// our big G value GAMMA
	m_dGAMMA = G4*G3*G2*G1;

	m_dSG1 =  G4*G3*G2;
	m_dSG2 =  G4*G3;
	m_dSG3 =  G4;
	m_dSG4 =  1.0;

	// set alphas
	m_LPF1.m_dAlpha = g/(1.0 + g);
	m_LPF2.m_dAlpha = g/(1.0 + g);
	m_LPF3.m_dAlpha = g/(1.0 + g);
	m_LPF4.m_dAlpha = g/(1.0 + g);

	// set betas
	m_LPF1.m_dBeta = 1.0/(1.0 + g - g*G2);
	m_LPF2.m_dBeta = 1.0/(1.0 + g - 0.5*g*G3);
	m_LPF3.m_dBeta = 1.0/(1.0 + g - 0.5*g*G4);
	m_LPF4.m_dBeta = 1.0/(1.0 + g);

	// set gammas
	m_LPF1.m_dGamma = 1.0 + G1*G2;
	m_LPF2.m_dGamma = 1.0 + G2*G3;
	m_LPF3.m_dGamma = 1.0 + G3*G4;
	// m_LPF4.m_dGamma = 1.0; // constant - done in constructor

	// set deltas
	m_LPF1.m_dDelta = g;
	m_LPF2.m_dDelta = 0.5*g;
	m_LPF3.m_dDelta = 0.5*g;
	// m_LPF4.m_dDelta = 0.0; // constant - done in constructor

	// set epsilons
	m_LPF1.m_dEpsilon = G2;
	m_LPF2.m_dEpsilon = G3;
	m_LPF3.m_dEpsilon = G4;
	// m_LPF4.m_dEpsilon = 0.0; // constant - done in constructor
}
```

doFilter()
- implement delay-less feedback loop; first get feedback outputs
- then form our global SIGMA value
- apply budget NLP if wanted
- calculate u, the input to the first LPF
- calculate and set all the necessary coefficients on the member LPFs

```
double CVADiodeLadderFilter::doFilter(double xn)
{
//	m_LPF4.setFeedback(0.0); // constant - done in constructor
	m_LPF3.setFeedback(m_LPF4.getFeedbackOutput());
	m_LPF2.setFeedback(m_LPF3.getFeedbackOutput());
	m_LPF1.setFeedback(m_LPF2.getFeedbackOutput());

	// form input
	double SIGMA = m_dSG1*m_LPF1.getFeedbackOutput() +
	               m_dSG2*m_LPF2.getFeedbackOutput() +
	               m_dSG3*m_LPF3.getFeedbackOutput() +
	               m_dSG4*m_LPF4.getFeedbackOutput();

	// "cheap" nonlinear model; just process input
	if(m_NonLinearProcessing == ON)
	{
		// Normalized Version
		if(m_uNLPType == NORM)
			xn = (1.0/tanh(m_dSaturation))*tanh(m_dSaturation*xn);
		else
			xn = tanh(m_dSaturation*xn);
	}

	// form the input to the loop
	double un = (xn - m_dK*SIGMA)/(1 + m_dK*m_dGAMMA);

	// cascade of series filters
	return m_LPF4.doFilter(m_LPF3.doFilter(m_LPF2.doFilter(m_LPF1.doFilter(un))));
}
```

Please download the sample code and play with this filter in RackAFX. If you want a pure VST or AU version, use the Make VST or Make AU buttons and then you can analyze for yourself in Visual Studio or XCode. I have added a Forum Topic for this filter under the DSP Algorithm group at the Forum at www.willpirkle.com

References:

Civaloni, Marco & Fontana, Fredrico. 2008. *A Nonlinear Digital Model of the EMS VCS3 Voltage Controlled Filter*, Proc. of the 11th Int. Conference on Digital Audio Effects, Finland: DAFXx-08

Fontana, Fredrico. 2006. *Modeling the EMS VCS3 Voltage Controlled Filter as a Nonlinear Filter Network*, IEEE Transactions on Audio, Speech, and Language Processing, Vol. 18, No. 4, pp. 760-772

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

Välimäki, Vesa & Huovilainen, Antti. 2006. *Oscillator and Filter Algorithms for Virtual Analog Synthesis*, Computer Music Journal, 30:2, pp 19-31, Massachusetts: MIT 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