Copyright © 2013 Will Pirkle

Virtual Analog (VA) Korg35 Highpass Filter
Will Pirkle
July 25, 2013

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.

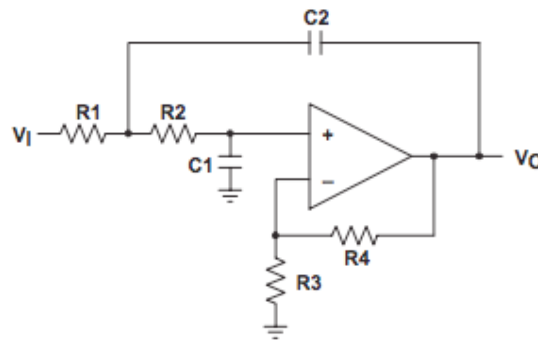

Figure 7.1: The Sallen-Key LPF

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.

Copyright © 2013 Will Pirkle

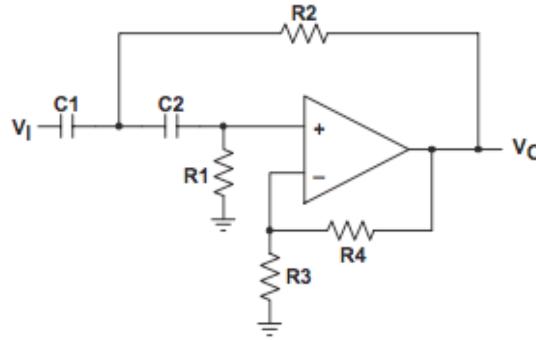

Figure 7.2: the Sallen-Key HPF

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 in software we need to dig deeper into the Sallen-Key *highpass* filter topology which would result from the RC:CR transformation. If you are unfamiliar with analog audio electronics, this might be difficult to follow but you should still look it over as it contains the method for this filter emulation. We are also going to use the fact that R1 = R2 and C1 = C2 in the design. Figure 7.3 shows the generalized Sallen-Key highpass filter topology.

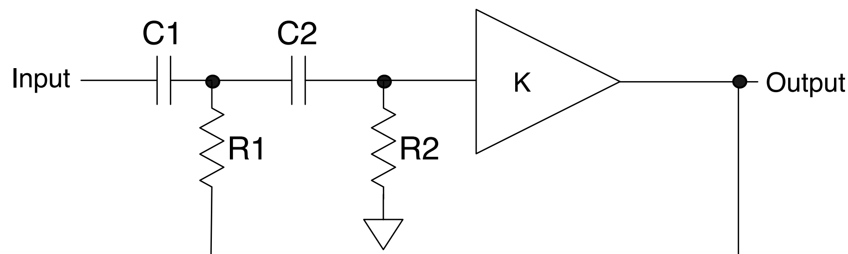

Figure 7.3: The Sallen Key highpass filter without the op-amp details.

By the (analog electronics) principle of superposition Figure 7.3 can be re-drawn in block diagram form. It consists of two synchronously tuned RC HPFs (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 7.4. The BPF/feedback path reinforces the Q of the filter.

Copyright © 2013 Will Pirkle

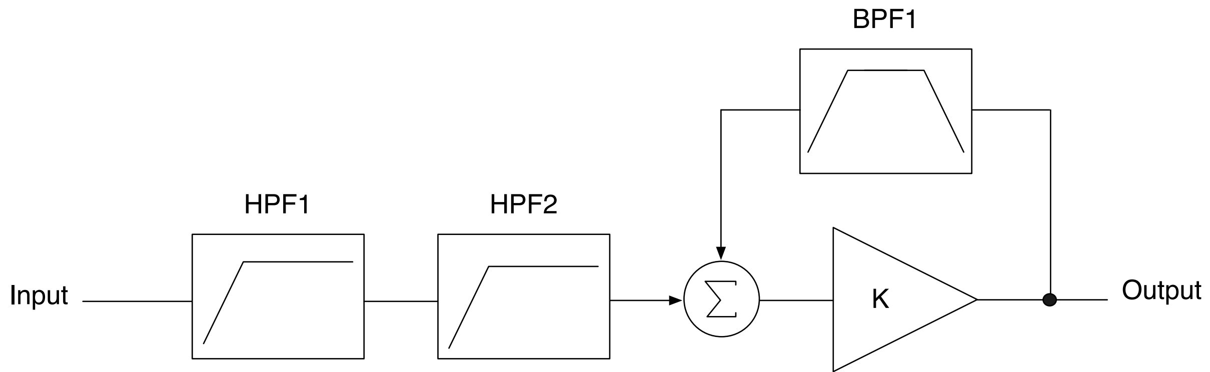

Figure 7.4: The Sallen-Key HPF Block Diagram

HPF 1 and HPF2 are first order and since R1 = R2 and C1 = C2, 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). Thus Figure 7.4 could be redrawn as Figure 7.5. Note all filters (including the two that make the BPF) are synchronously tuned.

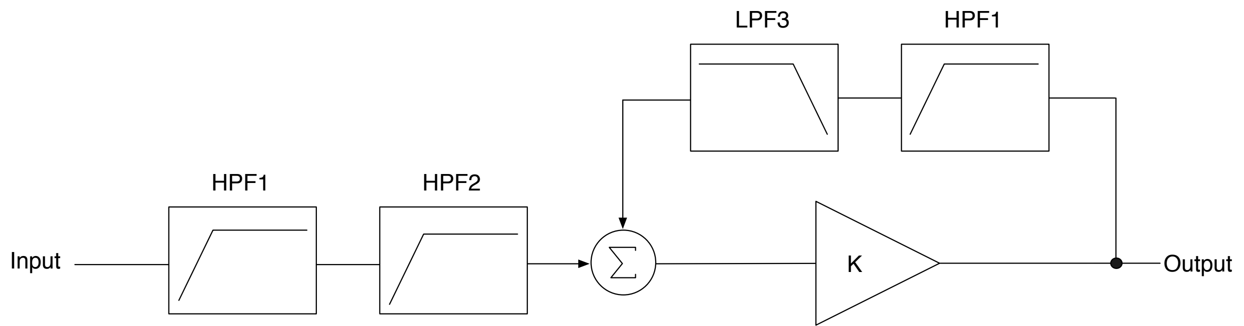

Figure 7.5: The Sallen-Key HPF topology.

Figure 7.5 will produce the complementary highpass filter with the same 12dB/octave rolloff. To turn this into the Korg35 HPF, we need to analyze the analog circuit. Stinchcombe's excellent paper [2006] revealed that grounding the input and driving the ground produces a transfer function with an extra BPF filtering term. This produces a parallel BPF on the input section as shown in Figure 7.6. The effect of the second parallel BPF is to offset the rolloff below the cutoff frequency so that it shifts to 6dB/octave. Figure 7.7 resolves the second BPF into HPF and LPF sections.

Copyright © 2013 Will Pirkle

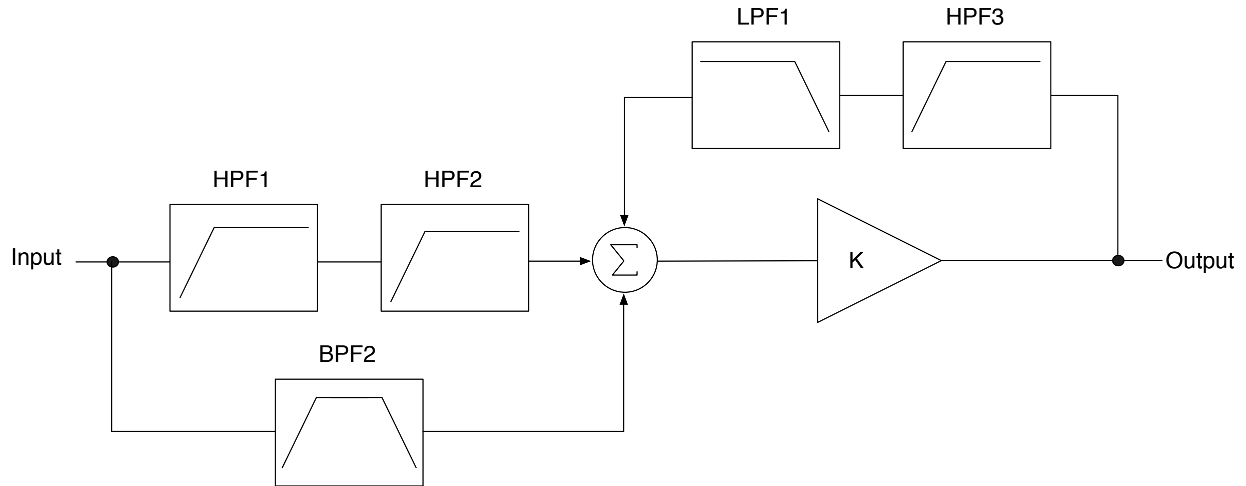

Figure 7.6: The Korg35 HPF topology.

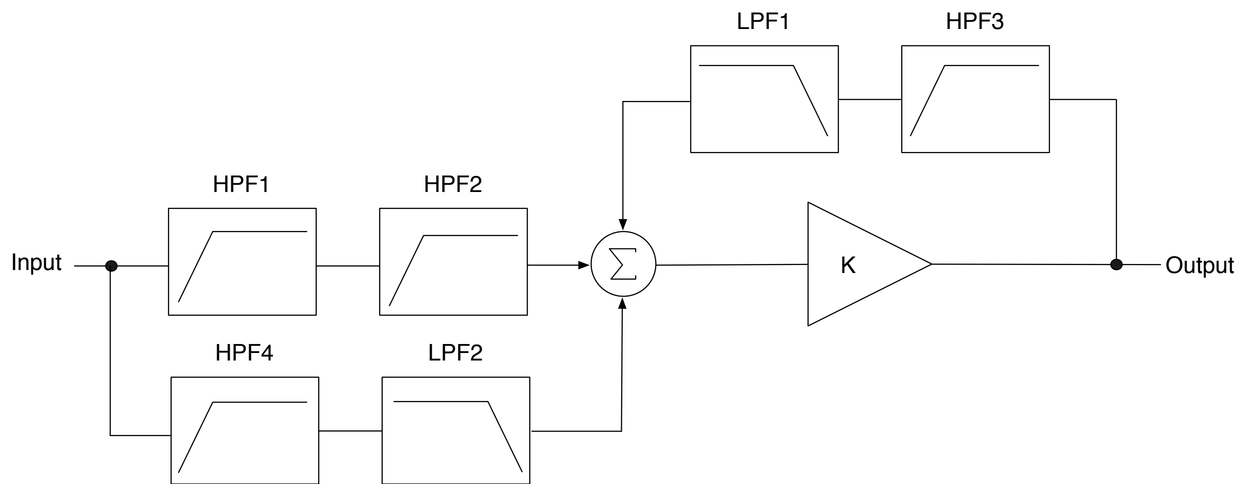

Figure 7.7: The Korg35 HPF topology with the second BPF converted into the series HPF and LPF.

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. As before, placing the NLP block inside the feedback path is consistent with the original design, but will have consequences that are covered in the next section. 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 7.8.

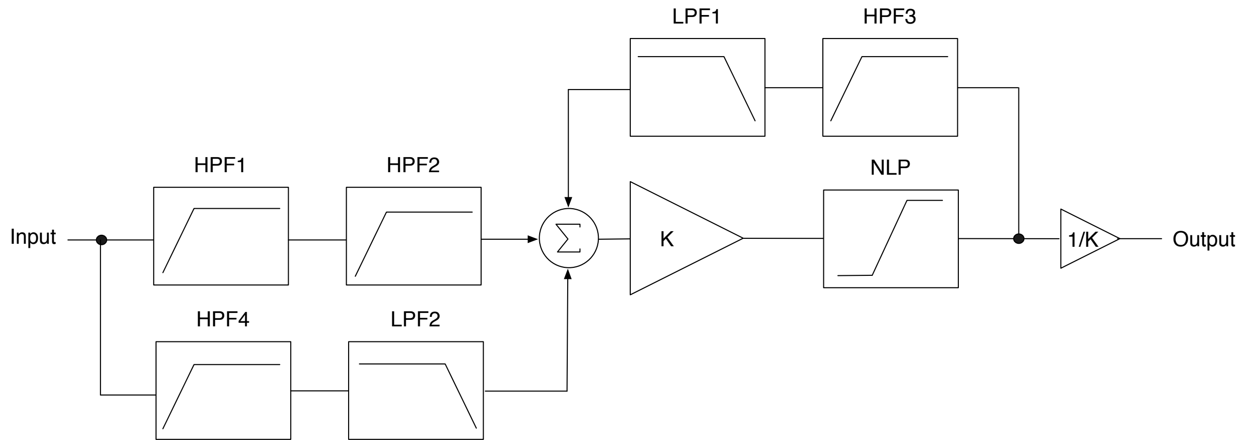

Figure 7.8: The completed Korg35 Highpass Filter Block Diagram with NLP and auto-normalizing blocks

The filter equations for the Sallen-Key highpass filter are:

$$H(s) = \frac{H_0 s^2}{s^2/\omega_c^2 + 2\zeta s/\omega_c + 1}$$

$$= \frac{Ks^2(R_1C_1R_2C_2)}{s^2R_1C_1R_2C_2 + s((1-K)R_1C_2 + R_1C_1 + R_2C_2) + 1}$$

We can then find the cutoff fc and Q and DC gain values as:

$$H_0 = K$$

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

$$2\zeta = \frac{1}{Q} \text{ (the well known relationship between damping and Q)}$$

$$Q = \frac{\sqrt{R_1C_1R_2C_2}}{(1-K)R_1C_2 + R_2C_1 + R_2C_2}$$

It can also be shown that for the standard Sallen-Key highpass filter

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

However, since we have two bandpass filters tuned to f_c, we need to cut the K value in half. Thus for Butterworth (maximally flat magnitude) Q = 0.707, K = 0.5285/2 = 0.2643. In the original design, with the resonance pot grounded the filter turns into a critically damped filter with Q = 0.5 (i.e. a simple cascade of two first order sections), or K = 0.3333/2 = 0.1667. Self oscillation still occurs when K = 2.0 (as with the LPF version) since the BPF in the feedback loop has not been altered. 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. Thus our emulation can have

Copyright © 2013 Will Pirkle

K vary from 0.2643 to 2.0 (in the sample code, K is fixed on the range of [0.2643...2.0] but you may wish to experiment with over-damped variations by allowing K to go below 0.2643; obviously, K can not go all the way to 0.0).

VA Korg35 Design Equations - Linear Model

The block diagram in Figure 7.8 has an interesting feature; it 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 7.9 with intermediate nodes:

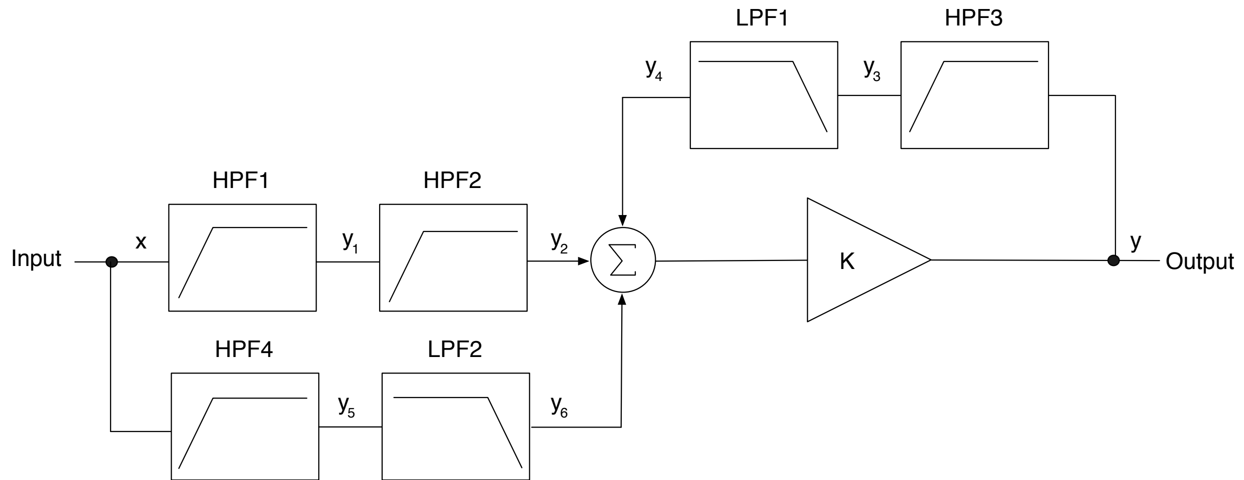

Figure 7.9: The nodes are labeled without the (n) notation for convenience (i.e. x rather than x(n), etc...)

The circuit equations are then found as:

HPF1:

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

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

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

HPF2:

$$\begin{aligned} y_2 &= y_1 - (Gy_1 + S2) \\ &= x - Gx - S1 - G(x - Gx - S1) - S2 \\ &= x - 2Gx + G^2x + GS1 - S1 - S2 \end{aligned}$$

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

Copyright © 2013 Will Pirkle

HPF3:

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

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

LPF1:

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

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

HPF4:

$$\begin{aligned} y_5 &= x-(Gx+S5) \\ &= x-Gx-S5 \end{aligned}$$

$$S5=\frac{s_5}{1+g}$$

LPF2:

$$\begin{aligned} y_6 &= Gy_5+S6 \\ &= G(x-Gx-S5)+S6 \\ &= Gx-G^2x-GS5+S6 \end{aligned}$$

$$S6=\frac{s_6}{1+g}$$

The final output is then

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

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

$$y=\frac{(K-KG)x+(KG-K)S1-KS2-KGS3+KS4-KGS5+KS6}{1-KG+KG^2}$$

Copyright © 2013 Will Pirkle

Or in more familiar VA terms:

$$y = G35Hx + S35H$$

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

$$S35H = \frac{(KG - K)S1 - KS2 - KGS3 + KS4 - KGS5 + KS6}{1 - KG + KG^2}$$

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 7.10 and 7.11 show the two building blocks of the design, the first order TPT LPF and HPF respectively.

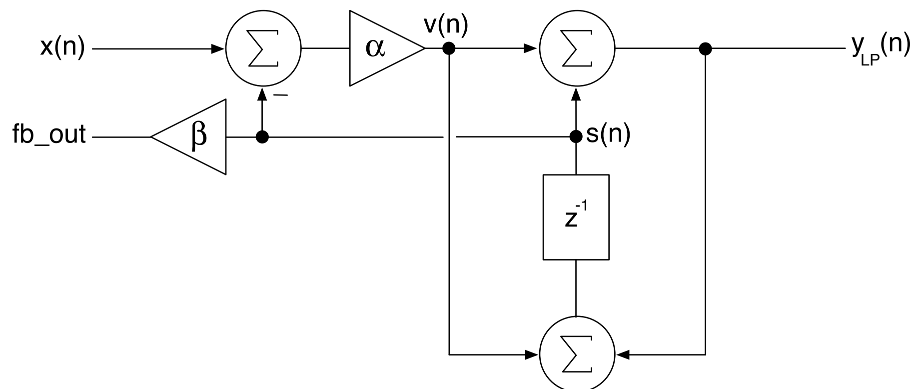

Figure 7.10: 1st order TPT LPF Block Diagram

Copyright © 2013 Will Pirkle

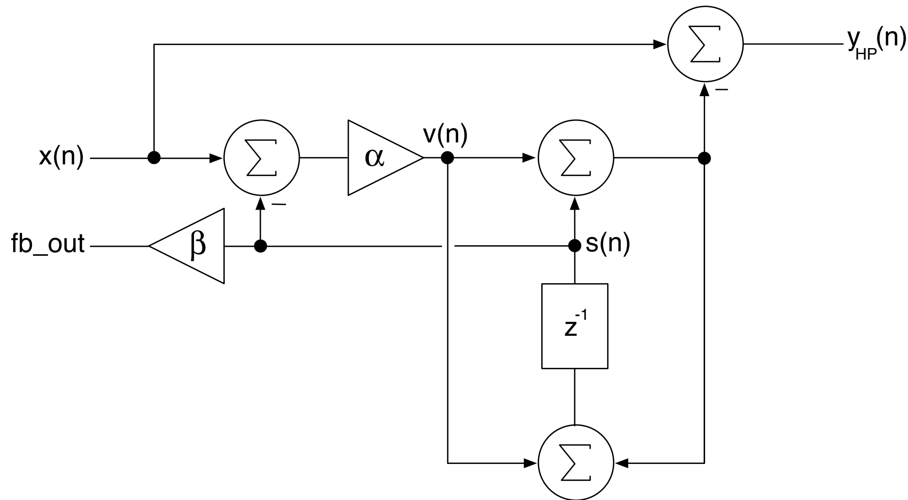

Figure 7.11: 1st Order TPT HPF Block Diagram

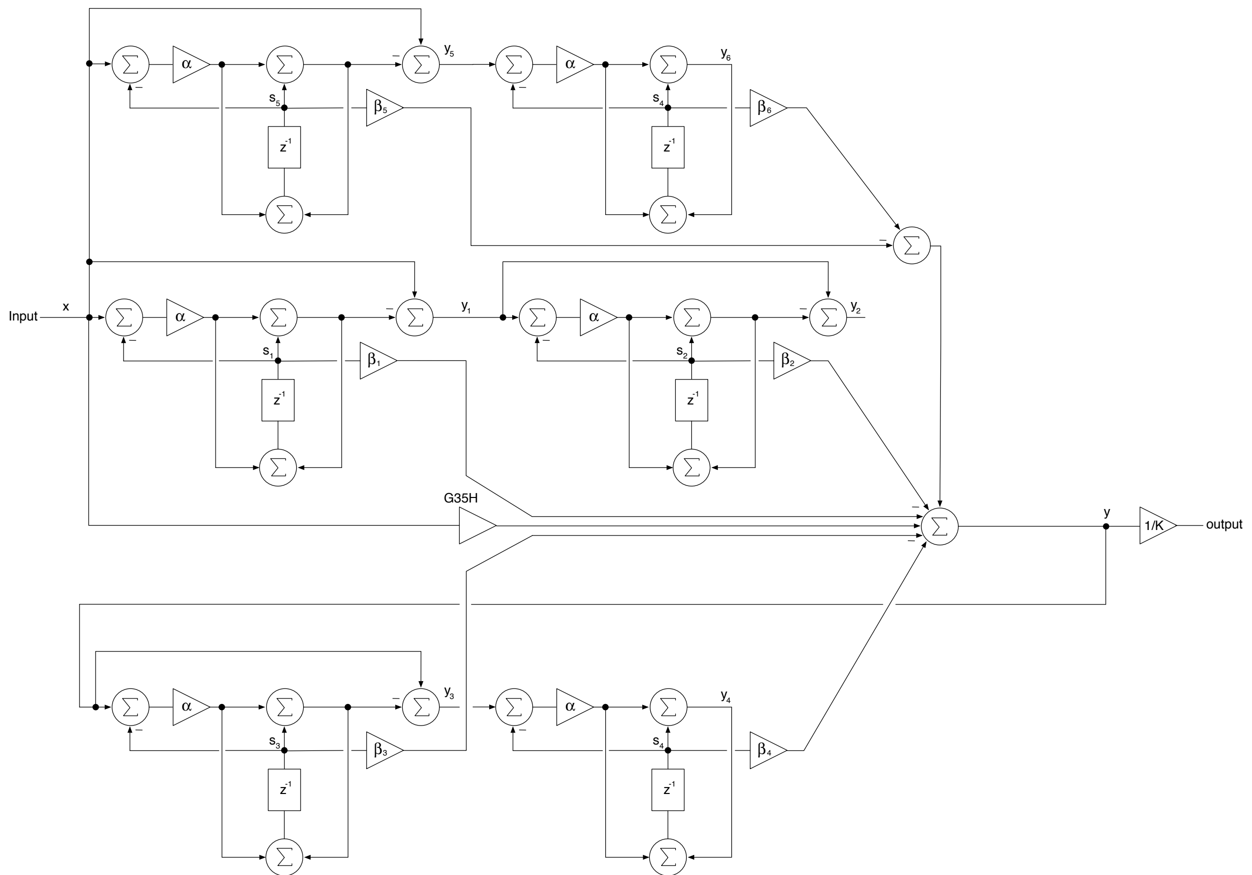

Figure 7.12: The completed VA Korg35 Highpass Filter - Linear Model

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

Copyright © 2013 Will Pirkle

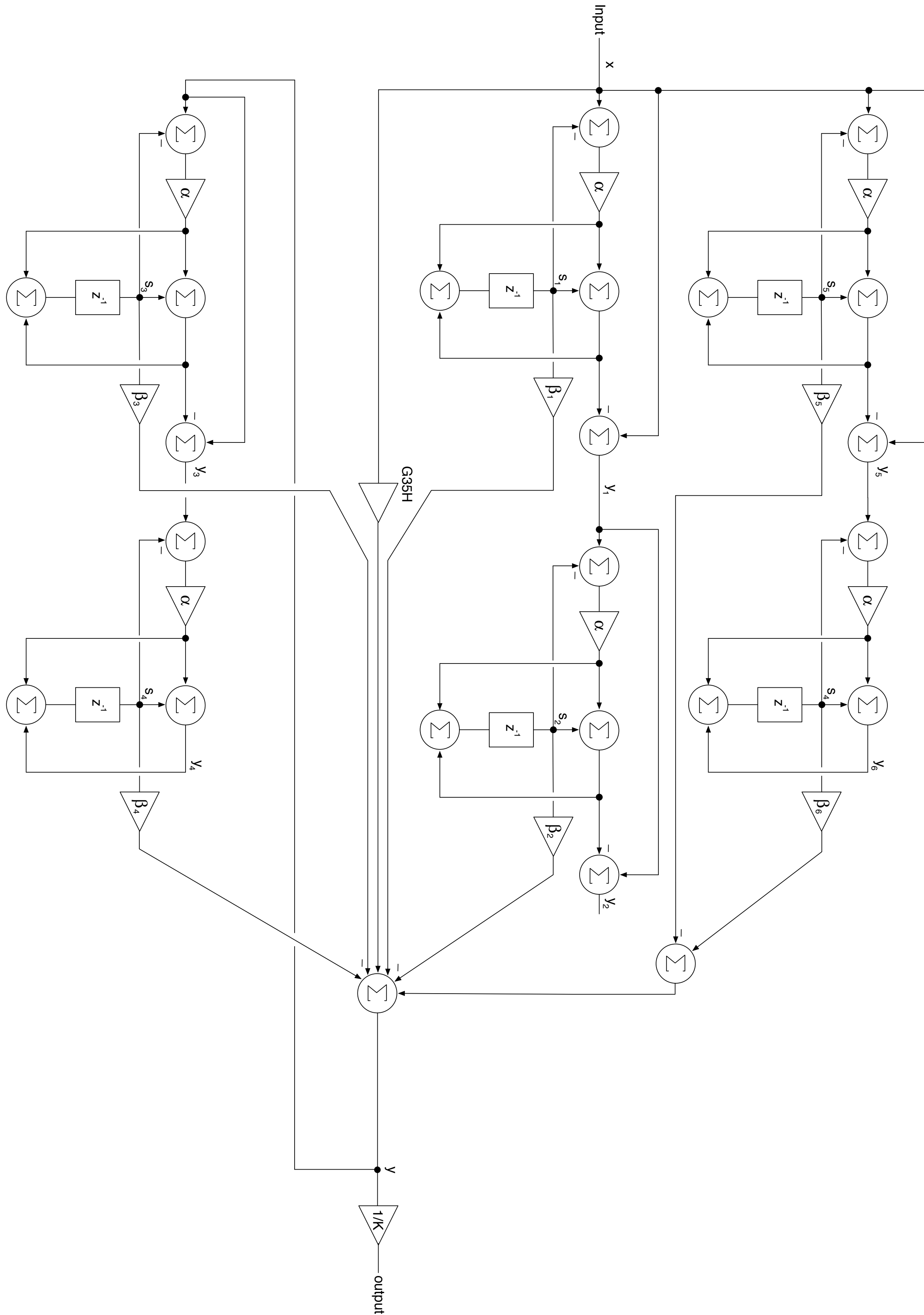

Figure 7.12: the completed Linear Model (landscape)

Copyright © 2013 Will Pirkle

Figure 7.12 shows the completed filter auto-normalizing (1/K) coefficient included. Interestingly, you can see that the output y_2, y_4 and y_6 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:

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

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

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

$$\beta_2 = \beta_4 = \beta_6 = \frac{K}{(1+g)(1 - KG + KG^2)}$$

$$\beta_3 = \beta_5 = \frac{KG}{(1+g)(1 - KG + KG^2)}$$

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)

Second Order Variation (*optional*)

Since we can control the design and don't need to reuse a filter module, we can obtain a true second order version of the filter by implementing the straight Sallen-Key HPF topology and leaving out the parallel BPF in Figure 7.6. The overall block diagram only changes by omitting the second (parallel) BPF. The block diagram is shown in Figure 7.13. In doing so, the equation for Q reverts back to the original and does not need to be cut in half, thus for Butterworth (maximally flat magnitude) Q = 0.707, K = 0.5285. The sample project implements both the original (default) and the optional second order versions.

Copyright © 2013 Will Pirkle

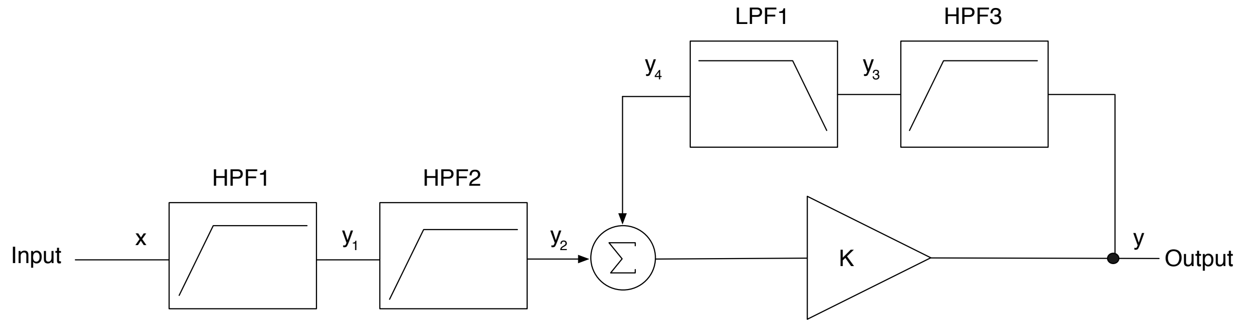

Figure 7.13: the “true” second order Sallen-Key HPF topology with nodes labeled

The filter equations for HPF1, HPF2, HPF3 and LPF1 do not change. However, we need to derive a new output equation for y as follows:

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

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

$$y = \frac{(K - 2KG + KG^2)x + (KG - K)S1 - KS2 - KGS3 + KS4}{1 - KG + KG^2}$$

Or in more familiar VA terms:

$$y = G35Hx + S35H$$

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

$$S35H = \frac{(KG - K)S1 - KS2 - KGS3 + KS4}{1 - KG + KG^2}$$

This is the final equation for the output of the filter. Again, we will have an added step of resolving the S terms into the s values i.e.

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

If you implement this variation, you will notice a slight difference in gain at the resonant peak; the 6dB/octave version will have a slightly higher peak value. This is because there is a hidden scalar of 2 feeding the parallel BPF branch [Stinchcombe]. This factor is there in the original Korg35 HPF as a consequence of grounding the input and driving the ground so I left it in place.

Copyright © 2013 Will Pirkle

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 7.14. 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.

We could try a *budget* implementation by moving the NLP block outside the loop, prior to the auto-normalizing coefficient. This would leave the filter equations undisturbed. Figure 7.14 shows the block diagram for both the naive and budget versions. Complete block diagrams of the Nonlinear Models are at the end of the document.

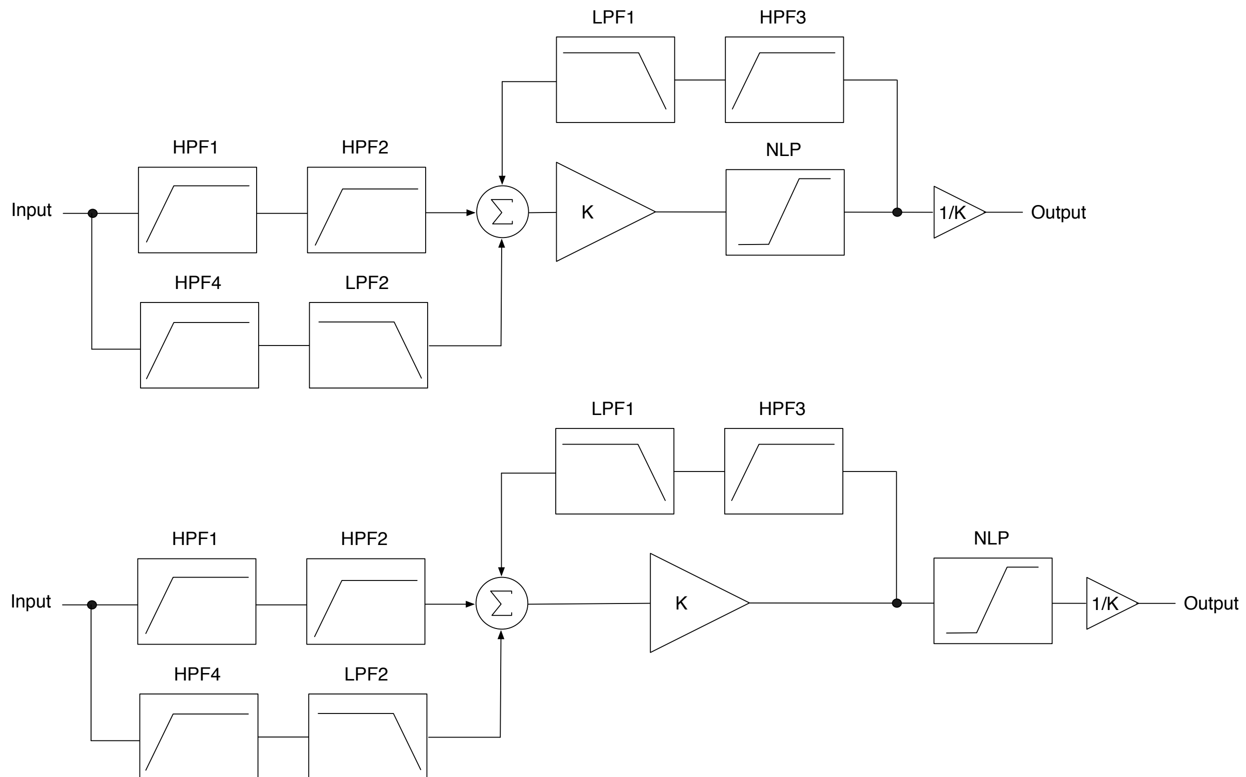

Figure 7.14: the naive NLP implementation (top) places the block in the feedback loop while the budget version (bottom) pulls it outside

Copyright © 2013 Will Pirkle

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 7.15.

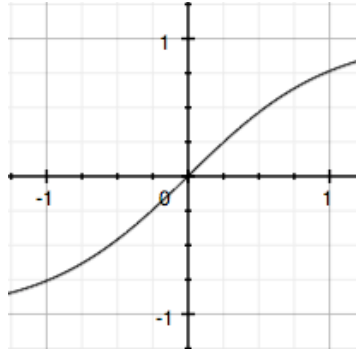

Figure 7.15: tanh(x) produces about 0.8 when x = 1

In the naive implementation, the filter may not self oscillate under this condition unless the loop gain K is increased beyond 2.0 - this is another issue with the naive implementation; 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 7.16 (notice that this changes the overall shape considerably, but since this is still an approximation we can experiment with it):

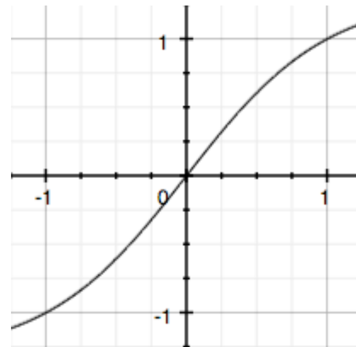

Figure 7.16: The normalized tanh() function transfer curve

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 7.17 shows this new function with *sat* = 3 but be careful: *for the naive NLP implementation, large values for the saturation variable will cause self oscillation to occur earlier as this is a form of gain and will interfere with the resonance (K) value.*

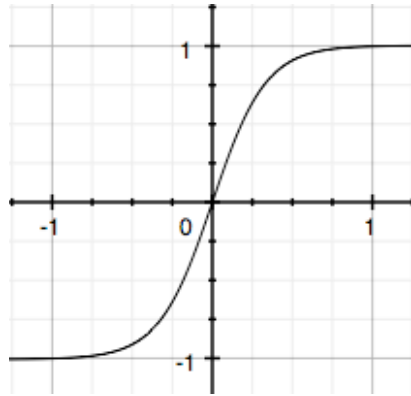

Figure 7.17: A steeper curve is obtained with *sat* > 1, here it is 3

In the code I provide both the normalized and regular (unmodified) versions of the waveshaper. The normalized version is the default; you can comment this out and un-comment out the regular version for experimentation. The saturation control will still work in both cases. I also provide both the naive and budget variations for you to experiment with. In the code below you would replace the part feeding the parallel BPF with this:

The sample RackAFX project allows you to choose either the *naive* or *budget* implementations as well as allowing you to turn on and off the NLP however as with any NLP, aliasing can and will occur. This is mitigated by oversampling which is covered in other App Notes so I have omitted it from the sample project for clarity.

Naive NLP Implementation

When you enable the NLP section in the naive implementation, self oscillation may occur at a lower resonance (K) value if the nonlinearity adds gain; the gain is also increased as you increase the saturation control. The filter remains stable but the range of K values must be tweaked. If you choose to experiment with the naive implementation, *the most important issue is the NLP block's interaction with overall filter gain K*. So, you are urged to do your own experimenting with different NLP functions, symmetrical/asymmetrical behavior, and gain ranges. You can also experiment with normalized versus regular tanh() waveshaping. Once you lock down your final NLP block, you will want to tweak the range of K values to give self oscillation at the maximum value, however you may find that this is not constant across the spectrum. Also beware that the distortion this is going to add can be severe depending on the saturation and resonance (K) settings! If you are a lo-fi aficionado, you may appreciate this, otherwise you may find it irritating.

Budget NLP Implementation

When you enable the NLP section in the budget implementation the filter is much more well behaved. You can add saturation to increase the grittiness of the filter but you can't alter the asymmetrical resonance behavior by making this waveshaper asymmetrical. In the code, this is not an option however you can choose between normalized and regular tanh() functions. The sample code defaults to the budget implementation. You may want to limit K to 1.99 if you find the self oscillation annoying.

Copyright © 2013 Will Pirkle

NOTE: With either of the NLP versions the overall gain of the filter can rise above unity since the NLP blocks are gain blocks. If you use NLP you will need to adjust the maximum K value (for self oscillation) and possibly the final gain stage to compensate for gains above unity. Experimentation and tweaking will be necessary depending on your final decision regarding NLP type, location and saturation level.

Figures 7.18 and 7.19 show the frequency responses of the filter for various values of fc and Q with NLP turned off. **NOTE: the low frequency error you see is NOT due to the filter, it is due to poor resolution of the FFT at low frequencies. The analyzer uses a 1024 point FFT which produces a resolution of 43Hz/bin. Therefore, the response below 43 Hz only has one more data point at f = 0Hz, thus the error.**

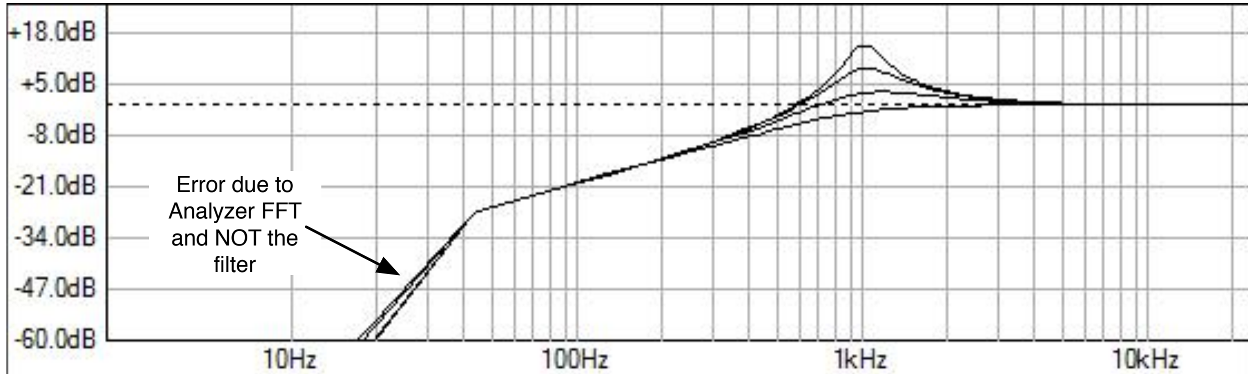

Figure 7.18: the Korg35 HPF emulation with fc = 1kHz and K = 0.2643, 1.0, 1.5, and 1.75, NLP = OFF

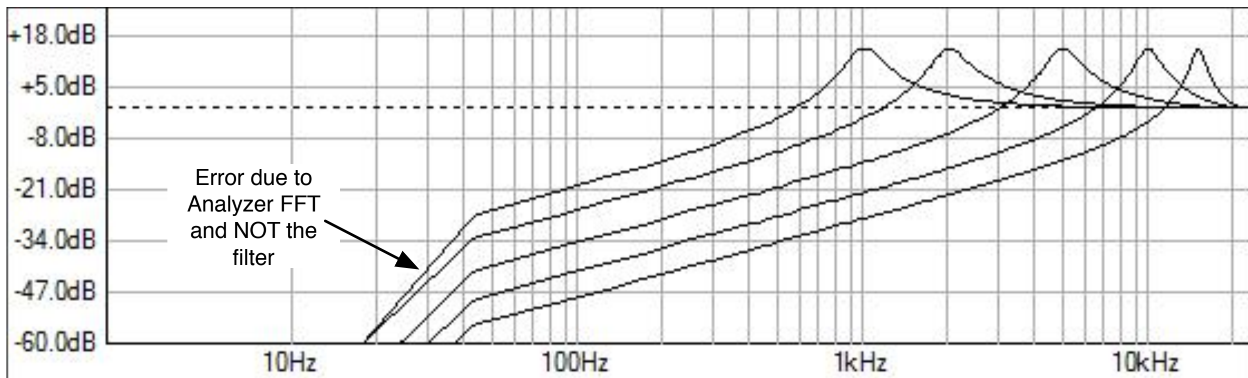

Figure 7.19: the Korg35 emulation with fc = 1kHz, 2kHz, 5kHz, 10kHz and 15kHz. On the original Korg35, fc is variable from 50Hz to 15kHz. NLP = OFF

Sample Code

The sample code is a typical RackAFX project called *KorgThreeFiveHPF.* I created an object to encapsulate the VAOnePoleFilter, similar to App Note 4. The filter self oscillates at K = 2.0 but with either NLP on (budget or naive) or off (no limiting) the oscillation will turn into a square wave. With NLP on, the waveshaper does this. With it off, the filter's output gain is far beyond the +/-1 boundaries of the Plug-In system. The filter will remain stable, however. Again, you may wish to limit K to 1.99 on the upper end, depending on your taste.

Copyright © 2013 Will Pirkle

In the .h File

```
	// Add your code here: ------------------------------------------------- //
	CVAOnePoleFilter m_HPF1;
	CVAOnePoleFilter m_HPF2;

	// HPF->LPF = BPF in FB loop
	CVAOnePoleFilter m_LPF1;
	CVAOnePoleFilter m_HPF3;

	// HPF->LPF = BPF in parallel for 6dB/oct slope (normal Korg35 HPF)
	CVAOnePoleFilter m_LPF2;
	CVAOnePoleFilter m_HPF4;


	// 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 CKorgThreeFiveHPF::prepareForPlay()
{
	// Add your code here:
	// use this is f you want to let filters update themselves;
	// since we calc everything here, it would be redundant
	/*
	m_LPF1.m_fSampleRate = (float)m_nSampleRate;
	m_LPF2.m_fSampleRate = (float)m_nSampleRate;
	m_LPF3.m_fSampleRate = (float)m_nSampleRate;
	m_HPF1.m_fSampleRate = (float)m_nSampleRate;
	*/

	// set types
	m_HPF1.m_uFilterType = HPF1;
	m_HPF2.m_uFilterType = HPF1;

	// FB BPF
	m_LPF1.m_uFilterType = LPF1;
	m_HPF3.m_uFilterType = HPF1;

	// parallel BPF
	m_LPF2.m_uFilterType = LPF1;
	m_HPF4.m_uFilterType = HPF1;
```

Copyright © 2013 Will Pirkle

```
	// flush everything
	m_HPF1.reset();
	m_HPF2.reset();
	m_LPF1.reset();
	m_HPF3.reset();
	m_LPF2.reset();
	m_HPF4.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 CKorgThreeFiveHPF::updateFilters()
{
	// use this is f you want to let filters update themselves;
	// since we calc everything here, it would be redundant
	/*
	m_LPF1.m_dFc = this->m_dFc;
	m_LPF2.m_dFc = this->m_dFc;
	m_LPF3.m_dFc = this->m_dFc;
	m_HPF1.m_dFc = this->m_dFc;

	m_LPF1.updateFilter();
	m_LPF2.updateFilter();
	m_LPF3.updateFilter();
	m_HPF1.updateFilter();
	*/

	// 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_HPF2.m_fAlpha = G;
	m_LPF1.m_fAlpha = G;
	m_HPF3.m_fAlpha = G;
	m_LPF2.m_fAlpha = G;
	m_HPF4.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 = (m_dK*G - m_dK)/fDenominator;
	m_HPF2.m_fBeta = m_dK/fDenominator;
```

Copyright © 2013 Will Pirkle

```
        m_HPF3.m_fBeta = m_dK*G/fDenominator;
        m_LPF1.m_fBeta = m_dK/fDenominator;
        m_HPF4.m_fBeta = m_dK*G/fDenominator;
        m_LPF2.m_fBeta = m_dK/fDenominator;

        // calc our G value; see App Note for details
        if(m_uHPFType == SixdB) // original Korg35 version
                G35H = (m_dK - m_dK*G)/(1.0 - m_dK*G + m_dK*G*G);
        else // optional 12dB/octave version
                G35H = (m_dK - 2.0*m_dK*G + m_dK*G*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
- add budget NLP if enabled
- auto-normalize by 1/K

```
double CKorgThreeFiveHPF::doFilter(double xn)
{
        // FIRST: form feedback and feed forward values (read before write)
        if(m_uHPFType == SixdB)
                S35H = m_HPF1.getFeedbackOutput() - m_HPF2.getFeedbackOutput() -
                       m_HPF3.getFeedbackOutput() + m_LPF1.getFeedbackOutput() -
                       m_HPF4.getFeedbackOutput() + m_LPF2.getFeedbackOutput();
        else
                S35H = m_HPF1.getFeedbackOutput() - m_HPF2.getFeedbackOutput() -
                       m_HPF3.getFeedbackOutput() + m_LPF1.getFeedbackOutput();

        // y = G35Hx + S35H
        double y = G35H*xn + S35H;


        // NAIVE NLP
        if(m_uNonLinearProcessing == ON && m_uNLPType == naive)
        {
                // Normalized Version
                if(y >= 0)    // positive half first:
                        y = (1.0/tanh(m_dSaturation))*tanh(m_dSaturation*y);
                else
                        // lower half adds 1.25x the saturation
                        // uncomment to try asymmetrical clipping
                        // y = (1.0/tanh(1.25*m_dSaturation))*tanh(1.25*m_dSaturation*y);

                        // default is symmetrical
                        y = (1.0/tanh(m_dSaturation))*tanh(m_dSaturation*y);

                /*
                // Regular Version
                if(y >= 0)    // positive half first:
                        y = tanh(m_dSaturation*y);
                else
                        // lower half adds 1.25x the saturation]
                        // uncomment to try asymmetrical clipping
                //      y = tanh(1.25*m_dSaturation*y);
```

Copyright © 2013 Will Pirkle

```
                    // default is symmetrical
                    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 two HPFs
    m_HPF2.doFilter(m_HPF1.doFilter(xn));

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

    // process x through parallel BPF
    if(m_uHPFType == SixdB)
            m_LPF2.doFilter(m_HPF4.doFilter(xn));

    // BUDGET NLP
    if(m_uNonLinearProcessing == ON && m_uNLPType == budget)
    {
            // Normalized Version
            y = (1.0/tanh(m_dSaturation))*tanh(m_dSaturation*y);

            /*
            // Regular Version
            y = tanh(m_dSaturation*y);
            */
    }

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

    return y;
}
```

Analog Simulation

Before coding any of the Virtual Analog filters (App Notes 4,5,6,7) I simulate the digital emulation in analog. If the analog simulation works as I expect, then I can move on to the coding. The simulations are done in CircuitMaker. 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 7.20 shows the circuit used in the simulation of the Korg35 HPF (original version) while Figure 7.21 shows the simulated frequency response. Time domain analysis confirms oscillation at K = 2.0.

Copyright © 2013 Will Pirkle

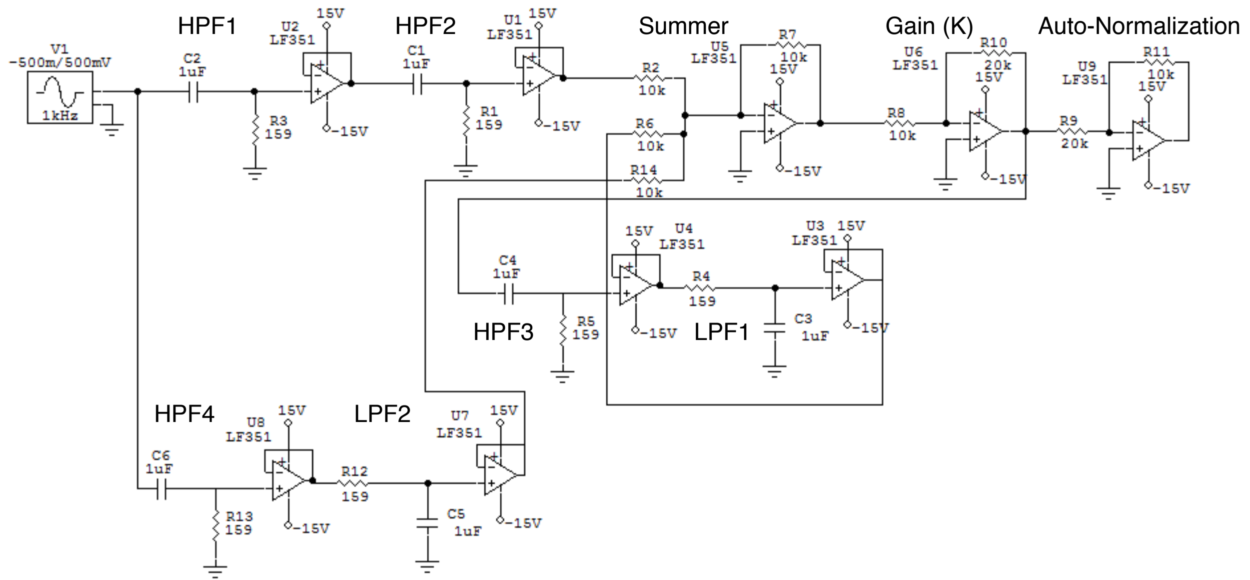

Figure 7.20: CircuitMaker simulation of the digital emulation; this is shown with K = 2.0 (R10 and op amp U6) R2, R6, and R14 sum the three signal paths together

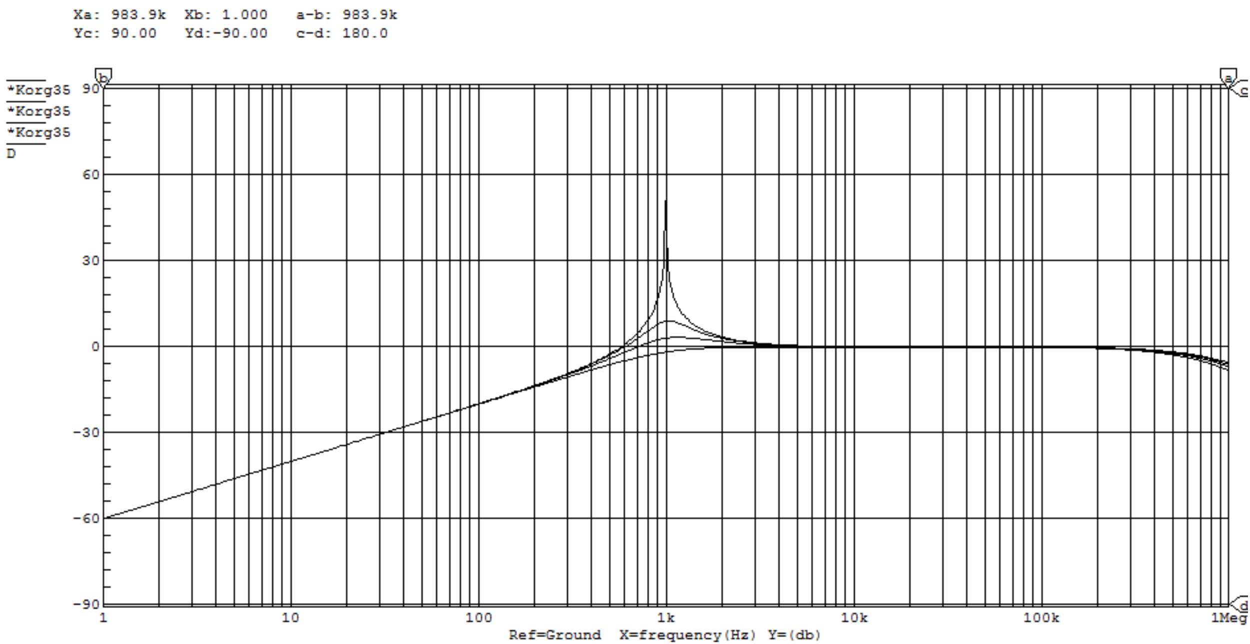

Figure 7.21: simulated frequency response with K = 0.2643, 1.0, 1.5, and 2.0; note the 6dB/octave rolloff

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:

Copyright © 2013 Will Pirkle

The original Korg35 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).

Nonlinear Models

Figure 7.22 shows the block diagram of the complete filter with the naive implementation of NLP, while Figure 7.23 shows the budget implementation. Landscape versions are also provided. My preference is to keep the NLP block in the feedback path which stays closer to the original and soft-clips the feedback signal, use the standard tanh() function rather than the normalized version, and adjust saturation and K values accordingly. I also implement oversampling to lessen the severity of aliasing. Other nonlinear functions should also be investigated. A future App Note will address nonlinear processing in more detail.

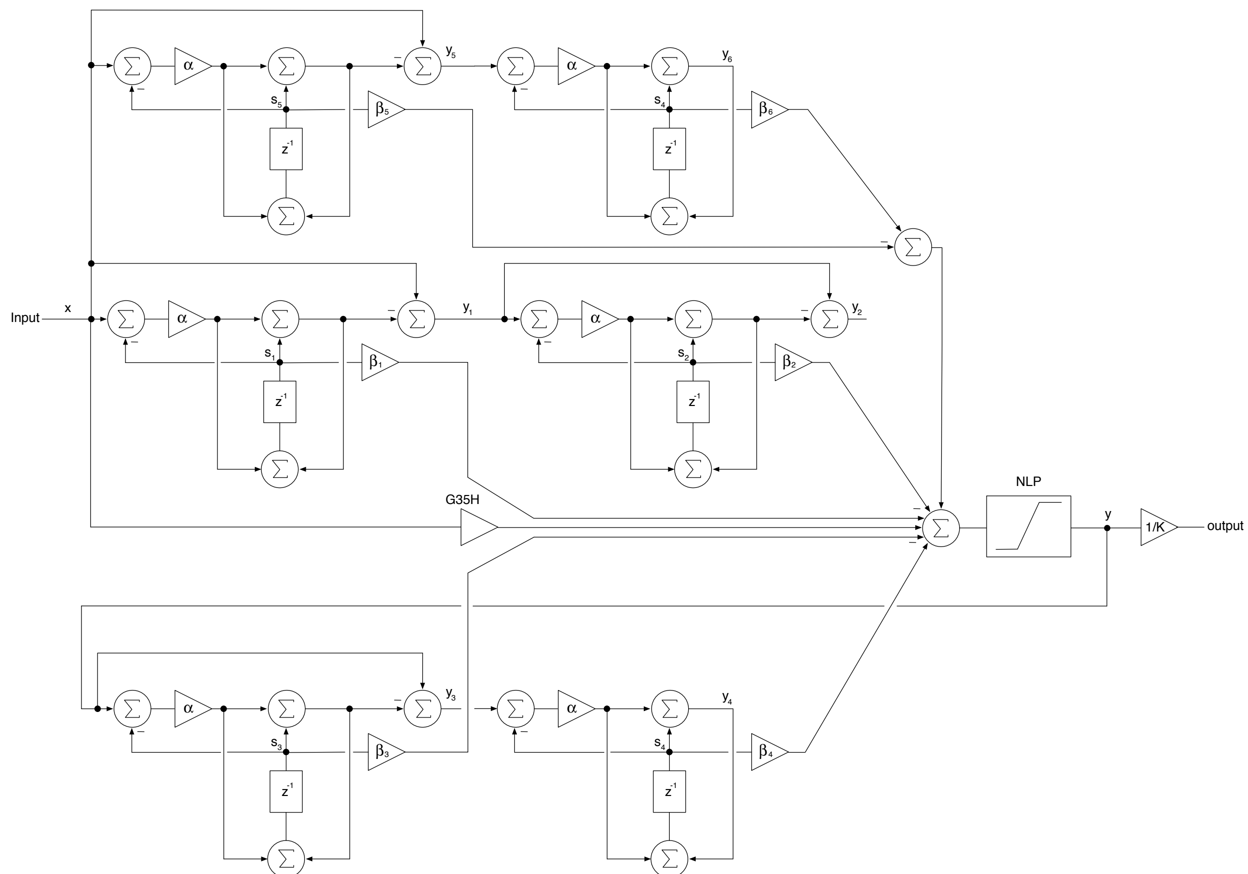

Figure 7.22: naive NLP implementation (landscape version on next page)

Copyright © 2013 Will Pirkle

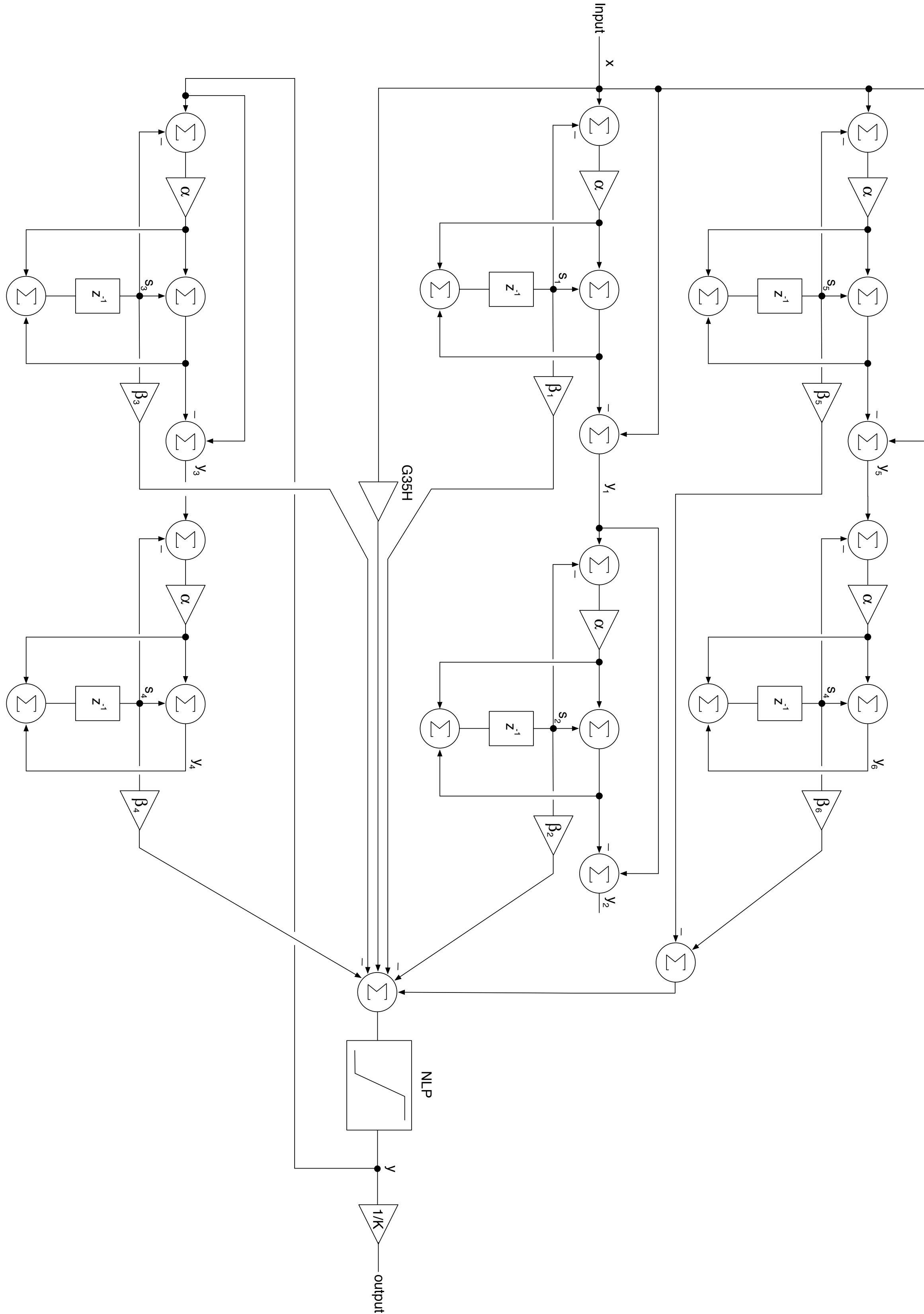

Figure 7.22: naive NLP, landscape

Copyright © 2013 Will Pirkle

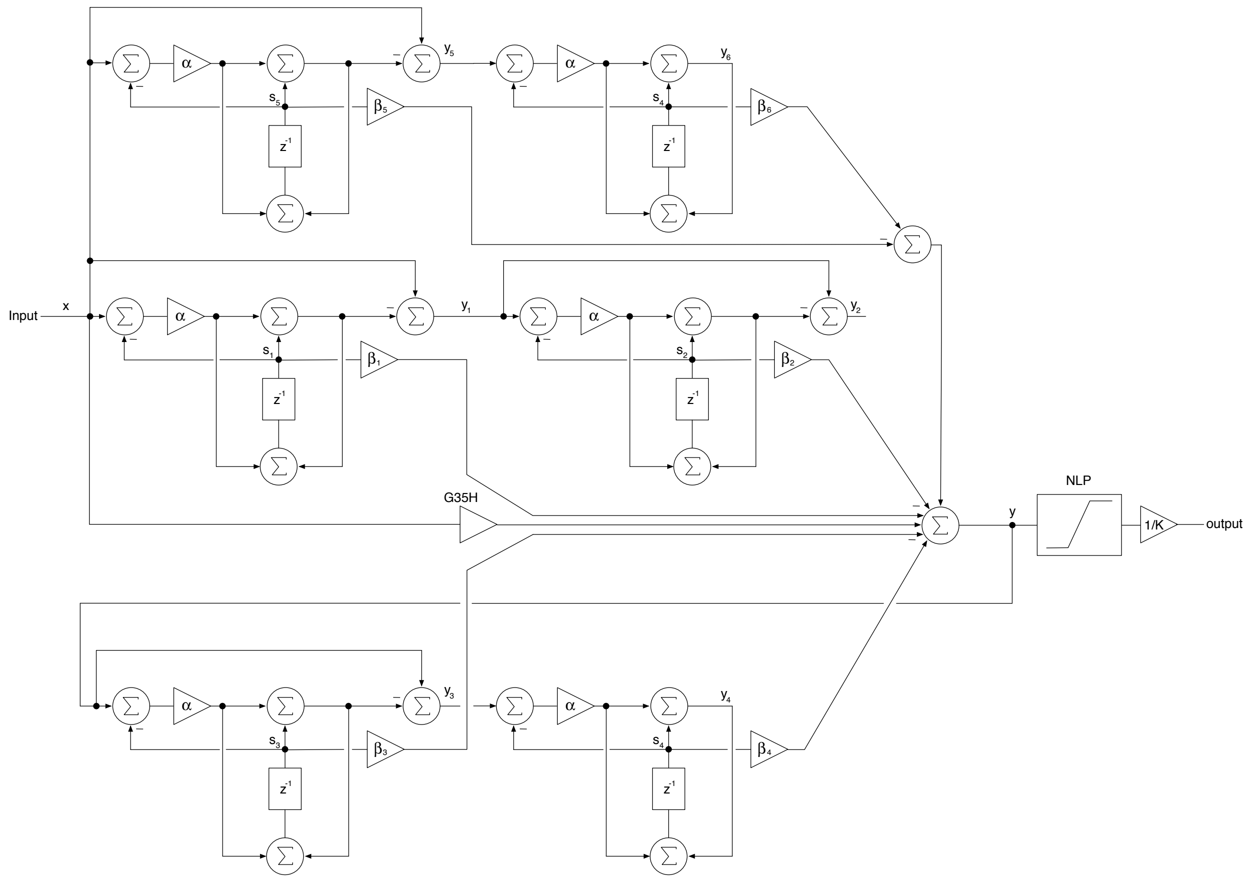

Figure 7.23: budget NLP implementation (landscape version on next page)

Copyright © 2013 Will Pirkle

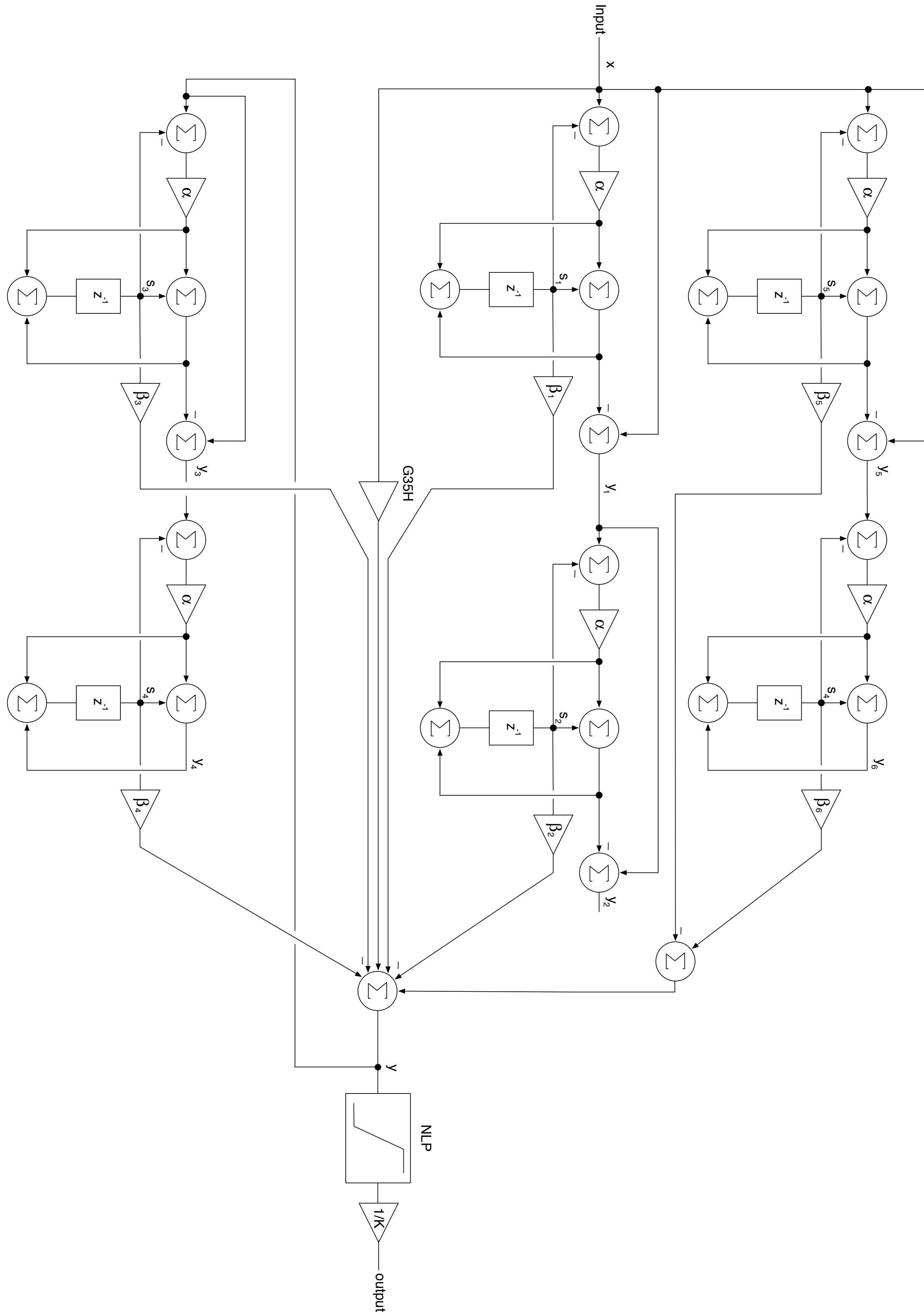

Figure 7.23: budget NLP (landscape)

Copyright © 2013 Will Pirkle

Revision History:

1.1: Initial Release, *July 25, 2013*
1.2: added Huovilainen reference and changed min Q to 0.5 rather than 0.707
1.3: added note about peak value differences between the 6 and 12 dB versions
1.4: corrected block diagrams to remove redundant K amplifier; all equations and code remain unchanged

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