DSP Trick: Fixed-Point DC Blocking Filter with Noise-Shaping

Trick

```
From: robert bristow-johnson <rbj@gisco.net>
Subject: Fixed-Point DC Blocking Filter with Noise-Shaping
Date: 22 Jun 2000 00:00:00 GMT, Updated: 17 Apr 2001
Newsgroups: comp.dsp
THIS WORK IS PLACED IN THE PUBIC DOMINION
```

Name: Fixed-Point DC Blocking Filter With Noise-Shaping

Category: Algorithmic

Application: Used anytime DC offsets need to be removed (audio level metering or nearly any audio signal processing to avoid clicks). Since A/D converters and their front end circuitry cannot be expected to be totally DC free, this is useful partly to insure that 'silence' is equavalent to 'zero'.

Advantages: Utterly wipes out DC *without* introducing its own DC

Introduction:

The use of a digital differentiator along with a 'leaky' integrator (essentially a one-pole LPF) to accomplish DC blocking is nothing new. It is essentially a 1st order HPF.

- Differentiator: y[n]=x[n] - x[n-1]
- Leaky Integrator: y[n]=pole*y[n-1] + x[n]
  (0 < 1-pole << 1) if pole=1, then it is a non-leaky integrator and completely undoes the differentiator.

However, doing this with a fixed-point DSP (such as the 56K) introduces a couple of problems. First the order of operations has to be considered.

Leaky Integrator first followed by differentiator: Since the differentiator truly wipes out DC no matter what the numerical issues are, this has the advantage that no DC gets into the output. The disadvantage is that if the pole is very close to 1 (which is necessary if you want your DC blocker to just block DC and other very low frequencies), the the gain of the leaky integrator blows up at low frequencies, enough that saturation is vitually assured and the post differentiation will not fix that horrible distortion.

Differentiator first followed by leaky integrator: DC is killed in the first stage and the gains at low frequecies are reduced enough to guarantee that the integrator does not saturate. Also, there is _no_ quantization error introduced by the differentiator so there isn't the problem of the integrator boosting quantization noise at low frequencies. The disadvantage is that the integrator creates its own DC problems due to quantization (rounding or truncating down) and limit cycling. This limit cycling happens when the amount that y[n] is reduced by multiplying by pole is equal to 1/2 LSB and rounding returns y[n+1] back to its previous non-zero value y[n]. it gets stuck on a non-zero value that gets measurably large as pole gets close to 1.

The Trick:

We use the 2nd option above (differentiator first) and deal with the limit-cycling problem using error-shaping or noise-shaping with a zero on DC (Randy Yates called this "fraction saving"). By feeding back the quantization error that occurs in the one-pole LPF (the leaky integrator), one can design an error-to-output transfer function to have a desired frequency response (to literally shape the noise or error spectrum to steer noise away from where you don't want it to where you can tolerate it). With a zero of this noise transfer placed right on z=1 (or at DC) we force the amplitude of the error in the output to be zero at the frequency of 0 (at the expense of increasing the error amplitude at Nyquist). We literally will have infinite S/N ratio at DC. Therefore if no DC goes into the LPF (and the differentiator sees to that), then no DC comes out. Even if the quantization is not rounding to nearest but always rounding down (definitely a DC bias here) any DC introduced gets killed by the error shaping (this is what happens with 'fraction saving').

```
Let:
        x[n] = input to DC blocking filter (and to differentiator)
        d[n] = output of differntiator and input to LPF
        y[n] = output of LPF and of DC blocking filter
        e[n] = quantization error of LPF (or leaky integrator)
        d[n] = x[n] - x[n-1]    (no quantization error)
        y[n] = Quantize{ pole*y[n-1] + d[n] - e[n-1] }
        e[n] = y[n] - (pole*y[n-1] + d[n] - e[n-1])
or
        y[n] = (pole*y[n-1] + d[n] - e[n-1])            + e[n]
or
        y[n] = (pole*y[n-1] + x[n] - x[n-1] - e[n-1])   + e[n]
or
        y[n] = (y[n-1] - (1-pole)*y[n-1] + x[n] - x[n-1] - e[n-1]) + e[n]
```

I got this down to 3 instructions per sample in a 56K (using sample buffering, another trick, and pipelining) but here is what it might look like using C code:

```
// let's say sizeof(short) = 2 (16 bits) and sizeof(long) = 4 (32 bits)
short x[], y[];
long acc, A, prev_x, prev_y;
double pole;
unsigned long n, num_samples;
pole = 0.9999;
A = (long)(32768.0*(1.0 - pole));
acc = 0;
prev_x = 0;
prev_y = 0;
for (n=0; n<num_samples; n++)
    {
    acc   -= prev_x;
    prev_x = (long)x[n]<<15;
    acc   += prev_x;
    acc   -= A*prev_y;
    prev_y = acc>>15;               // quantization happens here
    y[n]   = (short)prev_y;
    // acc has y[n] in upper 17 bits and -e[n] in lower 15 bits
    }
```

‹ DSP Trick: Fast Floating Point to Mu-Law Conversion   up   DSP Trick: Dealing With Propagating Truncation Errors ›

» Printer-friendly version | Login to post comments