Virtual Analog (VA) Korg35 Lowpass Filter

Will Pirkle

July 18, 2013

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.

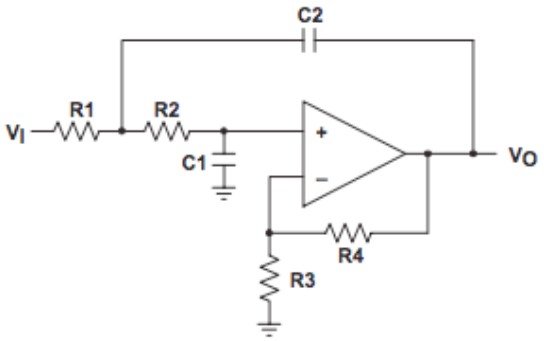

Figure 5.1: The Sallen-Key LPF

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).

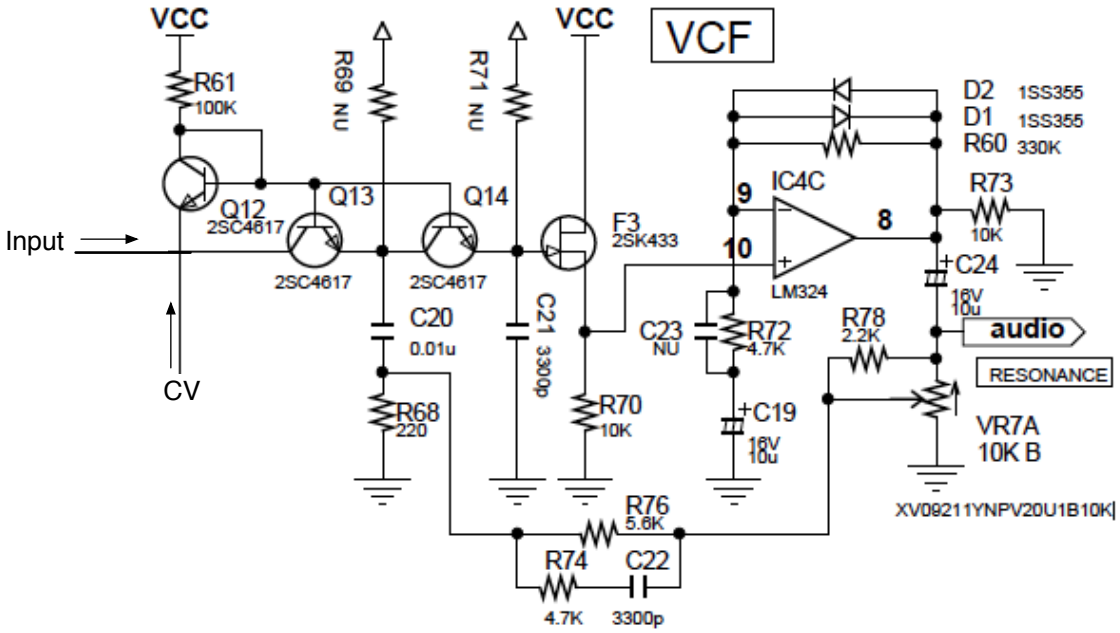

Figure 5.2: The Korg35 lowpass filter in the Monotron

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. 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 5.3 shows the generalized Sallen-Key topology.

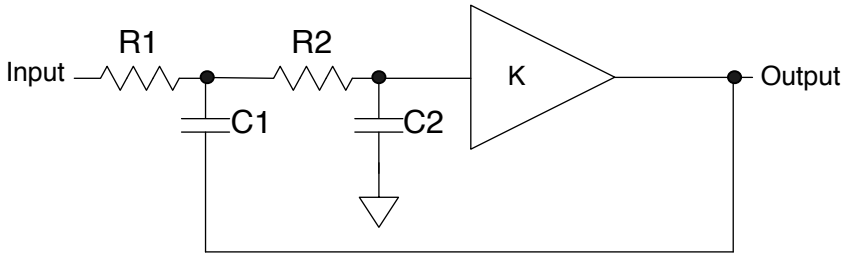

Figure 5.3: The Sallen-Key lowpass filter without the op-amp details

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.

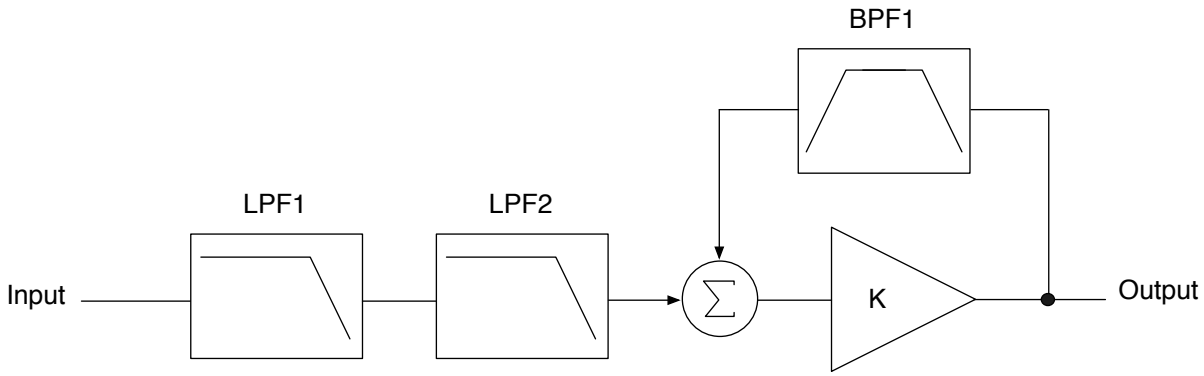

Figure 5.4: The Sallen-Key Block Diagram

LPF 1 and LPF2 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 5.4 could be redrawn as Figure 5.5. Note all filters (including the two that make the BPF) are synchronously tuned.

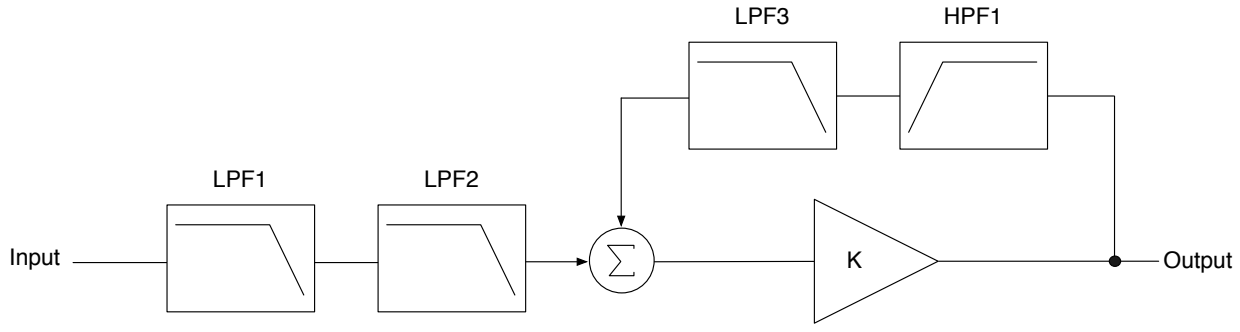

Figure 5.5: The Korg35 lowpass filter topology.

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 5.5 to get Figure 5.6 - the completed model. Placing the NLP block here 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).

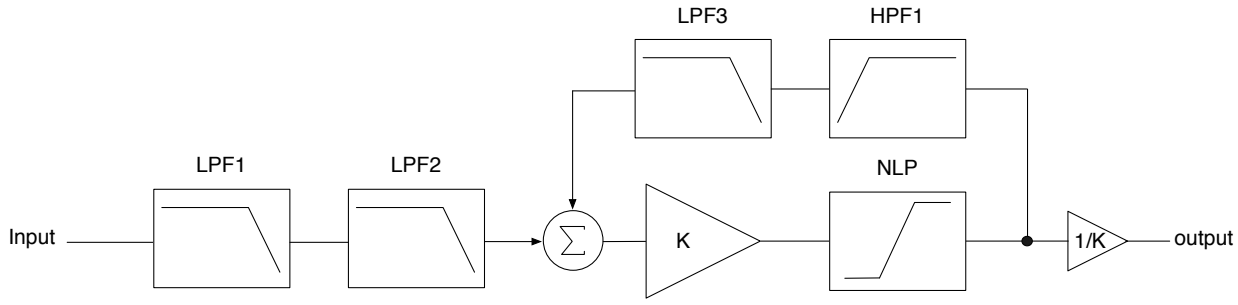

Figure 5.6: The completed Korg35 lowpass filter Block Diagram

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

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

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

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

$$H_0 = K$$

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

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

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

It can also be shown that

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

For Butterworth (maximally flat magnitude) Q = 0.707, K = 0.5285. 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. Self oscillation occurs when K = 2.0. 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 K vary from 0.3333 to 2.0 (in the sample code, K is fixed on the range of [0.3333...2.0] but you may wish to experiment with over-damped variations by allowing K to go below 0.3333; obviously, K can not go all the way to 0.0). Note - if you refer to the Stinchcombe reference, you will see that due to loading factors in the original design, the self oscillation occurs at K = 2.333; since our VA LPF and HPF filters do not suffer from loading, we can stick with K = 2.0 as the oscillation point.

VA Korg35 Design Equations - Linear Model

The block diagram in Figure 5.6 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 5.5 with intermediate nodes:

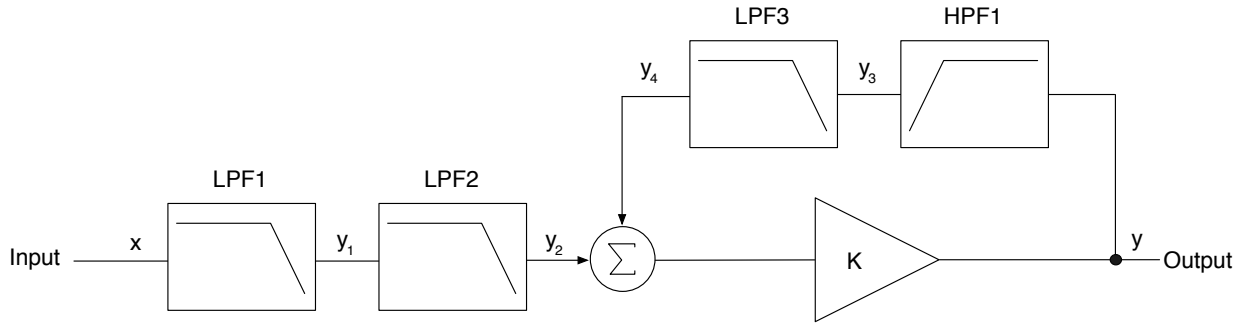

Figure 5.7: The nodes are labeled without the (n) notation for convenience

The circuit equations are then found as:

LPF1:

$$y_1 = Gx + S1$$

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

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

LPF2:

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

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

HPF1:

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

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

LPF3:

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

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

The final output is then

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

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

$$y = K(G^2x + GS1 + S2 + Gy - G^2y - GS3 + S4)$$

$$y - KGy + KG^2y = KG^2x + KGS1 + KS2 - KGS3 + KS4$$

$$y = \frac{KG^2x + KGS1 + KS2 - KGS3 + KS4}{1 - KG + KG^2}$$

Or in more familiar VA terms:

$$y = G35x + S35$$

$$G35 = \frac{KG^2}{1 - KG + KG^2}$$

$$S35 = \frac{KGS1 + KS2 - KGS3 + KS4}{1 - KG + KG^2}$$

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 (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.

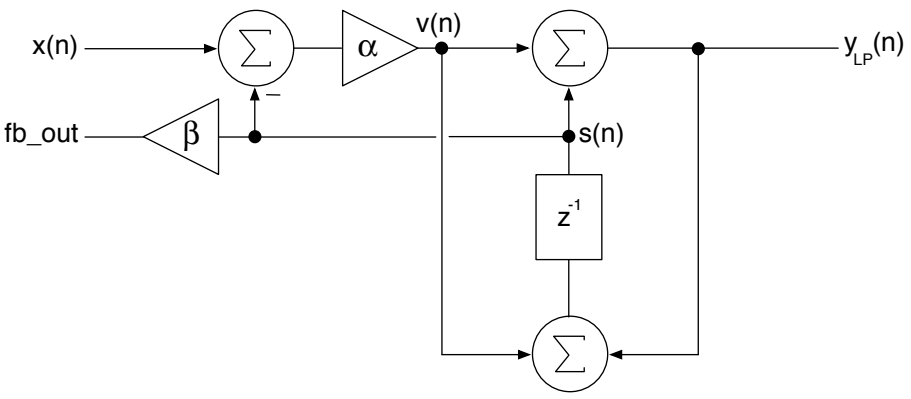

Figure 5.8: 1st order TPT LPF Block Diagram

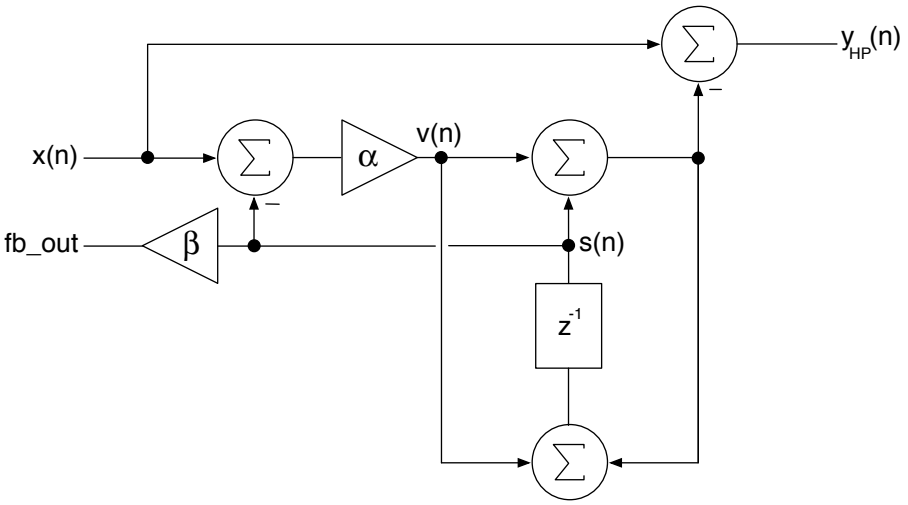

Figure 5.9: 1st Order TPT HPF Block Diagram

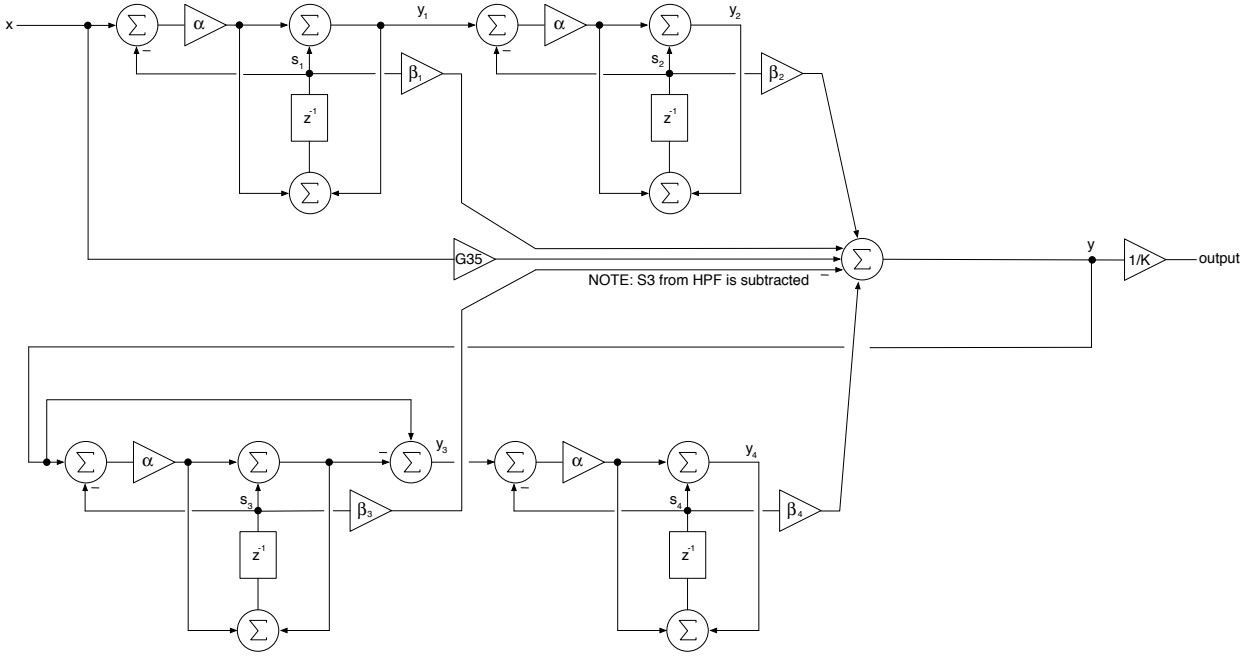

Figure 5.10: The completed VA Korg35 lowpass filter - Linear Model

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

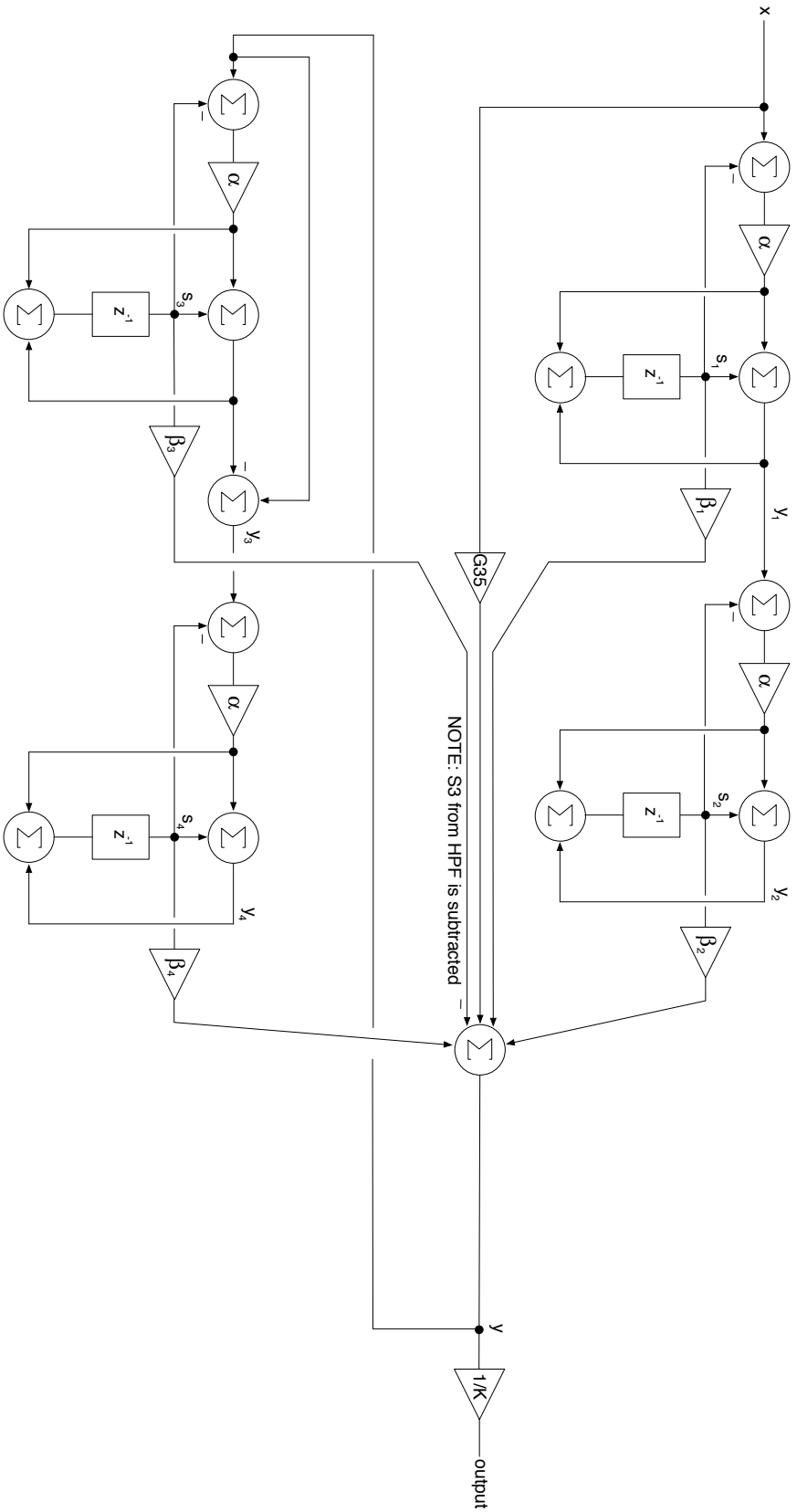

Figure 5.10: the completed Linear Model (landscape)

Figure 5.10 shows the completed filter auto-normalizing (1/K) coefficient included. Interestingly, you can see that the outputs of LPF2 and LPF4 don't lead anywhere. We took care of that when we resolved the delay-less feedback loop. Look at the main summer that feeds the loop. Make sure you can figure out that it implements the equation

$$y = G35x + S35$$

where the alpha and beta coefficients are as follows:

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

$$G35 = \frac{KG^2}{1 - KG + KG^2}$$

$$\beta_1 = \beta_3 = \frac{KG}{(1+g)(1 - KG + KG^2)}$$

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

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 the line from the HPF (S3) is subtracted as per the equation above.

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 5.6. 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

$$y = \tanh(K(y_2 + y_4))$$
$$= \tanh(K(G^2x + GS1 + S2 + Gy - G^2y - GS3 + S4))$$

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, but would do nothing to stop the overload/distortion caused by the massive gain near self-oscillation. Figure 5.11 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.

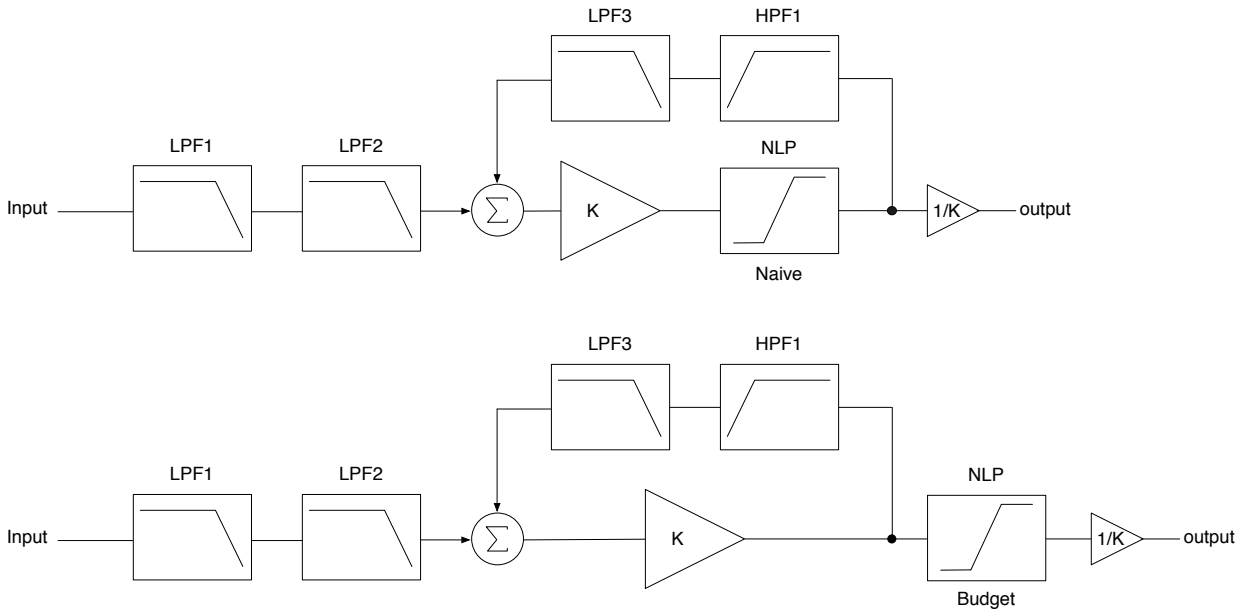

Figure 5.11: the naive NLP implementation places the block in the feedback loop while the budget version pulls it outside

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 5.12.

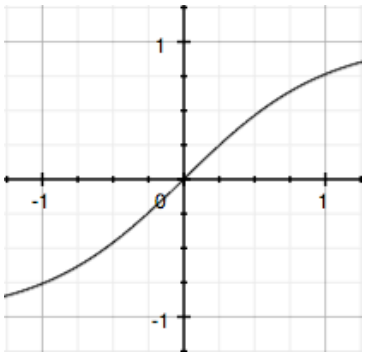

Figure 5.12: tanh(x) produces about 0.8 when x = 1

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

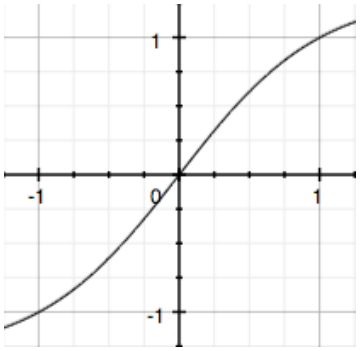

Figure 5.13: The normalized tanh() function transfer curve

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 5.14 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.*

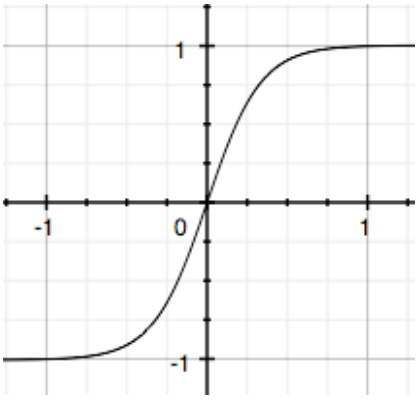

Figure 5.14: A steeper curve is obtained with *sat* > 1, here it is 3

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.

Asymmetrical Resonance - an approximation (naive implementation only)
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.

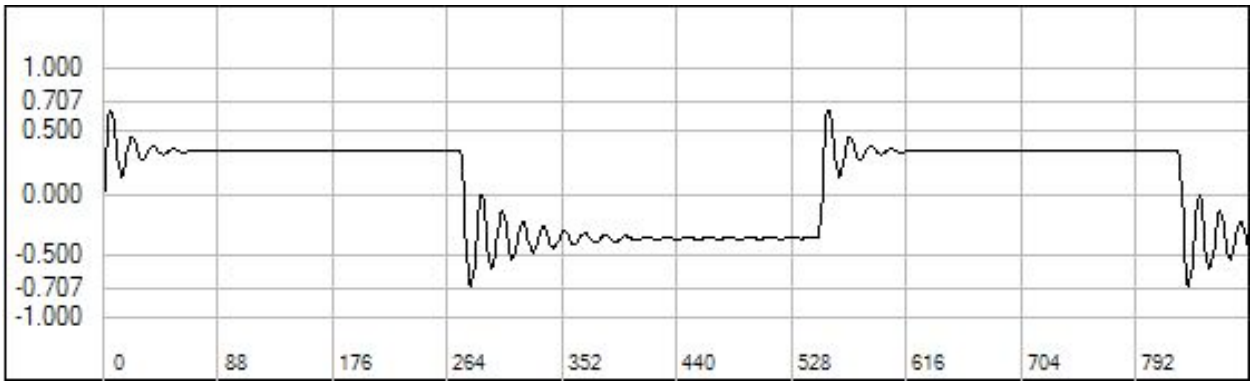

Figure 5.15: the asymmetrical response obtained with the asymmetrical clipping

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 and there may be tuning errors at high frequencies. 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 but the output will be severely distorted due to clipping in the filter. You can add saturation to smooth 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 so that you can get self-oscillation right at K = 2.0.

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 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.

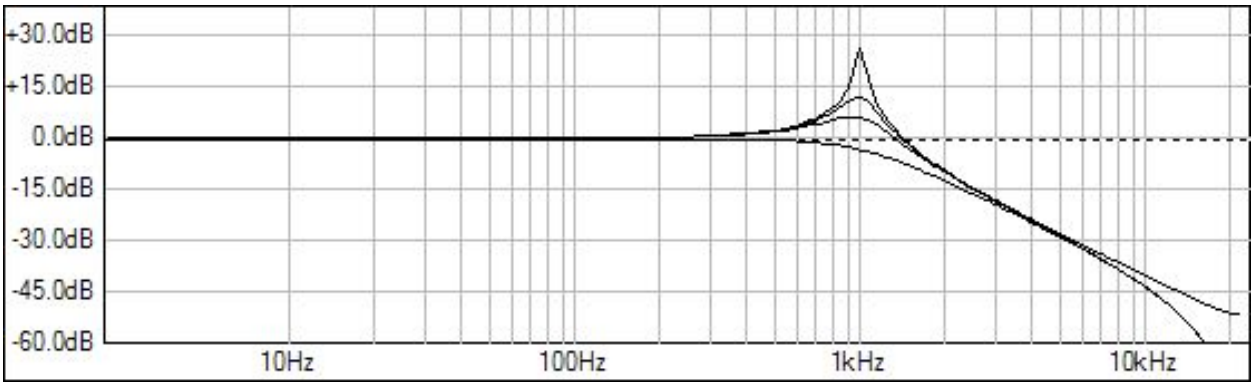

Figure 5.16: the Korg35 emulation with fc = 1kHz and K = 0.5285, 1.0, 1.5, and 1.75, NLP = OFF

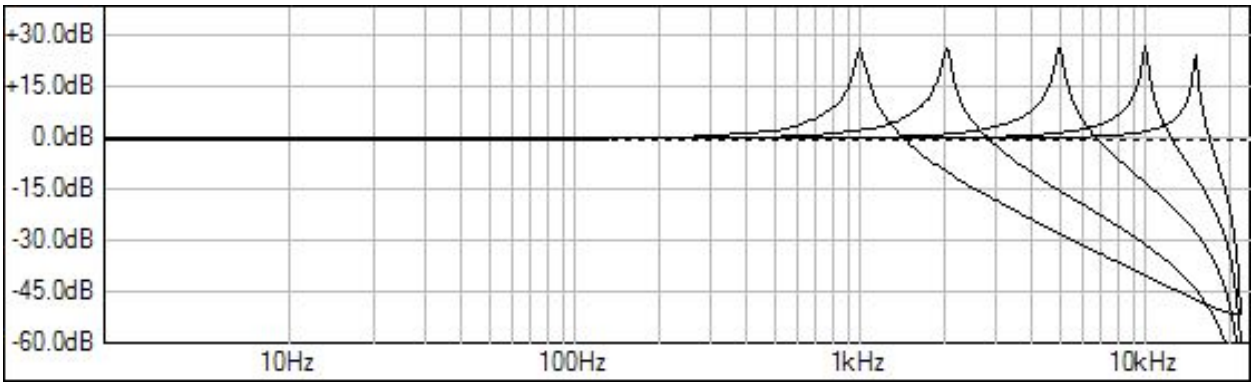

Figure 5.17: the Korg35 emulation with fc = 1kHz, 2kHz, 5kHz, 10kHz and 15kHz. On the original Korg35, fc is variable from 50Hz to 15kHz. NLP = OFF

Sample Code

The sample code is a typical RackAFX project called *KorgThreeFiveFilter.* 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 or off 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. You may wish to limit K to 1.99999 on the upper end, depending on your taste.

In the .h File

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

	// HPF->LPF = BPF
	CVAOnePoleFilter m_LPF3;
	CVAOnePoleFilter m_HPF1;

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

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

	// variables
	double G35; // our G value
	double S35; // 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 CKorgThreeFiveFilter::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_LPF1.m_uFilterType = LPF1;
	m_LPF2.m_uFilterType = LPF1;
	m_LPF3.m_uFilterType = LPF1;
	m_HPF1.m_uFilterType = HPF1;

	// flush everything
	m_LPF1.reset();
	m_LPF2.reset();
	m_LPF3.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 G35 coefficient

```
void CKorgThreeFiveFilter::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_LPF1.m_fAlpha = G;
	m_LPF2.m_fAlpha = G;
	m_LPF3.m_fAlpha = G;
	m_HPF1.m_fAlpha = G;

	// set betas all are in the form of  <something>/((1 + g)(1 - kG + kG^2))
	m_LPF1.m_fBeta = m_dK*G/((1.0 + g)*(1.0 - m_dK*G + m_dK*G*G));
	m_LPF2.m_fBeta = m_dK/((1.0 + g)*(1.0 - m_dK*G + m_dK*G*G));
	m_HPF1.m_fBeta = m_dK*G/((1.0 + g)*(1.0 - m_dK*G + m_dK*G*G));
	m_LPF3.m_fBeta = m_dK/((1.0 + g)*(1.0 - m_dK*G + m_dK*G*G));

	// calc our G value; see App Note 5
	G35 = 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 CKorgThreeFiveFilter::doFilter(double xn)
{
	// FIRST: form feedback and feed forward values (read before write)
	S35 = m_LPF1.getFeedbackOutput() + m_LPF2.getFeedbackOutput() -
		m_HPF1.getFeedbackOutput() + m_LPF3.getFeedbackOutput();

	// y = G35x + S35
	double y = G35*xn + S35;

	// 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);

			// 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 LPFs
	m_LPF2.doFilter(m_LPF1.doFilter(xn));

	// process y through the feedback path
	m_LPF3.doFilter(m_HPF1.doFilter(y));

	// 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). Figure 5.18 shows the circuit used in the simulation of the Korg35 LPF while Figure 5.19 shows the simulated frequency response. Time domain analysis confirms oscillation at K = 2.0.

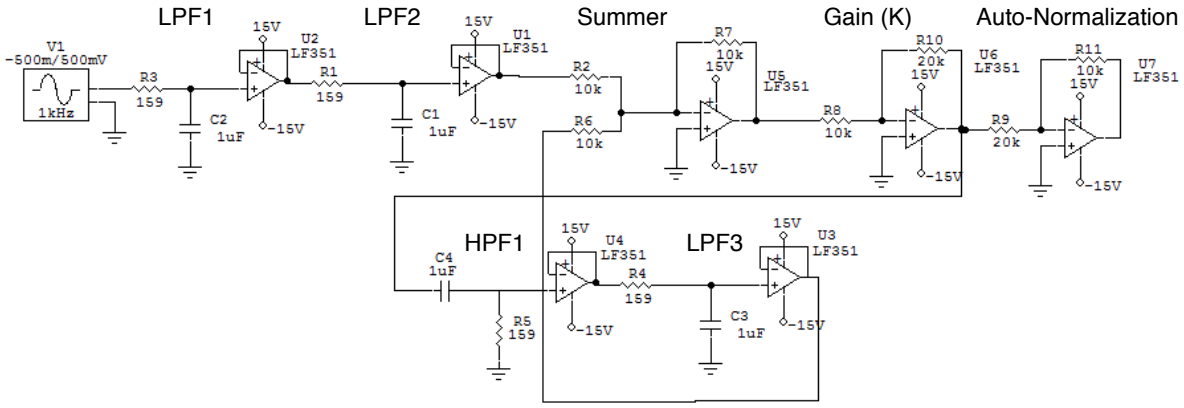

Figure 5.18: the CircuitMaker circuit used to simulate the digital emulation of the Korg35 LPF; the circuit is shown with K = 2.0 (controlled with R10 in the U6 sub-circuit) - there is no NLP block in the simulation.

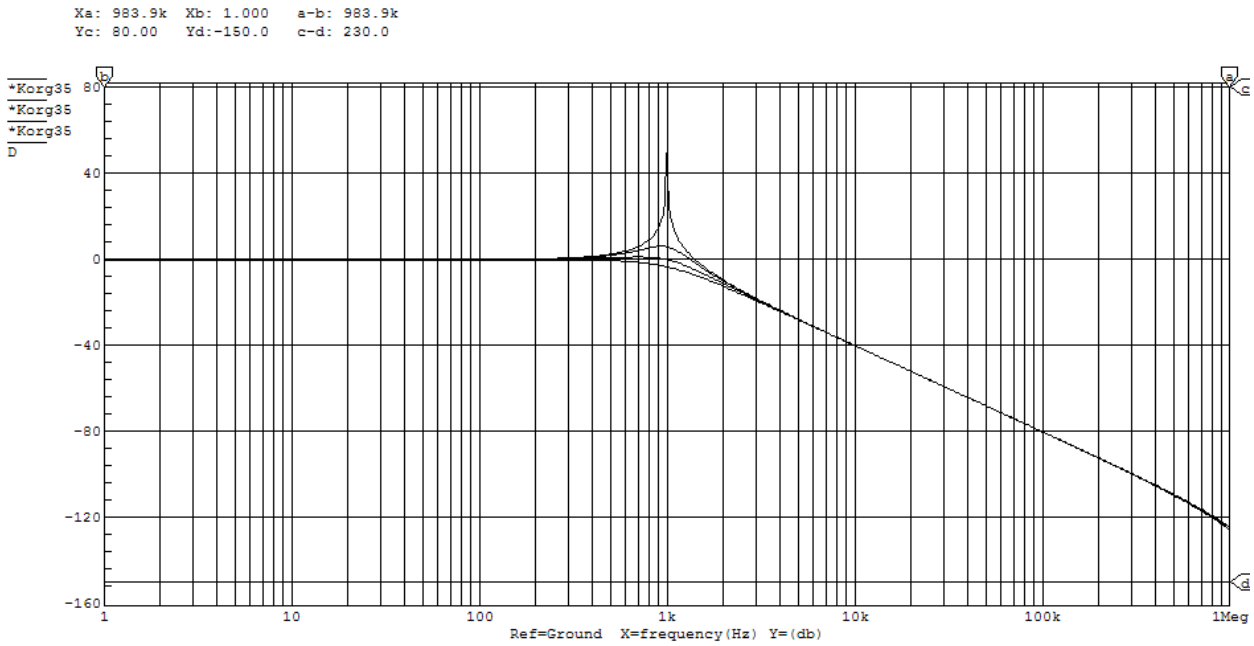

Figure 5.19: simulated frequency responses with K = 0.5285, 1.0, 1.5 and 2.0

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.

Nonlinear Models

Figure 5.20 shows the block diagram of the complete filter with the naive implementation of NLP, while Figure 5.21 shows the budget implementation. Landscape versions are also provided. It can not be emphasized enough that the NLP implementation is the tricky part of this emulation. 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.

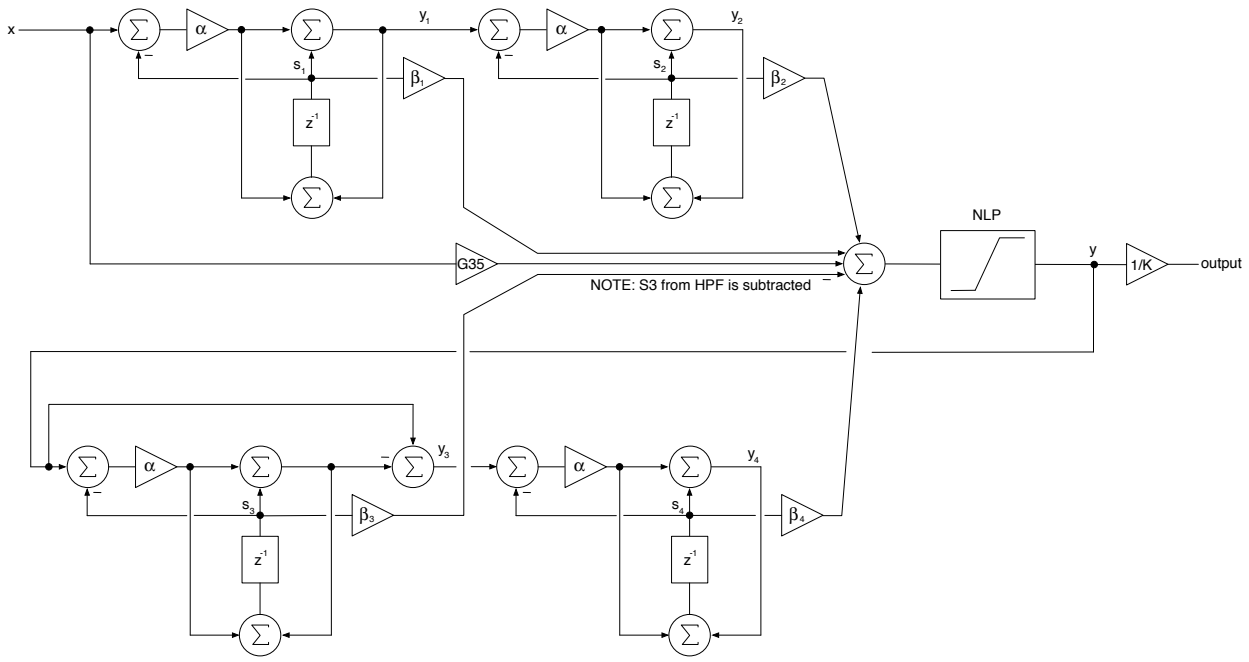

Figure 5.20: naive NLP implementation (landscape version on next page)

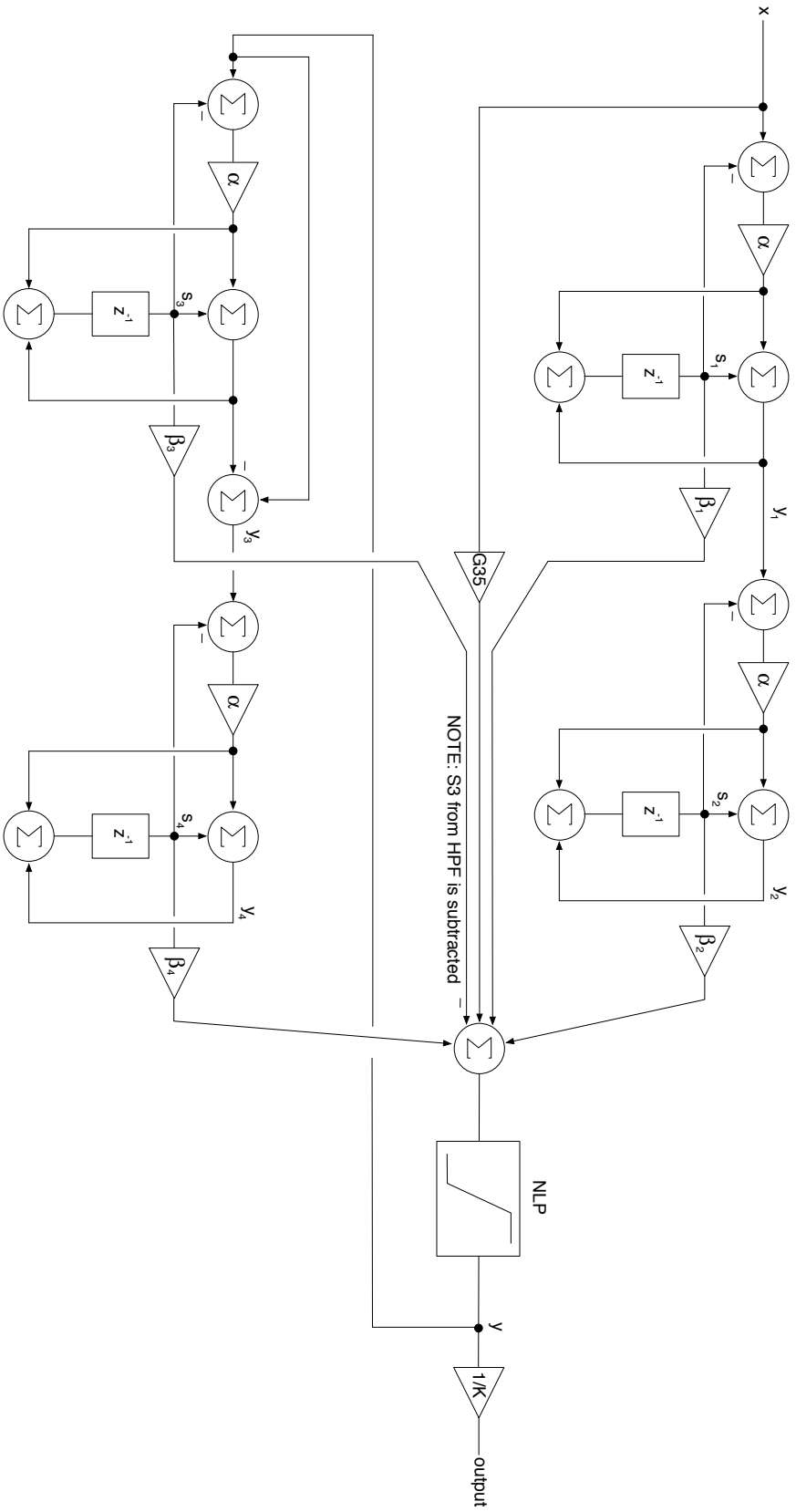

Figure 5.20: naive NLP, landscape

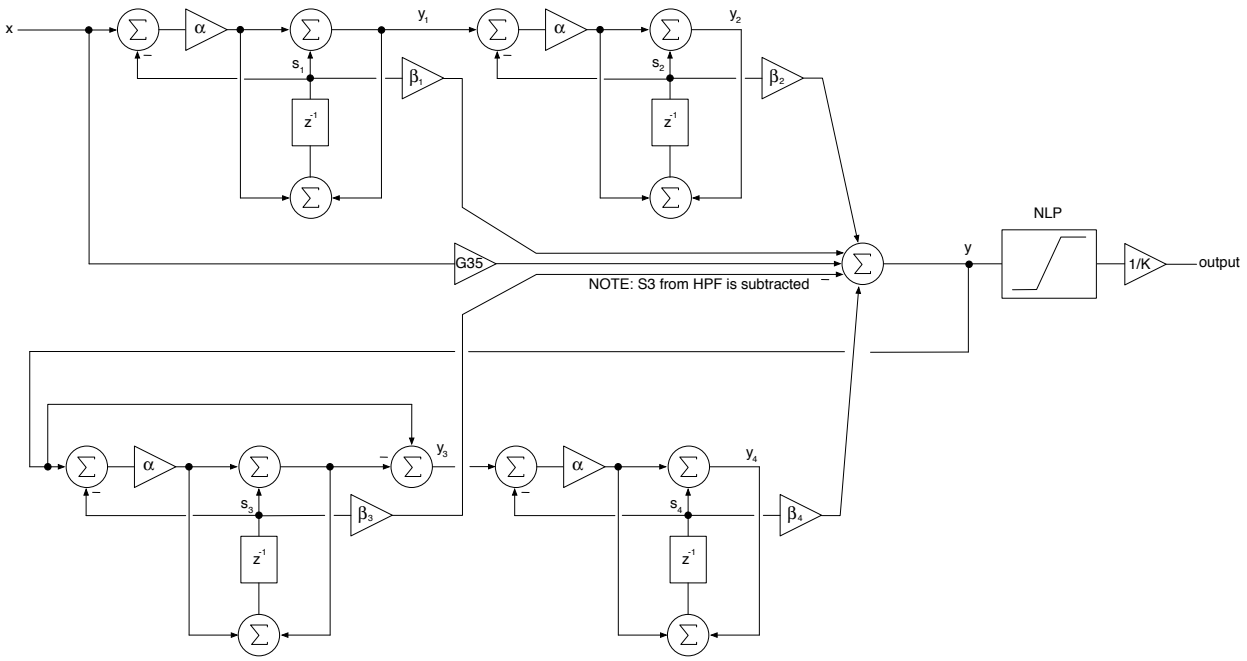

Figure 5.21: budget NLP implementation (landscape version on next page)

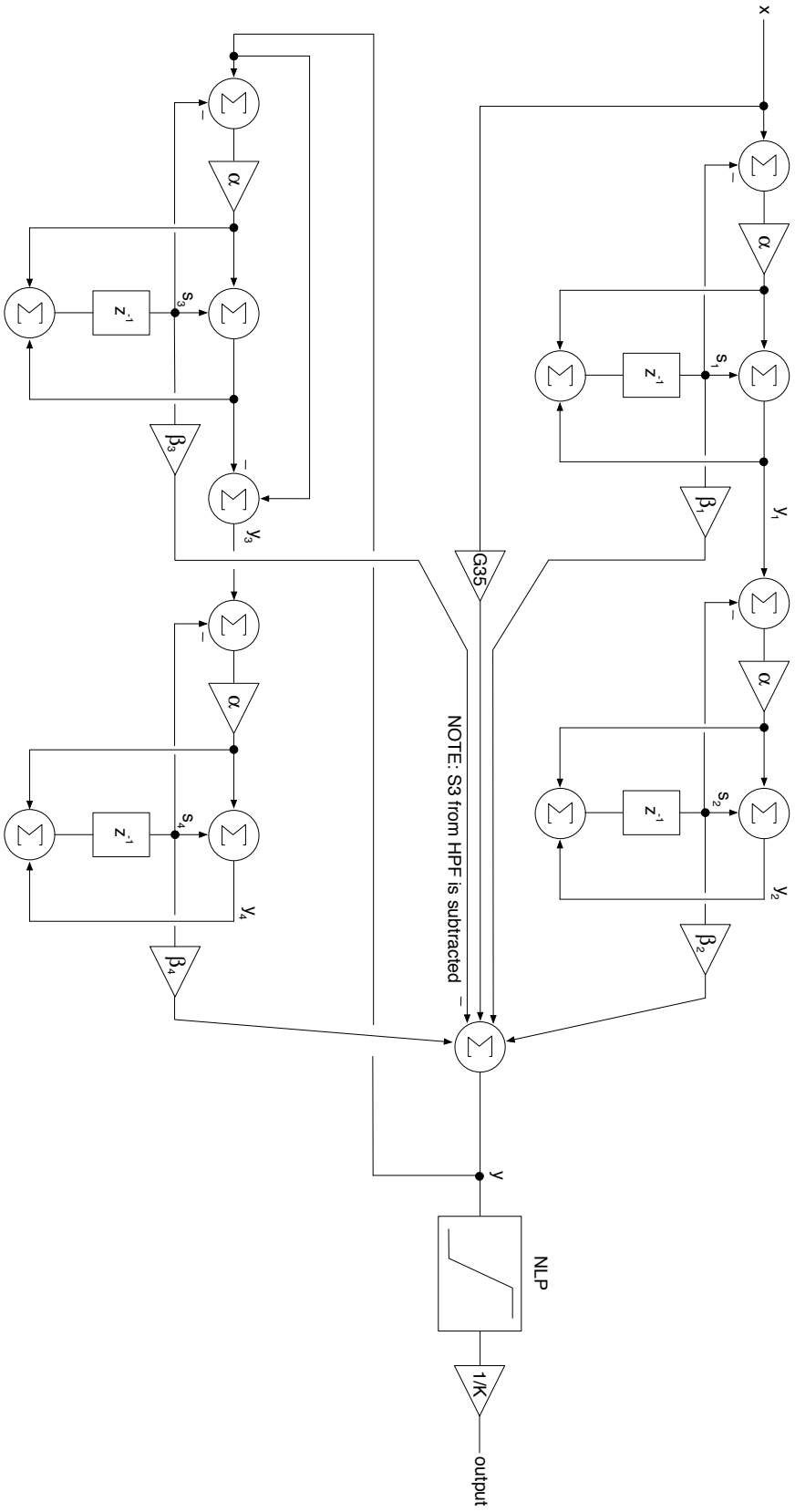

Figure 5.21: budget NLP (landscape)

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)

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