A11 *IIRFilters*: The Moog Ladder Filter Biquad Style

Robert Moog's Ladder Filter (aka the *Moog Ladder Filter* or *Moog LPF*) is a 4th order lowpass filter designed specifically as an analog voltage-controlled filter circuit. In his original design, a control voltage was used to set and modulate the cutoff frequency f_c. The filter's Q was controlled by a potentiometer and was not modulate-able. The filter exhibits a 4th order slope roll-off of -24dB/octave. However as the Q is raised above 0.707, the filter's overall gain drops, producing a massive reduction in bass frequencies. In addition, you can see in Figure A11.1 that although the cutoff frequency is set to 1kHz, it is clear that neither the -3dB point nor the resonant frequency are 1kHz for low Q settings. This means that the resonant frequency of the filter changes (albeit slightly) as the Q is increased, which is evident in Figure A11.1.

These anomalies, that EE professors would scorn as un-usable, are exactly what makes the Moog LPF one of the most musical sounding filters ever designed. Figure A11.1 shows how the overall gain changes with increase in filter Q. In addition, the Q may be increased all the way to infinity, placing the filter poles on the unit circle and causing the filter to go into self-oscillation. In older analog synths, it was not uncommon to use the filter as an oscillator itself. The heartbeat sound in the *Stranger Things* soundtrack is actually a filter in self-oscillation at a low frequency with modulation applied.

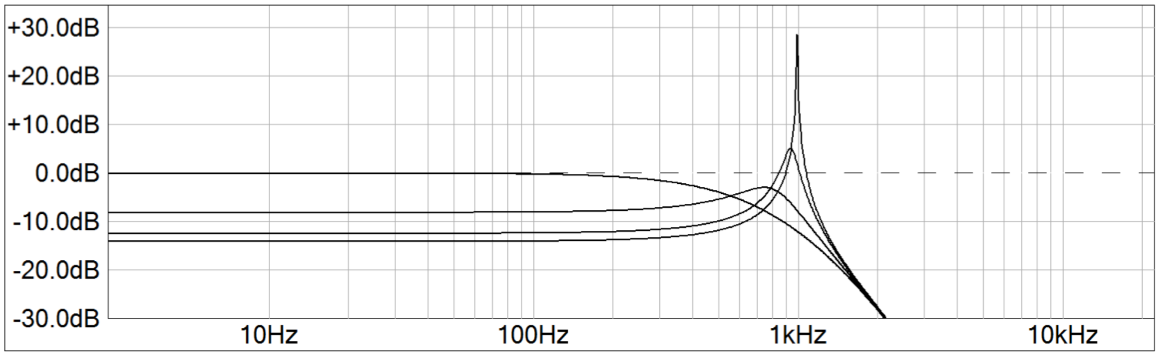

Figure A11.1: the Moog LPF with f_c = 1kHz and a variety of Q settings; as the resonance increases, the filter gain drops

A11.1 The 1st Order Filter Revisited

To understand how the Moog LPF works, you first have to go back to the old 1st order LPF. The plots for the frequency and phase responses are shown in Figure A11.2. For the Moog filter is it crucial to understand the *phase response'*s action on the signal.

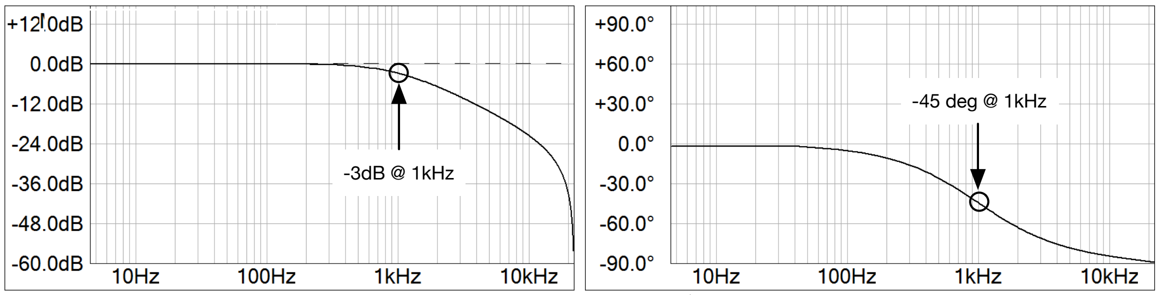

Figure A11.2: frequency and phase responses for a 1st order LPF

In Figure A11.2 you can see that the phase is shifted by -45 degrees at the cutoff frequency. In the past we've simply accepted this fact and moved on – but Moog bases the architecture of his filter on this phase-response-feature.

A11.2 The Moog LPF Architecture

Figure A11.3 shows the block diagram for the Moog LPF. Notice the following features:

- There are 4 first-order LPFs in series; their cascade produces the 4th order rolloff response
- The output of the serial chain of filters is fed back into the input, scaled by *K*, and *subtracted* from it; another way to think of that is that the feedback signal is scaled by –*K* and *added* to the input
- The *K* control is the *Q* control; when *K* = 0 the *Q* = 0.2 and when *K* = -4 the *Q* = infinity (self oscillation)
- The f_c is controlled by synchronously tuning the 4 LPFs to the new f_c; when you do that, the bandwidth of the filter shrinks and you can see this in Figure A11.1 on the lowest *Q* curve; the -3db point is far below 1kHz because of the shrinkage

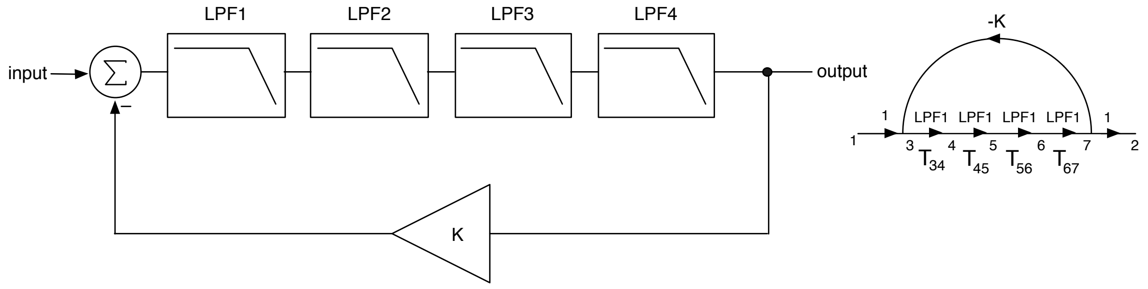

Figure A11.3: the Moog Ladder Filter block diagram and signal flow (right); the f_c is controlled by tuning the LPFs together and is omitted from the diagram

A11.3 The Moog LPF Operation

How does it work? The phase shift at the cutoff frequency is -45 degrees for a first order LPF stage. Each stage then adds another -45 degrees of phase shift. After going through four of these filters the phase shift at the cut-off frequency will be -180 degrees. The output is exactly out of phase with the input. The output is fed back into the input through a negative scalar -*K*, which flips the phase at the cut-off so it is in phase with the input. This amplifies the cutoff frequency and frequencies that are very close to it resulting in a resonant peak. If $K = 0$ there is no feedback and no resonance. As soon as *K* becomes non-zero, the *Q* increases and peaking occurs. When $K = 4$ we achieve 100% feedback through the loop; *K* needs to be 4 (rather than 1) because the feedback energy is spread across the four 1st order filters. When $K = 4$ the filter self oscillates. Figure A11.4 shows this graphically. Adding the re-inverted signal back to the input increases gain at that frequency which is the definition of resonance.

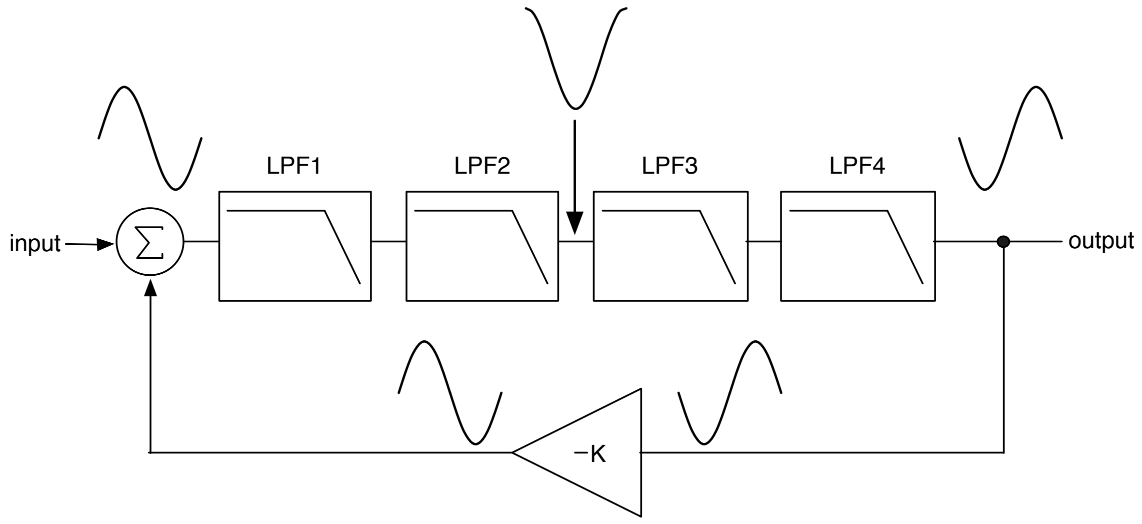

Figure A11.4 the phase of the input sinusoid oscillating exactly at f_c is shifted through the filters by -45 degrees each time; at the output of the 2nd filter, the signal is -90 degrees out of phase; at the output of the 4th filter, the signal is fully flipped in phase.

A11.3.1 Dude, Where's my Bass?

The Moog LPF decimates the bass frequencies as the *Q* is increased. Why? Go back to the phase response plot in Figure A11.1, and reprinted here in Figure A11.5. Notice that the low frequencies incur little or no phase shift through the filter.

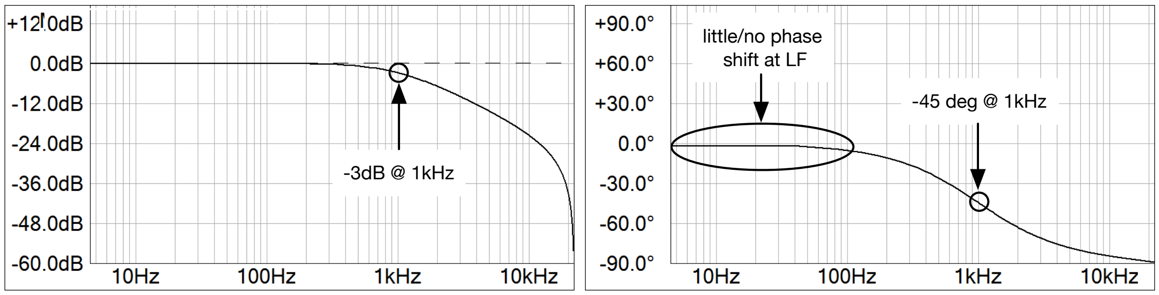

Figure A11.5: the low frequencies do not get phase shifted by very much

Since the low frequencies get little or no phase shift and then they are fed back through a negative coefficient that inverts them, these frequencies are then ***subtracted*** from the input causing a reduction in the amplitudes of these low frequencies. You can accommodate for the bass loss by amplifying the signal prior to sending it through the series of filters. You can also scale the amplifier with a coefficient that allows any amount of bass compensation from fully off (normal operation, with bass losses) or fully on (full bass, watch out for distortion).

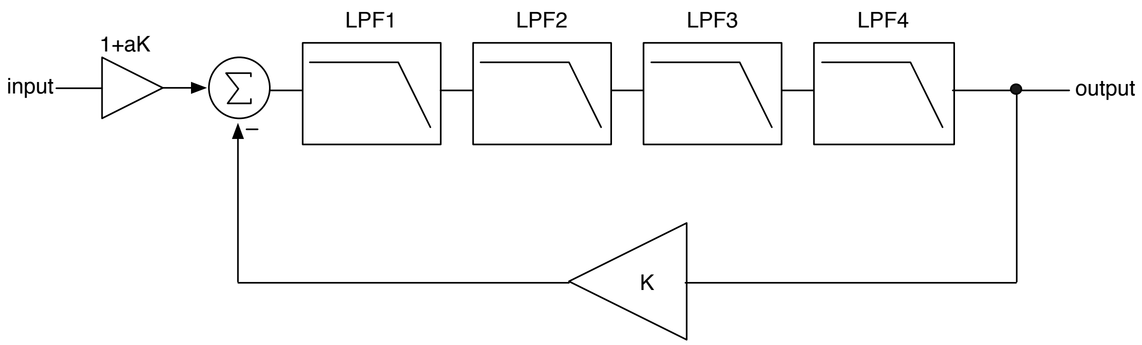

Figure A11.6: an input amplifier scaled with a coefficient *a* on the range [0.0, 1.0] can be used to restore some ($0.0 < a < 1.0$) or all ($a = 1.0$) of the bass response

A11.4 Taming the Self Oscillation

When the Moog LPF goes into self-oscillation, all of its poles are distributed on the unit circle. In theory, a single impulse will set the filter into oscillation. However, the amplitude of the oscillation is dependent on the amplitude of the impulse. When driving the filter with audio, either synthesized or a full range soundtrack, the filter's oscillation amplitude depends greatly on the frequency content of the signal. When the frequency content of the signal matches closely to the resonant self-oscillation frequency, the self-oscillation can become repeatedly re-energized. This causes the oscillations to grow and grow in amplitude, eventually clipping out a digital system: note that the clipping is not occurring within the algorithm itself. The pure sinusoidal output of the filter is growing to far exceed the [-1.0, +1.0] limits of the digital system and ***this*** is causing the clipping. It turns out that there are nonlinearities in the Moog LPF, which act as a kind of built-in limiter. Typically, you model these nonlinearities by placing waveshaper blocks within the structure. There have been several different variations on the waveshapers. Välimäki and Zavalishin both proposed "cheap" implementations using a single waveshaper; Välimäki places the waveshaper outside of the loop and Zavalishin places it inside the loop.

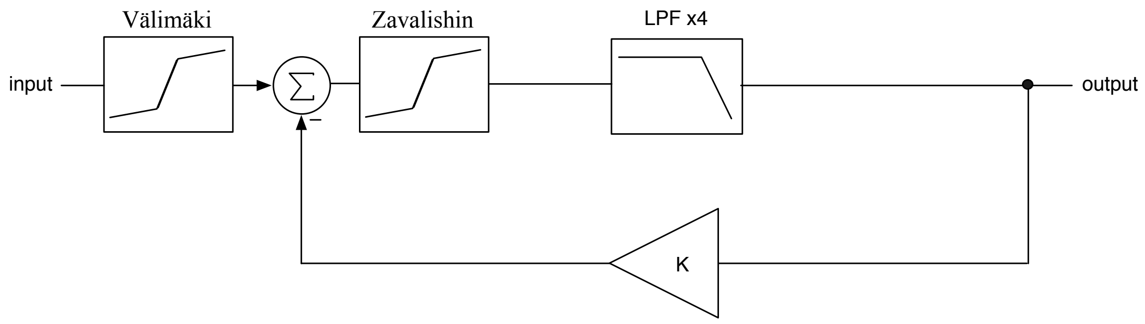

Figure A11.7: two different locations for a non-linear waveshaper within the Moog Ladder Filter structure

Figure A11.8 shows the affect of applying the nonlinear waveshaper to the Moog LPF during self-oscillation. You can see that the output becomes soft and rounded, what I call "test-tube shaped." This soft clipping can sound good (or bad) and prevents hard clipping from occurring.

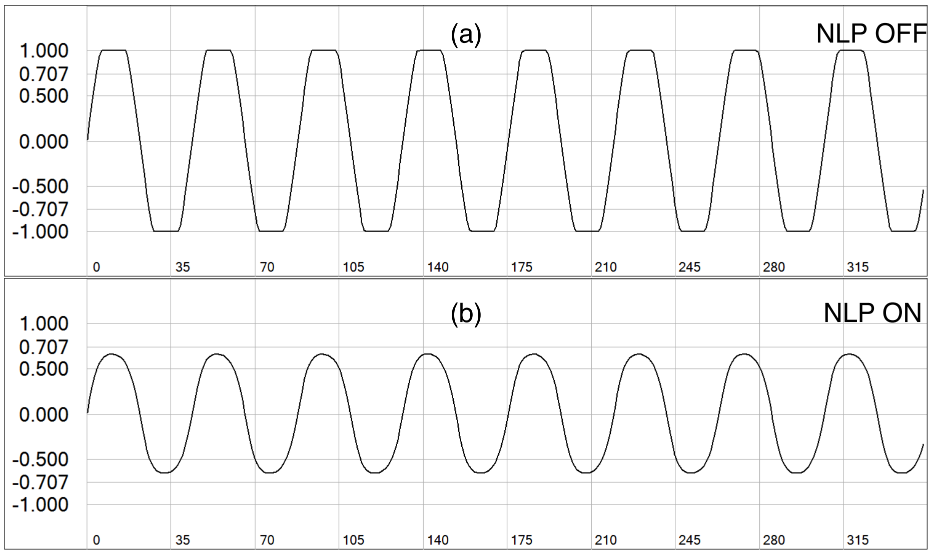

Figure A11.8: the Moog LPF with nonlinear processing (NLP) turned off and on; here the NLP is a hyperbolic tangent waveshaper (*tanh*)

A11.4.1 My Distortion Solution: Hard Limiting

I have spent years dealing with the Moog LPFs nonlinearities and self-oscillation using waveshapers. The fundamental issue I have is my own experience; when I hear an analog Moog LPF self oscillate, to me it sounds like a nearly perfect sinusoid. The waveshapers always throw out extra harmonics, so you have to oversample or do other stuff to try to alleviate the aliasing, the analog Moog doesn't go into clipping. In addition, placing waveshapers within the main loop, or the feedback loop of any sub-filter may cause instabilities if feedback values cause the poles to leave the unit circle, and it may also result in the filter cutoff frequency drifting and being incorrect.

My solution is to avoid the waveshapers altogether and use a hard peak-limiter on the output. It tames the oscillations and gives pure-sinusoidal tones instead – no oversampling required. This is shown in Figure A11.9. In Figure A11.10, you can see the effect of the auto-limiter – check out how pure the sinusoids look compared to the fat and squashed version in Figure A11.8 (b).

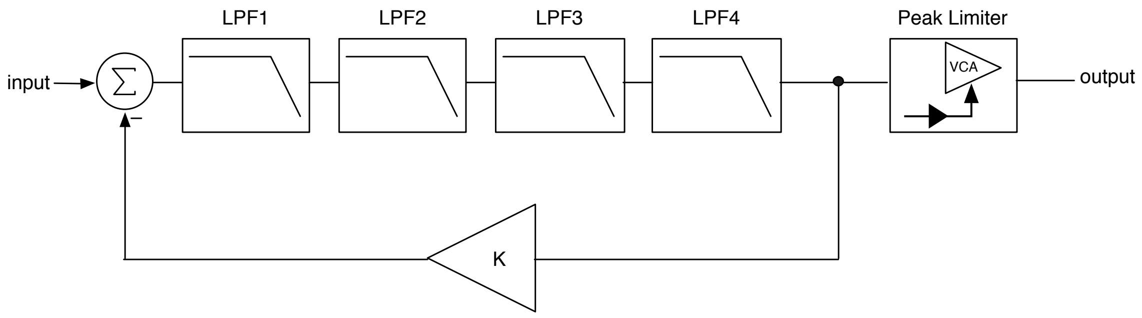

Figure A11.9: Pirkle's solution is to use a peak limiter with threshold set to just *under* 0.0dBFS with fast attack and release times

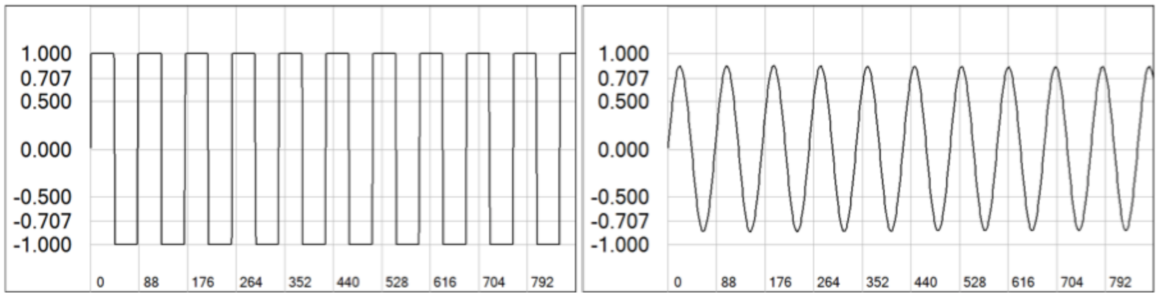

Figure A11.10: filter output in self-oscillation without peak limiter (left) and with auto-limiting (right)

For the limiter there are a few requirements:

- detect PEAK envelope value
- convert detected value to dB
- allow detection of signals > 0dBFS
- attack = 5mSec
- release = 25mSec

Setting up the *AudioDetector* for this is easy:

```
// --- init; true = analog time-constant
detector.setSampleRate(_sampleRate);

AudioDetectorParameters detectorParams = detector.getParameters();

detectorParams.detect_dB = true;
detectorParams.attackTime_mSec = 5.0;
detectorParams.releaseTime_mSec = 25.0;
detectorParams.clampToUnityMax = false;
detectorParams.detectMode = ENVELOPE_DETECT_MODE_PEAK;

detector.setParameters(detectorParams);
```

A11.5 Delay-Free Loop Resolution

A fundamental design issue with the Moog LPF is that the block diagram reveals a delay-free loop from the last filter output back to the input. This feedback loop is not the same as in a delay line, where we queue up samples for outputting from a delay-loop. In the delay-free case, we have a difference equation as shown below where *f( )* is the function of low-pass filtering through four series filters.

$$y(n) = f(Ky(n) + x(n))$$

You can see that *y(n)* is on both sides of the equation. This is equivalent to saying "the current output *y(n)* depends on the current output *y(n)* and ..." In other words, you need to know *y(n)* to calculate *y(n)*. Note that this is only a problem in digital algorithms; analog systems with continuous (delay-free) loops are commonplace and the methods for solving them (either differential equations or Laplace transform) are well known and long established.

In 2012, Zavalishin published a book on Virtual Analog Filter Design (we use it as a reference in Synth Plugins class). In it, he outlined a method for resolving the delay free loops by temporarily throwing out the discrete time variable *n*, and then using simple algebra to solve the loop equation. Afterwards, you restore the discrete time variable. I personally had a lot of difficulty with this, mainly involving discarding the discrete time variable – we spend many university classes learning the effects of time-discretization and simply ignoring it was not an option for me. At the request of a friend Korg, I started back-researching the delay-free loop problem and found a paper by Härmä (1992) that demonstrated how to resolve one particular type of delay free loop that turns out to be the *opposite* of the Moog LPF structure; in the original Härmä architecture, the filtering or other processing stuff is in the feedback loop. For the Moog LPF, the filters are in the *feed-forward* branch with a feedback loop containing only a scalar multiplier.

In 2013, I derived a new version the Härmä method and published an AES paper on it; I named the new system the *Modified Härmä Method*. My extended version took into account any kind of delay-free loop architecture and showed how to solve it. That also includes nested structures of loops within loops. In that paper, I used the Moog Ladder filter with biquad structures as one of the example projects (the other is the Sallen-Key filter from the Korg MS-20).

For the project here, I am going to stick with a specific biquad structure. The Zavalishin version with trapezoidal integrators has already been "done to death" and there is plenty of information on it. So, this will show you the basics on how to use the biquad structures in delay-free loop systems that may be applied to any system, not just the Moog ladder filter. With a little practice of the Modified Härmä Method you can implement many interesting filters. I did this in the companion paper on *Hybrid Virtual Analog Filters.* See the bibliography for these references.

A11.5.1 The Biquad Structure as *Gx(n) + S(n)*

One of the things you need to do to use either my *Modified Härmä Method* or Zavalishin's algebraic method is to refactor the signal processing blocks so the difference equation has the form of:

$$y(n) = Gx(n) + S(n)$$

This rearranges the difference equation into two parts: the first part contains only the current input *x(n)* scaled by some coefficient *G*. The second part consists of all of the stuff that has been delayed, scaled by the appropriate coefficients. Figure A11.11 shows the two forms of the biquad structure that may be shoehorned into this arrangement. The first is the direct form and the second is the transposed canonical form. In both of these structures, you can see that the output is a combination of the input and some delayed sample(s). Make sure you understand how these structures may be refactored this way by observing that the output is the output of a summer that contains one current sample input plus one or more previous samples.

You can see that the two structures are different and there is more detail in the 2nd FX book. For modulated filters using biquads, the general belief is that the transposed canonical form produces the best/smoothest results with less noise or distortion. Some of this notion comes from the days fixed-point DSP ICs, which have the same issue with the output size range being [-1.0, +1.0] but unlike floating point systems, the fixed point range must always stay between -1.0 and +1.0 so the internal signal can't grow beyond those limits.

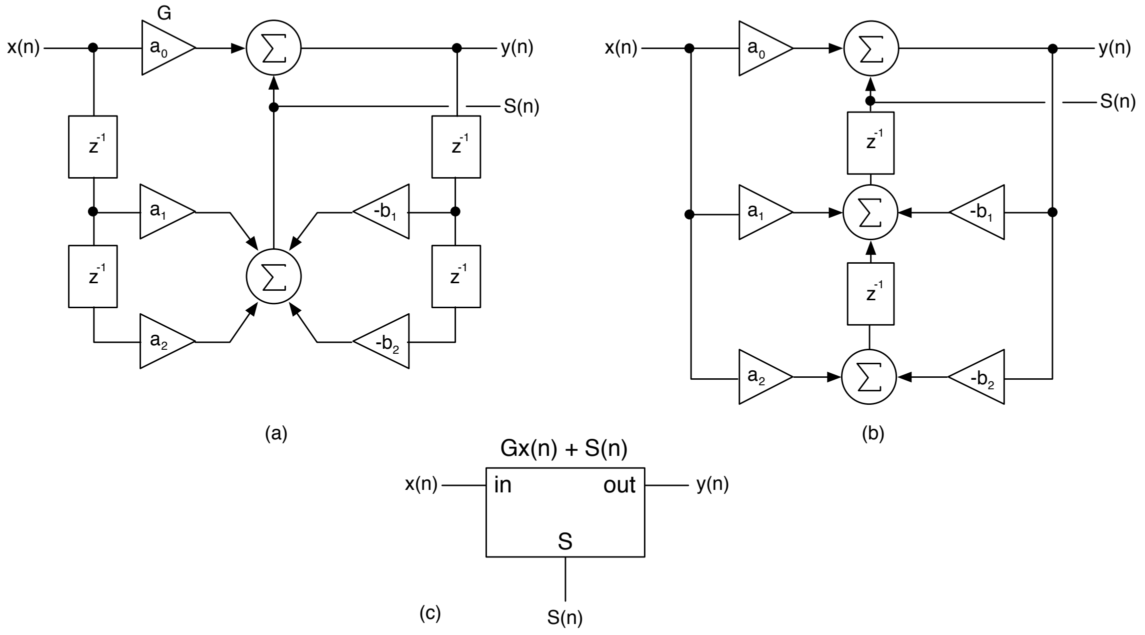

Figure A11.11 (a) the direct form and (b) transposed canonical forms allow refactoring of the difference equation into the form *y(n) = Gx(n) + S(n)* and (c) the simplified schematic symbol for a structure that can be refactored this way; the "S-port" at the bottom is where you access the delayed components

A *canonical* structure has exactly the same number of delays as the order of the transfer function *H(z)* that it implements; for *2nd* order systems, that would be two delay elements. This means that the direct form biquad structure has *redundant delays*. The effect of this is on the sensitivity to parameter variations – that is, how close the filter's performance is to the actual desired transfer function. Since cutoff frequency modulation is a kind of brutally forced "parameter variation" then we'd like the structure that is most accurate as the parameters are fluctuating. One advantage to both forms in Figure A11.11 is that the order of processing the numerator and denominator of the transfer function, or as some people say, the order of processing the zeros and poles of the transfer function. These forms both process the zeros (numerator) before the poles (denominator) and that usually results in a reduced probability of overflow errors. The canonical form is preferred – it requires less CPU operations and is more stable at the same time. The transposed canonical form is one of the most popular biquad structures for realizing real-time audio algorithms. For the *AudioFilter* object from the FX book, the default structure is the transposed canonical form, so that is automatically set when you use the object.

With the biquad in the form of Figure A11.11 (c) we can run the *Modified Härmä Method* and generate the block diagram in Figure A11.12. Notice that the original delay-free loop from the final filter output back to the input has been replaced with a set of feedback values taken directly from the *S(n)* output from the structure. You can also see some more coefficients have been added in addition to the feedback *K* value: the feedback branches are scaled before summing, and there is a new *alpha0* coefficient in the input path. These two results happen every time you resolve delay free loops using the *Modified Härmä Method* and this is covered in my paper (see bibliography).

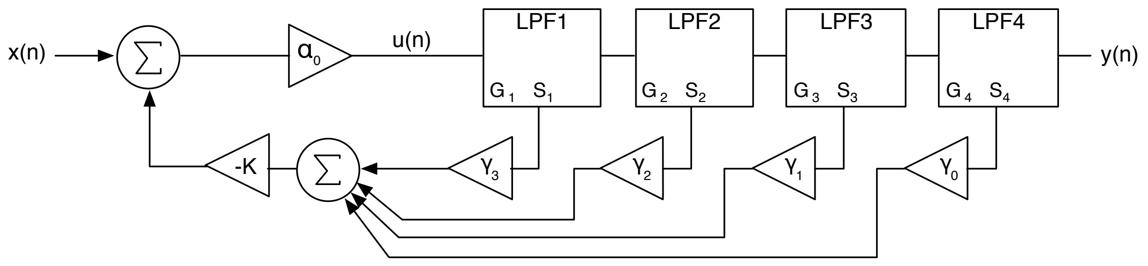

Figure A11.12: the Moog LPF block diagram with the delay free-loop resolved; notice the node *u(n)* which is the input into the first filter in the series

The equations for the coefficients are as follows:

$G = a_0$ since all filters are sync tuned they share the same G (a_0) value

$$\gamma_0 = 1.0$$

$$\gamma_1 = G$$

$$\gamma_2 = G^2$$

$$\gamma_3 = G^3$$

$$\alpha_0 = \frac{1}{1+KG^4}$$

$$u(n) = \alpha_0\left[x(n) + K(\gamma_3 S_1 + \gamma_2 S_2 + \gamma_1 S_3 + \gamma_0 S_4)\right]$$

A11.6 Coding the *MoogLPF* Object

The *MoogLPF* object follows the same design pattern we've used all semester so it is fairly straightforward. The only new addition involves using two helper functions that are already built-into the *Biquad* and *AudioFilter* objects that allow you to access the *G* (*a_0*) and *S(n)* values from within the structure.

A11.6.1 *MoogLPFParameters* Custom Data Structure

The GUI data structure for the *MoogLPF* contains four member items that correspond to the attributes we've discussed so far. The user can adjust the bass compensation to restore the missing bass using a % parameter; at 100%, all bass will be fully restored.

```
struct MoogLPFParameters
{
      MoogLPFParameters() {}

      MoogLPFParameters& operator=(const MoogLPFParameters& params)

      <SNIP SNIP SNIP>

      double fc = 0.0;                 // --- Hz
      double Q = 1.0;                  // --- Q = [1.0, 10.0]
      double bassBoost_Pct = 0.0;      // --- restore some (or all) bass
      bool enableAutoLimiter = false;
};
```

A11.6.2 *MoogLPF* Operation

The *MoogLPF* operates the same way as the rest of the signal processing objects so I won't go into any major detail there – reset the object with the sample rate, update the GUI parameters, and process samples. But, we can surely take a look at some of the functions.

A11.6.2.1 reset( )

The *reset( )* function is where you initialize the filters and the peak limiter. Notice that we setup the filter type as *filterAlgorithm::kLPF1* – do not be tempted to using one of the all-pole or other analog-matched filters. They do not have the proper -45 degrees of phase shift at the cutoff frequency.

```
virtual bool reset(double _sampleRate)
{
      // --- store the sample rate
      sampleRate = _sampleRate;

      // --- setup the audio filters
      AudioFilterParameters params = lowpassFilters[0].getParameters();
      params.algorithm = filterAlgorithm::kLPF1;
      params.fc = parameters.fc;

      for (int i = 0; i < 4; i++)
      {
            lowpassFilters[i].reset(sampleRate);
            lowpassFilters[i].setParameters(params);
      }

      // --- setup the peak limiter
      peakLimiter.reset(_sampleRate);
```

```
	// --- 3dB threshold is good compromise
	peakLimiter.setThreshold_dB(-3.0);

	return true;
}
```

A11.6.2.2 calculateCoefficients( )

The *calculateCoefficients( )* function just converts the GUI's *Q* control from its range of [1, 10] to the *MoogLPF's K* range of [0, 4] and then the audio filters are updated. This is a result of calling the *MoogLPF::setParameters( )* method with GUI updates.

```
void calculateCoefficients()
{
	// --- Q is 1 -> 10 for my plugins; just map it to 0 -> 4
	K = (4.0)*(parameters.Q - 1.0) / (10.0 - 1.0);

	// --- update filters
	AudioFilterParameters params = lowpassFilters[0].getParameters();
	params.fc = parameters.fc;

	for (int i = 0; i < 4; i++)
	{
		lowpassFilters[i].setParameters(params);
	}
}
```

A11.6.2.3 processAudioSample( )

The *processAudioSample( )* function is fairly straightforward. To understand it, compare the code with the block diagrams in Figures 11.6 and 11.12 which involve the bass compensation and the formation of the sigma and feedback components. The limiter is optionally applied at the end of the function. The audio filter sub-components handle their state updates. Make sure you understand:

- how to use the *getS_value( )* function from the *AudioFilter* object
- how to use the *getG_value( )* function from the *AudioFilter* object
- the calculation of the coefficients
- scaling the input for bass-restoration
- application of peak limiter outside the loop

```
virtual double processAudioSample(double xn)
{
	// --- gather the S(n) values
	double S1 = lowpassFilters[0].getS_value();
	double S2 = lowpassFilters[1].getS_value();
	double S3 = lowpassFilters[2].getS_value();
	double S4 = lowpassFilters[3].getS_value();

	// --- setup coefficients
	double G = lowpassFilters[0].getG_value();
	double dSigma = G*G*G*S1 + G*G*S2 + G*S3 + S4;
	double alpha0 = 1.0 / (1.0 + K*G*G*G*G);
```

```
      // --- bass compensation (see notes) gain = 1 + aK
      double compensationGain = 1.0 +
                        (parameters.bassBoost_Pct / 100.0) * K;

      // --- scale input BEFORE entering loop
      xn *= compensationGain;

      // calculate input to first filter
      double u = (xn - K*dSigma)*alpha0;

      // --- process all filters in sequence
      double LPF0 = lowpassFilters[0].processAudioSample(u);
      double LPF1 = lowpassFilters[1].processAudioSample(LPF0);
      double LPF2 = lowpassFilters[2].processAudioSample(LPF1);
      double LPF3 = lowpassFilters[3].processAudioSample(LPF2);

      // --- peak limiter is OUTSIDE the loop!
      double limiterOutput = parameters.enableAutoLimiter ?
                        peakLimiter.processAudioSample(LPF3) : LPF3;

      // --- done
      return limiterOutput;
}
```

Note that the code above is purposefully verbose – the filtering operation could certainly be compressed into a single line of code with nested function calls, and the other variables like the bass boost could also be combined to create reduced code. Remember that many of the FX book readers are relatively new to C++ and I do this to help make the code a bit less confusing, even if it results in slightly less efficient code. Please don't email me about this.

A11.7 Bibliography

Lathi, B.P. and Green, R.A. 2014. *Essentials of Digital Signal Processing*. Cambridge University Press, New York.

Pirkle, W. 2014. *Designing Software Synthesizer Plugins in C++*. Focal Press, New York.

Pirkle, W. 2014. *Novel Hybrid Virtual Analog Filters Based on the Sallen-Key Architecture*, presented at the AES137th Convention, Los Angeles, USA. Paper 9194

Pirkle, W. 2014, *Resolving Delay-Free Loops in Recursive Filters using the Modified Härmä Method*, presented at the AES137th Convention, Los Angeles, USA. Paper 9195

Zavalishin, Vadim. 2012. *The Art of VA Filter Design*, Accessed June 2014, http://www.native-instruments.com/fileadmin/ni_media/downloads/pdf/VAFilterDesign_1.0.3.pdf