kvraudio.com

KVR Forum: Does anyone here understand FAUST Programming Language? (Or can you intuit it for this one question?) - DSP and Plugin Development Forum

Post by mystran » Wed Apr 15, 2020 5:14 pm

7–8 minutes

Got it working I think. For anyone who is curious, here's the derivation and code - might be useful to others as well.

Here's the three "classes" in Faust that generate this:

Code: Select all

```
//-------------------------- spectral_tilt
--------------------------------
// Spectral tilt filter, providing an arbitrary
spectral rolloff factor
// alpha in (-1,1), where
//  -1 corresponds to one pole (-6 dB per octave), and
//  +1 corresponds to one zero (+6 dB per octave).
// In other words, alpha is the slope of the ln
magnitude versus ln frequency.
// For a "pinking filter" (e.g., to generate 1/f noise
from white noise),
// set alpha to -1/2.
//
```

```
// USAGE:
//   _ : spectral_tilt(N,f0,bw,alpha) : _
// where
//    N = desired integer filter order (fixed at
compile time)
//   f0 = lower frequency limit for desired roll-off
band
//   bw = bandwidth of desired roll-off band
// alpha = slope of roll-off desired in nepers per
neper
//        (ln mag / ln radian freq)
//
// EXAMPLE:
//   spectral_tilt_demo below
//
// REFERENCE:
//   http://arxiv.org/abs/1606.06154
//
spectral_tilt(N,f0,bw,alpha) = seq(i,N,sec(i)) with {
  sec(i) = g * tf1s(b1,b0,a0,1) with {
    g = a0/b0; // unity dc-gain scaling
    b1 = 1.0;
    b0 = mzh(i);
    a0 = mph(i);
    mzh(i) = prewarp(mz(i),SR,w0); // prewarping for
bilinear transform
    mph(i) = prewarp(mp(i),SR,w0);
    prewarp(w,SR,wp) = wp * tan(w*T/2) / tan(wp*T/2)
with { T = 1/SR; };
    mz(i) = w0 * r ^ (-alpha+i); // minus zero i in s
plane
    mp(i) = w0 * r ^ i; // minus pole i in s plane
```

```
    w0 = 2 * PI * f0; // radian frequency of first pole
    f1 = f0 + bw; // upper band limit
    r = (f1/f0)^(1.0/float(N-1)); // pole ratio (2 =>
octave spacing)
  };
};

//-------------------------- spectral_tilt_demo
------------------------------
// USAGE:
//   _ : spectral_tilt_demo(N) : _ ;
// where
//   N = filter order (integer)
// All other parameters interactive
//
spectral_tilt_demo(N) = spectral_tilt(N,f0,bw,alpha)
with {
    alpha = hslider("[1] Slope of Spectral Tilt across
Band",-1/2,-1,1,0.001);
    f0 = hslider("[2] Band Start Frequency
[unit:Hz]",100,20,10000,1);
    bw = hslider("[3] Band Width
[unit:Hz]",5000,100,10000,1);
};
```

Code: Select all

```
//------------------------------- tf1s
--------------------------------
// First-order direct-form digital filter,
// specified by ANALOG transfer-function polynomials
B(s)/A(s),
// and a frequency-scaling parameter.
```

```
//
// USAGE: tf1s(b1,b0,a0,w1), where
//
//       b1 s + b0
// H(s) = ----------
//          s + a0
//
// and w1 is the desired digital frequency (in radians/
second)
// corresponding to analog frequency 1 rad/sec (i.e., s
= j).
//
// EXAMPLE: A first-order ANALOG Butterworth lowpass
filter,
//         normalized to have cutoff frequency at 1
rad/sec,
//         has transfer function
//
//          1
// H(s) = -------
//        s + 1
//
// so b0 = a0 = 1 and b1 = 0.  Therefore, a DIGITAL
first-order
// Butterworth lowpass with gain -3dB at SR/4 is
specified as
//
// tf1s(0,1,1,PI*SR/2); // digital half-band order 1
Butterworth
//
// METHOD: Bilinear transform scaled for exact mapping
of w1.
```

```
// REFERENCE:
//   https://ccrma.stanford.edu/~jos/pasp/
Bilinear_Transformation.html
//
tf1s(b1,b0,a0,w1) = tf1(b0d,b1d,a1d)
with {
  c   = 1/tan(w1*0.5/SR); // bilinear-transform scale-
factor
  d   = a0 + c;
  b1d = (b0 - b1*c) / d;
  b0d = (b0 + b1*c) / d;
  a1d = (a0 - c) / d;
};
```

Code: Select all

```
//------------------------------ tf1, tf2
---------------------------------
// tfN = N'th-order direct-form digital filter
tf1(b0,b1,a1) = _ <: *(b0), (mem : *(b1)) :> + ~ *(0-
a1);
```

Here's my C++ version (I consolidated the two filter classes into one since they were essentially just working off each other):

Code: Select all

```
#pragma once
#include "JuceHeader.h"

class SpectralTiltFilter_TF1S {
public:

       void setSampleRate(double sampleRateIn) {
              SR = sampleRateIn;
```

```
	}

	void set_tf1s(double b1, double b0, double a0, double w1) {
		double c = 1 / tan(w1*0.5 / SR);
		double d = a0 + c;
		b1d = (b0 - b1 * c) / d;
		b0d = (b0 + b1 * c) / d;
		a1d = (a0 - c) / d;
		g = a0 / b0;
	}

	double process_tf1(double sampleIn) {
		input_1 = input;
		input = sampleIn;
		output_1 = output;

		output = ((input * b0d) + (input_1 * b1d)) + (output_1 * -a1d);
		double output_gained = g * output;
		return output_gained;
	}

private:
	double SR = 44100.0;
	double b1d = 0.0;
	double b0d = 0.0;
	double a1d = 0.0;
	double g = 1.0;
	double input = 0.0;
	double output = 0.0;
	double input_1 = 0.0;
```

```
        double output_1 = 0.0;

};

class SpectralTiltFilter {

public:

        void setSampleRate(double sampleRateIn) {
                SR = sampleRateIn;
                T = 1 / SR;

                for (int i = 0; i < filterArray.size();
i++) {
                        SpectralTiltFilter_TF1S* unit =
filterArray.getUnchecked(i);
                        unit-
>setSampleRate(sampleRateIn);
                }
        }
        void setFilterN(int N) {
                filterN = N;    //N must be 2 or more
                filterArray.clear();
                for (int i = 0; i < N; i++) {
                        SpectralTiltFilter_TF1S* unit =
new SpectralTiltFilter_TF1S();
                        unit->setSampleRate(SR);
                        filterArray.add(unit);
                }
        }

        void setFilter(double f0, double bw, double
```

```
alphaIn) {
		alpha = alphaIn;
		w0 = 2 * MathConstants<float>::pi * f0;
		f1 = f0 + bw;
		r = pow((f1 / f0), (1.0 /
static_cast<double>(filterN - 1))); //N must be at
least 2

		for (int i = 0; i < filterN; i++) {
			SpectralTiltFilter_TF1S* unit =
filterArray.getUnchecked(i);
			unit->set_tf1s(1.0, mzh(i),
mph(i), 1);
		}
	}

	double processSample(double sampleIn) {
		double input_filtered = sampleIn;
		for (int i = 0; i < filterN; i++) {
			SpectralTiltFilter_TF1S* unit =
filterArray.getUnchecked(i);
			input_filtered = unit-
>process_tf1(input_filtered);
		}
		return input_filtered;
	}

	double prewarp(double w, double T, double wp) {
		return wp * tan(w*T / 2) / tan(wp*T / 2);
	}

	double mz(int i) {
```

```
                return w0 * pow(r, (-alpha + i));
        }

        double mp(int i) {
                return w0 * pow(r, i);
        }

        double mzh(int i) {
                return prewarp(mz(i), T, w0);
        }

        double mph(int i) {
                return prewarp(mp(i), T, w0);
        }

private:
        OwnedArray<SpectralTiltFilter_TF1S>
filterArray;
        int filterN = 2;
        double SR = 44100.0;
        double T = 1 / 44100.0;
        double w0 = 10.0;
        double f1 = 220.0;
        double r = 40.0;
        double alpha = 0.5;

};
```

Usage would be for example:

Code: [Select all]

```
//Initialization:
SpectralTiltFilter spectralTilt;
```

```
spectralTilt.setSampleRate(sampleRate);
spectralTilt.setFilterN(12); // N must be 2 or more
spectralTilt.setFilter(cornerFreqHz,
widthOfRollOffBand, slope); //slope is set as 0 = flat,
-0.5 = 3dB/oct LPF, -1 = 6 db/oct LPF

//Sample processing:
filteredNoise =
spectralTilt.processSample(whiteNoise);
```

I haven't tested it with visualizers to confirm it's doing exactly as it should, but it sounds good to me.

I notice with n=8 and a -1 slope (-6 db/oct) it sounds very different from a one pole LPF, but when I put it to n=12 I can't hear a difference anymore. Maybe this is the sweet spot.