Virtual Analog (VA) Korg35 Highpass Filter v2.0 Simplified

Will Pirkle

August 15, 2013

While experimenting with some analog simulations in CircuitMaker, I realized that the Korg35 HPF model could be greatly simplified without sacrificing anything. The original design in App Note 7 follows the model of the filter using the analog transfer function for this modified Sallen-Key filter, it used Stinchcombe's analog transfer function directly to fashion the model; this simplified version produces the same transfer function but with three fewer filter components. The equations are simpler to derive and less processing is needed.

I also made changes and simplifications to the nonlinear processing, eliminating the "budget" processing from the original document in favor of the better sounding "naive" version.

This App Note derives the Virtual Analog (VA) equations for the Korg35 Highpass 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. Additionally, the HPF version does not match the original analog filter's rolloff of 6dB/octave and can also become unstable. 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 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 highpass version of the Korg35. Please refer to App Note 5 (VA Korg35 Lowpass Filter) first as this document builds upon it as well as refers to it in multiple locations.

Background

The Korg35 highpass filter is found in the Korg MS-10 and early MS-20 synthesizers. This is an interesting filter for several reasons; the most important is that while it is a second order resonant filter, it has a rolloff slope of a first order filter (6dB/oct instead of 12dB/oct)! Like the lowpass version, it will also self oscillate. It can not be implemented using the standard BZT -> Biquad topology.

Korg35 Highpass Filter Design

The Korg35 highpass filter is actually a voltage controlled version of the well known Sallen-Key *lowpass* filter but with an old analog filtering trick employed to make it a highpass type. Please refer to App Note 5 for the background on the Korg35 lowpass filter. Take a look at the basic Sallen-Key *lowpass* filter.

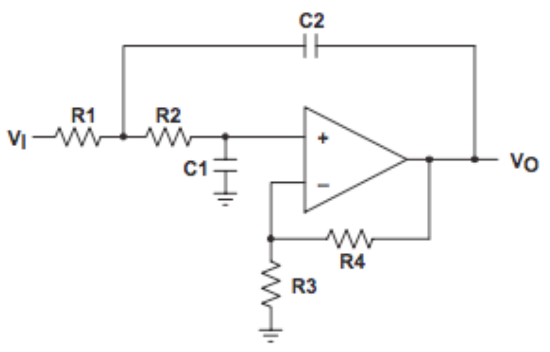

Figure 7.1: The Sallen-Key LPF

In order to convert the lowpass filter into a highpass filter, engineers typically use the RC:CR transformation in which the Rs and Cs are swapped. This produces the Sallen-Key highpass filter shown in Figure 7.2.

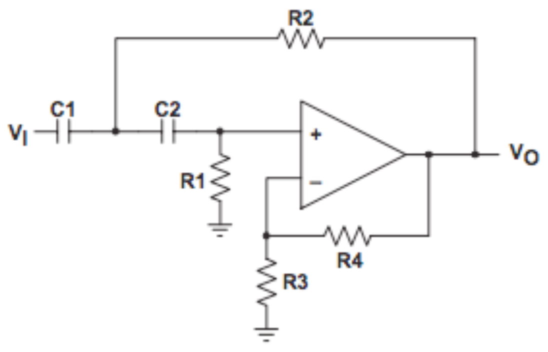

Figure 7.2: the Sallen-Key HPF

Using identical R and C values produces a second order HPF with the same cutoff frequency f_c as the LPF and the standard second order rolloff slope of 12dB/octave; this is called a *Complementary Filter* in analog filter jargon. However, there is another way to make a complementary filter to turn a LPF into a HPF (or vice versa). This trick dates back to the origins of analog filter design. It is called "grounding the input and driving the ground" and is implemented exactly like that - you connect the filter input to ground, then drive the ground with the input signal [Lindquist 1977].

However, with the Sallen-Key topology, this technique results in a hybrid highpass version; it has 6dB/octave rolloff instead of 12dB/octave. This is what the engineers at Korg did, presumably to avoid designing a second voltage control module (the Korg35 voltage control section was actually on a separate "chip" consisting of discrete components potted in epoxy and mounted on a tiny PCB with legs like a chip). This allowed them to reuse the module.

VA Korg35 HPF Block Diagram

To emulate this filter we start with the circuit that results from grounding the input and driving the ground of the Korg35 LPF. In this diagram, there are two amplifiers, K1 and K2 so it is a buffered filter. We are using this version of the Sallen-Key filter (aka "Class 1B") because our digital version mimics buffered filter blocks. There is no impedance loading between our digital filter circuits.

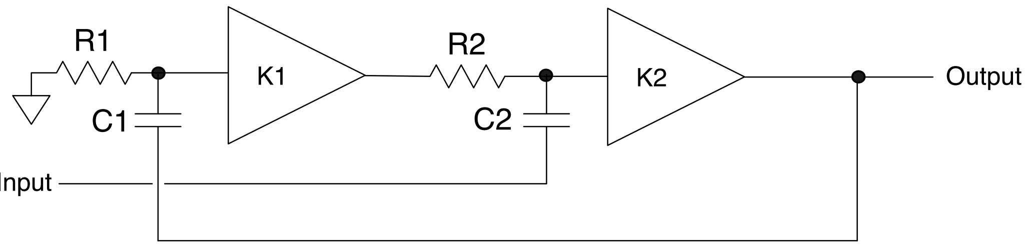

Figure 7.3: The buffered form of the Sallen Key LPF converted to the HPF

Stinchcombe [2006] derived the analog transfer function resulting from grounding the input and driving the ground of the unbuffered version (“Class 1B”) using nodal analysis. It produced a parallel BPF on the input section as shown in Figure 7.4. The effect of the second parallel BPF is to offset the rolloff below the cutoff frequency so that it shifts to 6dB/octave. In effect, it forms a 1st order HPF when combined with the 2nd order HPF input.

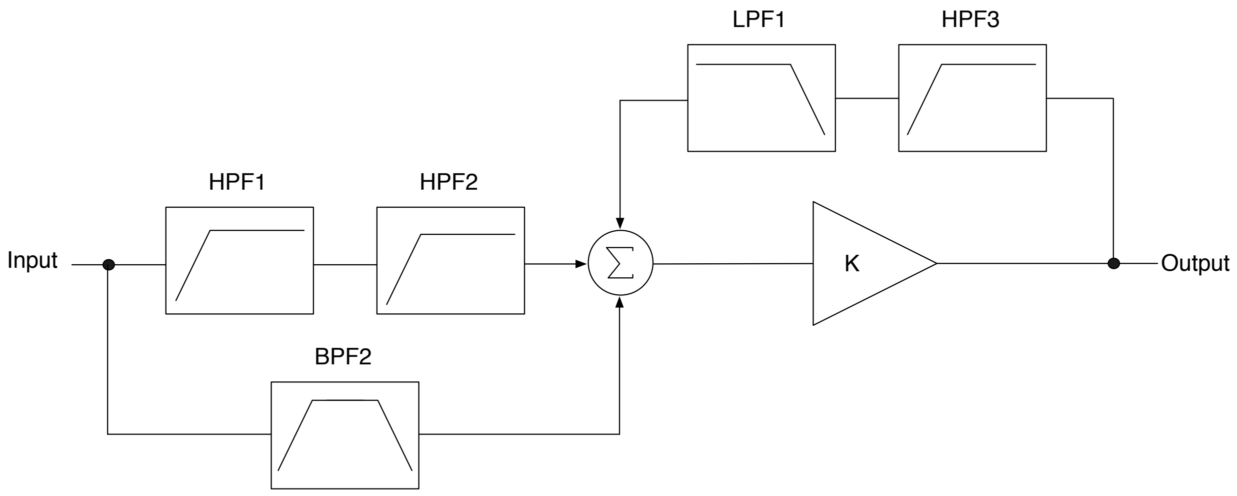

Figure 7.4: The Korg35 HPF topology.

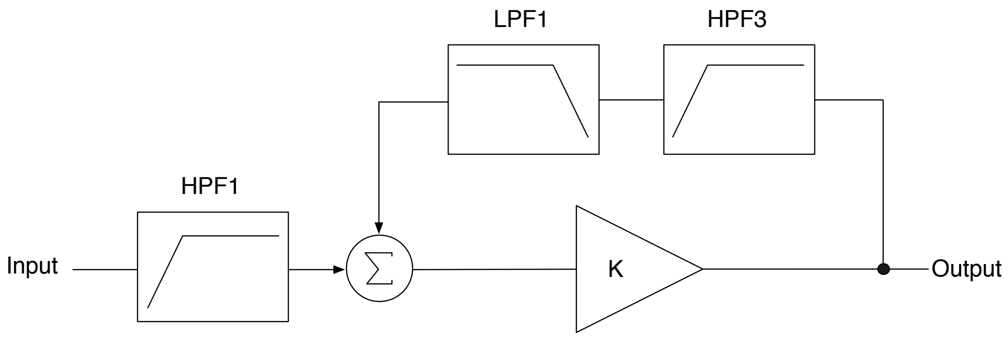

Figure 7.5: The Simplified Korg35 HPF topology

To complete the block diagram, we need to add the diode clipping circuit as a Non Linear Processing (NLP) block as discussed in App Note 5. I have also included the auto-normalizing block at the output (1/K) to keep the DC gain constant (see the Sallen-Key equations below). The completed block diagram is shown in Figure 7.6.

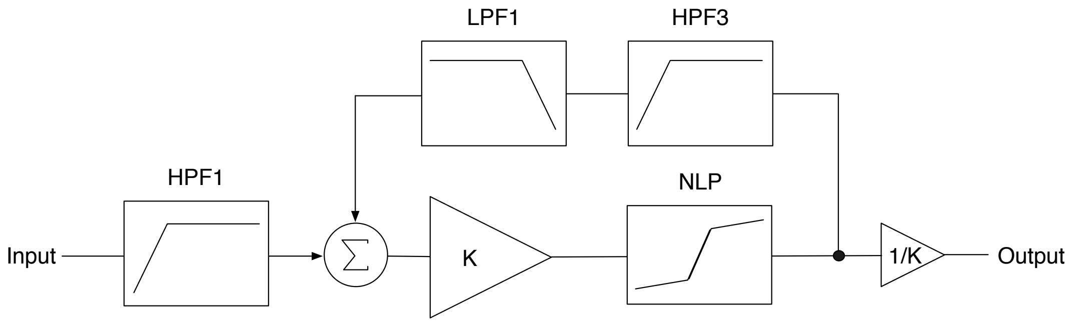

Figure 7.6: The completed Korg35 Highpass Filter Block Diagram with NLP and auto-normalizing blocks

To derive our analog transfer function, we start with the signal flow graph in Figure 7.7. The input transmission T_{13} is a first order HPF and the feedback T_{63} is a 2nd order buffered BPF (meaning a buffer between the HPF and LPF stages as in Figure 7.3).

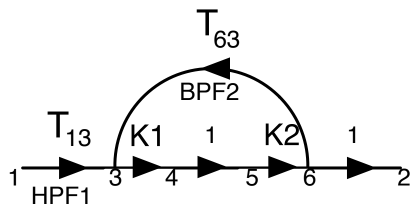

Figure 7.7: the signal flow graph for our hybrid HPF

What makes Figure 7.7 interesting is the T_{45} transmission component is 1 where normally there would be either a HPF1 or LPF1 block in its place for a 2nd order filter. Using Mason's Gain Equation, we can write the transfer directly from the signal flow graph:

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

$$T_{63} = \frac{sR_1C_1}{s^2R_1C_1R_2C_2 + s\left(R_1C_1 + R_2C_2\right)+1} = \frac{sR_1C_1}{(1+sR_1C_1)(1+sR_2C_2)}$$

$$K = K1K2$$

$$
\begin{aligned}
H(s) &= \frac{KT_{13}}{1-KT_{63}} \\
&= \frac{K\dfrac{sR_1C_1}{1+sR_1C_1}}{1-K\left(\dfrac{sR_1C_1}{(1+sR_1C_1)(1+sR_2C_2)}\right)} \\
&= \frac{KsR_1C_1}{1+sR_1C_1-K\left(\dfrac{sR_1C_1}{(1+sR_2C_2)}\right)} \\
&= \frac{KsR_1C_1}{\dfrac{s^2R_1C_1R_1C_1+s\left(R_1C_1(1-K)+R_2C_2\right)+1}{1+sR_2C_2}} \\
&= \frac{K(sR_1C_1+s^2R_1C_1R_2C_2)}{s^2R_1C_1R_1C_1+s\left(R_1C_1(1-K)+R_2C_2\right)+1} \\
&= K\left[\frac{sR_1C_1}{s^2R_1C_1R_1C_1+s\left(R_1C_1(1-K)+R_2C_2\right)+1}+\frac{s^2R_1C_1R_2C_2}{s^2R_1C_1R_1C_1+s\left(R_1C_1(1-K)+R_2C_2\right)+1}\right]
\end{aligned}
$$

This reveals a first order BPF and a 2nd order HPF in parallel. From it we can extract the gain, cutoff and Q:

$$H_0 = K = K1K2$$

$$\omega = \frac{1}{\sqrt{R_1C_1R_1C_1}}$$

$$Q = \frac{\sqrt{R_1C_1R_1C_1}}{(1-K1K2)R_1C_1+R_2C_2}$$

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. For the unbuffered version (i.e. the original analog filter) the equations are different and K must be 3.0 for self oscillation. I derive this using signal flow graphing at the end of the document. Since H_0 = K we will need to normalize the output at 1/K to keep the overall filter gain at 1.0.

However, we have a conundrum when trying to find a minimum value for K. In order to match an actual 1st order HPF with 6dB/octave rolloff, K will need to be 0.0 so that none of the bandpass signal is added to the input filter. But, with K = 0 there is no forward gain through the filter. So, we can let K take on a very small non-zero value. With K = 0.01 we are within a fraction of a dB of achieving the same 1st order HPF edge response. Thus our emulation can have K vary from 0.01 to 2.0.

VA Korg35 Design Equations Simplified - Linear Model

The block diagram in Figure 7.8 has a delay-less feedback path. We can resolve this delay-less path to make a VA emulation. We start by ignoring the NLP and auto-normalizing blocks and labeling Figure 7.11 with intermediate nodes. I have kept the same component numbers (HPF1, LPF1, HPF3) as in App Note 7 so the equations and code would match:

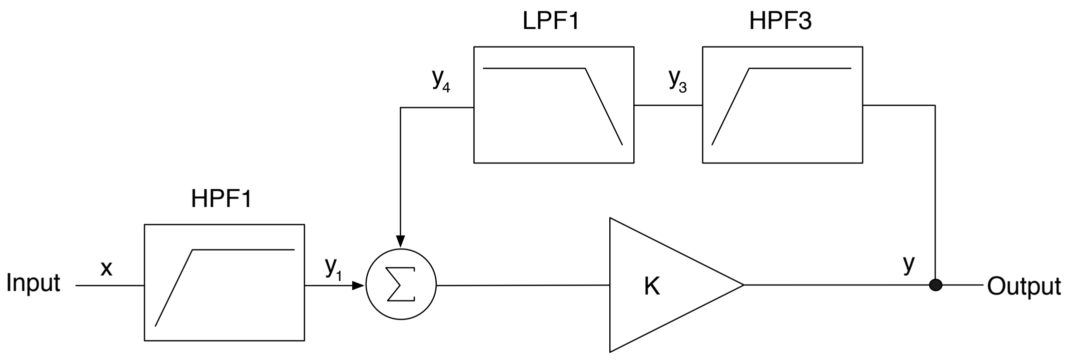

Figure 7.8: The nodes are labeled without the (n) notation for convenience (i.e. x rather than x(n), etc...)

The circuit equations are then found as:

HPF1:

$$\begin{aligned} y_1 &= x - (Gx + S1) \\ &= x - Gx - S1 \end{aligned}$$

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

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

HPF3:

$$\begin{aligned} y_3 &= y - (Gy + S3) \\ &= y - Gy - S3 \end{aligned}$$

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

LPF1:

$$\begin{aligned} y_4 &= Gy_3 + S4 \\ &= G(y - Gy - S3) + S4 \\ &= Gy - G^2 y - GS3 + S4 \end{aligned}$$

$$S4 = \frac{s_4}{1+g}$$

The final output is then

$$\begin{aligned} y &= K(y_1 + y_4) \\ &= K(x - Gx - S1 + Gy - G^2 y - GS3 + S4) \end{aligned}$$

We then eliminate the delay-less feedback path by separating variables and isolating y vs. x:

$$y = \frac{(K - KG)x - KS1 - KGS3 + KS4}{1 - KG + KG^2}$$

Or in more familiar VA terms:

$$\begin{aligned} y &= G35Hx + S35H \\ G35H &= \frac{K - KG}{1 - KG + KG^2} \\ S35H &= \frac{-KS1 - KGS3 + KS4}{1 - KG + KG^2} \end{aligned}$$

I named the terms G35H and S35H to avoid confusion with G35 and S35 in the Korg35 LPF App note. This is the final equation for the output of the filter. We will have an added step of resolving the S terms into the s values i.e.

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

etc...

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. Figures 7.12 and 7.13 show the two building blocks of the design, the first order TPT LPF and HPF respectively.

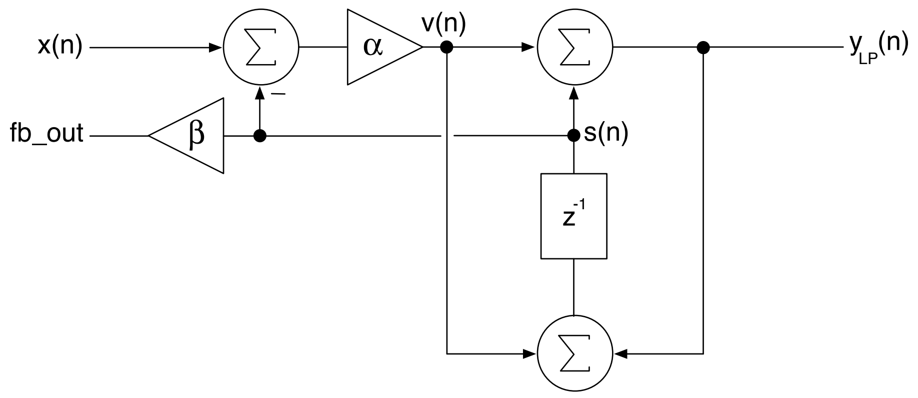

Figure 7.9: 1st order TPT LPF Block Diagram

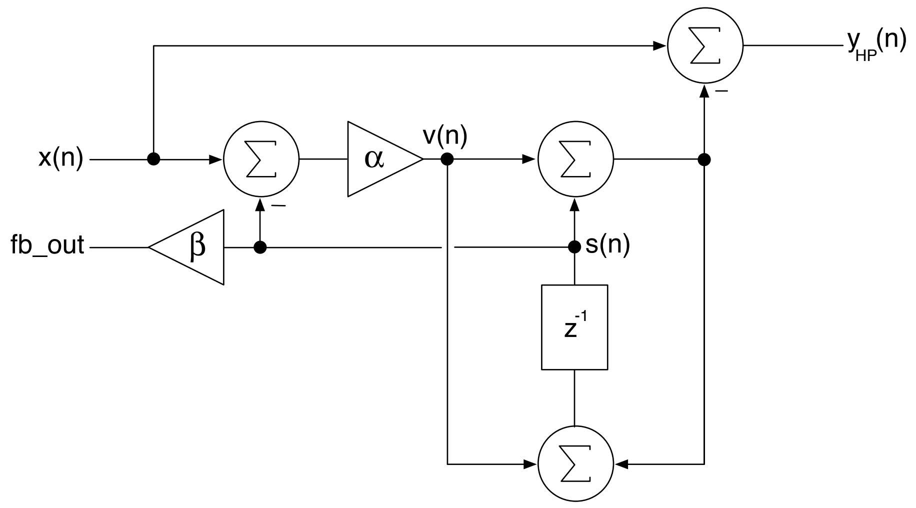

Figure 7.10: 1st Order TPT HPF Block Diagram

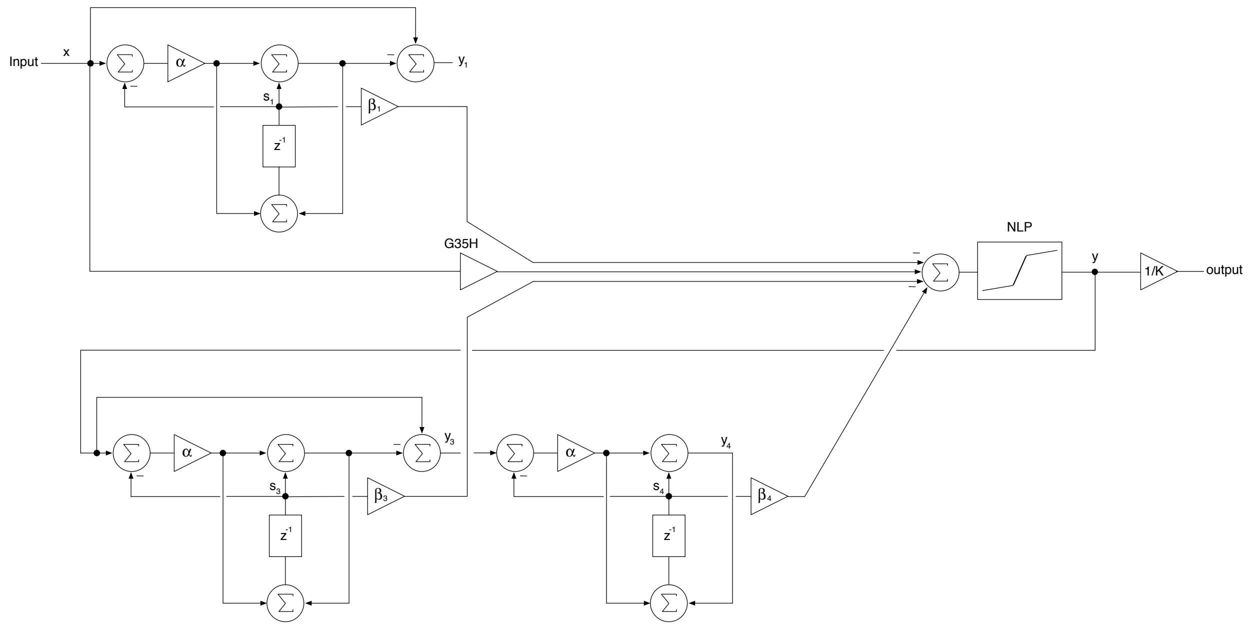

Figure 7.11: The completed VA Korg35 Highpass Filter - Simplified Linear Model

A Landscape version is also included; print out or rotate the App Note for better viewing.

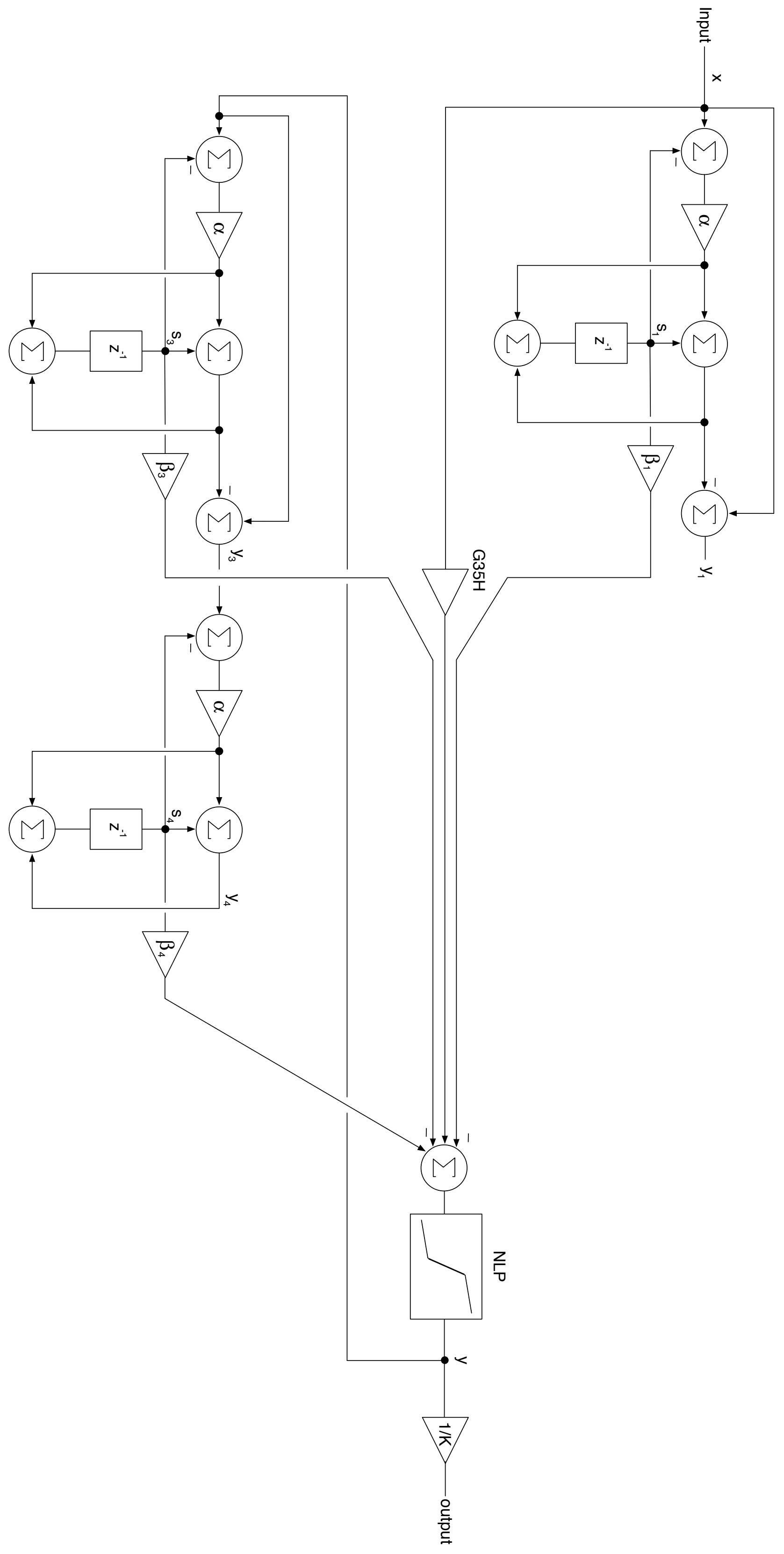

Figure 7.11: the completed Linear Model (landscape)

Figure 7.11 shows the completed filter auto-normalizing (1/K) coefficient included. Interestingly, you can see that the outputs y_1 and y_4 don't appear to lead anywhere. We took care of that when we resolved the delay-less feedback loop. Look at the main summer that feeds loop. Make sure you can figure out that it implements the equation

$$y = G35Hx + S35H$$

where the alpha and beta coefficients are as follows:

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

$$G35H = \frac{K-KG}{1-KG+KG^2}$$

$$\beta_1 = \frac{-K}{(1+g)(1-KG+KG^2)}$$

$$\beta_3 = \frac{KG}{(1+g)(1-KG+KG^2)}$$

$$\beta_4 = \frac{K}{(1+g)(1-KG+KG^2)}$$

Note: the added (1 + g) terms in the denominator of the beta coefficients come from resolving the S into s values, i.e.

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

Make sure you can connect the diagram to the equations; print out the App Note and label the nodes and trace through the branches if you need to. Also notice that some of the branches are subtracted (S2, S3, S5)

Nonlinear Model

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 7.8. This has several implications. First, it is going to alter final output equation. The output y is now tanh(K(y_2+y_4+y_6)), thus the final filter equation becomes

$$\begin{aligned} y &= \tanh(K(y_2+y_4+y_6)) \\ &= \tanh(K(x-2Gx+G^2x+GS1-S1-S2+Gy-G^2y-GS3+S4+Gx-Gx^2-GS5+S6)) \end{aligned}$$

This would lead to an unsolvable equation when we try to isolate y to resolve the delay-less loop. But, experiments show it will still work without blowing up and without tuning issues for high values of K (resonant peaks match locations using a 16384 point FFT with Blackman-Harris windowing and a bin spacing of 2.7Hz with and without the NLP engaged). However, as K is reduced, the resonant peak shifts slightly depending on the cutoff frequency. A 11.4% drift was observed for fc = 1kHz and K = 1.0 with NLP turned on and saturation = 1.0. However with K = 1.9, the resonant peak frequencies were within the 2.7Hz margin for the FFT (they measured the same value of 1001.29Hz).

Impulse response testing shows that for saturation = 1.0, the filter is on the verge of oscillation. Self oscillation can be induced by increasing the saturation however high values if saturation will also de-tune the filter.

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 7.12.

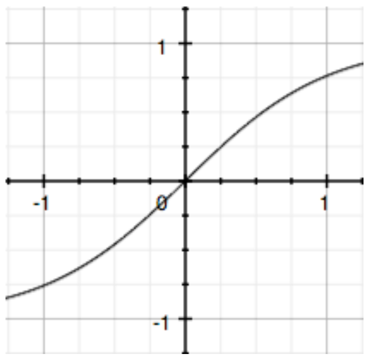

Figure 7.12: tanh(x) produces about 0.8 when x = 1

In the naive implementation, the filter may not self oscillate under this condition however it will whistle and lie on the verge of oscillation. In the sample code, I also provide the saturation variable that simply amplifies the signal going into the tanh() function which has the effect of squaring out the soft clipper, as shown in Figure 7.13.

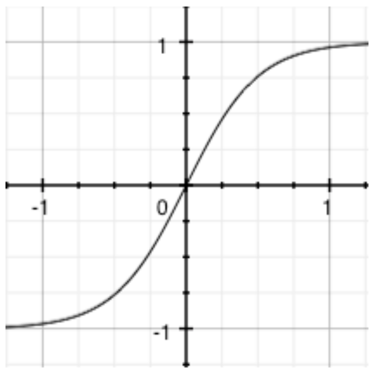

Figure 7.13: tanh(2x) produces gain and a more squared transfer function

Figures 7.14 and 7.15 show the frequency responses of the filter for various values of fc and Q with NLP turned off. It uses the new High Resolution FFT in the current RackAFX Beta (still under test). Figure 7.16 shows a worst case scenario with the input signal at 0dBFS with the same frequency as the cutoff. You can see in 7.16a that the signal is already clipping. Engaging the NLP will soft-clip the signal. Bringing the saturation up to 1.5 reproduces the full-scale amplitude and whistles and oscillates with input.

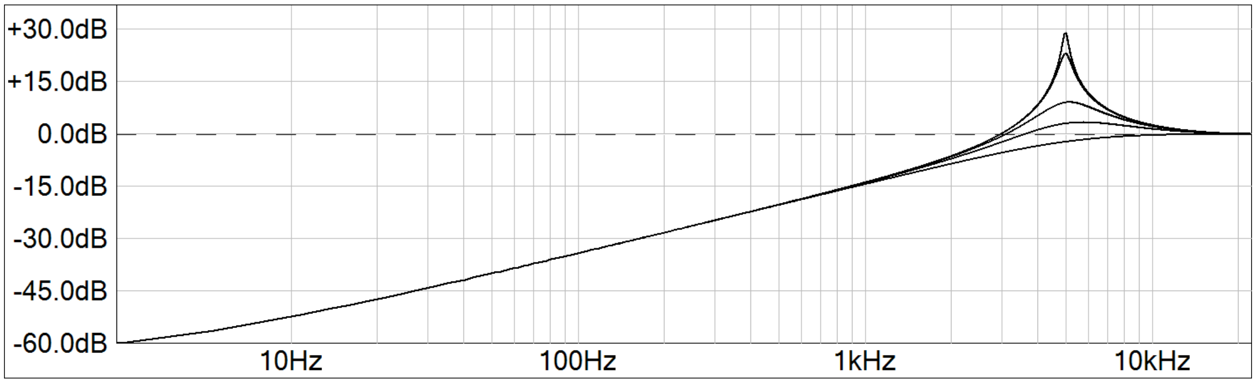

Figure 7.14: the Korg35 HPF emulation with fc = 1kHz and K = 0.170, 1.0, 1.5, 1.75, 1.95, NLP = OFF

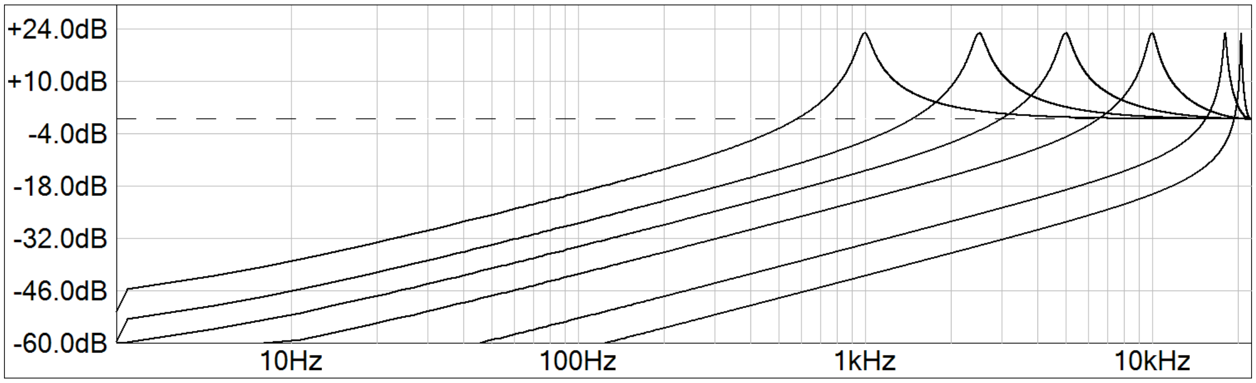

Figure 7.15: the Korg35 emulation with fc = 1kHz, 2.5kHz, 5kHz, 10kHz and 15kHz, 18kHz and 20480kHz. On the original Korg35, fc is variable from 50Hz to 15kHz. NLP = OFF. The sample code allows for a cutoff range of 10 octaves 20Hz - 20,480Hz.

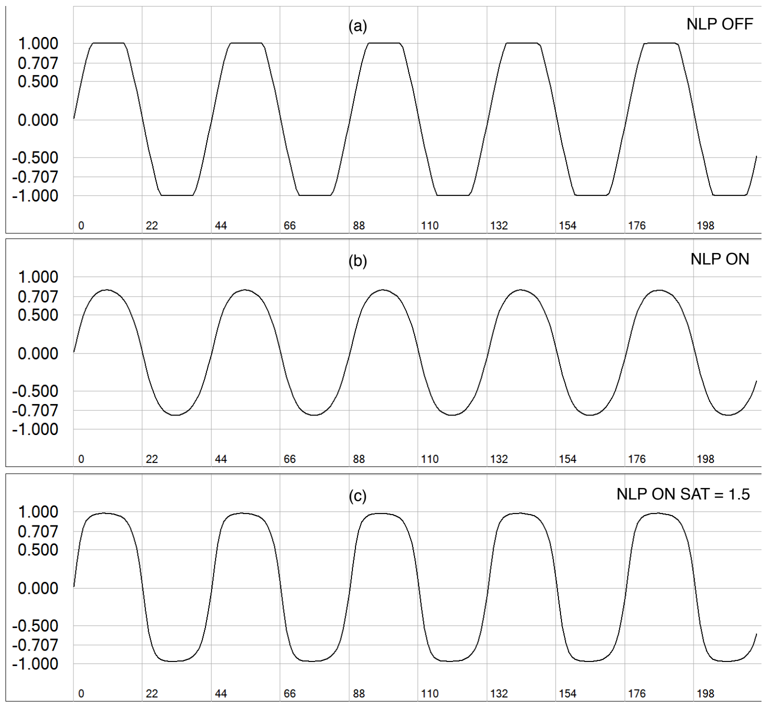

Figure 7.16: the effect of NLP in the time domain; fc = 1kHz, input f = 1kHz, 0dBFS, K = 1.0 (a) NLP off shows the output is clipping. (b) NLP on with saturation = 1.0 and (c) NLP on with saturation = 1.5

Sample Code

The sample code is a typical RackAFX project called *KorgThreeFiveHPFmkII.* I created an object to encapsulate the VAOnePoleFilter, similar to App Note 4.

In the .h File

```
// Add your code here: ----------------------------------------------------- //
// these match the App Note and the original K35HPF version naming
CVAOnePoleFilter m_HPF1;
CVAOnePoleFilter m_HPF3;
CVAOnePoleFilter m_LPF1;

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

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

// variables
double G35H; // our G value
double S35H; // our S value // y = Gx + S

// 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 CKorgThreeFiveHPFmkII::prepareForPlay()
{
	// Add your code here:
	// set types
	m_HPF1.m_uFilterType = HPF1;
	m_LPF1.m_uFilterType = LPF1;
	m_HPF3.m_uFilterType = HPF1;

	// flush everything
	m_HPF1.reset();
	m_LPF1.reset();
	m_HPF3.reset();

	// set initial coeff states
	updateFilters();

	return true;
}
```

updateFilters()
- called when GUI changes
- calculate and set the alpha and beta values
- calculate our own G35 coefficient

```
void CKorgThreeFiveHPFmkII::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_HPF1.m_fAlpha = G;
	m_LPF1.m_fAlpha = G;
	m_HPF3.m_fAlpha = G;

	// set betas all are in the form of  <something>/((1 + g)(1 - kG + kG^2))
	float fDenominator = ((1.0 + g)*(1.0 - m_dK*G + m_dK*G*G));

	m_HPF1.m_fBeta = -1.0*m_dK/fDenominator;
	m_HPF3.m_fBeta = m_dK*G/fDenominator;
	m_LPF1.m_fBeta = m_dK/fDenominator;


	// set the G35H variable
	G35H = (m_dK - m_dK*G)/(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
- auto-normalize by 1/K

```
double CKorgThreeFiveHPFmkII::doFilter(double xn)
{
        // FIRST: form feedback and feed forward values (read before write)
        S35H = m_HPF1.getFeedbackOutput() - m_HPF3.getFeedbackOutput() +
               m_LPF1.getFeedbackOutput();

        // y = G35x + S35
        double y = G35H*xn + S35H;

        // NAIVE NLP
        if(m_uNonLinearProcessing == ON)
               y = tanh(m_dSaturation*y);

        // THEN: update -- note the outputs of the sections are not used directly
        //                  because we resolved the zero-delay feedforward loop and
        //                  access the values that way (via G35 and S35)
        //
        // process x through first HPF
        m_HPF1.doFilter(xn);

        // process y through the feedback path
        m_LPF1.doFilter(m_HPF3.doFilter(y));

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

        return y;
}
```

Analog Simulation

Because our filter sections do not suffer from impedance loading, unity gain high-Z buffers are inserted between each stage (U1,U2,U3,U4,U7,U8). Figure 7.17 shows the circuit used in the simulation of the Korg35 HPF (original version) while Figure 7.18 shows the simulated frequency response. Time domain analysis confirms oscillation at K = 2.0.

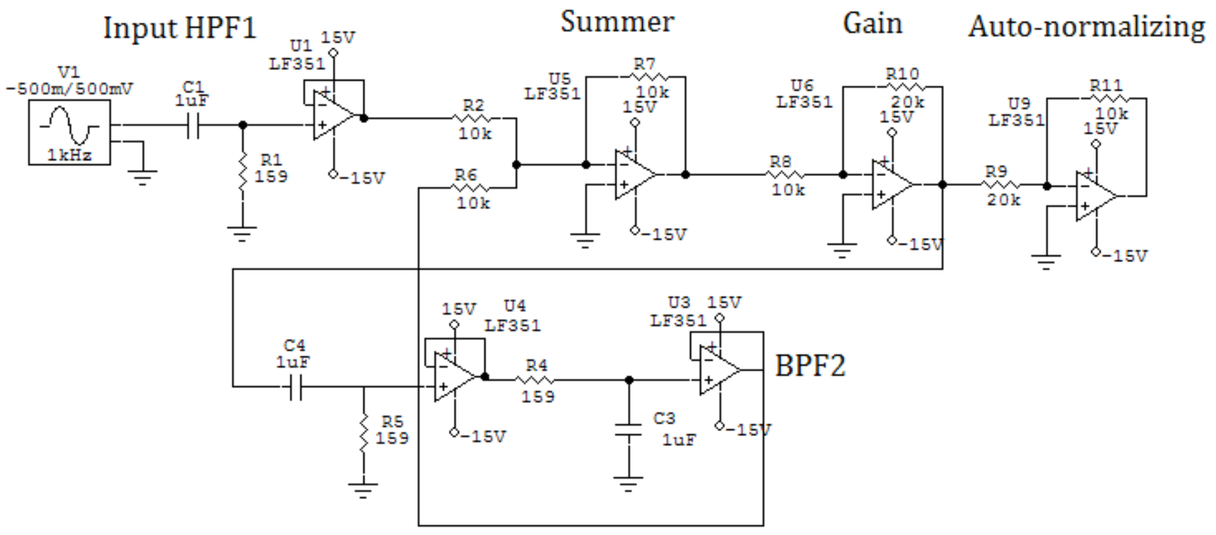

Figure 7.17: CircuitMaker simulation of the digital emulation; this is shown with K = 2.0 (R10 and op amp U6) R2, R6, and R14 sum the three signal paths together

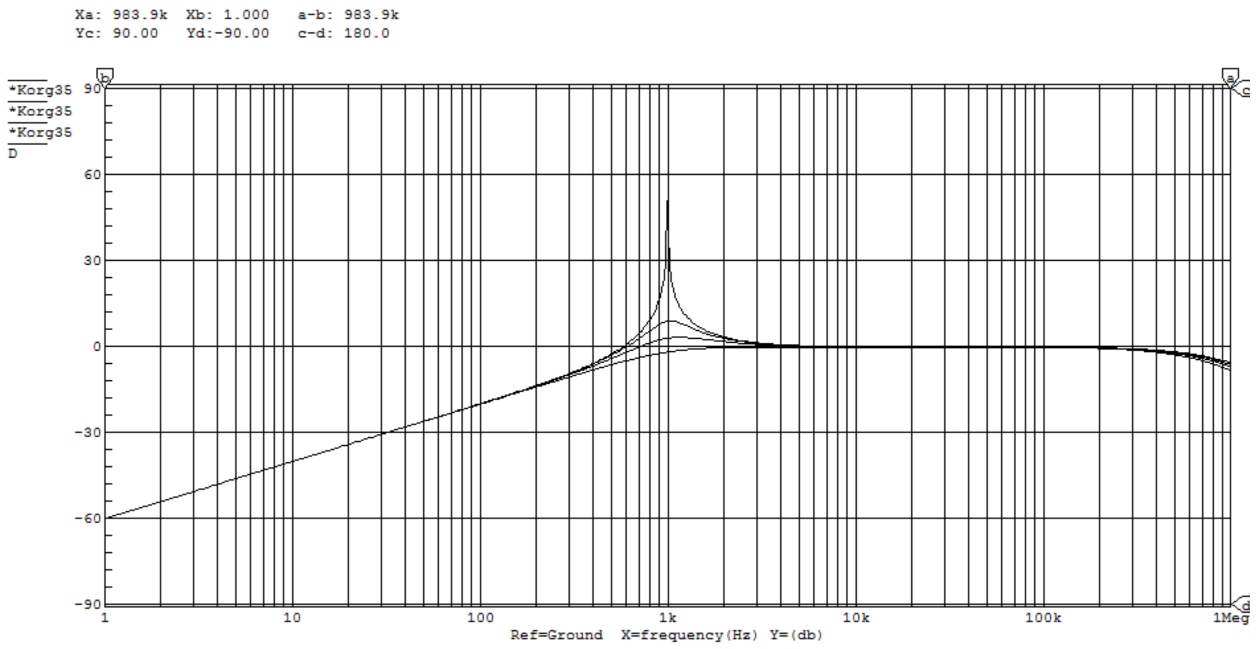

Figure 7.18: simulated frequency response with K = 0.3, 1.0, 1.5, and 2.0; note the 6dB/octave rolloff

Derivation of the unbuffered (analog) version

The unbuffered, single amplifier version is shown in Figure 7.19 along with the signal flow graph.

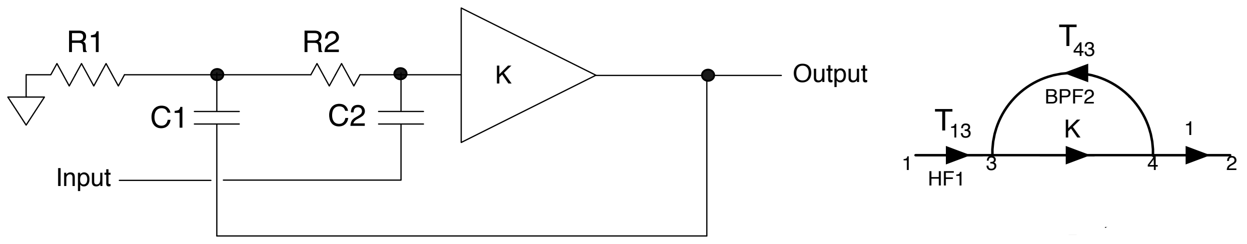

Figure 7.19: the unbuffered original analog filter

Start with the transmissions T13 and T43 and Mason's Gain Equation:

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

$$T_{43} = \frac{sR_1C_1}{s^2R_1C_1R_2C_2 + s\left(R_1C_1 + R_2C_2 + R_1C_2\right) + 1} = \frac{sR_1C_1}{(1+sR_1C_1)(1+sR_2C_2) + sR_1C_2}$$

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

$$= \frac{K\dfrac{sR_1C_1}{1+sR_1C_1}}{1-K\left(\dfrac{sR_1C_1}{(1+sR_1C_1)(1+sR_2C_2)+sR_1C_2}\right)}$$

$$= \frac{KsR_1C_1}{1+sR_1C_1 - K\left(\dfrac{sR_1C_1}{\left(1+sR_2C_2+sR_1C_2\right)}\right)}$$

$$= \frac{KsR_1C_1}{\dfrac{s^2R_1C_1R_2C_2 + s\left(R_1C_1(1-K) + R_2C_2 + R_1C_2\right)+1}{1+s(R_2C_2+R_1C_2)}}$$

$$= \frac{K(sR_1C_1 + s^2(R_1C_1R_2C_2 + R_1C_1R_1C_2)}{s^2R_1C_1R_2C_2 + s\left(R_1C_1(1-K) + R_2C_2 + R_1C_2\right)+1}$$

$$= K\left[\frac{sR_1C_1}{s^2R_1C_1R_2C_2 + s\left(R_1C_1(1-K) + R_2C_2 + R_1C_2\right)+1} + \frac{s^2(R_1C_1R_2C_2 + R_1C_1R_1C_2)}{s^2R_1C_1R_2C_2 + s\left(R_1C_1(1-K) + R_2C_2 + R_1C_2\right)+1}\right]$$

This reveals a first order BPF and a 2nd order HPF in parallel. From it we can extract the gain, cutoff and Q:

$$H_0 = K$$

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

$$Q = \frac{\sqrt{R_1C_1R_2C_2}}{(1-K)R_1C_1 + R_2C_2 + R_1C_2}$$

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

$$H(s) = K\left[\frac{s}{s^2 + s(3-K)+1} + \frac{2s^2}{s^2 + s(3-K)+1}\right]$$

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

Self oscillation occurs when K = 3.0.

Revision History:

1.1: Initial Release, *August 15, 2013*
2.0: simplified model
2.1: showed why self oscillation occurs at K = 2; set new minimum for K to get Q = 0.707
2.2: added full hybrid and unbuffered derivations

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

Lindquist, Claude. 1977. *Active Network Theory with Signal Filtering Applications*, Long Beach: Steward and Sons.

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