Virtual Analog (VA) Korg35 Lowpass Filter v3.5

Will Pirkle

July 18, 2013

Updated August 24, 2013

This updated App Note simplifies the sample code and project options. It also explains why the lack of analog loading in our digital model causes self-oscillation to occur at K = 2 rather than K = 3 for the original Sallen-Key version.

In v3.0 I introduce modeling the signal flow graph directly which eliminates one filter block. v3.2 streamlines the filter even more by simplifying the block diagram.

In v3.5 I add the analog and VA derivations and code for the loaded (un-buffered) version of the filter which then self oscillates at K = 3. The frequency responses are identical.

This App Note derives the Virtual Analog (VA) equations for the Korg35 lowpass filter. Huovilainen [2010] proposed both a trivial and bilinear transform discretized version of the filter however both result in a unit delay in the feedback path. As Huovilainen points out, the effect of the unit delay is a resonant peak amplitude that is not constant with cutoff frequency. The version here utilizes the VA derivation and Topology Preserving Transform (TPT) filters in Zavalishin's *The Art of VA Filter Design* in order to keep the feedback path delay-less, resulting in an essentially constant resonant peak amplitude across the spectrum. You will need to be familiar with this book to understand the derivation. Of course, you can always skip that and go right to the block diagram if you wish. This App Note only derives the lowpass version of the Korg35. Look for the highpass version in a future App Note.

Background

The Korg35 lowpass filter is found in the Korg MS-10 and early MS-20 synthesizers. It is currently incorporated in the new Korg Monotron synth. Variations are also found in other Korg products. The Monotron version differs slightly in components but is otherwise faithful to the original's functionality. The Korg35 lowpass filter is a 2nd order resonant lowpass type. Unlike its contemporaries of the time (the Moog Ladder and Diode Ladder filters) it does **not** reduce overall gain as the resonance is increased. Because of this it is often overlooked. However the skyrocketing popularity of the MS-20 (and new MS-20 Mini) as well as the Monotron make it worth investigating. Indeed there is more than meets the eye with this filter. It can not be implemented using the standard BZT -> Biquad design. The reason is that the Korg35 is capable of self-oscillation - this is what makes it attractive as a synth filter as well as for study.

Korg35 Lowpass Filter Design

The Korg35 lowpass filter needed to be voltage controlled, not just knob-controlled. In this way, it could be used in a modular synth so that its cutoff frequency could be modulated by other sources (LFOs, EGs, etc…). And, it needed to self oscillate. The brilliance of this design is that it is based on a lowpass filter topology that includes a positive feedback path. Note - the same could be said for the Moog Ladder and Diode Ladder filters too. The Korg35 is actually a voltage controlled version of the well known Sallen-Key lowpass filter. The voltage control comes in the form of two current controlled transistors operating as resistors. First, take a look at the basic Sallen-Key lowpass filter.

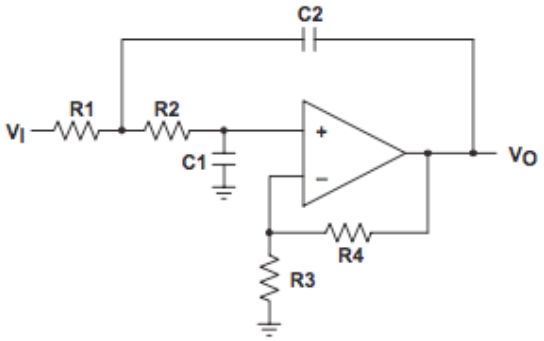

Figure 5.1: The Sallen-Key LPF

You can see the positive feedback path through C2 - it leads back to the non-inverting input. If you are one of my students who has taken my Audio Electronics classes, you know that positive feedback into an op-amp always raises red-flags. If you see positive feedback without any negative feedback compensation, you know you are looking at an oscillator. If there is negative feedback compensation, the circuit can still oscillate if the negative feedback value is not high enough. In Figure 5.1 the negative feedback is accomplished with R3 and R4. In many books, you won't see R3 or R4 - R4 will simply be a short (0 ohms) and R3 is omitted as unneeded. This is because these books are looking at a specific version of the filter. Figure 5.2 shows the Korg35 lowpass filter in the new Monotron implementation (there is currently still some debate over whether it is legal to show the original circuit, so I am using the newer one instead; in the end it won't matter as far as our emulation goes).

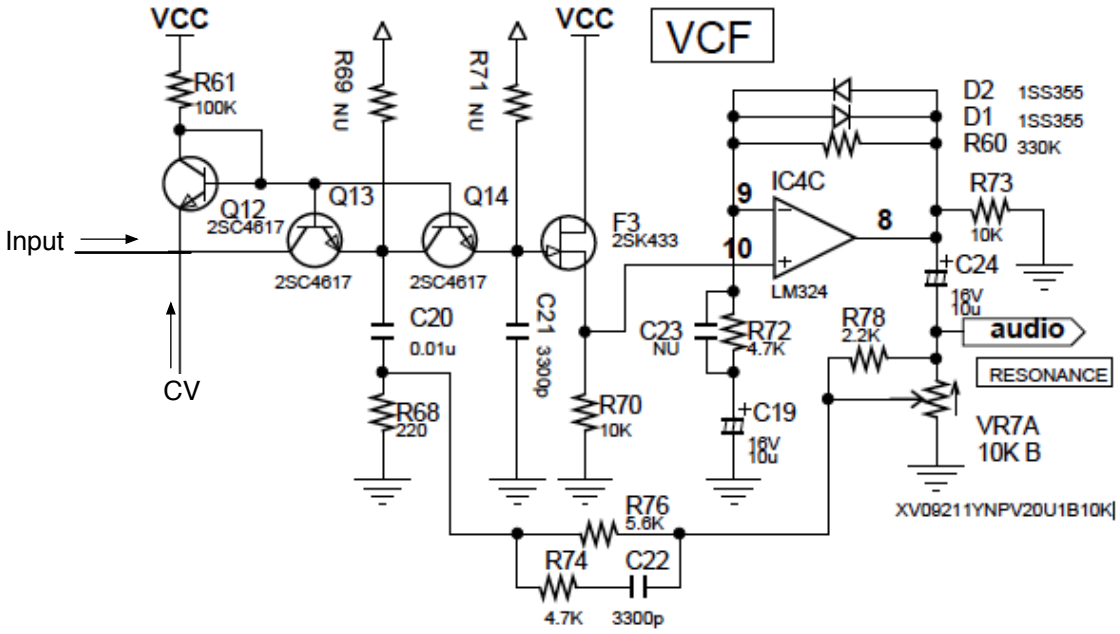

Figure 5.2: The Korg35 lowpass filter in the Monotron

In this topology, Q13 acts as R1 while Q14 is R2; likewise C20 and C21 are C1 and C2 in the schematic in Figure 5.1. Q13 and Q14 are reverse biased and behave as current controlled resistors. A reverse biased BJT has a resistance R_{CE} that is proportional to the base current, which is provided via Q12 and

R61 and the control voltage CV. C20 = 3C21 but the resistances formed by Q13 (R1) and Q14 (R2) are inverse to that i.e. R1 = R2/3 so this sets up a Sallen-Key LPF with R1 = R2 and C1 = C2 (a standard design consideration [see Texas Instruments in references]). The FET F3 is in unity gain buffer mode and is only to provide an ultra-high input impedance into the LM324 op amp, IC4C.

Diodes D1 and D2 limit the overall output level to about $1.4V_{P-P}$. The IC4C circuit is in fact a standard diode clipping circuit found in numerous distortion boxes (pedals or circuits). The reason it is there is to limit the amplitude of the signal. When the resonance is increased the output signal increases dramatically at the resonant frequency causing distortion and overload of the signal. Theoretically at self-oscillation the signal would be be swinging close to the rail voltages.

The output of IC4C is fed back into C20 via the potentiometer VR7A a 10K linear taper variety. This pot controls the feedback gain and thus the resonance of the filter. We seek to emulate this filter with a Virtual Analog type of circuit. And, much like the VA version of the Moog and Diode Ladder filters, we are emulating the topology of the circuit and not trying to implement a SPICE equivalent (i.e. we are not trying to emulate the components themselves or voltages/currents as with waveguide designs).

VA Korg35 Block Diagram

To emulate this filter in software we need to dig deeper into the Sallen-Key topology. Figure 5.3 shows the two-amplifier Sallen-Key topology (“Class 1B”). The reason for using this model is that the amplifiers K1 and K2 buffer the two filter stages eliminating the effect of R_2C_2 loading the first stage in the feed-forward path and again in the feedback path. Since our digital filters have no impedance loading, we need to use this version. The one-amplifier version produces the same frequency response but with different coefficients in the transfer function. This is derived and implemented at the end of the document.

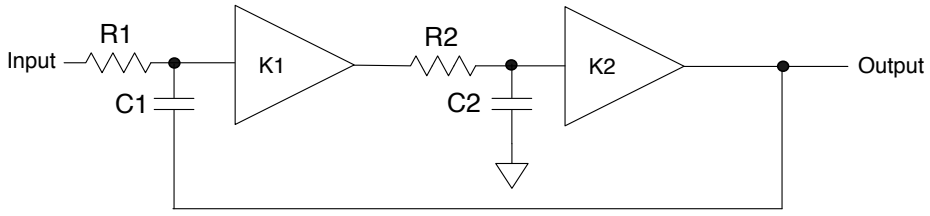

Figure 5.3: The Sallen-Key lowpass filter with two amplifiers K1 and K2

By the (analog electronics) principle of superposition Figure 5.3 can be re-drawn in block diagram form. It consists of two synchronously tuned RC LPFs (formed by R1/C1 and R2/C2) feeding a summer that adds a positive feedback signal via a 2nd order BPF as shown in Figure 5.4. Conceptually this makes sense; the BPF signal in the feedback path reinforces the Q since increases in K will amplify this component. So, K controls the Q of the filter.

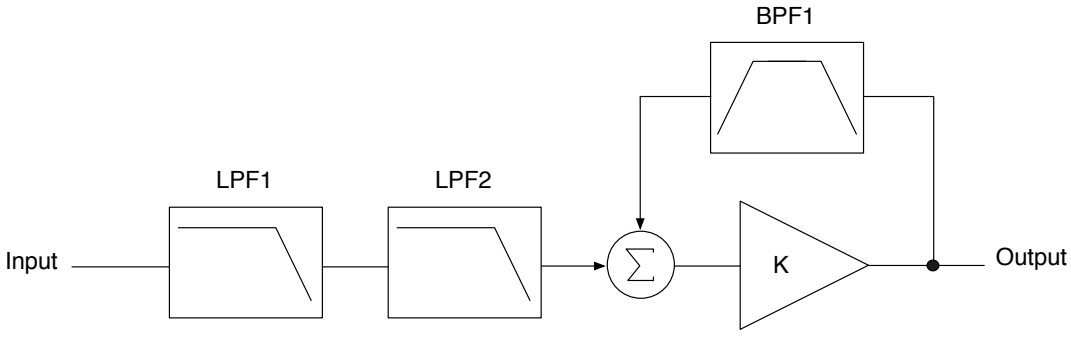

Figure 5.4: The Sallen-Key Block Diagram

LPF 1 and LPF2 are first order and since R1C1 = R2C2, they are synchronously tuned. BPF1 is a 2nd order BPF consisting of two first order sections, a first order HPF in series with a first order LPF (the standard BPF topology). In the first version of this App Note I used this model to design the Korg35 model. Another option is to model the filter by modeling its signal flow graph *exactly*. There is no difference in the transfer function, frequency response or sound of the filter in the original App Note but we can eliminate one filter block and simplify the calculations.

To derive the analog transfer function, start with the signal flow graph shown in Figure 5.5. Transmission T_{13} is the LPF formed by R_1C_1, T_{45} is the LPF formed by R_2C_2 and T_{63} is the HPF formed by look backward into the R_1C_1 circuit.

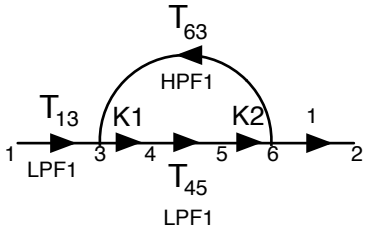

Figure 5.5: the signal flow graph for the buffered (2 amplifier) Sallen-Key filter

Start with the transmissions:

$$T_{13} = \frac{1}{1 + sR_1C_1}$$

$$T_{45} = \frac{1}{1 + sR_2C_2}$$

$$T_{63} = \frac{sR_1C_1}{1 + sR_1C_1}$$

$$K = K1K2$$

Mason's Gain Equation reveals the 2nd order LPF in the forward branch ($T_{13}T_{45}$) and the BPF in the feedback path ($T_{45}T_{65}$).

$$H(s) = \frac{KT_{13}T_{45}}{1 - KT_{45}T_{63}}$$

$$= \frac{K\dfrac{1}{(1 + sR_1C_1)}\dfrac{1}{(1 + sR_2C_2)}}{1 - K\left(\dfrac{1}{(1 + sR_2C_2)}\dfrac{(sR_1C_1)}{(1 + sR_1C_1)}\right)}$$

$$= \frac{K}{(1 + sR_1C_1)(1 + sR_2C_2) - KsR_1C_1}$$

$$= \frac{K}{s^2R_1C_1R_2C_2 + s(R_1C_1(1 - K) + R_2C_2) + 1}$$

From this we can extract the gain, cutoff frequency and Q:

$$H_0 = K = K1K2$$

$$\omega_c = \sqrt{\frac{1}{R_1C_1R_2C_2}}$$

$$Q = \frac{\sqrt{R_1C_1R_2C_2}}{(1-K1K2)R_1C_1 + R_2C_2}$$

For a normalized filter with $R_1C_1 = R_2C_2 = 1$

$$Q = \frac{1}{2-K1K2}$$

Letting K1 = 1.0, self oscillation occurs when K2 = 2.0 which is the value for our model. Also notice that since $H_0 = K$ we will need to normalize the output at 1/K to keep the overall filter gain at 1.0. In the original filter, the feedback into C2 can be set to 0.0 resulting in a critically damped LPF with Q = 0.5. The problem is that when K = 0 there is no forward gain through our model. So, we can let K assume a very small, non-zero value to emulate the zero feedback condition. Thus our emulation can let K vary from 0.01 to 2.0.

From the signal flow graph, we see that the LPF (T_{45}) is actually being *shared* between the feed forward and feed back paths. Implementing the signal flow graph directly results in Figure 5.6.

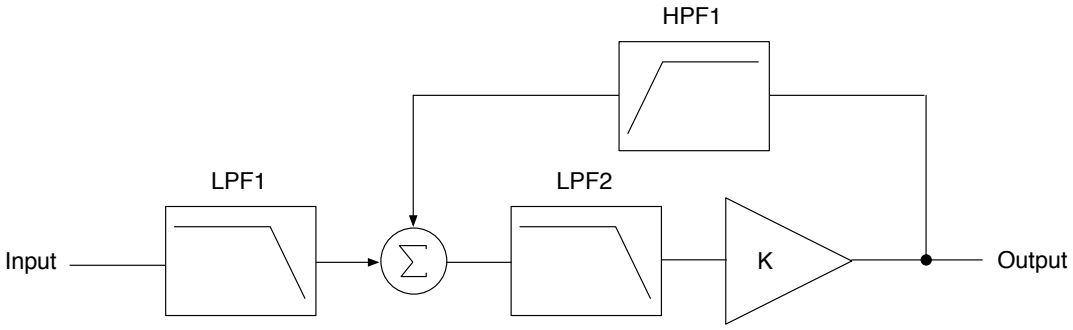

Figure 5.6: an exact model of the signal flow graph

Notice in Figure 5.6 only one amplifier block is used. We are going to let K1 (or K2) = 1.0 and adjust the other one. It makes no difference whether the amplifier block is placed before or after LPF2; the results are the same.

VA Korg35 Design Equations - Linear Model

The block diagram in Figure 5.6 has a delay-less feedback path. We need to resolve this delay-less path to make a VA emulation. We start by ignoring the NLP and auto-normalizing blocks and labeling Figure 5.7 with intermediate nodes:

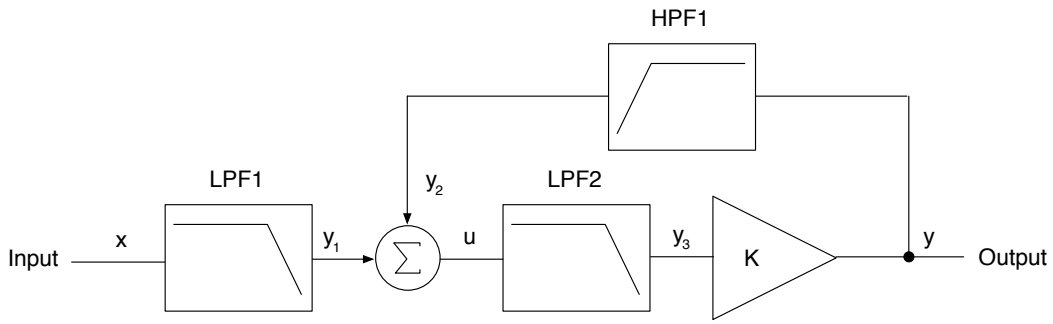

Figure 5.7: The nodes are labeled without the (n) notation for convenience

There are two strategies for calculating the equations for the filter. One option is to calculate the output directly as I have done in the original App Note. In this way, you use a single equation that creates the output *y* directly. The result shows that the entire filter may be thought of as a VA building block as you can show the output y is in the form Gx + S. However, this leads to a slightly more complex filter (in the calculation of the coefficients; the block diagram is the same). Another method is to calculate the value *u* to apply to LPF2. Both strategies produce the same result but the second version is the simpler of the two.

First, define the filter equations for each component:

$$\text{LPF1} \qquad \text{HPF1}$$

$$y_1 = Gx + S1 \qquad y_2 = y - (Gy + S2)$$

$$= y - Gy - S2$$

$$G = \frac{g}{1+g} \qquad S2 = \frac{s_2}{1+g}$$

$$S1 = \frac{s_1}{1+g}$$

$$\text{LPF2}$$

$$y_3 = Gu + S3$$

$$S3 = \frac{s_3}{1+g}$$

Now form u:

$$y = K(G3u + S3)$$
$$y_2 = y - G2y - S2$$
$$u = y_1 + y_2$$
$$= y_1 + K(G3u + S3) - G2(K(G3u + S3)) - S2$$
$$= \frac{y_1 - S2 + S3(K - KG)}{1 - KG + KG^2}$$

$$\alpha_0 = \frac{1}{1 - KG + KG^2}$$

$$S2 = \beta_2 s_2$$
$$S3 = \beta_3 s_3$$
$$\beta_2 = \frac{-1}{(1+g)}$$
$$\beta_3 = \frac{(K - KG)}{(1+g)}$$

then

$$u = \alpha_0(y_1 + \beta_2 s_2 + \beta_3 s_3)$$

VA Korg35 Block Diagram Synthesis

The block diagram is synthesized directly from the above equations. The first order TPT filters are used as building blocks. I am using my modified TPT structure that allows a feedback path to be extracted as well (same as App Note 4's Moog Ladder Filter). The feed-forward coefficient (labeled G in Zavalishin) is named alpha while the feedback coefficient is beta. This allows easy synthesis from the above equations. There are two simple variations on the block diagram; the sample code implements the one shown here. Synthesizing the other structure is left as an exercise for the reader (it's very straightforward). Figures 5.8 and 5.9 show the two building blocks of the design, the first order TPT LPF and HPF respectively.

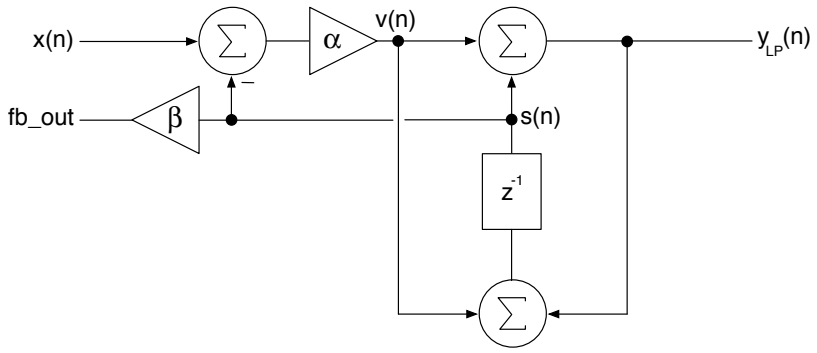

Figure 5.8: 1st order TPT LPF Block Diagram

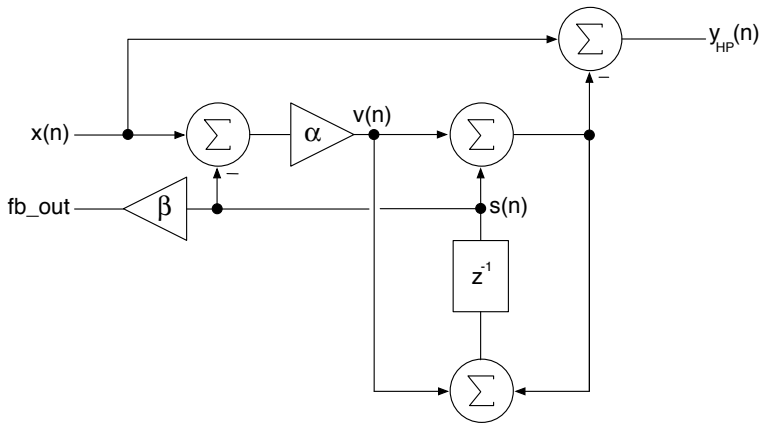

Figure 5.9: 1st Order TPT HPF Block Diagram

We can then synthesize the block diagram in Figure 5.10. Each filter block is labeled along with the nodes .

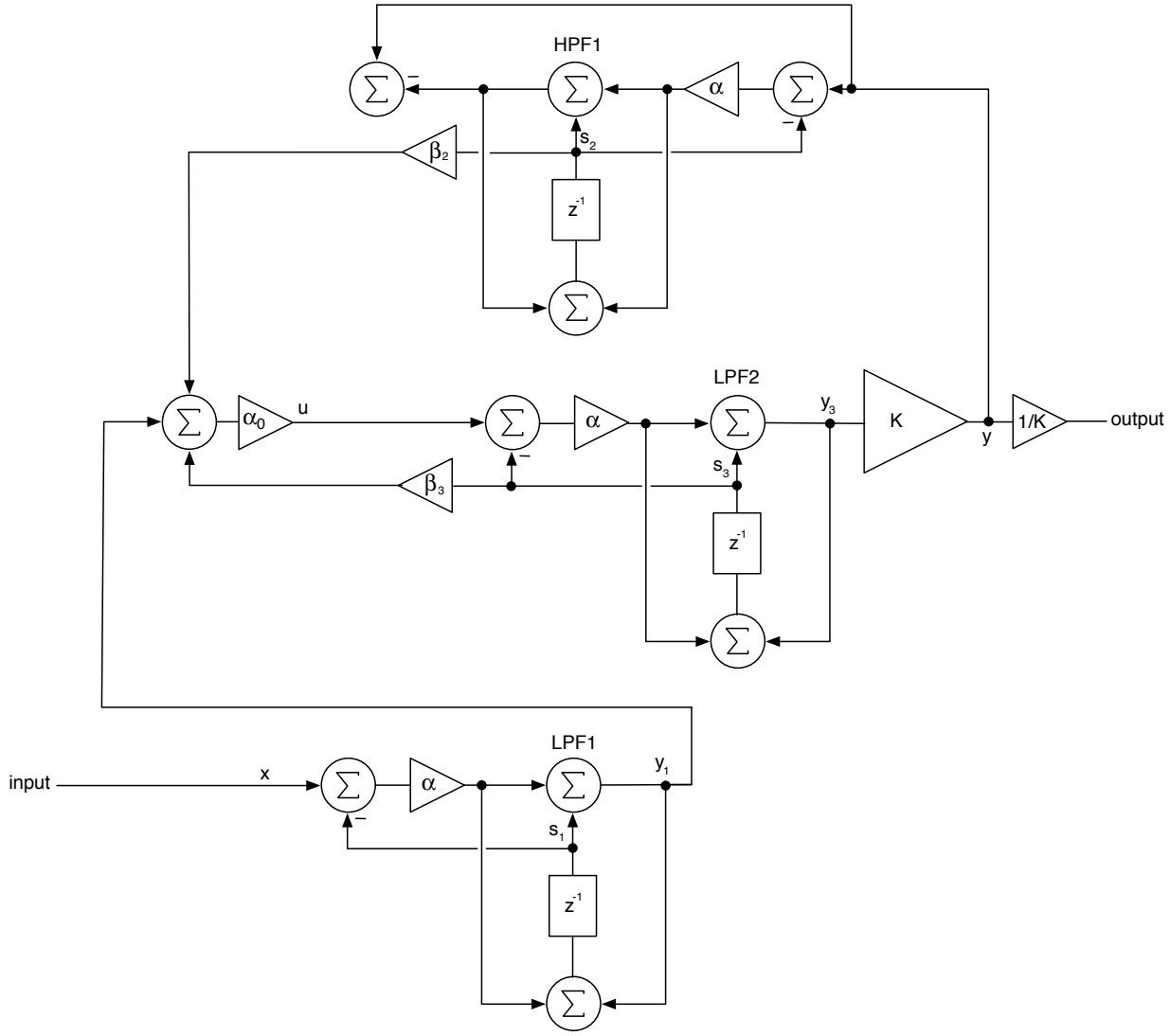

Figure 5.10: The completed VA Korg35 lowpass filter - Linear Model

Look at the node u and see that it is indeed:

$$u = \frac{y_1 - S2 + S3(K - KG)}{1 - KG + KG^2}$$

Figure 5.10 shows the completed filter auto-normalizing (1/K) coefficient included. Interestingly, you can see that the output of LPF2 doesn't seem to lead anywhere. We took care of that when we resolved the delay-less feedback loop with K wrapped up inside the equations so there is no K block directly in the diagram.

Nonlinear Model

To complete the block diagram, we need to add the diode clipping circuit as a Non Linear Processing (NLP) block. Referring back to Figure 5.2 we observe that the diode clipper circuit is inside the feedback

loop (i.e. if there was no positive feedback with the resonance pot grounded, the signal would still go through the NLP) so we can add it to Figure 5.7 to get Figures 5.11 and 5.12 - the completed model. The location of the NLP before LPF2 is important because LPF2 is a shared filter that implements half the bandpass filter in the feedback loop. The NLP must precede this bandpass filter to ensure stability.

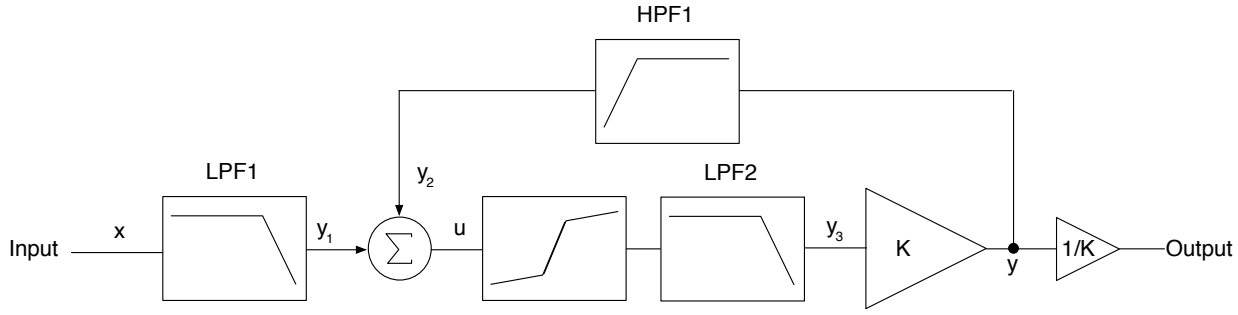

Figure 5.11: The completed Korg35 lowpass filter Block Diagram with NLP block in the feedback loop

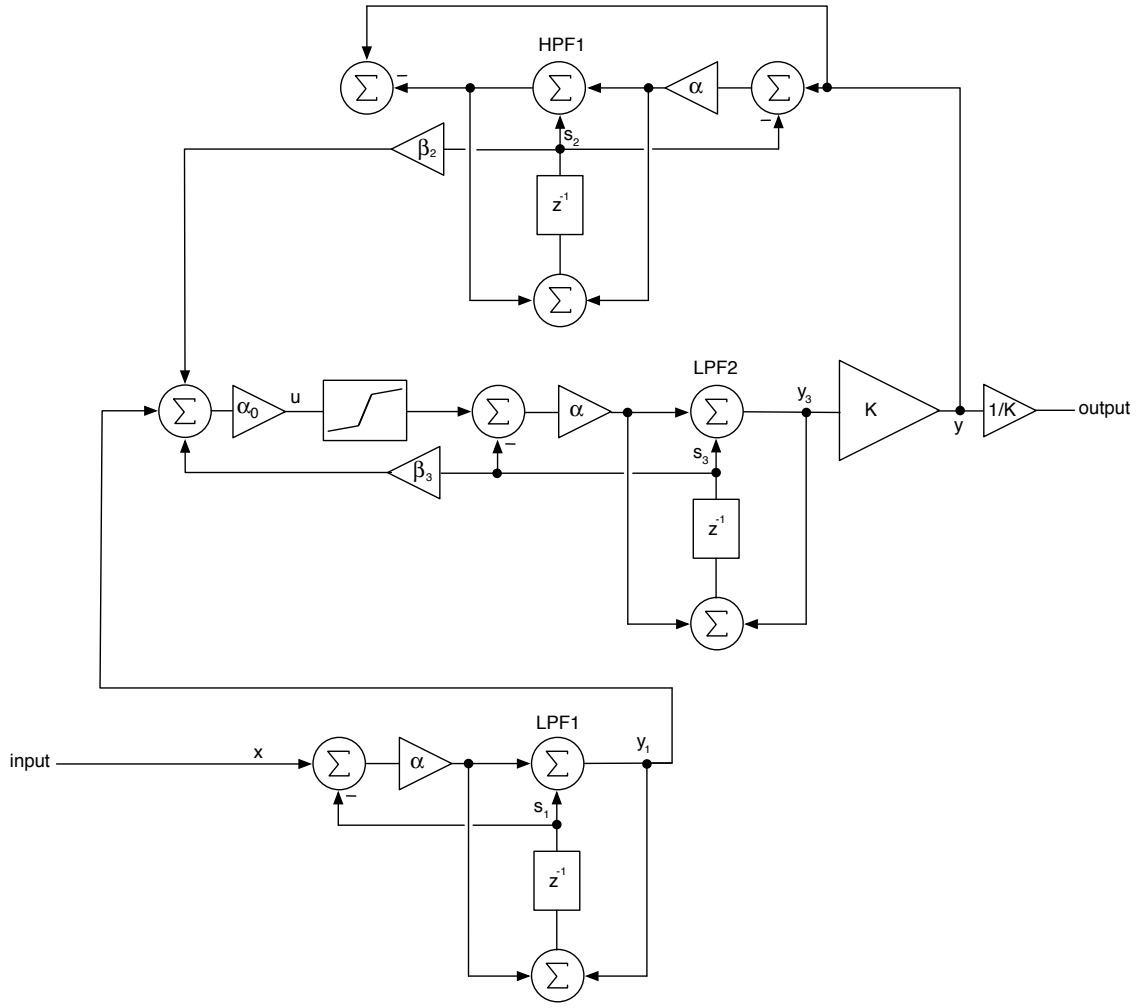

Figure 5.12: location of the NLP block in the model

When an op-amp oscillator oscillates, its output can swing close to the rail voltages. The diode clipper was added to prevent this in the analog version. In our model because we are limited to a range of -1.0 to +1.0, the massive gain as K becomes close to 2.0 will cause clipping and distortion - even worse than just a gigantic signal. The nonlinear processing helps to cure this issue. In the sample code it is turned OFF by default; when you drive the filter into oscillation you will get square waves out unless you enable the nonlinear processing.

The hyperbolic tangent function tanh() is often used as a nonlinear processing element. However, as Välimäki points out, any smooth saturation (sigmoid) function can be used as an approximation. But, an exact match to the analog version requires finessing the transfer function. See the Stinchcombe reference for a details of the diode transfer function in the clipper. You are encouraged to experiment with different NLP blocks as they will have a very big influence on the sound of the filter. See the references [Välimäki, Huovilainen]; *for simplicity only the tanh() function is considered here.*

An issue with the Linear Model is that the lack of a clipping device creates an overloaded and distorted output as the filter approaches self oscillation. We could *naively* place the NLP block back into the feedback loop in the model, as shown in Figure 5.11. This has several implications. First, it is going to alter final output equation. The output y is now tanh(K(y_2+y_4)), thus the final filter equation becomes

$$y = \tanh(K(Gu + S3))$$

This would lead to an unsolvable equation when we try to isolate y to resolve the delay-less loop. However it is easy to implement and the filter remains stable. But, the filter tuning can drift when the NLP is applied. For NLP with high saturation/gain I have found drifts of 10% (i.e. fc = 1kHz becomes fc = 1.1kHz) however for the standard tanh() saturator, the drift is low.

With the tanh() function, for the range of x = [-1..+1], tanh(x) outputs a value y that is less than [-1..+1] as shown in Figure 5.13.

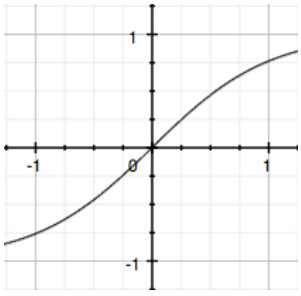

Figure 5.13: tanh(x) produces about 0.8 when x = 1

For more experimentation, you can add a saturation variable *sat* to the equation which controls the steepness of the sigmoid; going above sat = 2.0 produces a lot of aliasing.

$$y = \tanh((sat)x)$$

Figure 5.14 shows this new function with *sat* = 1.5.

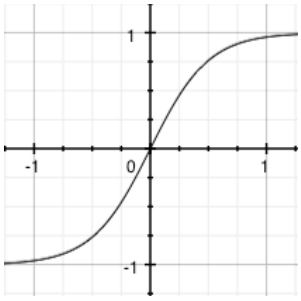

Figure 5.14: A steeper curve is obtained with *sat* > 1, here it is 1.5

Asymmetrical Resonance - an approximation
The original Korg35 lowpass filter adds resonance asymmetrically such that the lower portion of a square wave will have more rippling (see Stinchcombe). The asymmetry is not caused by asymmetrical clipping, however we can *approximate* it by making the diode clipper asymmetrical so as to amplify the lower portion slightly more than the upper portion, but only in the naive implementation. This can be done by using a bipolar tanh() waveshaper that treats each half of the waveform separately. By making the *sat* value about 1.25 times as large for the lower half, we can achieve a similar result (this was done by starting with Stinchcombe's measurement of about 1.177:1 for the ringing frequency ratio lower:upper and then adjusting; again this is an approximation). Figure 5.15 shows the response to a square wave input with greater rippling on the lower half using this approximation. However bear in mind that this approximation will alter the response of the filter. I provide the code for both cases; the symmetrical version is the default. See the code in the next section.

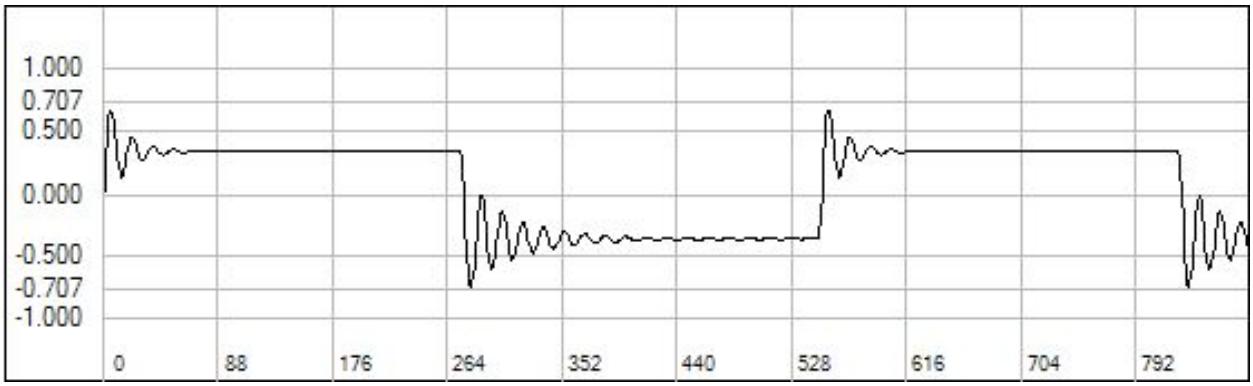

Figure 5.15: the asymmetrical response obtained with the asymmetrical clipping

NOTE: NLP engaged aliasing can occur which may require oversampling to mitigate. Another issue is that it can change the loop gain depending on the function used and the saturation level so tweaking may be necessary to get a good balance of distortion and whistling (oscillation).

Figures 5.16 and 5.17 show the frequency responses of the filter for various values of fc and Q. Try the sample code and play with the various NLP options, noting the interaction between NLP and K in the naive version.

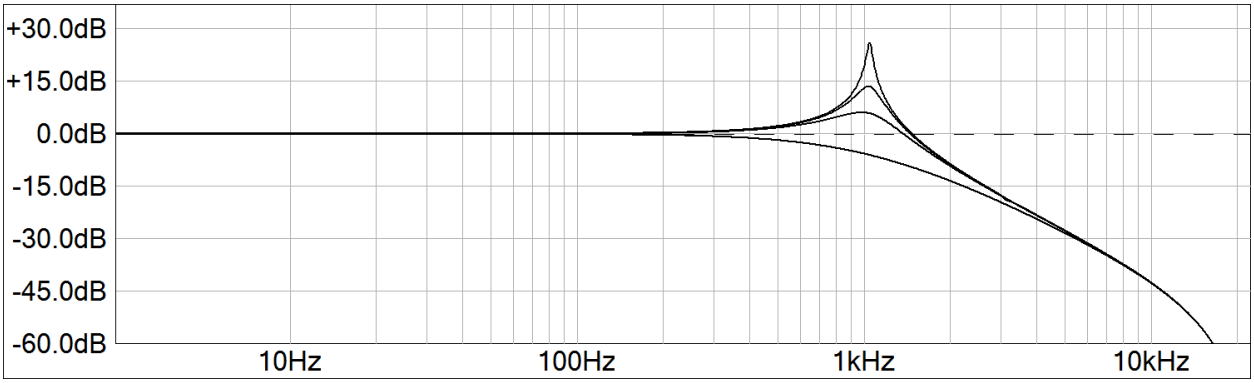

Figure 5.16: the Korg35 emulation with fc = 1kHz and K = 0.5285, 1.0, 1.5, and 1.75, NLP = OFF

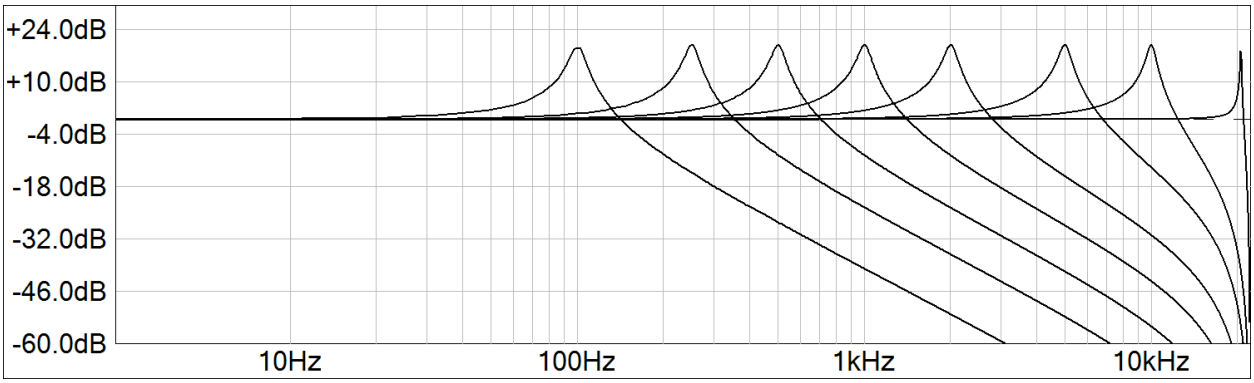

Figure 5.17: the Korg35 emulation with K = 1.9 and fc = 100Hz, 250Hz, 1kHz, 2kHz, 5kHz, 10kHz and 20,480Hz

Sample Code

The sample code is a typical RackAFX project called *KorgThreeFiveLPF.* I created an object to encapsulate the VAOnePoleFilter, similar to App Note 4. The filter self oscillates at K = 2.0 but the harshness of the huge gain and clipping is softened considerably when you engage the NLP block.

The sequence of operations in the filter code is:

1) form S35 from the feedbacks of each filter
2) calculate y = G35x + S35
3) add NLP if wanted
4) process x through LPF1 to get y_1
5) process y through HPF1 to get y_2
6) process y_1+ y_2 through LPF2
7) scale y by 1/K

In the .h File

```
// Add your code here: ------------------------------------------------ //
CVAOnePoleFilter m_LPF1;
CVAOnePoleFilter m_LPF2;
CVAOnePoleFilter m_HPF1;

// fn to update when UI changes
```

```
	void updateFilters();

	// main do function
	double doFilter(double xn);

	// variables
	double alpha0; // our u scalar

	// enum needed for child members
	enum{LPF1,HPF1}; /* one short string for each */
	// END OF USER CODE ------------------------------------------------- //
```

In the .cpp File

prepareForPlay()
- initialize the member filters
- set the m_uFilterType properly
- call the update function

```
bool __stdcall CKorgThreeFiveLPF::prepareForPlay()
{
	// Add your code here:
	// set types
	m_LPF1.m_uFilterType = LPF1;
	m_LPF2.m_uFilterType = LPF1;
	m_HPF1.m_uFilterType = HPF1;

	// flush everything
	m_LPF1.reset();
	m_LPF2.reset();
	m_HPF1.reset();

	// set initial coeff states
	updateFilters();
	return true;
}
```

updateFilters()
- called when GUI changes
- calculate and set the alpha and beta values
- calculate our own alpha0 coefficient

```
void CKorgThreeFiveLPF::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
	float G = g/(1.0 + g);

	// set alphas
```

```
        m_LPF1.m_fAlpha = G;
        m_LPF2.m_fAlpha = G;
        m_HPF1.m_fAlpha = G;

        // set betas all are in the form of  <something>/((1 + g)
        m_LPF2.m_fBeta = (m_dK - m_dK*G)/(1.0 + g);
        m_HPF1.m_fBeta = -1.0/(1.0 + g);

        // set m_dAlpha0 variable
        m_dAlpha0 = 1.0/(1.0 - m_dK*G + m_dK*G*G); ;
}
```

doFilter()
- first, get the feedback outputs of each filter N which is (beta)(s_N) and form our S35
- form y directly
- add naive NLP if enabled
- update each filter
- add budget NLP if enabled
- auto-normalize by 1/K

```
double CKorgThreeFiveLPF::doFilter(double xn)
{
        // process input through LPF1
        double y1 = m_LPF1.doFilter(xn);

        // form feedback value
        double S35 = m_HPF1.getFeedbackOutput() + m_LPF2.getFeedbackOutput();

        // calculate u
        double u = m_dAlpha0*(y1 + S35);

        // NAIVE NLP
        if(m_uNonLinearProcessing == ON)
        {
                // Regular Version
                u = tanh(m_dSaturation*u);
        }

        // feed it to LPF2
        double y = m_dK*m_LPF2.doFilter(u);

        // feed y to HPF
        m_HPF1.doFilter(y);

        // auto-normalize
        if(m_dK > 0)
                y *= 1/m_dK;

        return y;
}
```

Implementing the Single-Amplifier Sallen-Key Filter

The single-amplifier version of the Sallen-Key filter is shown in Figure 5.18. It is the exact version used in the Korg35 filters. It can also be implemented using the simple virtual analog building blocks but loading between the stages needs to be taken in account. The single amplifier version produces an identical conceptual block diagram shown in Figure 5.19. Here the BPF has been split into two first order filters. The loading is indicated by dotted lines. LPF2 loads the output of LPF1 while LPF3 loads the output of HPF1.

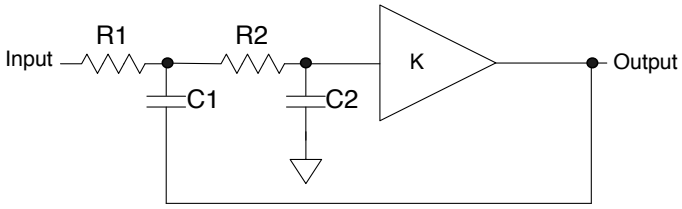

Figure 5.18: the single amplifier version of the Sallen-Key filter

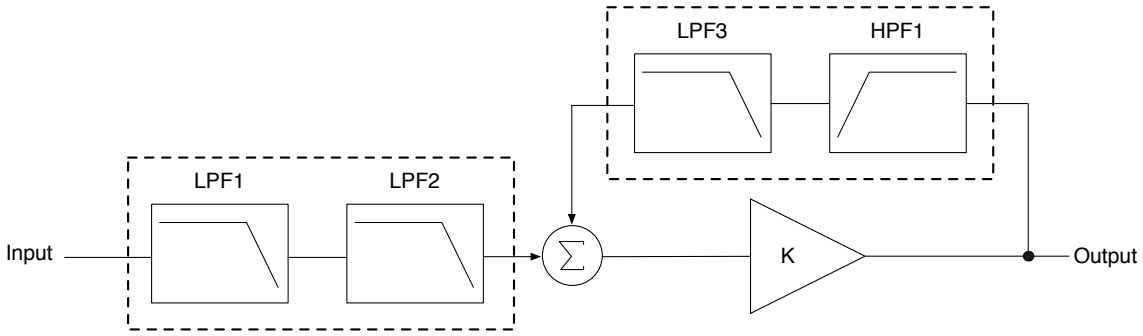

Figure 5.19: the dotted lines indicate loaded pairs in this conceptual block diagram of the filter

Loaded LPF

The analog transfer function of a pair of RC lowpass stages connected without buffering is found with standard signal flow graph techniques as:

$$H(s) = \frac{1}{s^2 R_1 C_1 R_2 C_2 + s\left(R_1 C_1 + R_1 C_2 + R_2 C_2\right) + 1}$$

The normalized transfer function is then:

$$H(s) = \frac{1}{s^2 + 3s + 1}$$

So we see the first effect of loading: the Q of this filter is 1/3 or 0.333 making it over-damped. This effectively pulls back the f_{-3dB} location for the cascade pair, although the resonant frequency is still:

$$\omega_o = \frac{1}{\sqrt{R_1 C_1 R_2 C_2}}$$

The roots of the normalized transfer function's denominator are:

$$s_1 = 0.5\left(-3 - \sqrt{5}\right)$$
$$s_2 = 0.5\left(\sqrt{5} - 3\right)$$
$$Let$$
$$P_1 = -s_1$$
$$P_2 = -s_2$$

The roots are reciprocals of one another. The transfer function can then be split back into two first order sections as:

$$H(s) = \frac{1}{\left(s + P_1\right)\left(s + P_2\right)} = \left(\frac{P_1}{s/P_1 + 1}\right)\left(\frac{P_2}{s/P_2 + 1}\right)$$

This represents a cascade of two 1st order LPFs with different cutoff frequencies, one higher than the original cutoff and one lower than it. The original cutoff frequency is scaled by the roots. So:

$$\omega_L = P_1 \omega_o$$
$$\omega_H = P_2 \omega_o$$

The passband gain of the filter is 1.0 because it is a lowpass type and $P_1P_2 = 1.0$.

Virtual Analog Model

The VA model is simply a cascade of two LPFs with different cutoff frequencies. The VA Equations are the same for each filter as:

$$y_{LPF1} = G_{LPF1} x + S_{LPF1}$$
$$y_{LPF2} = G_{LPF2} y_{LPF1} + S_{LPF2}$$

Loaded BPF

The analog transfer function of a pair of RC filter stages connected in a BPF configuration (HPF -> LPF) without buffering is:

$$H(s) = \frac{sR_1C_1}{s^2 R_1 C_1 R_2 C_2 + s\left(R_1C_1 + R_1C_2 + R_2C_2\right) + 1}$$

As expected, the two transfer functions share the same denominator.

The normalized transfer function is then:

$$H(s) = \frac{s}{s^2 + 3s + 1}$$

Since the denominator is the same as the LPF case, the roots are the same and we can similarly split the transfer function as:

$$H(s) = \frac{s}{(s + P_1)(s + P_2)} = \left(\frac{sP_1}{s/P_1 + 1}\right)\left(\frac{P_2}{s/P_2 + 1}\right)$$

This represents a cascade of a 1st order HPF feeding a 1st order LPF with the same two different cutoff frequencies, one higher than the original cutoff and one lower than it.

$$\omega_L = \omega_{HPF} = P_1\omega_o$$
$$\omega_H = \omega_{LPF} = P_2\omega_o$$

However, the passband gain in a passive unbuffered bandpass filter is comprised of two parts. There is gain loss due to the band edges crossing over each other. The closer their frequencies, the more loss. This is the same as the loss for a buffered BPF as:

$$A_v(pass) = \frac{R_1C_1}{R_1C_1 + R_2C_2}$$

There is also a passband loss from the loading of the second stage on the first. Together, they make up the total passband loss as:

$$A_v(total) = \frac{R_1C_1}{R_1C_1 + R_1C_2 + R_2C_2}$$

This loss value is 1/3 for a normalized filter. In order to model a lossy cascade of HPF and BPF, a loss factor needs to be calculated which also takes into account the loss contribution from the band edges crossing. The loss correction is simply:

$$L = \frac{A_v(total)}{A_v(pass)}$$

Virtual Analog Model

The VA model is slightly more complicated because of the loss correction. It is still a cascade of two filters with the loss correction factor L inserted after the HPF as shown in Figure 5.20. Inserting the loss factor here also simulates the loading of the second section.

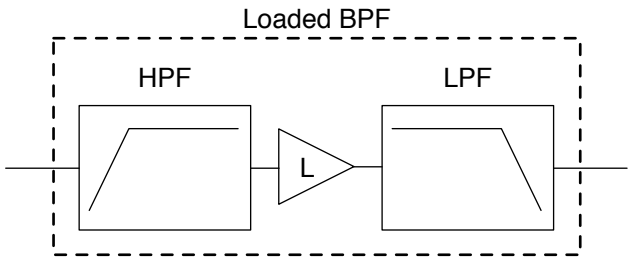

Figure 5.20: model of the loaded or lossy BPF.

The VA Equations are different as the loss has to be taken into account:

$$y_{HPF} = x - G_{HPF}x - S_{HPF}$$
$$y_{LPF} = G_{LPF}L(y_{HPF}) + S_{LPF}$$

Implementation of the Signal Flow Graph

To derive the analog transfer function, start with the signal flow graph shown in Figure 5.21. Transmission T_{13} is the LPF formed by R_1C_1, T_{45} is the LPF formed by R_2C_2 and T_{63} is the HPF formed by look backward into the R_1C_1 circuit.

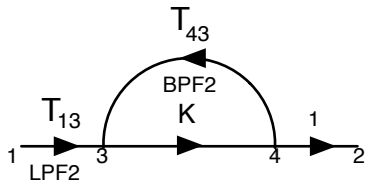

Figure 5.21: the signal flow graph for the single amplifier Sallen-Key filter

Start with the transmissions:

$$T_{13} = \frac{1}{s^2R_1C_1R_2C_2 + s(R_1C_1 + R_1C_2 + R_2C_2) + 1}$$
$$T_{43} = \frac{sR_1C_1}{s^2R_1C_1R_2C_2 + s(R_1C_1 + R_1C_2 + R_2C_2) + 1}$$

Mason's Gain Equation reveals the 2nd order LPF in the forward branch (T_{13}) and the BPF in the feedback path (T_{43}).

$$H(s) = \frac{KT_{13}}{1 - KT_{43}}$$

$$= \frac{\dfrac{1}{s^2 R_1 C_1 R_2 C_2 + s\left(R_1 C_1 + R_1 C_2 + R_2 C_2\right) + 1}}{1 - K\left(\dfrac{R_1 C_1}{s^2 R_1 C_1 R_2 C_2 + s\left(R_1 C_1 + R_1 C_2 + R_2 C_2\right) + 1}\right)}$$

$$= \frac{K}{s^2 R_1 C_1 R_2 C_2 + s\left(R_1 C_1 (1 - K) + R_1 C_2 + R_2 C_2\right) + 1}$$

From this we can extract the gain, cutoff frequency and Q:

$$H_0 = K$$

$$\omega_c = \sqrt{\frac{1}{R_1 C_1 R_2 C_2}}$$

$$Q = \frac{\sqrt{R_1 C_1 R_2 C_2}}{(1 - K) R_1 C_1 + R_1 C_2 + R_2 C_2}$$

For a normalized filter with $R_1C_1 = R_2C_2 = 1$

$$Q = \frac{1}{3 - K}$$

Thus self oscillation occurs when K=3. This is the fundamental difference between the two designs. Looking at the circuit itself, you can see that the second LPF (R_2 and C_2) is shared - in the forward path it is shared with the first LPF to create the second order transfer function T_{13}. In the feedback path, it is shared with the HPF to make the BPF. Therefore, we can create the Virtual Analog version in the same manner as before using the same block diagram shown in Figure 5.22.

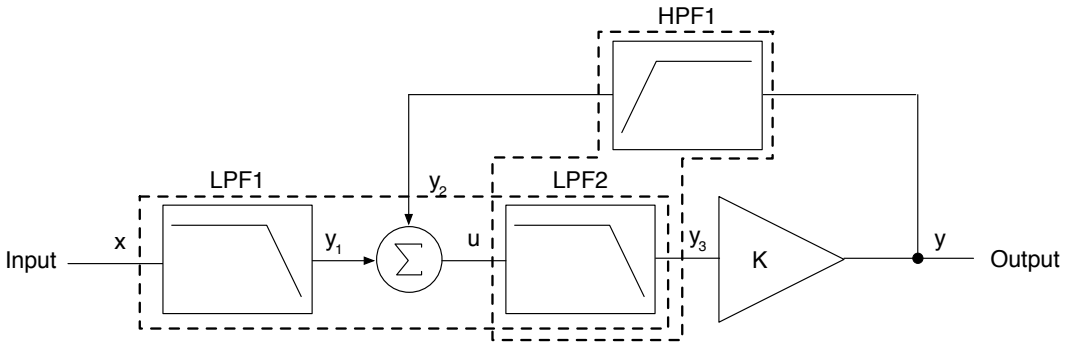

Figure 5.22: the same block diagram is implemented using the loaded filters instead

To find the VA equations: define the filter equations for each component. LPF1and HPF1 have the lower cutoff frequency while LPF2 has the higher one. Notice that the BPF loss factor has been absorbed into HPF1.

LPF1

$$y_1 = Gx + S1$$

HPF1

$$y_2 = L(y - (G3y + S2))$$
$$= Ly - LG3y - LS2$$

$$G1 = \frac{g_L}{1 + g_L}$$

$$G2 = \frac{g_L}{1 + g_L}$$

$$S1 = \frac{s_1}{1 + g_L}$$

$$S2 = \frac{s_2}{1 + g_L}$$

LPF2

$$y_3 = G3u + S3$$

$$G3 = \frac{g_H}{1 + g_H}$$

$$S3 = \frac{s_3}{1 + g_H}$$

Now form u:

$$y = K(G3u + S3)$$
$$y_2 = Ly - LG2y - LS2$$
$$u = y_1 + y_2$$
$$= y_1 + L(K(G3u + S3)) - LG2(K(G3u + S3)) - LS2$$
$$= \frac{y_1 - LS2 + S3(LK - LKG)}{1 - LKG3 + LKG2G3}$$

$$\alpha_0 = \frac{1}{1 - LKG3 + LKG2G3}$$

$$S2 = \beta_2 s_2$$
$$S3 = \beta_3 s_3$$
$$\beta_2 = \frac{-L}{(1 + g_L)}$$
$$\beta_3 = \frac{(LK - LKG2)}{(1 + g_H)}$$

then

$$u = \alpha_0(y_1 + \beta_2 s_2 + \beta_3 s_3)$$

VA Block Diagram Synthesis

The block diagram is synthesized directly from the above equations. It is identical to the block diagram in Figure 5.10. The difference is that the filters model the loaded or lossy versions of the components. The frequency response is identical to the first design; the only difference is the point of self-oscillation at K=3 rather than K=2. It also does not affect the symmetry of the resonance. Since the output of HPF leads nowhere in the diagram, we don't have to implement the loss L after the filter (notice the L value has percolated through the equations for u). *However, if you are going to use the lossy BPF in another project where the output of the HPF is used as the input to another section, you need to insert it between the HPF and LPF. In any event, it still must be taken into account when writing loop equations.*

For the nonlinear version you also need to place the nonlinear processing before the LPF in the loop as in the first design.

Sample Code

The sample code is a typical RackAFX project called *KorgThreeFiveFilter.* It uses the same CVAOnePoleFilter building blocks as the first design.

In the .h File (nothing has changed)

```
	// Add your code here: ------------------------------------------------ //
	CVAOnePoleFilter m_LPF1;
	CVAOnePoleFilter m_LPF2;
	CVAOnePoleFilter m_HPF1;

	// fn to update when UI changes
	void updateFilters();

	// main do function
	double doFilter(double xn);

	// variables
	double alpha0; // our u scalar

	// enum needed for child members
	enum{LPF1,HPF1}; /* one short string for each */
	// END OF USER CODE ------------------------------------------------- //
```

In the .cpp File the only changes are to updateFilters() and doFilters().

updateFilters()
- called when GUI changes
- calculate and set the alpha and beta values
- calculate our own alpha0 coefficient

```
void CKorgThreeFiveLPF::updateFilters()
{
	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;

	// roots when R1C1 = R2C2 - these can be pre-calculated in the constructor!
	double highMult = -0.5*(-3.0 - pow((double)5.0, (double)0.5));
	double lowMult =  -0.5*(pow((double)5.0, (double)0.5) - 3.0);
```

```
	double gLowEdge  = lowMult*wa*T/2;
	double gHighEdge  = highMult*wa*T/2;

	double GLow = gLowEdge/(1.0 + gLowEdge);
	double GHigh = gHighEdge/(1.0 + gHighEdge);

	// simulates lossy RC-CR BPF
	double L = (1.0/3.0)*(highMult + lowMult)/highMult;

	m_LPF1.m_fAlpha = GLow;
	m_LPF2.m_fAlpha = GHigh;
	m_HPF1.m_fAlpha = GLow;

	m_HPF1.m_fBeta = -L/(1.0 + gLowEdge);
	m_LPF2.m_fBeta = (L*m_dK - L*m_dK*GLow)/(1.0 + gHighEdge);

	// set alpha0 variable
	m_fAlpha0 = 1.0/(1.0 - L*m_dK*GHigh + L*m_dK*GHigh*GLow);
}
```

doFilter():

```
double CKorgThreeFiveLPF::doFilter(double xn)
{
	// first lpf
	double y2 = m_LPF1.doFilter(xn);

	// make u
	double u = m_fAlpha0*(y2 + m_HPF1.getFeedbackOutput() +
				m_LPF2.getFeedbackOutput());

	// NAIVE NLP
	if(m_uNonLinearProcessing == ON)
		// Regular Version
		u = tanh(m_dSaturation*u);

	// process u to get y
	double y = m_dK*m_LPF2.doFilter(u);

	// update feedback filter
	m_HPF1.doFilter(y);

	return y;
}
```

Considerations and Future Work

Fast tanh() approximation

In the sample code I simply call the tanh() method available via math.h however in a synth plug-in you might want to modulate the NLP section. In this case, you will want to replace the tanh() function call with a fast approximation. A google search will yield many variations you can try.

Exponential Control:

The original Korg35 lowpass filter has exponential control over the cutoff frequency fc. This is because the resistances of Q12 and Q13 vary exponentially to the base current. This is also musically useful and

generally not a bad thing. You might want to make your slider react the same way (in RackAFX this is easy by making the slider exponential in the setup).

Asymmetrical Resonance
The asymmetry in the resonance is due to other interactions within the filter and not the diode clipper, though we are able to obtain a similar response using asymmetrical clipping. Modeling the asymmetry more accurately would be an area of future work as well as further investigations into the best NLP function to match the original diode clipper transfer function. Other improvements include modeling aging transistors Q13 and Q14 and leaky capacitors C20 and C21.

Revision History:

1.1: Initial Release, *July 18, 2013*
1.2: split NLP into Naive and Budget versions
1.3: added check for K = 0 to code to avoid div by zero condition
1.4: aded reference to Huovilainen
1.5: changed minimum Q to 0.5 rather than 0.707
1.6: corrected block diagrams to remove redundant K in loop; all else unchanged (equations, code)
2.0: simplified project, removed some NLP options, derivation of K = 2 for self oscillation
2.1: added full derivation of analog transfer function of our model
3.0: changed model to directly implement signal flow graph
3.1: added alternate implementation based on u
3.3: streamlined code
3.5: added the single amp version and code

References:

Huovilainen, Antti. 2010. *Design of a scalable polyphony MIDI-synthesizer*. MS Thesis, Aalto University School of Science and Technology, Espoo Finland. http://lib.tkk.fi/Dipl/2010/urn100219.pdf

Korg Inc. 2013. *Montron Schematic for Public Release*, Tokyo: Korg Inc. http://www.korg-datastorage.jp/Manual/monotron_sch.pdf

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

Stinchcombe, Tim. 2006. *A Study of the Korg MS10 and MS20 Filters*, http://www.timstinchcombe.co.uk/synth/MS20_study.pdf

Texas Instruments. 1999. *Analysis of the Sallen-Key Architecture.* http://lorien.die.upm.es/~macias/docencia/datasheets/varios/sallenkey-ti.pdf

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