# A matrix approach to the Moog ladder filter

Raph Levien

June 23, 2013

## 1 Background

The original Moog ladder filter remains a highly desirable building block for electronic music, both in its original form and in the form of digital simulations. There is now a considerable literature on characterizing the original filter and on techniques for digital simulation.

In spite of advances, it is safe to say that there is not yet a definitive method for simulating the Moog ladder filter.

The contribution of this paper is an extremely accurate, efficient, and simple approach to the *linear* fragment of the problem.

The fundamental approach is to model the state as a vector, and the evolution of state as applying a transition matrix. These calculations lend themselves especially well to evaluation on a microprocessor with SIMD extensions, as the fundamental operation is a 4x4 matrix multiplication, which typically is a standard library operation and has an esepcially highly optimized implementation.

## 2 Review of the model

Under the linear assumption, the Moog ladder filter can accurately be modeled as four one-pole lowpass filters in series, with a feedback loop. The gain of the feedback loop controls the filter resonance.

Most approaches to digital simulation of the filter rely on *discretization* of the analog one-pole filters, replacing each with a corresponding one-pole IIR. However, the frequency and phase responses of such discretized filters do not exactly match that of the original analog filters, especially as the cutoff frequency becomes a nontrivial fraction of the sampling rate.

Under the linear assumption, the Moog ladder filter can be characterized by coupled differential equations:

$$dy_0/dt = \alpha(y_0 - x + ky_3)$$

$$dy_1/dt = \alpha(y_1 - y_0)$$

$$dy_2/dt = \alpha(y_2 - y_1)$$

$$dy_3/dt = \alpha(y_3 - y_2)$$

Here, $\alpha$ is simply equal to $F_c$, the "cutoff" frequency, which corresponds to the resonant peak for high values of k, or the frequency at which there is 12dB attenuation in the k=0 case.

## 3 Approach of this paper

The state of the system is represented as a four-valued vector. The most basic approach to solving these differential equations is Euler integration. For very small values of $\alpha$, or for small step sizes, the simulation is accurate. But at large values, integration errors become nontrivial. One approach would be to use a more sophisticated integration approach, such as trapezoidal or Runge-Kutta. Another is simply to divide time into finer steps. Both of these approaches add complexity and computational cost.

Rather, we model the Euler integration as a matrix describing the transition from one state of the 4-vector to the next.

To correctly take into account the input, we extend the state to a 5-vector and assume that the x parameter will be constant for the duration of the integration. Thus, the transition matrix is 5x5.

The resulting matrix is:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ \alpha\Delta t & 1-\alpha\Delta t & 0 & 0 & -k\alpha\Delta t \\ 0 & \alpha\Delta t & 1-\alpha\Delta t & 0 & 0 \\ 0 & 0 & \alpha\Delta t & 1-\alpha\Delta t & 0 \\ 0 & 0 & 0 & \alpha\Delta t & 1-\alpha\Delta t \end{bmatrix}$$

Repeated transition of state can now be modeled as exponentiation of this matrix. This is especially cheap for exponentiation by $2^N$ – it can be accomplished by repeatedly squaring the matrix N times. In practice, values of N of 10 (1/1024 step size) yield accurate results across the range of filter parameters.

Now we can model the linear filter with arbitrarily low error. We model the input process as "sample and hold" of the input, with x(t) constant for the entire sample period. We model the output process as simple sampling. This potentially yields errors of up to 3dB attenuation at the Nyquist frequency, but since this is a lowpass filter, such effects may be ignored. The filter resonant frequency and sharpness (Q) are more important perceptually.

## 4 Experimental results

The results of implementing the above filter are shown in Figure 1. This figure shows the frequency responses of the filter for 10 values of $F_c$, starting at 20kHz

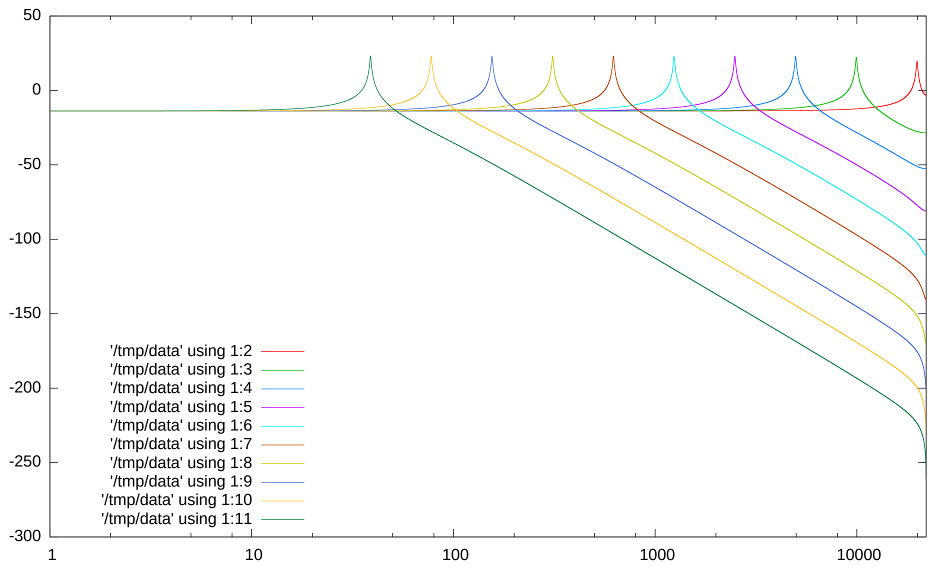

Figure 1: Frequency response at k = 3.9

and successively each an octave lower. The even spacing shows that the tuning is very precise, and the consistent shape of the peak shows that resonance is completely independent of frequency. There is a small amount of error at the far right of the graph, but only significant at frequencies greater than 20kHz, so insignificant audibly.

Note that frequencies extremely near the Nyquist limit are still stable and accurate.

## 5 Extension to nonlinearity

The actual Moog ladder filter is not linear, but rather relies on differential pairs of transistors, which have a transfer function shaped like the tanh function. Huovilainen gave nonlinear differential equations that accurately capture the circuit behavior (evaluation of these differential equations match detailed simulation of the circuit schematic diagram with general purpose circuit simulators such as SPICE).

The nonlinear equations are:

$$dy_0/dt = \alpha(\tanh(y_0) - \tanh(x - ky_3))$$

$$dy_1/dt = \alpha(\tanh(y_1) - \tanh(y_0))$$

$$dy_2/dt = \alpha(\tanh(y_2) - \tanh(y_1))$$

$$dy_3/dt = \alpha(\tanh(y_3) - \tanh(y_2))$$

The general approach is to compute the same matrix as for the linear case, but then to apply tanh() to each of the five inputs of the state. We modify the kernel so that we compute a delta to the $y$ state vector, rather than simply the new value of $y$. The kernel is also modified so that the $-ky_3$ contribution is subtracted from the rightmost column, so it can be included under the tanh of the $x$ term.

The kernel is then:

$$\mathbf{C} = \begin{bmatrix} A_{00} - 1 & A_{10} & A_{20} & A_{30} + kB_0 \\ A_{01} & A_{11} - 1 & A_{21} & A_{31} + kB_1 \\ A_{02} & A_{12} & A_{22} - 1 & A_{32} + kB_2 \\ A_{03} & A_{13} & A_{23} & A_{33} + kB_3 - 1 \end{bmatrix}$$

$$y \mathrel{+}= B\tanh(x - ky_3) + \mathbf{C}[\tanh(y0)\tanh(y1)\tanh(y2)\tanh(y3)]^T$$

For the small-signal assumption, the resulting filter is clearly equivalent to the linear variant. (note that the terminology has shifted a bit; the B of this section is the bottom four entries of the left column of the A of last, and the A of this section is the lower right 4x4 block of the A of last. I hope to fix this up for the real publication paper)

Note that the nonlinearities can introduce aliasing. Oversampling is likely the best technique for combatting this aliasing. Unlike the implementation described in Huovilainen, however, the filter is not dependent on oversampling in order to get correct tuning or frequency response.

Note also that all 5 tanh functions can be computed in parallel, as they depend only on the last iteration of state, with no serial dependencies.

## 6 Conclusion

The implementation proposed in this paper is faster on modern processors, more accurate (especially at higher cutoff frequencies), and simpler, not requiring arbitrary tuning tables. We have presented both a linear version with an extremely high degree of accuracy and a nonlinear version which accurately captures the behavior of the original circuit.

## 7 References

Huovilainen, Antti. Non-linear digital implementation of the Moog ladder filter. DAFx '04.

Daly, Paul. A comparison of virtual analogue Moog VCF models. MSc research project, University of Edinburgh. August 2012.

Stilson and Smith. Analyzing the Moog VCF with considerations for digital implementation.

Robelly, J.P, et al. Implementation of recursive digital filters into vector SIMD DSP architectures. ICASSP '04.

## A Python code

As a more precise specification of the algorithm, here is Python code that implements the linear case for arbitrary $\alpha$ and $k$, computing the impulse response.

```
import numpy as np

def reso(alpha, k, n = 16):
    a = alpha / (1 << n)
    na = 1 - a
    A = np.matrix([[1, 0, 0, 0, 0],
        [a, na, 0, 0, -k * a],
        [0, a, na, 0, 0],
        [0, 0, a, na, 0],
        [0, 0, 0, a, na]])
    for i in range(n):
        A = A * A

    # now crank the impulse response
    B = A[1:, 0]
    A = A[1:, 1:]
    y = B

    result = []
    for i in range(65536 * 16):
        result.append(y[3, 0])
        y = A * y
    return result
```