

musicdsp.org source code archive

This PDF was autogenerated on January 28, 2009, 7:27 am, server time
All names in the first section are links, you can click'm ;-)

Analysis

- Beat Detector Class
- Coefficients for Daubechies wavelets 1-38
- DFT
- Envelope detector
- Envelope Detector class (C++)
- Envelope follower with different attack and release
- Fast in-place Walsh-Hadamard Transform
- FFT
- FFT classes in C++ and Object Pascal
- Frequency response from biquad coefficients
- Java FFT
- Look ahead limiting
- LPC analysis (autocorrelation + Levinson-Durbin recursion)
- Magnitude and phase plot of arbitrary IIR function, up to 5th order
- Measuring interpolation noise
- QFT and DQFT (double precision) classes
- Simple peak follower
- tone detection with Goertzel
- Tone detection with Goertzel (x86 ASM)

Effects

- 2 Wave shaping things
- Alien Wah
- Band Limited PWM Generator
- Bit quantization/reduction effect
- Class for waveguide/delay effects
- Compressor
- Decimator
- Delay time calculation for reverberation
- DIRAC - Free C/C++ Library for Time and Pitch Manipulation of Audio Based on Time-Frequency Transforms
- dynamic convolution
- Early echo's with image-mirror technique
- ECE320 project: Reverberation w/ parameter control from PC
- fold back distortion
- Guitar feedback
- Lo-Fi Crusher
- Most simple and smooth feedback delay
- Most simple static delay
- Parallel combs delay calculation
- Phaser code
- Polynominal Waveshaper
- Reverberation Algorithms in Matlab
- Reverberation techniques
- Simple Compressor class (C++)
- smsPitchScale Source Code
- Soft saturation
- Stereo Enhancer
- Stereo Field Rotation Via Transformation Matrix
- Stereo Width Control (Obtained Via Transformation Matrix)
- Time compression-expansion using standard phase vocoder
- transistor differential amplifier simulation
- Variable-hardness clipping function
- WaveShaper
- Waveshaper

- Waveshaper
- Waveshaper (simple description)
- Waveshaper :: Gloubi-boulga

Filters

- 1 pole LPF for smooth parameter changes
- 1-RC and C filter
- 18dB/oct resonant 3 pole LPF with tanh() dist
- 1st and 2nd order pink noise filters
- 3 Band Equaliser
- 303 type filter with saturation
- All-Pass Filters, a good explanation
- Another 4-pole lowpass...
- Bass Booster
- Biquad C code
- Butterworth Optimized C++ Class
- C++ class implementation of RBJ Filters
- C-Weighed Filter
- Cascaded resonant lp/hp filter
- Cool Sounding Lowpass With Decibel Measured Resonance
- DC filter
- Delphi Class implementation of the RBJ filters
- Digital RIAA equalization filter coefficients
- Direct form II
- Fast Downsampling With Antialiasing
- Formant filter
- frequency warped FIR lattice
- Hilbert Filter Coefficient Calculation
- High quality /2 decimators
- Karlsen
- Karlsen Fast Ladder
- Lowpass filter for parameter edge filtering
- LP and HP filter
- LPF 24dB/Oct
- Moog Filter
- Moog VCF
- Moog VCF, variation 1
- Moog VCF, variation 2
- Notch filter
- One pole filter, LP and HP
- One pole LP and HP
- One pole, one zero LP/HP
- One zero, LP/HP
- One-Liner IIR Filters (1st order)
- Peak/Notch filter
- Perfect LP4 filter
- Phase equalization
- Pink noise filter
- Plot Filter (Analyze filter characteristics)
- Plotting R B-J Equalisers in Excel
- Polyphase Filters
- Polyphase Filters (Delphi)
- Poor Man's FIWIZ
- Prewarping
- RBJ Audio-EQ-Cookbook
- RBJ Audio-EQ-Cookbook
- Remez Exchange Algorithm (Parks/McClellan)
- Remez Remez (Parks/McClellan)
- Resonant filter
- Resonant IIR lowpass (12dB/oct)
- Resonant low pass filter
- Reverb Filter Generator
- Simple biquad filter from apple.com
- Spuc's open source filters
- State variable
- State Variable Filter (Chamberlin version)

- State Variable Filter (Double Sampled, Stable)
- Stilson's Moog filter code
- Time domain convolution with $O(n \log_2(3))$
- Time domain convolution with $O(n \log_2(3))$
- Type : LPF 24dB/Oct
- Various Biquad filters
- Windowed Sinc FIR Generator
- Zoelzer biquad filters

Other

- VST SDK GUI Switch without
- 16-Point Fast Integer Sinc Interpolator.
- 16-to-8-bit first-order dither
- 3rd order Spline interpolation
- 5-point spline interpolation
- Allocating aligned memory
- Antialiased Lines
- Automatic PDC system
- Base-2 exp
- Bit-Reversed Counting
- Block/Loop Benchmarking
- Branchless Clipping
- Calculate notes (java)
- Center separation in a stereo mixdown
- Center separation in a stereo mixdown
- Cheap pseudo-sinusoidal lfo
- Clipping without branching
- Constant-time exponent of 2 detector
- Conversion and normalization of 16-bit sample to a floating point number
- Conversions on a PowerPC
- Copy-protection schemes
- Cubic interpolation
- Cure for malicious samples
- Denormal DOUBLE variables, macro
- Denormal numbers
- Denormal numbers, the meta-text
- Denormalization preventer
- Denormalization preventer
- Dither code
- Dithering
- Double to Int
- Envelope Follower
- Exponential curve for
- Exponential parameter mapping
- fast abs/neg/sign for 32bit floats
- Fast binary log approximations
- Fast cube root, square root, and reciprocal for x86/SSE CPUs.
- fast exp() approximations
- Fast exp2 approximation
- Fast log2
- fast power and root estimates for 32bit floats
- Fast rounding functions in pascal
- Fast sign for 32 bit floats
- Float to int
- Float to int (more intel asm)
- Float to integer conversion
- Float-to-int, converting an array of floats
- Gaussian dithering
- Gaussian random numbers
- Hermite Interpolator (x86 ASM)
- Hermite interpolation
- Linear interpolation
- Lock free fifo
- Matlab Time Domain Impulse Response Inverter/Divider
- MATLAB-Tools for SNDAN
- MIDI note/frequency conversion

- Millimeter to DB (faders...)
- Motorola 56300 Disassembler
- Noise Shaping Class
- Nonblocking multiprocessor/multithread algorithms in C++
- Piecewise quadratic approximate exponential function
- please add it as a comment to the Denormalization preventer entry (no comments are allowed now) thanks
- $\text{pow}(x,4)$ approximation
- rational tanh approximation
- Reading the compressed WA! parts in gigasampler files
- Real basic DSP with Matlab (+ GUI) ...
- real value vs display value
- Really fast x86 floating point sin/cos
- Reasonably accurate/fastish tanh approximation
- resampling
- Saturation
- Sin(x) Aproximation (with SSE code)
- Sin, Cos, Tan approximation

Synthesis

- (Allmost) Ready-to-use oscillators
- Alias-free waveform generation with analog filtering
- AM Formantic Synthesis
- Another cheap sinusoidal LFO
- another LFO class
- antialiased square generator
- Arbitrary shaped band-limited waveform generation (using oversampling and low-pass filtering)
- Audiable alias free waveform gen using width sine
- Bandlimited sawtooth synthesis
- Bandlimited waveform generation
- Bandlimited waveform generation with hard sync
- Bandlimited waveforms synopsis.
- Bandlimited waveforms...
- Butterworth
- C# Oscilator class
- C++ gaussian noise generation
- chebyshev waveshaper (using their recursive definition)
- Cubic polynomial envelopes
- Direct pink noise synthesis with auto-correlated generator
- Discrete Summation Formula (DSF)
- Drift generator
- DSF (super-set of BLIT)
- Easy noise generation
- Fast & small sine generation tutorial
- Fast Exponential Envelope Generator
- Fast LFO in Delphi...
- Fast SIN approximation for usage in e.g. additive synthesizers
- Fast sine and cosine calculation
- Fast sine wave calculation
- Fast square wave generator
- Fast Whitenoise Generator
- Gaussian White noise
- Gaussian White Noise
- Generator
- Inverted parabolic envelope
- matlab/octave code for minblep table generation
- PADsynth synthesys method
- Parabolic shaper
- Phase modulation Vs. Frequency modulation
- Phase modulation Vs. Frequency modulation II
- Pseudo-Random generator
- PulseQuad
- Pulsewidth modulation
- Quick & Dirty Sine
- quick and dirty sine generator
- RBJ Wavetable 101
- Rossler and Lorenz Oscillators

- SawSin
- Simple Time Stretching-Granular Synthesizer
- Sine calculation
- Square Waves
- Trammell Pink Noise (C++ class)
- Waveform generator using MinBLEPS
- Wavetable Synthesis
- Weird synthesis

[\(Almost\) Ready-to-use oscillators](#) (click this to go back to the index)

Type : waveform generation

References : Ross Bencina, Olli Niemitalo, ...

Notes :

Ross Bencina: original source code poster

Olli Niemitalo: UpdateWithCubicInterpolation

Code :

```
//this code is meant as an EXAMPLE

//uncomment if you need an FM oscillator
//define FM_OSCILLATOR

/*
members are:

float phase;
int TableSize;
float sampleRate;

float *table, dtable0, dtable1, dtable2, dtable3;

->these should be filled as follows... (remember to wrap around!!!)
table[i] = the wave-shape
dtable0[i] = table[i+1] - table[i];
dtable1[i] = (3.f*(table[i]-table[i+1])-table[i-1]+table[i+2])/2.f
dtable2[i] = 2.f*table[i+1]+table[i-1]-(5.f*table[i]+table[i+2])/2.f
dtable3[i] = (table[i+1]-table[i-1])/2.f
*/

float Oscillator::UpdateWithoutInterpolation(float frequency)
{
    int i = (int) phase;

    phase += (sampleRate/(float)TableSize)/frequency;

    if(phase >= (float)TableSize)
        phase -= (float)TableSize;

#ifdef FM_OSCILLATOR
    if(phase < 0.f)
        phase += (float)TableSize;
#endif

    return table[i] ;
}

float Oscillator::UpdateWithLinearInterpolation(float frequency)
{
    int i = (int) phase;
    float alpha = phase - (float) i;

    phase += (sampleRate/(float)TableSize)/frequency;

    if(phase >= (float)TableSize)
        phase -= (float)TableSize;

#ifdef FM_OSCILLATOR
    if(phase < 0.f)
        phase += (float)TableSize;
#endif

    /*
    dtable0[i] = table[i+1] - table[i]; //remember to wrap around!!!
    */

    return table[i] + dtable0[i]*alpha;
}

float Oscillator::UpdateWithCubicInterpolation( float frequency )
{
    int i = (int) phase;
    float alpha = phase - (float) i;

    phase += (sampleRate/(float)TableSize)/frequency;

    if(phase >= (float)TableSize)
        phase -= (float)TableSize;

#ifdef FM_OSCILLATOR
    if(phase < 0.f)
        phase += (float)TableSize;
#endif

    /* //remember to wrap around!!!
    dtable1[i] = (3.f*(table[i]-table[i+1])-table[i-1]+table[i+2])/2.f
    dtable2[i] = 2.f*table[i+1]+table[i-1]-(5.f*table[i]+table[i+2])/2.f
    dtable3[i] = (table[i+1]-table[i-1])/2.f
    */
}
```

```
*/  
return ((dtable1[i]*alpha + dtable2[i])*alpha + dtable3[i])*alpha+table[i];  
}
```

Alias-free waveform generation with analog filtering (click this to go back to the index)

Type : waveform generation

References : Posted by Magnus Jonsson

Linked file : [synthesis001.txt](#) (this linked file is included below)

Notes :
(see linkfile)

Linked files

alias-free waveform generation with analog filtering

Ok, here is how I did it. I'm not 100% sure that everything is correct.
I'll demonstrate it on a square wave instead, although i havent tried it.

The impulse response of an analog pole is:

$$r(t) = \begin{cases} \exp(p*x) & \text{if } t \geq 0, \\ 0 & \text{if } t < 0 \end{cases}$$

notice that if we know $r(t)$ we can get $r(t+1) = r(t)*\exp(p)$.

You all know what the waveform looks like. It's a constant -1 followed by constant 1 followed by

We need to convolve the impulse response with the waveform and sample it at discrete intervals.
What if we assume that we already know the result of the last sample?
Then we can "move" the previous result backwards by multiplying it with $\exp(p)$ since
 $r(t+1) = r(t)*\exp(p)$

Now the problem is reduced to integrate only between the last sample and the current sample, and add that to the result.

some pseudo-code:

```
while forever
{
    result *= exp(pole);
    phase += freq;
    result += integrate(waveform(phase-freq*t), exp(t*pole), t=0..1);
}
```

```
integrate(square(phase-freq*t), exp(t*pole), t=0..1)
```

The square is constant except for when it changes sign.
Let's find out what you get if you integrate a constant multiplied with $\exp(t*pole)$ =)

```
integrate(k*exp(t*pole)) = k*exp(t*pole)/pole
k = square(phase-freq*t)
```

and with t from 0 to 1, that becomes
 $k*\exp(pole)/pole - k/pole = (\exp(pole) - 1)*k/pole$

the only problem left to solve now is the jumps from +1 to -1 and vice versa.

you first calculate $(\exp(pole) - 1)*k/pole$ like you'd normally do

then you detect if phase goes beyond 0.5 or 1. If so, find out exactly where between the samples this change takes place.

subtract $\text{integrate}(-2*\exp(t*pole), t=0..place)$ from the result to undo the error

Since I am terribly bad at explaining things, this is probably just a mess to you :)

Here's the (unoptimised) code to do this with a saw:
note that sum and pole are complex.

```
float getsample()
{
    sum *= exp(pole);

    phase += freq;

    sum += (exp(pole)*((phase-freq)*pole+freq)-(phase*pole+freq))/pole;

    if (phase >= 0.5)
    {
```



```
        float x = (phase-0.5)/freq;
        sum -= exp(pole*x)-1.0f;
        phase -= 1;
    }
    return sum.real();
}
```

There's big speedup potential in this i think.

Since the filtering is done "before" sampling, aliasing is reduced, as a free bonus. with high cutoff and high frequencies it's still audible though, but much less than without the filter.

If aliasing is handled in some other way, a digital filter will sound just as well, that's what i think.

-- Magnus Jonsson <zeal@mail.kuriren.nu>

AM Formantic Synthesis (click this to go back to the index)

References : Posted by Paul Sernine

Notes :

Here is another tutorial from Doc Rochebois.

It performs formantic synthesis without filters and without grains. Instead, it uses "double carrier amplitude modulation" to pitch shift formantic waveforms. Just beware the phase relationships to avoid interferences. Some patches of the DX7 used the same trick but phase interferences were a problem. Here, Thierry Rochebois avoids them by using cosine-phased waveforms.

Various formantic waveforms are precalculated and put in tables, they correspond to different formant widths.

The runtime uses many instances (here 4) of these and pitch shifts them with double carriers (to preserve the harmonicity of the signal).

This is a tutorial code, it can be optimized in many ways.

Have Fun

Paul

Code :

```
// FormantsAM.cpp

// Thierry Rochebois' "Formantic Synthesis by Double Amplitude Modulation"

// Based on a tutorial by Thierry Rochebois.
// Comments by Paul Sernine.

// The spectral content of the signal is obtained by adding amplitude modulated formantic
// waveforms. The amplitude modulations spectrally shift the formantic waveforms.
// Continuous spectral shift, without losing the harmonic structure, is obtained
// by using crossfaded double carriers (multiple of the base frequency).
// To avoid unwanted interference artifacts, phase relationships must be of the
// "cosine type".

// The output is a 44100Hz 16bit stereo PCM file.

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

//Approximates cos(pi*x) for x in [-1,1].
inline float fast_cos(const float x)
{
    float x2=x*x;
    return 1+x2*(-4+2*x2);
}

//Length of the table
#define L_TABLE (256+1) //The last entry of the table equals the first (to avoid a modulo)
//Maximal formant width
#define I_MAX 64
//Table of formants
float TF[L_TABLE*I_MAX];

//Formantic function of width I (used to fill the table of formants)
float fonc_formant(float p,const float I)
{
    float a=0.5f;
    int hmax=int(10*I)>L_TABLE/2?L_TABLE/2:int(10*I);
    float phi=0.0f;
    for(int h=1;h<hmax;h++)
    {
        phi+=3.14159265359f*p;
        float hann=0.5f+0.5f*fast_cos(h*(1.0f/hmax));
        float gaussienne=0.85f*exp(-h*h/(I*I));
        float jupe=0.15f;
        float harmonique=cosf(phi);
        a+=hann*(gaussienne+jupe)*harmonique;
    }
    return a;
}

//Initialisation of the table TF with the fonction fonc_formant.
void init_formant(void)
{
    float coef=2.0f/(L_TABLE-1);
    for(int I=0;I<I_MAX;I++)
        for(int P=0;P<L_TABLE;P++)
            TF[P+I*L_TABLE]=fonc_formant(-1+P*coef,float(I));
}

//This function emulates the function fonc_formant
// thanks to the table TF. A bilinear interpolation is
// performed
float formant(float p,float i)
{
    i=i<0?0:i>I_MAX-2?I_MAX-2:i; // width limitation
    float P=(L_TABLE-1)*(p+1)*0.5f; // phase normalisation
    int P0=(int)P; float fP=P-P0; // Integer and fractional
    int I0=(int)i; float fI=i-I0; // parts of the phase (p) and width (i).
    int i00=P0+L_TABLE*I0; int i10=i00+L_TABLE;
    //bilinear interpolation.
```

```

return (1-fI)*(TF[i00] + fP*(TF[i00+1]-TF[i00]))
      + fI*(TF[i10] + fP*(TF[i10+1]-TF[i10]));
}

// Double carrier.
// h : position (float harmonic number)
// p : phase
float porteuse(const float h,const float p)
{
float h0=floor(h); //integer and
float hf=h-h0; //decimal part of harmonic number.
// modulus pour ramener p*h0 et p*(h0+1) dans [-1,1]
float phi0=fmodf(p* h0 +1+1000,2.0f)-1.0f;
float phi1=fmodf(p*(h0+1)+1+1000,2.0f)-1.0f;
// two carriers.
float Porteuse0=fast_cos(phi0); float Porteuse1=fast_cos(phi1);
// crossfade between the two carriers.
return Porteuse0+hf*(Porteuse1-Porteuse0);
}
int main()
{
//Formant table for various french vowels (you can add your own)
float F1[]={ 730, 200, 400, 250, 190, 350, 550, 550, 450};
float A1[]={ 1.0f, 0.5f, 1.0f, 1.0f, 0.7f, 1.0f, 1.0f, 0.3f, 1.0f};
float F2[]={ 1090, 2100, 900, 1700, 800, 1900, 1600, 850, 1100};
float A2[]={ 2.0f, 0.5f, 0.7f, 0.7f, 0.35f, 0.3f, 0.5f, 1.0f, 0.7f};
float F3[]={ 2440, 3100, 2300, 2100, 2000, 2500, 2250, 1900, 1500};
float A3[]={ 0.3f, 0.15f, 0.2f, 0.4f, 0.1f, 0.3f, 0.7f, 0.2f, 0.2f};
float F4[]={ 3400, 4700, 3000, 3300, 3400, 3700, 3200, 3000, 3000};
float A4[]={ 0.2f, 0.1f, 0.2f, 0.3f, 0.1f, 0.1f, 0.3f, 0.2f, 0.3f};

float f0,dp0,p0=0.0f;
int F=7; //number of the current formant preset
float f1,f2,f3,f4,a1,a2,a3,a4;
f1=f2=f3=f4=100.0f;a1=a2=a3=a4=0.0f;

init_formant();
FILE *f=fopen("sortie.pcm","wb");
for(int ns=0;ns<10*44100;ns++)
{
if(0==(ns%11025)){F++;F%=8;} //formant change
f0=12*powf(2.0f,4-4*ns/(10*44100.0f)); //sweep
f0*=1.0f+0.01f*sinf(ns*0.0015f); //vibrato
dp0=f0*(1/22050.0f);
float un_f0=1.0f/f0;
p0+=dp0; //phase increment
p0-=2*(p0>1);
{ //smoothing of the commands.
float r=0.001f;
f1+=r*(F1[F]-f1);f2+=r*(F2[F]-f2);f3+=r*(F3[F]-f3);f4+=r*(F4[F]-f4);
a1+=r*(A1[F]-a1);a2+=r*(A2[F]-a2);a3+=r*(A3[F]-a3);a4+=r*(A4[F]-a4);
}

//The f0/fn coefficients stand for a -3dB/oct spectral envelope
float out=
a1*(f0/f1)*formant(p0,100*un_f0)*porteuse(f1*un_f0,p0)
+0.7f*a2*(f0/f2)*formant(p0,120*un_f0)*porteuse(f2*un_f0,p0)
+ a3*(f0/f3)*formant(p0,150*un_f0)*porteuse(f3*un_f0,p0)
+ a4*(f0/f4)*formant(p0,300*un_f0)*porteuse(f4*un_f0,p0);

short s=short(15000.0f*out);
fwrite(&s,2,1,f);fwrite(&s,2,1,f); //fichier raw pcm stereo
}
fclose(f);
return 0;
}

```

Comments

from : Baltazar

comment : Quite interesting and efficient for an algo that does not use any filter ;-)

from : Wait.

comment : What header files are you including?

from : phoenix-69

comment : Very funny sound !

[Another cheap sinusoidal LFO](#) (click this to go back to the index)

References : Posted by info[at]e-phonic[dot]com

Notes :

Some pseudo code for a easy to calculate LFO.

You can even make a rough triangle wave out of this by subtracting the output of 2 of these with different phases.

PJ

Code :

```
r = the rate 0..1

-----
p += r
if(p > 1) p -= 2;
out = p*(1-abs(p));
-----
```

Comments

from : music-dsp [[a t]] umminger.com

comment : Slick! I like it!

Sincerely,
Frederick Umminger

from : balazs.szoradi [[a t]] essnet.se

comment : Great! just what I wanted a fast trinagle lfo. :D

```
float rate = 0..1;
float phase1 = 0;
float phase2 = 0.1f;
```

```
-----
phase1 += rate;
if (phase1>1) phase1 -= 2;
```

```
phase2 += rate;
if (phase2>1) phase2 -= 2;
```

```
out = (phase1*(1-abs(phase1)) - phase2*(1-abs(phase2))) * 10;
```

from : scoofy [[a t]] inf.elte.hu

comment : Nice! If you want the output range to be between -1..1 then use:

```
-----
p += r
if(p > 2) p -= 4;
out = p*(2-abs(p));
-----
```

from : scoofy [[a t]] inf.elte.hu

comment : A better way of making a triangle LFO (out range is -1..1):

```
rate = 0..1;
p = -1;
{
  p += rate;
  if (p>1) p -= 4.0f;
  out = abs(-(abs(p)-2))-1;
}
```

[another LFO class](#) (click this to go back to the index)

References : Posted by mdsp

Linked file : [LFO.zip](#)

Notes :
This LFO uses an unsigned 32-bit phase and increment whose 8 Most Significant Bits address a Look-up table while the 24 Least Significant Bits are used as the fractional part.

Note: As the phase overflow automatically, the index is always in the range 0-255.

It performs linear interpolation, but it is easy to add other types of interpolation.

Don't know how good it could be as an oscillator, but I found it good enough for a LFO.
BTW there is also different kind of waveforms.

Modifications:

We could use phase on 64-bit or change the proportion of bits used by the index and the fractional part.

Comments

from : aja [[a t]] sonardyne.co.uk

comment : This type of oscillator is known as a numerically controlled oscillator (nco) or phase accumulator synthesiser. Integrated circuits that implement it in hardware are available such as the AD7008 from Analog Devices.

The frequency resolution is very high and is $= (\text{SampleRate})/32^2$. So if clocked at 44.1Khz the frequency resolution would be 0.00001026Hz!

As you said the output waveform can be whatever shape you choose to put in the lookup table. The phase register is already in saw tooth form.

Regards,

Tony

from : thaddy [[a t]] thaddy.com

comment : It works great!

Here's a Delphi version I just knocked up. Both VCL and KOL supported.

code:

```
unit PALFO;
//
// purpose: LUT based LFO
// author: © 2004, Thaddy de Koning
// Remarks: Translated from c++ sources by Remy Mueller, www.musicdsp.org
```

```
interface
```

```
uses
{$IFDEF KOL}
  Windows, Kol, KolMath;
{$ELSE}
  Windows, math;
{$ENDIF}
```

```
const
k1Div24lowerBits = 1/(1 shl 24);
```

```
WFStrings:array[0..4] of string =
('triangle','sinus','sawtooth','square','exponent');
```

```
type
```

```
  Twaveform = (triangle, sinus, sawtooth, square, exponent);
```

```
{$IFDEF KOL}
  PPaLfo = ^TPaLfo;
  TPaLfo = object(TObj)
{$ELSE}
  TPaLfo = class
{$ENDIF}
  private
    FTable:array[0..256] of Single;// 1 more for linear interpolation
    FPhase,
    FInc:Single;
    FRate: Single;
    FSampleRate: Single;
    FWaveForm: TWaveForm;
    procedure SetRate(const Value: Single);
    procedure SetSampleRate(const Value: Single);
    procedure SetWaveForm(const Value: TWaveForm);
  public
{$IFDEF KOL}
    constructor create(SampleRate:Single);virtual;
{$ENDIF}
    // increments the phase and outputs the new LFO value.
    // return the new LFO value between [-1;+1]
    function WaveformName:String;
    function Tick:Single;
    // The rate in Hz
```

```

property Rate:Single read FRate write SetRate;
// The Samplerate
property SampleRate:Single read FSampleRate write SetSampleRate;
property WaveForm:TWaveForm read FWaveForm write SetWaveForm;
end;

{$IFDEF KOL}
function NewPaLfo(aSamplerate:Single):PPaLfo;
{$ENDIF}

implementation

{ TPaLfo }
{$IFDEF KOL}
function NewPaLfo(aSamplerate:Single):PPaLfo;
begin
  New(Result,Create);
  with Result^ do
  begin
    FPhase:=0;
    Finc:=0;
    FSamplerate:=aSamplerate;
    SetWaveform(triangle);
    FRate:=1;
  end;
end;
{$ELSE}
constructor TPaLfo.create(SampleRate: Single);
begin
  inherited create;
  FPhase:=0;
  Finc:=0;
  FSamplerate:=aSamplerate;
  SetWaveform(triangle);
  FRate:=1;
end;
{$ENDIF}

procedure TPaLfo.SetRate(const Value: Single);
begin
  FRate := Value;
  // the rate in Hz is converted to a phase increment with the following formula
  // f[ inc = (256*rate/samplerate) * 2^24]
  Finc := (256 * Frate / FsampleRate) * (1 shl 24);
end;

procedure TPaLfo.SetSampleRate(const Value: Single);
begin
  FSampleRate := Value;
end;

procedure TPaLfo.SetWaveForm(const Value: TWaveForm);
var
  i:integer;
begin
  FWaveForm := Value;
  Case Fwaveform of
  sinus:
    for i:=0 to 256 do
      FTable[i] := sin(2*pi*(i/256));
  triangle:
    begin
      for i:=0 to 63 do
        begin
          FTable[i] := i / 64;
          FTable[i+64] :=(64-i) / 64;
          FTable[i+128] := - i / 64;
          FTable[i+192] := - (64-i) / 64;
        end;
      FTable[256] := 0;
    end;
  sawtooth:
    begin
      for i:=0 to 255 do
        FTable[i] := 2*(i/255) - 1;
      FTable[256] := -1;
    end;
  square:
    begin
      for i:=0 to 127 do
        begin
          FTable[i] := 1;
          FTable[i+128] := -1;
        end;
    end;
  end;
end;

```

```

    FTable[256] := 1;
end;
exponent:
begin
    // symmetric exponent similar to triangle
    for i:=0 to 127 do
        begin
            FTable[i] := 2 * ((exp(i/128) - 1) / (exp(1) - 1)) - 1 ;
            FTable[i+128] := 2 * ((exp((128-i)/128) - 1) / (exp(1) - 1)) - 1 ;
        end;
        FTable[256] := -1;
    end;
end;
end;

```

```

function TPaLfo.WaveformName:String;
begin
    result:=WFStrings[Ord(Fwaveform)];
end;

```

```

function TPaLfo.Tick: Single;
var
    i:integer;
    frac:Single;
begin
    // the 8 MSB are the index in the table in the range 0-255
    i := PInteger(Fphase)^ shr 24;
    // and the 24 LSB are the fractional part
    frac := (PInteger(Fphase)^ and $0FFFFFFF) * k1Div24lowerBits;
    // increment the phase for the next tick
    Fphase :=Fphase + Finc; // the phase overflows itself
    Result:= Ftable[i]*(1-frac) + Ftable[i+1]* frac; // linear interpolation
end;

end.

```

from : thaddy [[a t]] thaddy.com
comment : Oops,

This one is correct:

```

code:
unit PALFO;
//
// purpose: LUT based LFO
// author: © 2004, Thaddy de Koning
// Remarks: Translated from c++ sources by Remy Mueller, www.musicdsp.org

```

```

interface
uses
{$IFDEF KOL}
    Windows, Kol,KolMath;
{$ELSE}
    Windows, math;
{$ENDIF}

```

```

const
k1Div24lowerBits = 1/(1 shl 24);

```

```

WFStrings:array[0..4] of string =
('triangle','sinus', 'sawtooth', 'square', 'exponent');

```

```

type
    Twaveform = (triangle, sinus, sawtooth, square, exponent);

```

```

{$IFDEF KOL}
    PPaLfo = ^TPaLfo;
    TPaLfo = object(TObj)
{$ELSE}
    TPaLfo = class
{$ENDIF}
private
    FTable:array[0..256] of Single;// 1 more for linear interpolation
    FPhase,
    FInc:dword;
    FRate: Single;
    FSampleRate: Single;
    FWaveForm: TWaveForm;
    procedure SetRate(const Value: Single);

```

```

    procedure SetSampleRate(const Value: Single);
    procedure SetWaveForm(const Value: TWaveForm);
public
{$IFDEF KOL}
    constructor create(SampleRate:Single);virtual;
{$ENDIF}
    // increments the phase and outputs the new LFO value.
    // return the new LFO value between [-1;+1]
    function WaveformName:String;
    function Tick:Single;
    // The rate in Hz
    property Rate:Single read FRate write SetRate;
    // The Samplerate
    property SampleRate:Single read FSampleRate write SetSampleRate;
    property WaveForm:TWaveForm read FWaveForm write SetWaveForm;
end;

```

```

{$IFDEF KOL}
function NewPaLfo(aSamplerate:Single):PPaLfo;
{$ENDIF}

```

implementation

```

{ TPaLfo }
{$IFDEF KOL}
function NewPaLfo(aSamplerate:Single):PPaLfo;
begin
    New(Result,Create);
    with Result^ do
    begin
        FPhase:=0;
        FSamplerate:=aSamplerate;
        SetWaveform(sinus);
        Rate:=1;
    end;
end;
{$ELSE}
constructor TPaLfo.create(SampleRate: Single);
begin
    inherited create;
    FPhase:=0;
    FSamplerate:=aSamplerate;
    SetWaveform(sinus);
    FRate:=1;
end;
{$ENDIF}

```

```

procedure TPaLfo.SetRate(const Value: Single);
begin
    FRate := Value;
    // the rate in Hz is converted to a phase increment with the following formula
    // f[ inc = (256*rate/samplerate) * 2^24]
    Finc := round((256 * Frate / FsampleRate) * (1 shl 24));
end;

```

```

procedure TPaLfo.SetSampleRate(const Value: Single);
begin
    FSampleRate := Value;
end;

```

```

procedure TPaLfo.SetWaveForm(const Value: TWaveForm);
var
    i:integer;
begin
    FWaveForm := Value;
    Case Fwaveform of
    sinus:
        for i:=0 to 256 do
            FTable[i] := sin(2*pi*(i/256));
    triangle:
        begin
            for i:=0 to 63 do
                begin
                    FTable[i] := i / 64;
                    FTable[i+64] :=(64-i) / 64;
                    FTable[i+128] := - i / 64;
                    FTable[i+192] := - (64-i) / 64;
                end;
            FTable[256] := 0;
        end;
    sawtooth:
        begin
            for i:=0 to 255 do
                FTable[i] := 2*(i/255) - 1;

```



```

    FTable[256] := -1;
end;
square:
begin
  for i:=0 to 127 do
    begin
      FTable[i] := 1;
      FTable[i+128] := -1;
    end;
    FTable[256] := 1;
  end;
exponent:
begin
  // symmetric exponent similar to triangle
  for i:=0 to 127 do
    begin
      FTable[i] := 2 * ((exp(i/128) - 1) / (exp(1) - 1)) - 1 ;
      FTable[i+128] := 2 * ((exp((128-i)/128) - 1) / (exp(1) - 1)) - 1 ;
    end;
    FTable[256] := -1;
  end;
end;
end;

```

```

function TPaLfo.WaveformName:String;
begin
  result:=WFStrings[Ord(Fwaveform)];
end;

```

```

function TPaLfo.Tick: Single;
var
  i:integer;
  frac:Single;
begin
  // the 8 MSB are the index in the table in the range 0-255
  i := Fphase shr 24;
  // and the 24 LSB are the fractionnal part
  frac := (Fphase and $00FFFFFF) * k1Div24lowerBits;
  // increment the phase for the next tick
  Fphase :=Fphase + Finc; // the phase overflows itself
  Result:= Ftable[i]*(1-frac) + Ftable[i+1]* frac; // linear interpolation
end;

end.

```

antialiased square generator (click this to go back to the index)

Type : 1st April edition

References : Posted by Paul Sernine

Notes :

It is based on a code by Thierry Rochebois, obfuscated by me.
It generates a 16bit MONO raw pcm file. Have Fun.

Code :

```
//sqrfish.cpp
#include <math.h>
#include <stdio.h>
//obfuscation P.Sernine
int main() {float ccc,cccc=0,CC=0,cc=0,CCCC,
CCC,C,c; FILE *CCCCCC=fopen("sqrfish.pcm",
"wb" ); int ccccc= 0; float CCCCC=6.89e-6f;
for(int CCCCC=0;CCCCC<1764000;CCCCC++) {
if(!(CCCCC%7350)){if(++cccc>=30){ ccccc =0;
CCCC*=2;}CCC=1;}ccc=CCCC*expf(0.057762265f*
"aiakahiafahadfaiakahiahafahadf"[cccc]);CCCC
=0.75f-1.5f*ccc;cccc+=ccc;CCC*=0.9999f;cccc-=
2*(cccc>1);C=cccc+CCCC*CC; c=cccc+CCCC*cc; C
-=2*(C>1);c-=2*(c>1);C+=2*(C<-1); c+=1+2
*(c<-1);c-=2*(c>1);C=C*C*(2 *C*C-4);
c=c*c*(2*c*c-4); short ccccc=short(15000.0f*
CCC*(C-c )*CCC);CC=0.5f*(1+C+CC);cc=0.5f*(1+
c+cc); fwrite(&cccc,2,1,CCCCC);}
//algo by Thierry Rochebois
fclose(CCCCC);
return 0000000;}
```

Comments

from : cortex [[a t]] gmx.net

comment : you can shove your obfuscated code up your ass :]

from : Paul

comment : So, what ? What's your problem ?

from : bonbon [[a t]] elisa.com

comment : obfuscated code is stupid but it was done on fools day...

it sounds good, no aliasing.

it is quite ez to deobfuscate but i still don't get the algo.

bonaveture rosignol

from : bonbon [[a t]] elisa.com

comment : i think i got it.

it looks like a sort of looped fm/waveshaping thing.

from : fisher man

comment : I tried it and it works fine.

It is easy to decipher.

It uses a single phase accumulator, it is offseted and it feeds two self FM modulated sinewaves (sort of bandlimited saw). The sinewaves are approximated with polynomials and are subtracted from each other to obtain a square like sound.

from : myk [[a t]] golg.com

comment : nice fishy pattern... a bit hard to read ;),

but very organized and readable

i wonder why pro's don't write their code like this????

whats with all the C's and

```
ccc=CCCC*expf(0.057762265f*
```

```
"aiakahiafahadfaiakahiahafahadf"[cccc]);CCCC
```

```
=0.75f-1.5f*ccc;cccc+=ccc;
```

insert quote from the first comment here :)

from : kfkqff [[a t]] gmail.com

comment : Wouldn't it be easier to create a tab of the waveform (in this case a 2 element table), and address it with float instead of int with linear interpolation?

from : cccfffaakakakak [[a t]] hotmail.com

comment : Complete waste of bandwidth thank you.

[Arbitrary shaped band-limited waveform generation \(using oversampling and low-pass filtering\)](#) (click this to go back to the index)

[References](#) : Posted by remage[AT]kac[DOT]poliod[DOT]hu

Code :

Arbitrary shaped band-limited waveform generation
(using oversampling and low-pass filtering)

There are many articles about band-limited waveform synthesis techniques, that provide correct and fast methods for generating classic analogue waveforms, such as saw, pulse, and triangle wave. However, generating arbitrary shaped band-limited waveforms, such as the "sawsin" shape (found in this source-code archive), seems to be quite hard using these techniques.

My analogue waveforms are generated in a very high sampling rate (actually it's 1.4112 GHz for 44.1 kHz waveforms, using 32x oversampling). Using this sample-rate, the amplitude of the aliasing harmonics are negligible (the base analogue waveforms has exponentially decreasing harmonics amplitudes).

Using a 511-tap windowed sinc FIR filter (with Blackman-Harris window, and 12 kHz cutoff frequency) the harmonics above 20 kHz are killed, the higher harmonics (that cause the sharp overshoot at step response) are dampened.

The filtered signal downsampled to 44.1 kHz contains the audible (non-aliased) harmonics only.

This waveform synthesis is performed for wavetables of 4096, 2048, 1024, ... 8, 4, 2 samples. The real-time signal is interpolated from these waveform-tables, using Hermite-(cubic-)interpolation for the waveforms, and linear interpolation between the two wavetables near the required note.

This procedure is quite time-consuming, but the whole waveform (or, in my implementation, the whole waveform-set) can be precalculated (or saved at first launch of the synth) and reloaded at synth initialization.

I don't know if this is a theoretically correct solution, but the waveforms sound good (no audible aliasing). Please let me know if I'm wrong...

Comments

from : Alex [[a t]] smartelectronix.com

comment : Why can't you use fft/iff

to synthesis directly wavetables of 2048,1024,..?

It'd be not so

"time consuming" comparing to FIR filtering.

Further cubic interpolation still might give you audible distortion in some cases.

--Alex.

from : remage [[a t]] kac.poliod.hu

comment : What should I use instead of cubic interpolation? (I had already some aliasing problems with cubic interpolation, but that can be solved by oversampling 4x the realtime signal generation)

Is this theory of generating waves from wavetables of 4096, 2084, ... 8, 4, 2 samples wrong?

from : Alex[AT]smartelectronix.com

comment : I think tablesize should not vary

depending on tone (4096,2048...)

and you'd better stay with the same table size for all notes (for example 4096, 4096...).

To avoid interpolation noise

(it's NOT caused by aliasing)

try to increase wavetable size

and be sure that waveform spectrum has

steep roll off

(don't forget Gibbs phenomena as well).

from : balazs[DOT]szoradi[At]essnet[DOT]se

comment : you say that the higher harmonics (that cause the sharp overshoot at step response) are dampened.

How ? Or is it a result of the filtering ?

from : remage [[a t]] makacs.poliod.hu

comment : Yes. The FIR-filter cutoff is set to 12 kHz, so it dampens the audible frequencies too. This way the frequencies above 20 kHz are about -90 dB (don't remember exactly, but killing all harmonics above 20 kHz was the main reason to set the cutoff to 12 kHz).

Anyway, as Alex suggested, FFT/IFFT seems to be a better solution to this problem.

Audiable alias free waveform gen using width sine (click this to go back to the index)

Type : Very simple

References : Posted by joakim[DOT]dahlstrom[AT]jongame[DOT]com

Notes :

Warning, my english abilities is terribly limited.

How ever, the other day when finally understanding what bandlimited wave creation is (i am a noobie, been doing DSP stuff on and off for a half/year) it hit me i can implement one little part in my synths. It's all about the freq (that i knew), very simple you can reduce alias (the alias that you can hear that is) extremely by keeping track of your frequency, the way i solved it is using a factor, $a_{fact} = 1 - \sin(f \cdot 2\pi)$. This means you can do audiable alias free synthesis without very complex algorithms or very huge tables, even though the sound becomes kind of low-filtered. Probably something like this is mentioned b4, but incase it hasn't this is worth looking up

The psuedo code describes it more.

// Druttis

Code :

```
f := freq factor, 0 - 0.5 (0 to half samplingrate)

afact(f) = 1 - sin(f*2PI)

t := time (0 to ...)
ph := phase shift (0 to 1)
fm := freq mod (0 to 1)

sine(t,f,ph,fm) = sin((t*f+ph)*2PI + 0.5PI*fm*afact(f))

fb := feedback (0 to 1) (1 max saw)

saw(t,f,ph,fm,fb) = sine(t,f,ph,fb*sine(t-1,f,ph,fm))

pm := pulse mod (0 to 1) (1 max pulse)
pw := pulse width (0 to 1) (1 square)

pulse(t,f,ph,fm,fb,pm,pw) = saw(t,f,ph,fm,fb) - (t,f,ph+0.5*pw,fm,fb) * pm
```

I am not completely sure about fm for saw & pulse since i cant test that atm. but it should work :) otherwise just make sure fm are 0 for saw & pulse.

As you can see the saw & pulse wave are very variable.

// Druttis

Comments

from : druttis [[a t]] darkface.pp.se

comment : Um, reading it I can see a big flaw...

$a_{fact}(f) = 1 - \sin(f \cdot 2\pi)$ is not correct!

should be

$a_{fact}(f) = 1 - \sqrt{f \cdot 2 / sr}$

where $sr := \text{samplingrate}$
f should be exceed half sr

from : laurent [[a t]] ohmforce.com

comment : f has already be divided by sr, right ? So it should become :

$a_{fact}(f) = 1 - \sqrt{f \cdot 2}$

And i see a typo (saw forgotten in the second expression) :

$\text{pulse}(t,f,ph,fm,fb,pm,pw) = \text{saw}(t,f,ph,fm,fb) - \text{saw}(t,f,ph+0.5*pw,fm,fb) * pm$

However I haven't checked the formula.

from : 909 [[a t]] gmx.de

comment : Hi Lauent,

I'm new to that DSP stuff and can't get the key to what'S the meaning of a_{fact} ? - Can you explain please!? - Thanks in advice!

from : druttis [[a t]] chello.se

comment : I've been playing around with this for some time. Expect a major update in a while, as soon as I know how to describe it :)

from : druttis [[a t]] chello.se

comment : a_{fact} is used as an amplitude factor for fm or fb depending on the carrier frequency. The higher frequency the lower a_{fact} . It's not completely resolving the problem with aliasing but it is a cheap way that dramatically reduces it.

Bandlimited sawtooth synthesis (click this to go back to the index)

Type : DSF BLIT

References : Posted by emanuel.landeholm [AT] telia.com

Linked file : [synthesis002.txt](#) (this linked file is included below)

Notes :

This is working code for synthesizing a bandlimited sawtooth waveform. The algorithm is DSF BLIT + leaky integrator. Includes driver code.

There are two parameters you may tweak:

- 1) Desired attenuation at nyquist. A low value yields a duller sawtooth but gets rid of those annoying CLICKS when sweeping the frequency up real high. Must be strictly less than 1.0!
- 2) Integrator leakiness/cut off. Affects the shape of the waveform to some extent, esp. at the low end. Ideally you would want to set this low, but too low a setting will give you problems with DC.

Have fun!

/Emanuel Landeholm

(see linked file)

Comments

from : rainbow_stash [[a t]] hotmail.com

comment :

there is no need to use a butterworth design for a simple leaky integrator, in this case actually the variable curcps can be used directly in a simple: leak += curcps * (blit - leak);

this produces a nearly perfect saw shape in almost all cases

Linked files

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Bandlimited synthesis of sawtooth by
 * leaky integration of a DSF BLIT
 *
 * Emanuel Landeholm, March 2002
 * emanuel.landeholm@telia.com
 *
 * Provided \"as is\".
 * Free as in Ef Are Ee Ee.
 */

double pi      = 3.1415926535897932384626433832795029L;
double twopi   = 6.2831853071795864769252867665590058L;

/* Leaky integrator/first order lowpass which
 * shapes the impulse train into a nice
 * -6dB/octave spectrum
 *
 * The cutoff frequency needs to be pretty lowish
 * or the sawtooth will suffer phase distortion
 * at the low end.
 */

typedef struct
{
    double x1, y1;
    double a, b;
} lowpass_t;

/* initializes a lowpass, sets cutoff/leakiness */

void init_lowpass(lowpass_t *lp, double cutoff)
{
    double Omega;

    lp->x1 = lp->y1 = 0.0;

    Omega = atan(pi * cutoff);
    lp->a = -(1.0 - Omega) / (1.0 + Omega);
    lp->b = (1.0 - lp->b) / 2.0;
}

double update_lowpass(lowpass_t *lp, double x)
{
```

```

double y;

y = lp->b * (x + lp->x1) - lp->a * lp->y1;

lp->x1 = x;
lp->y1 = y;

return y;
}

/* dsf blit datatype
*
*/

typedef struct
{
double phase;      /* phase accumulator */
double aNQ;        /* attenuation at nyquist */
double curcps;     /* current frequency, updated once per cycle */
double curper;     /* current period, updated once per cycle */
lowpass_t leaky;   /* leaky integrator */
double N;          /* # partials */
double a;          /* dsf parameter which controls roll-off */
double aN;         /* former to the N */
} blit_t;

/* initializes a blit structure
*
* The aNQ parameter is the desired attenuation
* at nyquist. A low value yields a duller
* sawtooth but gets rid of those annoying CLICKS
* when sweeping the frequency up real high. |aNQ|
* must be strictly less than 1.0! Find a setting
* which works for you.
*
* The cutoff parameter controls the leakiness of
* the integrator.
*/

void init_blit(blit_t *b, double aNQ, double cutoff)
{
b->phase = 0.0;
b->aNQ = aNQ;
b->curcps = 0.0;
b->curper = 0.0;
init_lowpass(&b->leaky, cutoff);
}

/* Returns a sawtooth computed from a leaky integration
* of a DSF bandlimited impulse train.
*
* cps (cycles per sample) is the fundamental
* frequency: 0 -> 0.5 == 0 -> nyquist
*/

double update_blit(blit_t *b, double cps)
{
double P2, beta, Nbeta, cosbeta, n, d, blit, saw;

if(b->phase >= 1.0 || b->curcps == 0.0)
{
/* New cycle, update frequency and everything
* that depends on it
*/

if(b->phase >= 1.0)
b->phase -= 1.0;

b->curcps = cps;          /* this cycle\'s frequency */
b->curper = 1.0 / cps;    /* this cycle\'s period */

P2 = b->curper / 2.0;
b->N = 1.0 + floor(P2); /* # of partials incl. dc */

/* find the roll-off parameter which gives
* the desired attenuation at nyquist
*/

b->a = pow(b->aNQ, 1.0 / P2);
b->aN = pow(b->a, b->N);
}

beta = twopi * b->phase;

```

```

Nbeta = b->N * beta;
cosbeta = cos(beta);

/* The dsf blit is scaled by 1 / period to give approximately the same
 * peak-to-peak over a wide range of frequencies.
 */

n = 1.0 -
  b->aN * cos(Nbeta) -
  b->a * (cosbeta - b->aN * cos(Nbeta - beta));
d = b->curper * (1.0 + b->a * (-2.0 * cosbeta + b->a));

b->phase += b->curcps; /* update phase */

blit = n / d - b->curcps; /* This division can only fail if |a| == 1.0
 * Subtracting the fundamental frq rids of DC
 */

saw = update_lowpass(&b->leaky, blit); /* shape blit spectrum into a saw */

return saw;
}

/* driver code - writes headerless 44.1 16 bit PCM to stdout */

static int clipped = 0;

static void ADC_out(double x)
{
  short s;

  if(x > 1.0)
  {
    ++clipped;
    x = 1.0;
  }
  else if(x < -1.0)
  {
    ++clipped;
    x = -1.0;
  }

  s = 32767.0 * x;

  fwrite(&s, sizeof(s), 1, stdout);
}

int main(int argc, char **argv)
{
  int i, L;
  double x, cps, cm;
  blit_t b;

  L = 1000000;

  init_blit(&b, 0.5, 0.0001);

  /* sweep from 40 to 20000 Hz */

  cps = 40.0 / 44100.0;
  cm = pow(500.0, 1.0 / (double)L);

  for(i = 0; i < L; ++i)
  {
    x = 2.0 * update_blit(&b, cps);
    ADC_out(x);

    cps *= cm;
  }

  fprintf(stderr, "%d values were clipped\n", clipped);

  return 0;
}

```


Bandlimited waveform generation (click this to go back to the index)

Type : waveform generation

References : Posted by Joe Wright

Linked file : [bandlimited.cpp](#) (this linked file is included below)

Linked file : [bandlimited.pdf](#)

Notes :
(see linkfile)

Linked files

```
// An example of generating the sawtooth and parabola wavetables
// for storage to disk.
//
// SPEED=sampling rate, e.g. 44100.0f
// TUNING=pitch of concert A, e.g. 440.0f

////////////////////////////////////
// Wavetable reverse lookup
// Given a playback rate of the wavetable, what is wavetables index?
//
// rate = f.wavesize/fs e.g. 4096f/44100
// max partials = nyquist/f = wavesize/2rate e.g. 2048/rate
//
// using max partials we could then do a lookup to find the wavetables index
// in a pre-calculated table
//
// however, we could skip max partials, and lookup a table based on a
// function of f (or rate)
//
// the first few midi notes (0 - 9) differ by < 1 so there are duplicates
// values of (int) f.
// therefore, to get an index to our table (that indexes the wavetables)
// we need 2f
//
// to get 2f from rate we multiply by the constant
// 2f = 2.fs/wavesize e.g. 88200/4096
//
// our lookup table will have a length>25087 to cover the midi range
// we'll make it 32768 in length for easy processing

int a,b,n;
float* data;
float* sinetable=new float[4096];
float* datap;
for(b=0;b<4096;b++)
    sinetable[b]=sin(TWOPI*(float)b/4096.0f);
int partials;
int partial;
int partialindex,reverseindex,lastnumpartials;
float max,m;
int* reverse;

// sawtooth

data=new float[128*4096];
reverse=new int[32768];

reverseindex=0;
partialindex=0;
lastnumpartials=-1;

for(n=0;n<128;n++)
{
    partials=(int)((SPEED*0.5f)/float(TUNING*(float)pow(2,(float) (n-69)/12.0f))); //(int) NYQUIST/f
    if(partial!=lastnumpartials)
    {
        datap=&data[partialindex*4096];
        for(b=0;b<4096;b++)
            datap[b]=0.0f; //blank wavetable
        for(a=0;a<partials;a++)
        {
            partial=a+1;
            m=cos((float)a*HALFPI/(float)partials); //gibbs
            m*=m; //gibbs
            m/=(float)partial;
            for(b=0;b<4096;b++)
                datap[b]+=m*sinetable[(b*partial)%4096];
        }
    }
}
```

```

    }
    lastnumpartials=partials;
    a=int(2.0f*TUNING*(float)pow(2,(float) (n-69)/12.0f)); //2f
    for(b=reverseindex;b<=a;b++)
        reverse[b]=partialindex;
    reverseindex=a+1;
    partialindex++;
}
}

for(b=reverseindex;b<32768;b++)
    reverse[b]=partialindex-1;

ar << (int) partialindex; //number of waveforms
ar << (int) 4096; //waveform size (in samples)

max=0.0;
for(b=0;b<4096;b++)
{
    if(fabs(*(data+b))>max) //normalise to richest waveform (0)
        max=(float)fabs(*(data+b));
}
for(b=0;b<4096*partialindex;b++)
{
    *(data+b)/=max;
}

//ar.Write(data,4096*partialindex*sizeof(float));
//ar.Write(reverse,32768*sizeof(int));

delete [] data;
delete [] reverse;
}
// end sawtooth

// parabola

data=new float[128*4096];
reverse=new int[32768];

reverseindex=0;
partialindex=0;
lastnumpartials=-1;

float sign;

for(n=0;n<128;n++)
{
    partials=(int)((SPEED*0.5f)/float(TUNING*(float)pow(2,(float) (n-69)/12.0f)));
    if(partials!=lastnumpartials)
    {
        datap=&data[partialindex*4096];
        for(b=0;b<4096;b++)
            datap[b]=PI*PI/3.0f;
        sign=-1.0f;
        for(a=0;a<partials;a++)
        {
            partial=a+1;
            m=cos((float)a*HALFPI/(float)partials); //gibbs
            m*=m; //gibbs
            m/=(float)(partial*partial);
            m*=4.0f*sign;
            for(b=0;b<4096;b++)
                datap[b]+=m*sinetable[((b*partial)+1024)%4096]; //note, parabola uses cos
            sign=-sign;
        }
        lastnumpartials=partials;
        a=int(2.0f*TUNING*(float)pow(2,(float) (n-69)/12.0f)); //2f
        for(b=reverseindex;b<=a;b++)
            reverse[b]=partialindex;
        reverseindex=a+1;
        partialindex++;
    }
}

for(b=reverseindex;b<32768;b++)
    reverse[b]=partialindex-1;

ar << (int) partialindex; //number of waveforms
ar << (int) 4096; //waveform size (in samples)

max=0.0;
for(b=0;b<4096;b++)

```

```

{
  if(fabs(*(data+b))>max) //normalise to richest waveform (0)
    max=(float)fabs(*(data+b));
}
max*=0.5;
for(b=0;b<4096*partialindex;b++)
{
  *(data+b)/=max;
  *(data+b)-=1.0f;
}

//ar.Write(data,4096*partialindex*sizeof(float));
//ar.Write(reverse,32768*sizeof(int));

delete [] data;
delete [] reverse;
}
// end parabola

////////////////////////////////////
// An example of playback of a sawtooth wave
// This is not optimised for easy reading
// When optimising you'll need to get this in assembly (especially those
// float to int conversions)
////////////////////////////////////

#define WAVETABLE_SIZE (1 << 12)
#define WAVETABLE_SIZEF WAVETABLE_SIZE.0f
#define WAVETABLE_MASK (WAVETABLE_SIZE - 1)

float index;
float rate;
int wavetableindex;
float ratetofloatfactor;
float* wavetable;

void setupnote(int midinote /*0 - 127*/)
{
  float f=TUNING*(float)pow(2,(float) (midinote-69)/12.0f));
  rate=f*WAVETABLE_SIZEF/SPEED;
  ratetofloatfactor=2.0f*SPEED/WAVETABLE_SIZEF;
  index=0.0f;
  wavetableindex=reverse[(int)(2.0f*f)];
  wavetable=&sawtoothdata[wavetableindex*WAVETABLE_SIZE];
}

void generatesample(float* buffer,int length)
{
  int currentsample,
  int nextsample;
  float m;
  float temprate;
  while(length--)
  {
    currentsample=(int) index;
    nextsample=(currentsample+1) & WAVETABLE_MASK;
    m=index-(float) currentsample; //fractional part
    *buffer++=(1.0f-m)*wavetable[currentsample]+m*wavetable[nextsample]; //linear interpolation
    rate*=slide; //slide coefficient if required
    temprate=rate*fm; //frequency modulation if required
    index+=temprate;
    if(index>WAVETABLE_SIZEF)
    {
      //new cycle, respecify wavetable for sliding
      wavetableindex=reverse[(int)(ratetofloatfactor*temprate)];
      wavetable=&sawtoothdata[wavetableindex*WAVETABLE_SIZE];
      index-=WAVETABLE_SIZEF;
    }
  }
}

```

[Bandlimited waveform generation with hard sync](#) (click this to go back to the index)

References : Posted by Emanuel Landeholm

Linked file : <http://www.algonet.se/~e-san/hardsync.tar.gz>

Comments

from : emanuel.landeholm [[a t]] telia.com

comment : The tar-gz archive has moved to

<http://web.telia.com/~u84605054/hardsync.tar.gz>

[Bandlimited waveforms synopsis.](#) (click this to go back to the index)

References : Joe Wright

Linked file : [waveforms.txt](#) (this linked file is included below)

Notes :
(see linkfile)

Comments

from : dflatccrmatdotstanforddotedu

comment : The abs(sin) method from the Lane CMJ paper is not bandlimited! It's basically just a crappy method for BLIT.

You forgot to mention Eli Brandt's minBLEP method. It's the best! You just have to know how to properly generate a nice minblep table... (slightly dilated, see Stilson and Smith BLIT paper, at the end regarding table implementation issues)

Linked files

From: "Joe Wright" <joe@nyrsound.com>
To: <music-dsp@shoko.calarts.edu>
Subject: Re: waveform discussions
Date: Tue, 19 Oct 1999 14:45:56 +0100

After help from this group and reading various literature I have finished my waveform engine. As requested, I am now going to share some of the things I have learnt from a practical viewpoint.

Problem:

The waveforms of interest are sawtooth, square and triangle.

The waveforms must be bandlimited (i.e. fitting under Nyquist). This precludes simple generation of the waveforms. For example, the analogue/continuous formula (which has infinite harmonics):

$$s(t) = (t * f_0) \bmod 1 \quad (t = \text{time}, f_0 = \text{frequency of note})$$

produces aliasing that cannot be filtered out when converted to the digital/discrete form:

$$s(n) = (f_0 * n * T_s) \bmod 1 \quad (n = \text{sample}, T_s \text{ equals } 1 / \text{sampling rate})$$

The other condition of this problem is that the waveforms are generatable in real-time. Additionally, bonuses are given for solutions which allow pulse-width modulation and triangle asymmetry.

The generation of these waves is non-trivial and below is discussed three techniques to solve the problem - wavetables, approximation through processing of $|\sin \text{wave}|$ and BLIT integration. BLIT integration is discussed in depth.

Wavetables:

You can generate a wavetable for the waveform by summing sine waves up to the Nyquist frequency. For example, the sawtooth waveform can be generated by:

$$s(n) = \sum_{k=1, n} (1/k * \sin(2\pi * f_0 * k * T_s)) \quad \text{where } f_0 * n < T_s/2$$

The wavetable can then be played back, pitched up or down subject that pitched $f * n < T_s/2$. Anything lower will not alias but it may lack some higher harmonics if pitched too low.

To cover these situations, use multiple wavetables describing different frequency ranges within which it is fine to pitch up or down.

You may need to compromise between number of wavetables and accuracy because of memory considerations (especially if over-sampling). This means some wavetables will have to cover larger ranges than they should. As long as the range is too far in the lower direction rather than higher, you will not alias (you will just miss some higher harmonics).

With wavetables you can add a sawtooth to an inverted sawtooth offset in time, to produce a pulse/square wave. Vary the offset to vary the pulse width. Asymmetry of triangle waves is not possible (as far as I know) though.

Approximation through processing of |sinewave|

This method is discussed in detail in 'Modeling Analog Synthesis with DSPs' - Computer Music Journal, 21:4 pp.23 - 41, Winter 1997, Lane, Hoory, Marinez and Wang.

The basic idea starts with the generation of a sawtooth by feeding $\text{abs}(\sin(n))$ into a lowpass followed by a highpass filter. This approximates (quite well supposedly) a bandlimited sawtooth. Unfortunately, details of what the cutoff for the lowpass should be were not precise in the paper (although the highpass cutoff was given).

Square wave and triangle wave approximation was implemented by subtracting $\text{abs}(\sin(n))$ and $\text{abs}(\sin(n/2))$. With different gains, pulse width (and I presume asymmetry) were possible.

For more information, consult the paper.

BLIT intergration

This topic refers to the paper 'Alias-Free Digital Synthesis of Classic Analog Waveforms' by Tim Stilson and Julius Smith of CCRMA. The paper can be found at <http://www-ccrma.stanford.edu/~stilti/papers>

BLIT stands for bandlimited impluse train. I'm not going to go into the theory, you'll have to read the paper for that. However, simply put, a pulse train of the form 10000100001000 etc... is not bandlimited.

The formular for a BLIT is as follows:

$$\text{BLIT}(x) = (m/p) * (\sin(\text{PI}*x*m/p)/ (m*\sin(\text{PI}*x/p)))$$

x =sample number ranging from 1 to period
 p =period in samples (f_s/f_0). Although this should theoretically not be an interger, I found pratically speaking it needs to be.
 $m=2*((\text{int})p/2)+1$ (i.e. when p =odd, $m=p$ otherwise $m=p+1$)

[note] in the paper they describe m as the largest odd interger not exceeding the period when in fact their formular for m (which is the one that works in practive) makes it $(\text{int})p$ or $(\text{int})p +1$

As an extension, we also have a bipolar version:

$$\text{BP-BLIT } k(x) = \text{BLIT}(x) - \text{BLIT}(x+k) \quad (\text{where } k \text{ is in the range } [0, \text{period}])$$

Now for the clever bit. Lets start with the square/rectangle wave. Through intergration we get:

$$\text{Rect}(n) = \text{Sum}(i=0, n) (\text{BP-BLIT } k_0(i) - C_4)$$

C_4 is a DC offset which for the BP-BLIT is zero. This gives a nice iteration for $\text{rect}(n)$:

$$\text{Rect}(n) = \text{Rect}(n-1) + \text{BP-BLIT } k_0(n) \quad \text{where } k_0 \text{ is the pulse width between } [0, \text{period}] \text{ or in practive } [1, \text{period}-1]$$

A triangle wave is given by:

$$\text{Tri}(n) = \text{Sum}(i=0, n) (\text{Rect}(k) - C_6)$$

$$\text{Tri}(n) = \text{Tri}(n-1) + \text{Rect}(n) - C_6$$

$$C_6 = k_0/\text{period}$$

The triangle must also be scaled:

$$\text{Tri}(b) = \text{Tri}(n-1) + g(f, d) * (\text{Rect}(n) - C_6)$$

$$\text{where } g(f, d) = 0.99 / (\text{period} * d * (d-1)) \quad d = k_0/\text{period}$$

Theoretically it could be 1.00 / ... but I found numerical error sometimes pushed it over the edge. The paper actually states 2.00/... but for some reason I find this to be incorrect.

Lets look at some rough and ready code. I find the best thing to do is to generate one period at a time and then reset everything. The numerical errors over one period are negligable (based on 32bit float) but if you keep on going without resetting, the errors start to creep in.

```
float period = samplingrate/frequency;
```

```

float m=2*(int)(period/2)+1.0f;
float k=(int)(k0*period);
float g=0.99f/(period*(k/period)*(1-k/period));
float t;
float bpblit;
float square=0.0f;
float triangle=0.0f;
for(t=1;t<=period;t++)
{
    bpblit=sin(PI*(t+1)*m/period)/(m*sin(PI*(t+1)/period));
    bpblit-=sin(PI*(t+k)*m/period)/(m*sin(PI*(t+k)/period));
    square+=bpblit;
    triangle+=g*(square+k/period);
}
square=0;
triangle=0;

```

Highly un-optimised code but you get the point. At each sample(t) the output values are square and triangle respectively.

Sawtooth:

This is given by:

Saw(n) = Sum(k=0,n) (BLIT(k) - C2) [as opposed to BP-BLIT]
Saw(n) = Saw(n-1) + BLIT(n) - C2

Now, C2 is a bit tricky. Its the average of BLIT for that period (i.e. 1/n * Sum (k=0,n) (BLIT(k))). [Note] Blit(0) = 0.

I found the best way to deal with this is to have a lookup table which you have generated and saved to disk as a file which contains a value of C2 for every period you are interested in. This is because I know of no easy way to generate C2 in real-time.

Last thing. My implementation of BLIT gives negative values. Therefore my sawtooth is +C2 rather than -C2.

I hope this helps, any questions don't hesitate to contact me.

Joe Wright - Nyr Sound Ltd
<http://www.nyrsound.com>
info@nyrsound.com

[Bandlimited waveforms...](#) (click this to go back to the index)

[References](#) : Posted by Paul Kellet

[Notes](#) :
(Quoted from Paul's mail)
Below is another waveform generation method based on a train of sinc functions (actually an alternating loop along a sinc between $t=0$ and $t=\text{period}/2$).

The code integrates the pulse train with a dc offset to get a sawtooth, but other shapes can be made in the usual ways... Note that 'dc' and 'leak' may need to be adjusted for very high or low frequencies.

I don't know how original it is (I ought to read more) but it is of usable quality, particularly at low frequencies. There's some scope for optimisation by using a table for sinc, or maybe a a truncated/windowed sinc?

I think it should be possible to minimise the aliasing by fine tuning 'dp' to slightly less than 1 so the sincs join together neatly, but I haven't found the best way to do it. Any comments gratefully received.

```
Code :
float p=0.0f;          //current position
float dp=1.0f;        //change in position per sample
float pmax;           //maximum position
float x;              //position in sinc function
float leak=0.995f;    //leaky integrator
float dc;             //dc offset
float saw;            //output
```

```
//set frequency...

pmax = 0.5f * getSampleRate() / freqHz;
dc = -0.498f/pmax;
```

```
//for each sample...
```

```
p += dp;
if(p < 0.0f)
{
    p = -p;
    dp = -dp;
}
else if(p > pmax)
{
    p = pmax + pmax - p;
    dp = -dp;
}
```

```
x= pi * p;
if(x < 0.00001f)
    x=0.00001f; //don't divide by 0

saw = leak*saw + dc + (float)sin(x)/(x);
```

[Comments](#)

[from](#) : sinewave [[a t]] chello.se

[comment](#) : Hi,

Has anyone managed to implement this in a VST?

If anyone could mail me and talk me through it I'd be very grateful. Yes, I'm a total newbie and yes, I'm after a quick-fix solution...we all have to start somewhere, eh?

As it stands, where I should be getting a sawtooth I'm getting a full-on and inaudible signal...!

Even a small clue would be nice.

Cheers,

A

[Butterworth](#) (click this to go back to the index)

Type : LPF 24dB/Oct

References : Posted by Christian[at]savioursofsoul[dot]de

Code :

First calculate the prewarped digital frequency:

```
K = tan(Pi * Frequency / Samplerate);
```

Now calc some intermediate variables: (see 'Factors of Polynoms' at http://en.wikipedia.org/wiki/Butterworth_filter, especially if you want a higher order like 48dB/Oct)

```
a = 0.76536686473 * Q * K;
```

```
b = 1.84775906502 * Q * K;
```

```
K = K*K; (to optimize it a little bit)
```

Calculate the first biquad:

```
A0 = (K+a+1);
```

```
A1 = 2*(1-K);
```

```
A2 = (a-K-1);
```

```
B0 = K;
```

```
B1 = 2*B0;
```

```
B2 = B0;
```

Calculate the second biquad:

```
A3 = (K+b+1);
```

```
A4 = 2*(1-K);
```

```
A5 = (b-K-1);
```

```
B3 = K;
```

```
B4 = 2*B3;
```

```
B5 = B3;
```

Then calculate the output as follows:

```
Stage1 = B0*Input + State0;
```

```
State0 = B1*Input + A1/A0*Stage1 + State1;
```

```
State1 = B2*Input + A2/A0*Stage1;
```

```
Output = B3*Stage1 + State2;
```

```
State2 = B4*Stage1 + A4/A3*Output + State2;
```

```
State3 = B5*Stage1 + A5/A3*Output;
```

Comments

[from](#) : Christian [[a t]] savioursofsoul.de

[comment](#) : Once you figured it out, it's even possible to do higher order butterworth shelving filters. Here's an example of an 8th order lowshelf.

First we start as usual prewarping the cutoff frequency:

```
K = tan(fW0*0.5);
```

Then we settle up the Coefficient V:

```
V = Power(GainFactor,-1/4)-1;
```

Finally here's the loop to calculate the filter coefficients:

```
for i = 0 to 3
```

```
{
```

```
cm = cos(PI*(i*2+1) / (2*8) );
```

```
B[3*i+0] = 1 / ( 1 + 2*K*cm + K*K + 2*V*K*K + 2*V*K*cm + V*V*K*K);
```

```
B[3*i+1] = 2 * ( 1 - K*K - 2*V*K*K - V*V*K*K);
```

```
B[3*i+2] = (-1 + 2*K*cm - K*K - 2*V*K*K + 2*V*K*cm - V*V*K*K);
```

```
A[3*i+0] = ( 1-2*K*cm+K*K);
```

```
A[3*i+1] = 2*(-1 +K*K);
```

```
A[3*i+2] = ( 1+2*K*cm+K*K);
```

```
}
```

[from](#) : scoofy [[a t]] inf.elte.hu

[comment](#) : Hmm... interesting. I guess the phase response/group delay gets quite funky, which is generally unwanted for an equalizer.

I think the 1/ is not necessary for the first B coefficient! (of course you divide all the other coeffs with the inverse of that coeff at the end...)

I guess the next will be Chebyshev shelving filters ;)

BTW did you check whether my 4 pole highpass Butterworth code is correct?

Peter

[from](#) : Christian [[a t]] savioursofsoul.de

[comment](#) : The 1/ is of course an error here. It's left of my own implementation, where I divide directly. Also I think A and B is exchanged.

I've already nearly done all the different filter types (except elliptic filters), but I won't post too much here. The highpass maybe a highpass, but not the exact complementary. At least my (working) implementation looks different.

The lowpass->highpass transform is to replace s with 1/s and by doing this, more than one sign is changing.

from : scoofy [[a t]] inf.elte.hu

comment : Different authors tend to mix up A and B coeffs.

If I take this lowpass derived by bilinear transform and change B0 and B2 to 1, and B1 to -2 then I get a perfect highpass. At least that's what I see in FilterExplorer. Probably you could get the same by replacing s with 1/s and deriving it by bilinear transform.

Well, there are many ways to get a filter working, for example if I replace $\tan(\pi w)$ with $1.0/\tan(\pi w)$, inverse the sign of B1, and replace $A1 = 2*(1-K)$ with $A1 = 2*(K-1)$, I also get the same highpass.

Well, the reason for the sign inversion is that the coeffs you named B1 and B2 here are responsible for the locations of the zeroes. B1 is responsible for the angle, and B2 is for the radius, so if you invert B1 then the zeroes get on the opposite side of the unit circle, so you get a highpass filter. You then need to adjust the gain coefficient (B0) so that the passband gain is 1. Well, this is not a very precise explanation, but this is the reason why this works.

from : Christian [[a t]] savioursofsoul.de

comment : Ok, you're right. I've been doing too much stuff these days that I missed that simple thing. Your version is even numerical better, because there is less potential of annihilation. Thanks for that.

Btw. the group delay really get a little bit 'funky', i've also noticed that, but for not too high orders it doesn't hurt that much.

from : scoofy [[a t]] inf.elte.hu

comment : Well, it isn't a big problem unless you start modulating the filter very fast... then you get this strange pitch-shifting effect ;)

Well, I read sometimes that when you do EQing, phase is a very important factor. I guess that's why ppl sell a lot of linear phase EQ plugins. Or just the marketing? Don't know, haven't compared linear and non-linear phase stuff very much..

from : scoofy [[a t]] inf.elte.hu

comment : If you have a Q factor different than 1, then filter won't be a Butterworth filter (in terms of maximally flat passband). So, your filter is a kind of a tweaked Butterworth filter with added resonance.

Highpass version should be:

$$A0 = (K+a+1);$$

$$A1 = 2*(1-K);$$

$$A2 = (a-K-1);$$

$$B0 = 1;$$

$$B1 = -2;$$

$$B2 = 1;$$

Calculate the second biquad:

$$A3 = (K+b+1);$$

$$A4 = 2*(1-K);$$

$$A5 = (b-K-1);$$

$$B3 = 1;$$

$$B4 = -2;$$

$$B5 = 1;$$

The rest is the same. You might want to leave out B0, B2, B3 and B5 completely, because they all equal to 1, and optimize the highpass loop as:

$$\text{Stage1} = \text{Input} + \text{State0};$$

$$\text{State0} = B1 * \text{Input} + A1/A0 * \text{Stage1} + \text{State1};$$

$$\text{State1} = \text{Input} + A2/A0 * \text{Stage1};$$

$$\text{Output} = \text{Stage1} + \text{State2};$$

$$\text{State2} = B4 * \text{Stage1} + A4/A3 * \text{Output} + \text{State2};$$

$$\text{State3} = \text{Stage1} + A5/A3 * \text{Output};$$

Anyone confirms this code works? (Being too lazy to throw this into a compiler...)

Cheers,
Peter

from : scoofy [[a t]] inf.elte.hu

comment : And of course it is not a good idea to do divisions in the process loop, because they are very heavy, so the best is to precalculate $A1/A0$, $A2/A0$, $A4/A3$ and $A5/A3$ after the calculation of coefficients:

$$\text{inv_A0} = 1.0/A0;$$

$$A1A0 = A1 * \text{inv_A0};$$

$$A2A0 = A2 * \text{inv_A0};$$

$$\text{inv_A3} = 1.0/A3;$$

$$A4A3 = A4 * \text{inv_A3};$$

$$A5A3 = A5 * \text{inv_A3};$$

(The above should be faster than writing

$$A1A0 = A1/A0;$$

$A2A0 = A2/A0;$
 $A4A3 = A4/A3;$
 $A5A3 = A5/A3;$

but I think some compilers do this optimization automatically.)

Then the lowpass process loop becomes

$Stage1 = B0*Input + State0;$
 $State0 = B1*Input + A1A0*Stage1 + State1;$
 $State1 = B2*Input + A2A0*Stage1;$

$Output = B3*Stage1 + State2;$
 $State2 = B4*Stage1 + A4A3*Output + State2;$
 $State3 = B5*Stage1 + A5A3*Output;$

Much faster, isn't it?

C# Oscillator class (click this to go back to the index)

Type : Sine, Saw, Variable Pulse, Triangle, C64 Noise

References : Posted by neotec

Notes :

Parameters:

Pitch: The Osc's pitch in Cents [0 - 14399] startig at A -> 6.875Hz

Pulsewidth: [0 - 65535] -> 0% to 99.99%

Value: The last Output value, a set to this property 'syncs' the Oscillator

Code :

```
public class SynthOscillator
{
    public enum OscWaveformType
    {
        SAW, PULSE, TRI, NOISE, SINE
    }

    public int Pitch
    {
        get
        {
            return this._Pitch;
        }
        set
        {
            this._Pitch = this.MinMax(0, value, 14399);
            this.OscStep = WaveSteps[this._Pitch];
        }
    }

    public int PulseWidth
    {
        get
        {
            return this._PulseWidth;
        }
        set
        {
            this._PulseWidth = this.MinMax(0, value, 65535);
        }
    }

    public OscWaveformType Waveform
    {
        get
        {
            return this._WaveForm;
        }
        set
        {
            this._WaveForm = value;
        }
    }

    public int Value
    {
        get
        {
            return this._Value;
        }
        set
        {
            this._Value = 0;
            this.OscNow = 0;
        }
    }

    private int _Pitch;
    private int _PulseWidth;
    private int _Value;
    private OscWaveformType _WaveForm;

    private int OscNow;
    private int OscStep;
    private int ShiftRegister;

    public const double BaseFrequency = 6.875;
    public const int SampleRate = 44100;
    public static int[] WaveSteps = new int[0];
    public static int[] SineTable = new int[0];

    public SynthOscillator()
    {
        if (WaveSteps.Length == 0)
            this.CalcSteps();
    }
}
```

```

    if (SineTable.Length == 0)
        this.CalcSine();

    this._Pitch = 7200;
    this._PulseWidth = 32768;
    this._WaveForm = OscWaveformType.SAW;

    this.ShiftRegister = 0x7ffff8;

    this.OscNow = 0;
    this.OscStep = WaveSteps[this._Pitch];
    this._Value = 0;
}

private void CalcSteps()
{
    WaveSteps = new int[14400];

    for (int i = 0; i < 14400; i++)
    {
        double t0, t1, t2;

        t0 = Math.Pow(2.0, (double)i / 1200.0);
        t1 = BaseFrequence * t0;
        t2 = (t1 * 65536.0) / (double)this.SampleRate;

        WaveSteps[i] = (int)Math.Round(t2 * 4096.0);
    }
}

private void CalcSine()
{
    SineTable = new int[65536];

    double s = Math.PI / 32768.0;

    for (int i = 0; i < 65536; i++)
    {
        double v = Math.Sin((double)i * s) * 32768.0;

        int t = (int)Math.Round(v) + 32768;

        if (t < 0)
            t = 0;
        else if (t > 65535)
            t = 65535;

        SineTable[i] = t;
    }
}

public override int Run()
{
    int ret = 32768;
    int osc = this.OscNow >> 12;

    switch (this._WaveForm)
    {
        case OscWaveformType.SAW:
            ret = osc;
            break;
        case OscWaveformType.PULSE:
            if (osc < this.PulseWidth)
                ret = 65535;
            else
                ret = 0;
            break;
        case OscWaveformType.TRI:
            if (osc < 32768)
                ret = osc << 1;
            else
                ret = 131071 - (osc << 1);
            break;
        case OscWaveformType.NOISE:
            ret = ((this.ShiftRegister & 0x400000) >> 11) |
                ((this.ShiftRegister & 0x100000) >> 10) |
                ((this.ShiftRegister & 0x010000) >> 7) |
                ((this.ShiftRegister & 0x002000) >> 5) |
                ((this.ShiftRegister & 0x000800) >> 4) |
                ((this.ShiftRegister & 0x000080) >> 1) |
                ((this.ShiftRegister & 0x000010) << 1) |
                ((this.ShiftRegister & 0x000004) << 2);
            ret <<= 4;
            break;
        case OscWaveformType.SINE:
            ret = SynthTools.SineTable[osc];
            break;
        default:
            break;
    }

    this.OscNow += this.OscStep;
}

```

```
if (this.OscNow > 0xffffffff)
{
    int bit0 = ((this.ShiftRegister >> 22) ^ (this.ShiftRegister >> 17)) & 0x1;
    this.ShiftRegister <<= 1;
    this.ShiftRegister &= 0x7ffff;
    this.ShiftRegister |= bit0;
}

this.OscNow &= 0xffffffff;

this._Value = ret - 32768;

return this._Value;
}

public int MinMax(int a, int b, int c)
{
    if (b < a)
        return a;
    else if (b > c)
        return c;
    else
        return b;
}
}
```

C++ gaussian noise generation (click this to go back to the index)

Type : gaussian noise generation

References : Posted by paul[at]expdigital[dot]co[dot]uk

Notes :

References :

Tobybears delphi noise generator was the basis. Simply converted it to C++.

Link for original is:

<http://www.musicdsp.org/archive.php?classid=0#129>

The output is in noise.

Code :

```
/* Include requisits */
#include <cstdlib>
#include <ctime>

/* Generate a new random seed from system time - do this once in your constructor */
srand(time(0));

/* Setup constants */
const static int q = 15;
const static float c1 = (1 << q) - 1;
const static float c2 = ((int)(c1 / 3)) + 1;
const static float c3 = 1.f / c1;

/* random number in range 0 - 1 not including 1 */
float random = 0.f;

/* the white noise */
float noise = 0.f;

for (int i = 0; i < numSamples; i++)
{
    random = ((float)rand() / (float)(RAND_MAX + 1));
    noise = (2.f * ((random * c2) + (random * c2) + (random * c2)) - 3.f * (c2 - 1.f)) * c3;
}
```

[chebyshev waveshaper \(using their recursive definition\)](#) (click this to go back to the index)

Type : chebyshev

References : Posted by mdsp

Notes :

someone asked for it on kvr-audio.

I use it in an unreleased additive synth.

There's no oversampling needed in my case since I feed it with a pure sinusoid and I control the order to not have frequencies above $F_s/2$. Otherwise you should oversample by the order you'll use in the function or bandlimit the signal before the waveshaper. unless you really want that aliasing effect... :)

I hope the code is self-explaining, otherwise there's plenty of sites explaining chebyshev polynoms and their applications.

Code :

```
float chebyshev(float x, float A[], int order)
{
    // T0 = 1
    // T1 = x
    // Tn = 2.x.Tn-1 - Tn-2
    // out = sum(Ai*Ti(x)) , i C {1,..,order}
    float Tn_2 = 1.0f;
    float Tn_1 = x;
    float Tn;
    float out = A[0]*Tn_1;

    for(int n=2;n<=order;n++)
    {
        Tn = 2.0f*x*Tn_1 - Tn_2;
        out += A[n-1]*Tn;
        Tn_2 = Tn_1;
        Tn_1 = Tn;
    }
    return out;
}
```

Comments

from : mdsp

comment : BTW you can achieve an interesting effect by feeding back the output in the input. it adds a kind of interesting pitched noise to the signal.

I think VirSyn is using something similar in microTERA.

from : dan [[a t]] bacterie.wanadoo.co.uk

comment : Hi, it was me that asked about this on KVR. It seems that it is possible to use such a waveshaper on a non-sinusoidal input without oversampling; split the input signal into bands, and use the highest frequency in each band to determine which order polynomials to send each band to. The idea about feeding back the output to the input occurred to me as well, good to know that such an effect might be interesting... If I come across any other points of interest while coding this plugin, I'll be glad to mention them on here.

Dan

from : mdsp

comment : of course you can use it on non sinusoidal input, but you won't achieve the same result.

if you express your input as a sum of sinusoids of frequencies [f0 f1 f2 ...] and use the chebyshev polynom of order 2 you won't have $2*[f0 f1 f2...]$ as the resulting frequencies.

As it's a nonlinear function you can't use the superposition theorem anymore.

beware that chebyshev polynoms are sensitive to

the range of your input. Your sinusoid has to have a gain exactly equal to 1 in order to work as expected.

that's a nice trick but it has its limits.

from : nobody [[a t]] nowhere.com

comment : Stop saying "emnhqwjyv rgpavs mbejfd kthz xmsvyif dikfo ugnfnjsa."

Cubic polynomial envelopes (click this to go back to the index)

Type : envelope generation

References : Posted by Andy Mucho

Notes :

This function runs from:

startlevel at Time=0

midlevel at Time/2

endlevel at Time

At moments of extreme change over small time, the function can generate out of range (of the 3 input level) numbers, but isn't really a problem in actual use with real numbers, and sensible/real times..

Code :

```
time = 32
startlevel = 0
midlevel = 100
endlevel = 120
k = startlevel + endlevel - (midlevel * 2)
r = startlevel
s = (endlevel - startlevel - (2 * k)) / time
t = (2 * k) / (time * time)
bigr = r
bigs = s + t
bigt = 2 * t
```

```
for(int i=0;i<time;i++)
{
    bigr = bigr + bigs
    bigs = bigs + bigt
}
```

Comments

from : thecourier [[a t]] infinito.it

comment : I have try this and it works fine, but what hell is bigs?????

bye bye

```
float time = (float)pRect.Width(); //time in sampleframes
float startlevel = (float)pRect.Height(); //max h vedi ma 1.0
float midlevel = 500.f;
float endlevel = 0.f;
```

```
float k = startlevel + endlevel - (midlevel * 2);
float r = startlevel;
float s = (endlevel - startlevel - (2 * k)) / time;
float t = (2 * k) / (time * time);
```

```
float bigr = r;
float bigs = s + t;
float bigt = 2 * t;
```

```
for(int i=0;i<time;i++)
{
    bigr = bigr + bigs;
    bigs = bigs + bigt; //bigs? co'è
    pDC->SetPixel(i,(int)bigr,RGB (0, 0, 0));
}
```

from : texmex [[a t]] iki.fi

comment : the method uses a technique called forward differencing, which is based on the fact that a successive values of an polynomial function can be calculated using only additions instead of evaluating the whole polynomial which would require huge amount of multiplications.

Actually the method presented here uses only a quadratic curve, not cubic. The number of the variables in the adder is N+1, where N is the order of the polynomial to be generated. In this example we have only three, thus second order function. For linear we would have two variables: the current value and the constant adder.

The trickiest part is to set up the adder variables...

Check out forward difference in mathworld for more info.

[Direct pink noise synthesis with auto-correlated generator](#) (click this to go back to the index)

Type : 16-bit fixed-point

References : Posted by RidgeRat <ltramme1476ATearthlinkDOTnet>

Notes :
Canonical C++ class with minimum system dependencies, BUT you must provide your own uniform random number generator. Accurate range is a little over 9 octaves, degrading gracefully beyond this. Estimated deviations ± 0.25 dB from ideal $1/f$ curve in range. Scaled to fit signed 16-bit range.

```
Code :
// Pink noise class using the autocorrelated generator method.
// Method proposed and described by Larry Trammell "the RidgeRat" --
// see http://home.earthlink.net/~ltramell/tech/newpink.htm
// There are no restrictions.
//
// -----
//
// This is a canonical, 16-bit fixed-point implementation of the
// generator in 32-bit arithmetic. There are only a few system
// dependencies.
//
// -- access to an allocator 'malloc' for operator new
// -- access to definition of 'size_t'
// -- assumes 32-bit two's complement arithmetic
// -- assumes long int is 32 bits, short int is 16 bits
// -- assumes that signed right shift propagates the sign bit
//
// It needs a separate URand class to provide uniform 16-bit random
// numbers on interval [1,65535]. The assumed class must provide
// methods to query and set the current seed value, establish a
// scrambled initial seed value, and evaluate uniform random values.
//
// ----- header -----
// pinkgen.h

#ifndef _pinkgen_h_
#define _pinkgen_h_ 1

#include <stddef.h>
#include <alloc.h>

// You must provide the uniform random generator class.
#ifndef _URand_h_
#include "URand.h"
#endif

class PinkNoise {
private:
    // Coefficients (fixed)
    static long int const pA[5];
    static short int const pPSUM[5];

    // Internal pink generator state
    long int contrib[5]; // stage contributions
    long int accum; // combined generators
    void internal_clear( );

    // Include a UNoise component
    URand ugen;

public:
    PinkNoise( );
    PinkNoise( PinkNoise & );
    ~PinkNoise( );
    void * operator new( size_t );
    void pinkclear( );
    short int pinkrand( );
};
#endif

// ----- implementation -----
// pinkgen.cpp

#include "pinkgen.h"

// Static class data
long int const PinkNoise::pA[5] =
{ 14055, 12759, 10733, 12273, 15716 };
short int const PinkNoise::pPSUM[5] =
{ 22347, 27917, 29523, 29942, 30007 };

// Clear generator to a zero state.
void PinkNoise::pinkclear( )
{
    int i;
    for (i=0; i<5; ++i) { contrib[i]=0L; }
    accum = 0L;
}
```

```

// PRIVATE, clear generator and also scramble the internal
// uniform generator seed.
void PinkNoise::internal_clear( )
{
    pinkclear();
    ugen.seed(0);    // Randomizes the seed!
}

// Constructor. Guarantee that initial state is cleared
// and uniform generator scrambled.
PinkNoise::PinkNoise( )
{
    internal_clear();
}

// Copy constructor. Preserve generator state from the source
// object, including the uniform generator seed.
PinkNoise::PinkNoise( PinkNoise & Source )
{
    int i;
    for (i=0; i<5; ++i) contrib[i]=Source.contrib[i];
    accum = Source.accum;
    ugen.seed( Source.ugen.seed( ) );
}

// Operator new. Just fetch required object storage.
void * PinkNoise::operator new( size_t size )
{
    return malloc(size);
}

// Destructor. No special action required.
PinkNoise::~PinkNoise( ) { /* NIL */ }

// Coding artifact for convenience
#define UPDATE_CONTRIB(n) \
{ \
    accum -= contrib[n]; \
    contrib[n] = (long)randv * pA[n]; \
    accum += contrib[n]; \
    break; \
}

// Evaluate next randomized 'pink' number with uniform CPU loading.
short int PinkNoise::pinkrand( )
{
    short int randu = ugen.urand() & 0x7fff;    // U[0,32767]
    short int randv = (short int) ugen.urand(); // U[-32768,32767]

    // Structured block, at most one update is performed
    while (1)
    {
        if (randu < pPSUM[0]) UPDATE_CONTRIB(0);
        if (randu < pPSUM[1]) UPDATE_CONTRIB(1);
        if (randu < pPSUM[2]) UPDATE_CONTRIB(2);
        if (randu < pPSUM[3]) UPDATE_CONTRIB(3);
        if (randu < pPSUM[4]) UPDATE_CONTRIB(4);
        break;
    }
    return (short int) (accum >> 16);
}

// ----- application -----

short int pink_signal[1024];

void example(void)
{
    PinkNoise pinkgen;
    int i;
    for (i=0; i<1024; ++i) pink_signal[i] = pinkgen.pinkrand();
}

```

Discrete Summation Formula (DSF) (click this to go back to the index)

References : Stylson, Smith and others... (posted by Alexander Kritov)

Notes :

Buzz uses this type of synth.

For cool sounds try to use variable,

for example `a=exp(-x/12000)*0.8 // x- num.samples`

Code :

```
double DSF (double x, // input
            double a, // a<1.0
            double N, // N<SmplFQ/2,
            double fi) // phase
{
    double s1 = pow(a,N-1.0)*sin((N-1.0)*x+fi);
    double s2 = pow(a,N)*sin(N*x+fi);
    double s3 = a*sin(x+fi);
    double s4 =1.0 - (2*a*cos(x)) +(a*a);
    if (s4==0)
        return 0;
    else
        return (sin(fi) - s3 - s2 +s1)/s4;
}
```

Comments

from : dfl[*AT*]ccrma.stanford.edu

comment : According to Stilson + Smith, this should be

```
double s1 = pow(a,N+1.0)*sin((N-1.0)*x+fi);
            ^
            !
```

Could be a typo though?

from : Alex

comment : yepp..

from : TT

comment : So what is wrong about "double" up there?

For DSF, do we have to update the phase (fi input) at every sample?

Another question is what's the input x supposed to represent? Thanks!

from : David Lowenfels

comment : input x should be the phase, and fi is the initial phase I guess? Seems redundant to me.

There is nothing wrong with the double, there is a sign typo in the original posting.

from : nobody [[a t]] nowhere.com

comment : I'm not so sure that there is a sign typo. (I know--I'm five years late to this party.)

The author of this code just seems to have an off-by-one definition of N. If you expand it all out, it looks like Stilson & Smith's paper, except you have N here where S&S had N+1, and you have N-1 where S&S had N.

I think the code is equivalent. You just have to understand how to choose N to avoid aliasing.

I don't have it working yet, but that's how it looks to me as I prepare a DSF oscillator. More later.

from : mysterious T

comment : Got it working nicely, but it took a few minutes to pluck it apart. Had to correct it for my pitch scheme, too. But it's quite amazing! Funny concept, though, it's like a generator with a built in filter. It holds up into very high pitches, too, in terms of aliasing, as far as I can tell... ehm...and without any further oversampling (so far).

Really, really nice! I was looking for a way to give my sinus an edge! ;)

[Drift generator](#) (click this to go back to the index)

Type : Random

References : Posted by quintosardo[AT]yahoo[DOT]it

Notes :

I use this drift to modulate any sound parameter of my synth.

It is very effective if it slightly modulates amplitude or frequency of an FM modulator.

It is based on an incremental random variable, sine-warped.

I like it because it is "continuous" (as opposite to "sample and hold"), and I can set variation rate and max variation.

It can go to upper or lower constraint (+/- max drift) but it gradually decreases rate of variation when approaching to the limit.

I use it exactly as an LFO (-1.f .. +1.f)

I use a table for sin instead of sin() function because this way I can change random distribution, by selecting a different curve (different table) from sine...

I hope that it is clear ... (sigh... :-)

Bye!!!

P.S. Thank you for help in previous submission ;-)

Code :

```
const int kSamples //Number of samples in fSinTable below
float fSinTable[kSamples] // Tabulated sin() [0 - 2pi[ amplitude [-1.f .. 1.f]
float fWhere// Index
float fRate // Max rate of variation
float fLimit //max or min value
float fDrift // Output

//I assume that random() is a number from 0.f to 1.f, otherwise scale it

fWhere += fRate * random()
//I update this drift in a long-term cycle, so I don't care of branches
if (fWhere >= 1.f) fWhere -= 1.f
else if (fWhere < 0.f) fWhere += 1.f

fDrift = fLimit * fSinTable[(long) (fWhere * kSamples)]
```

Comments

from : quintosardo [[a t]] yahoo.it

comment : ...sorry...

random() must be in [-1..+1] !!!

DSF (super-set of BLIT) (click this to go back to the index)

Type : matlab code

References : Posted by David Lowenfels

Notes :

Discrete Summation Formula ala Moorer

computes equivalent to $\sum_{k=0:N-1} (a^k * \sin(\beta + k*\theta))$
modified from Emanuel Landeholm's C code
output should never clip past [-1,1]

If using for BLIT synthesis for virtual analog:

```
N = maxN;  
a = attn_at_Nyquist ^ (1/maxN); %hide top harmonic popping in and out when sweeping frequency  
beta = pi/2;  
num = 1 - a^N * cos(N*theta) - a*( cos(theta) - a^N * cos(N*theta - theta) ); %don't waste time on beta
```

You can also get growing harmonics if $a > 1$, but the min statement in the code must be removed, and the scaling will be weird.

Code :

```
function output = dsf( freq, a, H, samples, beta )  
%a = rolloff coefficient  
%H = number of harmonic overtones (fundamental not included)  
%beta = harmonic phase shift  
  
samplerate = 44.1e3;  
freq = freq/samplerate; %normalize frequency  
  
bandlimit = samplerate / 2; %Nyquist  
maxN = 1 + floor( bandlimit / freq ); %prevent aliasing  
N = min(H+2,maxN);  
  
theta = 2*pi * phasor(freq, samples);  
  
epsilon = 1e-6;  
a = min(a, 1-epsilon); %prevent divide by zero  
  
num = sin(beta) - a*sin(beta-theta) - a^N*sin(beta + N*theta) + a^(N+1)*sin(beta+(N-1)*theta);  
den = (1 + a * ( a - 2*cos(theta) ));  
  
output = 2*(num ./ den - 1) * freq; %subtract by one to remove DC, scale by freq to normalize  
output = output * maxN/N; %OPTIONAL: rescale to give louder output as rolloff increases  
  
function out = phasor(normfreq, samples);  
out = mod( (0:samples-1)*normfreq , 1);  
out = out * 2 - 1; %make bipolar
```

Comments

from : David Lowenfels

comment : oops, there's an error in this version. frequency should not be normalized until after the maxN calculation is done.

[Fast & small sine generation tutorial](#) (click this to go back to the index)

Type : original document link: www.active-web.cc/html/research/sine/sin-cos.txt

References : Posted by office[AT]develotec[DOT]com

Linked file : <http://www.active-web.cc/html/research/sine/sin-cos.txt>

Notes :

original document link: www.active-web.cc/html/research/sine/sin-cos.txt

Comments

from : bobm.dsp [[a t]] gmail.com

comment : Excellent! Thank you.

from : duplicate

comment : The technique here is identical to "Fast sine wave calculation" posted here by me (James McCartney)

Fast Exponential Envelope Generator (click this to go back to the index)

References : Posted by Christian Schoenebeck

Notes :

The naive way to implement this would be to use a `exp()` call for each point of the envelope. Unfortunately `exp()` is quite a heavy function for most CPUs, so here is a numerical, much faster way to compute an exponential envelope (performance gain measured in benchmark: about factor 100 with a Intel P4, gcc -O3 --fast-math -march=i686 -mcpu=i686).

Note: you can't use a value of 0.0 for `levelEnd`. Instead you have to use an appropriate, very small value (e.g. 0.001 should be sufficiently small enough).

Code :

```
const float sampleRate = 44100;
float coeff;
float currentLevel;

void init(float levelBegin, float levelEnd, float releaseTime) {
    currentLevel = levelBegin;
    coeff = (log(levelEnd) - log(levelBegin)) /
            (releaseTime * sampleRate);
}

inline void calculateEnvelope(int samplePoints) {
    for (int i = 0; i < samplePoints; i++) {
        currentLevel += coeff * currentLevel;
        // do something with 'currentLevel' here
        ...
    }
}
```

Comments

from : citizenchunk [at] chunkware.com

comment : is there a typo in the runtime equation? or am i missing something in the implementation?

from : schoenebeck (at) software (minus) engineering.org

comment : Why should there be a typo?

Here is my benchmark code btw:

<http://stud.fh-heilbronn.de/~cschoene/studienarbeit/benchmarks/exp.cpp>

from : citizenchunk[at]chunkware[dot]com

comment : ok, i think i get it. this can only work on blocks of samples, right? not per-sample calc?

i was confused because i could not find the input sample(s) in the runtime code. but now i see that the equation does not take an input; it merely generates a defined envelope accross the number of samples. my bad.

from : schoenebeck (at) software (minus) engineering.org

comment : Well, the code above is only meant to show the principle. Of course you would adjust it for your application. The question if you are calculating on a per-sample basis or applying the envelope to a block of samples within a tight loop doesn't really matter; it would just mean an adjustment of the interface of the execution code, which is trivial.

from : meeloo [[a t]] meeloo.net

comment : This is not working for long envelopes because of numerical accury problems. Try calculating is over 10 seconds @ 192KHz to see what I mean: it drifts.

I have an equivalent system that permits to have linear to log and to exp curves with a simple parameter. I may submit it one of these days...

Sebastien Metrot

--

<http://www.usbsounds.com>

from : schoenebeck (at) software (minus) engineering.org

comment : No, here is a test app which shows the introduced drift:
<http://stud.fh-heilbronn.de/~cschoene/studienarbeit/benchmarks/expaccuracy.cpp>

Even with an envelope duration of 30s, which is really quite long, a sample rate of 192KHz and single-precision floating point calculation I get this result:

Calculated sample points: 5764846
Demanded duration: 30.000000 s
Actual duration: 30.025240 s

So the envelope just drifts about 25ms for that long envelope!

from : meeloo [[a t]] meeloo.net

comment : I believe you are seeing unrealistic results with this test because on x86 the fpu's internal format is 80bits and your compiler probably optimises this cases quite easily. Try doing the same test, calculating the same envelope, but by breaking the calculation in blocks of 256 or 512

samples at a time and then storing in memory the temp values for the next block. In this case you may see different results and a much bigger drift (that's my experience with the same algo).

Anyway my algo is a bit different as it permits to change the current type with a parameter, this makes the formula look like

$value = value * coef + constant;$

Maybe this leads to more calculation errors :).

from : schoenebeck (at) software (minus) engineering.org

comment : And again... no! :)

Replace the C equation by:

```
asm volatile (
    "movss %1,%xmm0    # load coeff\n\t"
    "movss %2,%xmm1    # load currentLevel\n\t"
    "mulss %%xmm1,%%xmm0 # coeff *= currentLevel\n\t"
    "addss %%xmm0,%%xmm1 # currentLevel += coeff * currentLevel\n\t"
    "movss %%xmm1,%0    # store currentLevel\n\t"
    : "=m" (currentLevel) /* %0 */
    : "m" (coeff), /* %1 */
    "m" (currentLevel) /* %2 */
);
```

This is a SSE1 assembly implementation. The SSE registers are only 32 bit large by guarantee. And this is the result I get:

Calculated sample points: 5764845

Demanded duration: 30.000000 s

Actual duration: 30.025234 s

So this result differs just 1 sample point from the x86 FPU solution! So believe me, this numerical solution is safe!

(Of course the assembly code above is NOT meant as optimization, it's just to demonstrate the accuracy even for 32 bit / single precision FP calculation)

from : m (at) mindplay (dot) dk

comment : in my tests, the following code produced the exact same results, and saves one operation (the addition) per sample - so it should be faster:

```
const float sampleRate = 44100;
```

```
float coeff;
```

```
float currentLevel;
```

```
void init(float levelBegin, float levelEnd, float releaseTime) {
    currentLevel = levelBegin;
    coeff = exp(log(levelEnd)) /
           (releaseTime * sampleRate);
}
```

```
inline void calculateEnvelope(int samplePoints) {
    for (int i = 0; i < samplePoints; i++) {
        currentLevel *= coeff;
        // do something with 'currentLevel' here
        ...
    }
}
```

```
...
```

Also, assuming that your startLevel is 1.0, to calculate an appropriate endLevel, you can use something like:

```
endLevel = 10 ^ dB/20;
```

where dB is your endLevel in decibels (and must be a negative value of course) - for amplitude envelopes, -90 dB should be a suitable level for "near inaudible"...

from : schoenebeck (at) software (minus) engineering.org

comment : Sorry, you are right of course; that simplification of the execution equation works here because we are calculating all points with linear discretization. But you will agree that your init() function is not good, because $\exp(\log(x)) = x$ and it's not generalized at all. Usually you might have more than one exp segment in your EG and maybe even have an exp attack segment. So we arrive at the following solution:

```
const float sampleRate = 44100;
```

```
float coeff;
```

```
float currentLevel;
```

```
void init(float levelBegin, float levelEnd, float releaseTime) {
    currentLevel = levelBegin;
    coeff = 1.0f + (log(levelEnd) - log(levelBegin)) /
           (releaseTime * sampleRate);
}
```

```

inline void calculateEnvelope(int samplePoints) {
    for (int i = 0; i < samplePoints; i++) {
        currentLevel *= coeff;
        // do something with 'currentLevel' here
        ...
    }
}

```

You can use a dB conversion for both startLevel and endLevel of course.

from : na
comment : i would say that calculation of coeff is still wrong. It should be :
coeff = pow(levelEnd / levelBegin, 1 / N);

from : na[eldar # starman # ee]
comment : or coeff = exp(log(levelEnd/levelBegin) /
(releaseTime * sampleRate));
not sure but it looks computationally more expensive

from : e.l.i. [[a t]] gmx.ch
comment : what's about?
coeff = 1.0f + (log(levelEnd) - log(levelBegin)) /
(releaseTime * sampleRate - 1);

from : e.l.i. [[a t]] gmx.ch
comment : sorry for the double post. and i'm now almost sure, that it should be:
coeff = 1.0f + (log(levelEnd) - log(levelBegin)) /
(releaseTime * sampleRate + 1);

[Fast LFO in Delphi...](#) (click this to go back to the index)

[References](#) : Posted by Dambrin Didier (gol [AT] e-officedirect [DOT] com)

[Linked file](#) : [LFOGenerator.zip](#)

[Notes](#) :

[from Didier's mail...]

[see attached zip file too!]

I was working on a flanger, & needed an LFO for it. I first used a Sin(), but it was too slow, then tried a big wavetable, but it wasn't accurate enough.

I then checked the alternate sine generators from your web site, & while they're good, they all can drift, so you're also wasting too much CPU in branching for the drift checks.

So I made a quick & easy linear LFO, then a sine-like version of it. Can be useful for LFO's, not to output as sound. If has no branching & is rather simple. 2 Abs() but apparently they're fast. In all cases faster than a Sin()

It's in delphi, but if you understand it you can translate it if you want.

It uses a 32bit integer counter that overflows, & a power for the sine output.

If you don't know delphi, \$ is for hex (h at the end in c++?), Single is 32bit float, integer is 32bit integer (signed, normally).

[Code](#) :

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, ComCtrls;

type
  TForm1 = class(TForm)
    PaintBox1: TPaintBox;
    Bevel1: TBevel;
    procedure PaintBox1Paint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.PaintBox1Paint(Sender: TObject);
var n, Pos, Speed: Integer;
    Output, Scale, HalfScale, PosMul: Single;
    OurSpeed, OurScale: Single;
begin
  OurSpeed:=100; // 100 samples per cycle
  OurScale:=100; // output in -100..100

  Pos:=0; // position in our linear LFO
  Speed:=Round($100000000/OurSpeed);

  // --- triangle LFO ---
  Scale:=OurScale*2;
  PosMul:=Scale/$80000000;

  // loop
  for n:=0 to 299 do
    Begin
      // inc our 32bit integer LFO pos & let it overflow. It will be seen as signed when read by the math unit
      Pos:=Pos+Speed;

      Output:=Abs(Pos*PosMul)-OurScale;

      // visual
      Paintbox1.Canvas.Pixels[n, Round(100+Output)]:=clRed;
    End;

  // --- sine-like LFO ---
  Scale:=Sqrt(OurScale*4);
  PosMul:=Scale/$80000000;
  HalfScale:=Scale/2;

  // loop
  for n:=0 to 299 do
    Begin
```

```

// inc our 32bit integer LFO pos & let it overflow. It will be seen as signed when read by the math unit
Pos:=Pos+Speed;

Output:=Abs(Pos*PosMul)-HalfScale;
Output:=Output*(Scale-Abs(Output));

// visual
Paintbox1.Canvas.Pixels[n,Round(100+Output)]:=c1Blue;
End;
end;

end.

```

Comments

from : Christian [[a t]] savioursofsoul.de

comment : LFO Class...

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, ComCtrls;

type
  TLFOType = (lfoTriangle,lfoSine);
  TLFO = class(TObject)
  private
    iSpeed   : Integer;
    fSpeed   : Single;
    fMax, fMin : Single;
    fValue   : Single;
    fPos     : Integer;
    fType    : TLFOType;
    fScale   : Single;
    fPosMul  : Single;
    fHalfScale : Single;
    function GetValue:Single;
    procedure SetType(tt: TLFOType);
    procedure SetMin(v:Single);
    procedure SetMax(v:Single);
  public
    { Public declarations }
    constructor Create;
  published
    property Value:Single read GetValue;
    property Speed:Single read FSpeed Write FSpeed;
    property Min:Single read FMin write SetMin;
    property Max:Single read FMax Write SetMax;
    property LFO:TLFOType read fType Write SetType;
  end;

  TForm1 = class(TForm)
    Bevel1: TBevel;
    PaintBox1: TPaintBox;
    procedure PaintBox1Paint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

constructor TLFO.Create;
begin
  fSpeed:=100;
  fMax:=1;
  fMin:=0;
  fValue:=0;
  fPos:=0;
  iSpeed:=Round($100000000/fSpeed);
  fType:=lfoTriangle;
  fScale:=fMax-((fMin+fMax)/2);
end;

procedure TLFO.SetType(tt: TLFOType);
begin
  fType:=tt;

```

```

if fType = lfoSine then
begin
fPosMul:=(Sqrt(fScale*2))/$80000000;
fHalfScale:=(Sqrt(fScale*2))/2;
end
else
begin
fPosMul:=fScale/$80000000;
end;
end;

procedure TLFO.SetMin(v: Single);
begin
fMin:=v;
fScale:=fMax-((fMin+fMax)/2);
end;

procedure TLFO.SetMax(v: Single);
begin
fMax:=v;
fScale:=fMax-((fMin+fMax)/2);
end;

function TLFO.GetValue:Single;
begin
if fType = lfoSine then
begin
Result:=Abs(fPos*fPosMul)-fHalfScale;
Result:=Result*(fHalfScale*2-Abs(Result))*2;
Result:=Result+((fMin+fMax)/2);
end
else
begin
Result:=Abs(fPos*(2*fPosMul))+fMin;
end;
fPos:=fPos+iSpeed;
end;

procedure TForm1.PaintBox1Paint(Sender: TObject);
var n : Integer;
LFO1 : TLFO;
begin

LFO1:=TLFO.Create;
LFO1.Min:=-100;
LFO1.Max:=100;
LFO1.Speed:=100;
LFO1.LFO:=lfoTriangle;

// --- triangle LFO ---
for n:=0 to 299 do Paintbox1.Canvas.Pixels[n,Round(100+LFO1.Value)]:=clRed;

LFO1.LFO:=lfoSine;
// --- sine-like LFO ---
for n:=0 to 299 do Paintbox1.Canvas.Pixels[n,Round(100+LFO1.Value)]:=clBlue;
end;

end.

from : Christian [ [ a t ] ] savioursofsoul.de
comment : Ups, i forgot something:

TLFO = class(TObject)
private
...
procedure SetSpeed(v:Single);
public
...
published
...
property Speed:Single read FSpeed Write SetSpeed;
...
end;

...

constructor TLFO.Create;
begin
...
Speed:=100;
...
end;

procedure TLFO.SetSpeed(v:Single);

```

```
begin
fSpeed:=v;
iSpeed:=Round($10000000/fSpeed);
end;
```

...

Fast SIN approximation for usage in e.g. additive synthesizers (click this to go back to the index)

References : Posted by neotec

Notes :

This code presents 2 'fastsin' functions. fastsin2 is less accurate than fastsin. In fact it's a simple taylor series, but optimized for integer phase.

phase is in 0 -> (2^32)-1 range and maps to 0 -> ~2PI

I get about 55000000 fastsin's per second on my P4,3.2GHz which would give a nice Kawai K5 emulation using 64 harmonics and 8->16 voices.

Code :

```
float fastsin(UINT32 phase)
{
    const float frf3 = -1.0f / 6.0f;
    const float frf5 = 1.0f / 120.0f;
    const float frf7 = -1.0f / 5040.0f;
    const float frf9 = 1.0f / 362880.0f;
    const float f0pi5 = 1.570796327f;
    float x, x2, asin;
    UINT32 tmp = 0x3f800000 | (phase >> 7);
    if (phase & 0x40000000)
        tmp ^= 0x007fffff;
    x = *((float*)&tmp) - 1.0f) * f0pi5;
    x2 = x * x;
    asin = (((frf9 * x2 + frf7) * x2 + frf5) * x2 + frf3) * x2 + 1.0f) * x;
    return (phase & 0x80000000) ? -asin : asin;
}

float fastsin2(UINT32 phase)
{
    const float frf3 = -1.0f / 6.0f;
    const float frf5 = 1.0f / 120.0f;
    const float frf7 = -1.0f / 5040.0f;
    const float f0pi5 = 1.570796327f;
    float x, x2, asin;
    UINT32 tmp = 0x3f800000 | (phase >> 7);
    if (phase & 0x40000000)
        tmp ^= 0x007fffff;
    x = *((float*)&tmp) - 1.0f) * f0pi5;
    x2 = x * x;
    asin = ((frf7 * x2 + frf5) * x2 + frf3) * x2 + 1.0f) * x;
    return (phase & 0x80000000) ? -asin : asin;
}
```

Comments

from : bob [[a t]] yahoot.com

comment : Woah! Seven multiplies, on top of those adds and memory lookup. Is this really all that fast?

from : neotec

comment : PS: To use this as an OSC you'll need the following vars/equ's:

UINT32 phase = 0;

UINT32 step = frequency * powf(2.0f, 32.0f) / samplerate;

Then it's just:

```
...
out = fastsin(phase);
phase += step;
...
```


Fast sine and cosine calculation (click this to go back to the index)

Type : waveform generation

References : Lot's or references... Check Julius O. SMith mainly

```
Code :
init:
float a = 2.f*(float)sin(Pi*frequency/samplerate);

float s[2];

s[0] = 0.5f;
s[1] = 0.f;

loop:
s[0] = s[0] - a*s[1];
s[1] = s[1] + a*s[0];
output_sine = s[0];
output_cosine = s[1]
```

Comments

from : nigel

comment : This is the Chamberlin state variable filter specialized for infinite Q oscillation. A few things to note:

Like the state variable filter, the upper frequency limit for stability is around one-sixth the sample rate.

The waveform symmetry is very pure at low frequencies, but gets skewed as you get near the upper limit.

For low frequencies, $\sin(n)$ is very close to n , so the calculation for "a" can be reduced to $a = 2*\text{Pi}*frequency/samplerate$.

You shouldn't need to resync the oscillator--for fixed point and IEEE floating point, errors cancel exactly, so the oscillator runs forever without drifting in amplitude or frequency.

from : DFL

comment : Yeah, this is a cool trick! :)

FYI you can set $s[0]$ to whatever amplitude of sinewave you desire. With 0.5, you will get +/- 0.5

from : bigtick [[a t]] pastnotecut.org

comment : After a while it may drift, so you should resync it as follows:

```
const float tmp=1.5f-0.5f*(s[1]*s[1]+s[0]*s[0]);
s[0]*=tmp; s[1]*=tmp;
```

This assumes you set $s[0]$ to 1.0 initially.

Tick

from : DFL

comment : Just to explain the above "resync" equation

$(3-x)/2$ is an approximation of $1/\sqrt{x}$

So the above is actually renormalizing the complex magnitude.

[$\sin^2(x) + \cos^2(x) = 1$]

from : antiprosynthesis [[a t]] hotmail.com

comment : I made a nice little console 'game' using your cordic sinewave approximation. Download it at <http://users.pandora.be/antipro/Other/Ascillator.zip> (includes source code). Just for oldschool fun :).

from : hplus

comment : Note that the peaks of the waveforms will actually be between samples, and the functions will be phase offset by one half sample's worth. If you need exact phase, you can compensate by interpolating using cubic hermite interpolation.

from : bob [[a t]] yahoo.com

comment : How do I set a particular phase for this? I've tried setting $s[0] = \cos(\text{phase})$ and $s[1] = \sin(\text{phase})$, but that didn't seem to be accurate enough.

Thanks

from : more on that topic...

comment : ... can be found in Jon Dattorro, Effect Design, Part 3, a paper that can be easily found in the web.

Funny, this is just a complex multiply that is optimized for small angles (low frequencies)

When the CPU rounding mode is set to nearest, it should be stable, at least for small frequencies.

from : mail [[a t]] mroc.de

comment : More on that can be found in Jon Dattorro, Effect Design, Part 3, a paper that can be easily found in the web.

Funny, this is just a complex multiply that is optimized for small angles (low frequencies)

When the CPU rounding mode is set to nearest, it should be stable, at least for small frequencies.

Fast sine wave calculation (click this to go back to the index)

Type : waveform generation

References : James McCartney in Computer Music Journal, also the Julius O. Smith paper

Notes :
(posted by Niels Gorisse)
If you change the frequency, the amplitude rises (pitch lower) or lowers (pitch rise) a LOT I fixed the first problem by thinking about what actually goes wrong. The answer was to recalculate the phase for that frequency and the last value, and then continue normally.

Code :
Variables:
ip = phase of the first output sample in radians
w = freq*pi / samplerate
b1 = 2.0 * cos(w)

Init:
y1=sin(ip-w)
y2=sin(ip-2*w)

Loop:
y0 = b1*y1 - y2
y2 = y1
y1 = y0

output is in y0 (y0 = sin(ip + n*freq*pi / samplerate), n= 0, 1, 2, ... I *think*)

Later note by James McCartney:
if you unroll such a loop by 3 you can even eliminate the assigns!!

```
y0 = b1*y1 - y2
y2 = b1*y0 - y1
y1 = b1*y2 - y0
```

Comments

from : kainhart[at]hotmail.com

comment : try using this to make sine waves with frequency less than 1. I did and it gives very rough half triangle-like waves. Is there any way to fix this? I want to use a sine generated for LFO so I need one that works for low frequencies.

from : asynth [[a t]] io.com

comment : looks like the formula has gotten munged.

w = freq * twopi / samplerate

Fast square wave generator (click this to go back to the index)

Type : NON-bandlimited osc...

References : Posted by Wolfgang (wschneider[AT]nexoft.de)

Notes :

Produces a square wave -1.0f .. +1.0f.

The resulting waveform is NOT band-limited, so it's probably of not much use for synthesis. It's rather useful for LFOs and the like, though.

Code :

Idea: use integer overflow to avoid conditional jumps.

```
// init:
typedef unsigned long ui32;

float sampleRate = 44100.0f; // whatever
float freq = 440.0f; // 440 Hz
float one = 1.0f;
ui32 intOver = 0L;
ui32 intIncr = (ui32)(4294967296.0 / hostSampleRate / freq);

// loop:
*((ui32 *)&one) &= 0x7FFFFFFF; // mask out sign bit
*((ui32 *)&one) |= (intOver & 0x80000000);
intOver += intIncr;
```

Comments

from : musicdsp at lancej.com

comment : So, how would I get the output into a float variable like square_out, for instance?

Fast Whitenoise Generator (click this to go back to the index)

Type : Whitenoise

References : Posted by gerd[DOT]feldkirch[AT]web[DOT]de

Notes :

This is Whitenoise... :o)

Code :

```
float g_fScale = 2.0f / 0xfffffffff;
int g_x1 = 0x67452301;
int g_x2 = 0xefcdab89;

void whitenoise(
    float* _fpDstBuffer, // Pointer to buffer
    unsigned int _uiBufferSize, // Size of buffer
    float _fLevel ) // Noiselevel (0.0 ... 1.0)
{
    _fLevel *= g_fScale;

    while( _uiBufferSize-- )
    {
        g_x1 ^= g_x2;
        *_fpDstBuffer++ = g_x2 * _fLevel;
        g_x2 += g_x1;
    }
}
```

Comments

from : gerd.feldkirch [[a t]] web.de

comment : As I said! :-)

Take care

from : nobody [[a t]] nowhere.com

comment : I'm now waiting for pink and brown. :-)

from : scoofy [[a t]] inf.elte.hu

comment : To get pink noise, you can apply a 3dB/Oct filter, for example the pink noise filter in the Filters section.

To get brown noise, apply an one pole LP filter to get a 6dB/oct slope.

Peter

from : nobody [[a t]] nowhere.com

comment : Yeah, I know how to do it with a filter. I was just looking to see if this guy had anything else clever up his sleeve.

I'm currently using this great stuff:

vellocet.com/dsp/noise/VRand.html

from : tremblap [[a t]] gmail.com

comment : I compiled it, but I get some grainyness that a unsigned long LC algorithm does not give me... am I the only one?

pa

from : scoofy [[a t]] inf.elte.hu

comment : Did you do everything right? It works here.

from : gerd.feldkirch [[a t]] web.de

comment : I've noticed that my code is similar to a so called "feedback shift register" as used in the Commodore C64 Soundchip 6581 called SID for noise generation.

Links:

en.wikipedia.org/wiki/Linear_feedback_shift_register

en.wikipedia.org/wiki/MOS_Technology_SID

www.cc65.org/mailarchive/2003-06/3156.html

from : scoofy [[a t]] inf.elte.hu

comment : Works well! Kinda fast! The spectrum looks completely flat in an FFT analyzer.

from : Arif [[a t]] mail.---

comment : SID noise! cool.

Gaussian White noise (click this to go back to the index)

References : Posted by Alexey Menshikov

Notes :

Code I use sometimes, but don't remember where I ripped it from.

- Alexey Menshikov

Code :

```
#define ranf() ((float) rand() / (float) RAND_MAX)

float ranfGauss (int m, float s)
{
    static int pass = 0;
    static float y2;
    float x1, x2, w, y1;

    if (pass)
    {
        y1 = y2;
    } else {
        do {
            x1 = 2.0f * ranf () - 1.0f;
            x2 = 2.0f * ranf () - 1.0f;
            w = x1 * x1 + x2 * x2;
        } while (w >= 1.0f);

        w = (float)sqrt (-2.0 * log (w) / w);
        y1 = x1 * w;
        y2 = x2 * w;
    }
    pass = !pass;

    return ( (y1 * s + (float) m));
}
```

Comments

from : davidchristenATgmxDOTnet

comment : White Noise does !not! consist of uniformly distributed values. Because in white noise, the power of the frequencies are uniformly distributed. The values must be normal (or gaussian) distributed. This is achieved by the Box-Muller Transformation. This function is the polar form of the Box-Muller Transformation. It is faster and numeriacally more stable than the basic form. The basic form is coded in the other (second) post. Detailed information on this topic:

<http://www.taygeta.com/random/gaussian.html>

<http://www.eece.unm.edu/faculty/bsanthan/EECE-541/white2.pdf>

Cheers David

from : nick.a.shaw [[a t]] btopenworld.com

comment : I'm trying to implement this in C#, but y2 isn't initialized. Is this a typo?

[Gaussian White Noise](#) (click this to go back to the index)

[References](#) : Posted by remage[AT]netposta.hu

[Notes](#) :

SOURCE:

Steven W. Smith:
The Scientist and Engineer's Guide to Digital Signal Processing
<http://www.dspguide.com>

[Code](#) :

```
#define PI 3.1415926536f

float R1 = (float) rand() / (float) RAND_MAX;
float R2 = (float) rand() / (float) RAND_MAX;

float X = (float) sqrt( -2.0f * log( R1 )) * cos( 2.0f * PI * R2 );
```

[Comments](#)

[from](#) : pan[at]spinningkids.org

[comment](#) : The previous one seems better for me, since it requires only a rand, half log and half sqrt per sample. Actually, I used that one, but I can't remember where I found it, too. Maybe on Knuth's book.

Generator (click this to go back to the index)

Type : antialiased sawtooth

References : Posted by Paul Sernine

Notes :

This code generates a swept antialiasing sawtooth in a raw 16bit pcm file. It is based on the quad differentiation of a 5th order polynomial. The polynomial harmonics (and aliased harmonics) decay at 5*6 dB per oct. The differentiators correct the spectrum and waveform, while aliased harmonics are still attenuated.

Code :

```
/* clair.c          Examen Partiel 2b
   T.Rochebois
   02/03/98
*/
#include <stdio.h>
#include <math.h>
main()
{
    double phase=0,dphase,freq,compensation;
    double aw0=0,awl=0,ax0=0,ax1=0,ay0=0,ay1=0,az0=0,az1=0,sortie;
    short aout;
    int sr=44100;          //sample rate (Hz)
    double f_debut=55.0;//start freq (Hz)
    double f_fin=sr/6.0;//end freq (Hz)
    double octaves=log(f_fin/f_debut)/log(2.0);
    double duree=50.0;   //duration (s)
    int i;
    FILE* f;
    f=fopen("saw.pcm","wb");
    for(i=0;i<duree*sr;i++)
    {
        //exponential frequency sweep
        //Can be replaced by anything you like.
        freq=f_debut*pow(2.0,octaves*i/(duree*sr));
        dphase=freq*(2.0/sr);      //normalised phase increment
        phase+=dphase;           //phase incrementation
        if(phase>1.0) phase-=2.0; //phase wrapping (-1,+1)

        //polynomial calculation (extensive continuity at -1 +1)
        //      7      1 3      1 5
        //P(x) = --- x - -- x + --- x
        //      360     36      120
        aw0=phase*(7.0/360.0 + phase*phase*(-1/36.0 + phase*phase*(1/120.0)));
        // quad differentiation (first order high pass filters)
        ax0=awl-aw0; ay0=ax1-ax0; az0=ay1-ay0; sortie=az1-az0;
        //compensation of the attenuation of the quad differentiator
        //this can be calculated at "control rate" and linearly
        //interpolated at sample rate.
        compensation=1.0/(dphase*dphase*dphase*dphase);
        // compensation and output
        aout=(short)(15000.0*compensation*sortie);
        fwrite(&aout,1,2,f);
        //old memories of differentiators
        awl=aw0; ax1=ax0; ayl=ay0; az1=az0;
    }
    fclose(f);
}
```

Comments

from : Paul_Sernine75 [[a t]] hotmail.fr
comment : More infos and discussions in the KVR thread:
<http://www.kvraudio.com/forum/viewtopic.php?t=123498>

from : bonbon [[a t]] elisa.com
comment : nice but i prefer the fishy algo, it generates less alias.
bonaveture rosignol

[Inverted parabolic envelope](#) (click this to go back to the index)

Type : envelope generation

References : Posted by James McCartney

Code :

```
dur = duration in samples
midlevel = amplitude at midpoint
beglevel = beginning and ending level (typically zero)
```

```
amp = midlevel - beglevel;
```

```
rdur = 1.0 / dur;
rdur2 = rdur * rdur;
```

```
level = beglevel;
slope = 4.0 * amp * (rdur - rdur2);
curve = -8.0 * amp * rdur2;
```

```
...
```

```
for (i=0; i<dur; ++i) {
    level += slope;
    slope += curve;
}
```

Comments

from : krakengore [[a t]] libero.it

comment : This parabola approximation seems more like a linear than a parab/expo envelope... or i'm mistaking something but i tryed everything and is only linear.

from : ex0r0x0r [[a t]] hotmail.com

comment : slope is linear, but 'slope' is a function of 'curve'. If you imagine you threw a ball upwards, think of 'curve' as the gravity, 'slope' as the vertical velocity, and 'level' as the vertical displacement.

from : asynth(at)io(dot)com

comment : This is not an approximation of a parabola, it IS a parabola.

This entry has become corrupted since it was first posted. Should be:

```
for (i=0; i<dur; ++i) {
    out = level;
    level += slope;
    slope += curve;
}
```

[matlab/octave code for minblep table generation](#) (click this to go back to the index)

[References](#) : Posted by [dff\[at\]ccrma\[dot\]stanford\[dot\]edu](mailto:dff[at]ccrma[dot]stanford[dot]edu)

Notes :
When I tested this code, it was running with each function in a separate file... so it might need some tweaking (endfunction statements?) if you try and run it all as one file.

Enjoy!

PS There's a C++ version by Daniel Werner here.
<http://www.experimentalscene.com/?type=2&id=1>
Not sure if it the output is any different than my version.
(eg no thresholding in minphase calculation)

```
Code :
% Octave/Matlab code to generate a minblep table for bandlimited synthesis
%% original minblep technique described by Eli Brandt:
% http://www.cs.cmu.edu/~eli/L/icmc01/hardsync.html

% (c) David Lowenfels 2004
% you may use this code freely to generate your tables,
% but please send me a free copy of the software that you
% make with it, or at least send me an email to say hello
% and put my name in the software credits :)
% (IIRC: mps and clipdb functions are from Julius Smith)

% usage:
% fc = dilation factor
% Nzc = number of zero crossings
% omega = oversampling factor
% thresh = dB threshold for minimum phase calc

mbtable = minblep( fc, Nzc, omega, thresh );
mblen = length( mbtable );
save -binary mbtable.mat mbtable ktable nzc mblen;

*****
function [out] = minblep( fc, Nzc, omega, thresh )

out = filter( 1, [1 -1], minblip( fc, Nzc, omega, thresh ) );

len = length( out );
normal = mean( out( floor(len*0.7):len ) )
out = out / normal; %% normalize

%% now truncate so it ends at proper phase cycle for minimum discontinuity
thresh = 1e-6;
for i = len:-1:len-1000
% pause
a = out(i) - thresh - 1;
b = out(i-1) - thresh - 1;
% i
if( (abs(a) < thresh) & (a > b) )
break;
endif
endifor

%out = out';
out = out(1:i);

*****

function [out] = minblip( fc, Nzc, omega, thresh )
if (nargin < 4 )
thresh = -100;
end
if (nargin < 3 )
omega = 64;
end
if (nargin < 2 )
Nzc = 16;
end
if (nargin < 1 )
fc = 0.9;
end

blip = sinctable( omega, Nzc, fc );
%% length(blip) must be nextpow2! (if fc < 1 );

mag = fft( blip );
out = real( ifft( mps( mag, thresh ) ) );

*****

function [sm] = mps(s, thresh)
% [sm] = mps(s)
% create minimum-phase spectrum sm from complex spectrum s
```

```

if (nargin < 2 )
    thresh = -100;
endif

s = clipdb(s, thresh);
sm = exp( fft( fold( ifft( log( s ) ) ) ) );

*****
function [clipped] = clipdb(s,cutoff)
% [clipped] = clipdb(s,cutoff)
% Clip magnitude of s at its maximum + cutoff in dB.
% Example: clip(s,-100) makes sure the minimum magnitude
% of s is not more than 100dB below its maximum magnitude.
% If s is zero, nothing is done.

as = abs(s);
mas = max(as(:));
if mas==0, return; end
if cutoff >= 0, return; end
thresh = mas*10^(cutoff/20); % db to linear
toosmall = find(as < thresh);
clipped = s;
clipped(toosmall) = thresh;
*****

function [out, phase] = sinctable( omega, Nzc, fc )

if (nargin < 3 )
    fc = 1.0 %% cutoff frequency
end %if
if (nargin < 2 )
    Nzc = 16 %% number of zero crossings
end %if
if (nargin < 1 )
    omega = 64 %% oversampling factor
end %if

Nzc = Nzc / fc %% This ensures more flatness at the ends.

phase = linspace( -Nzc, Nzc, Nzc*omega*2 );

%sinc = sin( pi * fc * phase) ./ (pi * fc * phase);

num = sin( pi*fc*phase );
den = pi*fc*phase;

len = length( phase );
sinc = zeros( len, 1 );

%sinc = num ./ den;

for i=1:len
    if ( den(i) ~= 0 )
        sinc(i) = num(i) / den(i);
    else
        sinc(i) = 1;
    end
end %for

out = sinc;
window = blackman( len );
out = out .* window;

```

[PADsynth synthesys method](#) (click this to go back to the index)

Type : wavetable generation

References : Posted by zynaddsubfx[at]yahoo[dot]com

Notes :

Please see the full description of the algorithm with public domain c++ code here:
<http://zynaddsubfx.sourceforge.net/doc/PADsynth/PADsynth.htm>

Code :

It's here:

<http://zynaddsubfx.sourceforge.net/doc/PADsynth/PADsynth.htm>

You may copy it (everything is public domain).

Paul

Comments

from : thaddy [[a t]] thaddy.com

comment : Impressed at first hearing! Well documented.

from : n.eq [[a t]] mailcity.com

comment : 1. Isn't this plain additive synthesis

2. Isn't this the algorithm used by the waldorf microwave synths?

from : me [[a t]] yah00.kom

comment : 1. Nope. This is not a plain additive synthesis. It's a special kind :-P Read the doc again :)

2. No way.. this is NOT even close to waldord microwave synths. Google for this :)

[Parabolic shaper](#) (click this to go back to the index)

[References](#) : Posted by azertopia at free dot fr

Notes :

This function can be used for oscillators or shaper.

it can be driven by a phase accumulator or an audio input.

Code :

```
Function Parashape(inp:single):single;
var fgh,tgh:single;
begin
fgh := inp ;
fgh := 0.25-f_abs(fgh) ;
tgh := fgh ;
tgh := 1-2*f_abs(tgh);
fgh := fgh*8;
result := fgh*tgh ;
end;
// f_abs is the function of ddsputils unit.
```

Phase modulation Vs. Frequency modulation (click this to go back to the index)

References : Posted by Bram

Linked file : [SimpleOscillator.h](#) (this linked file is included below)

Notes :

This code shows what the difference is between FM and PM.
The code is NOT optimised, nor should it be used like this.
It is an EXAMPLE

See linked file.

Linked files

```
////////////////////////////////////
//
// this code was NEVER MEANT TO BE USED.
//
// use as EXPLANATION ONLY for the difference between
// Phase Modulation and Frequency Modulation.
// there are MANY ways to speed this code up.
//
// bram@musicdsp.org | bram@smartelectronix.com
//
// ps:
// we use the 'previous' value of the phase in all the algo's to make sure that
// the first call to the getSampleXX() function returns the wave at phase 'zero'
//
////////////////////////////////////

#include "math.h";

#define Pi 3.141592f

class SimpleOscillator
{
    SimpleOscillator(const float sampleRate = 44100.f, const long tableSize = 4096)
    {
        this->tableSize = tableSize;
        this->sampleRate = sampleRate;

        phase = 0.f;

        makeTable();
    }

    ~SimpleOscillator()
    {
        delete [] table;
    }

    // normal oscillator, no modulation
    //
    float generateSample(const float frequency)
    {
        float lookupPhase = phase;

        phase += frequency * (float)tableSize / sampleRate;
        wrap(phase);

        return lookup(lookupPhase);
    }

    // frequency modulation
    // the fm input should be in HZ.
    //
    // example:
    // osc1.getSampleFM(440.f, osc2.getSample(0.5f) * 5.f )
    // would give a signal where the frequency of the signal is
    // modulated between 435hz and 445hz at a 0.5hz rate
    //
    float generateSampleFM(const float frequency, const float fm)
    {
        float lookupPhase = phase;

        phase += (frequency + fm) * (float)tableSize / sampleRate;
        wrap(phase);

        return lookup(lookupPhase);
    }
}
```

```

// phase modulation
//
// a phase mod value of 1.f will increase the "phase" of the wave by a full cycle
// i.e. calling getSamplePM(440.f,1.f) will result in the "same" wave as getSamplePM(440.f,0.f)
//
float generateSamplePM(const float frequency, const float pm)
{
    float lookupPhase = phase + (float)tableSize * pm;
    wrap(lookupPhase)

    phase += frequency * (float)tableSize / sampleRate;
    wrap(phase);

    return lookup(lookupPhase);
}

// do the lookup in the table
// you could use different methods here
// like linear interpolation or higher order...
// see musicdsp.org
//
float lookup(const float phase)
{
    return table[(long)phase];
}

// wrap around
//
void wrap(float &in)
{
    while(in < 0.f)
        in += (float)tableSize;

    while(in >= (float)tableSize)
        in -= (float)tableSize;

    return in;
}

// set the sample rate
//
void setSampleRate(const float sampleRate)
{
    this->sampleRate = sampleRate;
}

// sets the phase of the oscillator
// phase should probably be in 0..Pi*2
//
void setPhase(const float phase)
{
    this->phase = phase / (2.f * Pi) * (float)tableSize;
    wrap(phase);
}

private:

float sampleRate;
float phase;

float *table;
long tableSize;

void makeTable()
{
    table = new float[tableSize];
    for(long i=0;i<tableSize;i++)
    {
        float x = Pi * 2.f * (float)i / (float)tableSize;
        table[i] = (float)sin(x);
    }
}
}

```

Phase modulation Vs. Frequency modulation II (click this to go back to the index)

References : Posted by James McCartney

Notes :
The difference between FM & PM in a digital oscillator is that FM is added to the frequency before the phase integration, while PM is added to the phase after the phase integration. Phase integration is when the old phase for the oscillator is added to the current frequency (in radians per sample) to get the new phase for the oscillator. The equivalent PM modulator to obtain the same waveform as FM is the integral of the FM modulator. Since the integral of sine waves are inverted cosine waves this is no problem. In modulators with multiple partials, the equivalent PM modulator will have different relative partial amplitudes. For example, the integral of a square wave is a triangle wave; they have the same harmonic content, but the relative partial amplitudes are different. These differences make no difference since we are not trying to exactly recreate FM, but real (or nonreal) instruments.

The reason PM is better is because in PM and FM there can be non-zero energy produced at 0 Hz, which in FM will produce a shift in pitch if the FM wave is used again as a modulator, however in PM the DC component will only produce a phase shift. Another reason PM is better is that the modulation index (which determines the number of sidebands produced and which in normal FM is calculated as the modulator amplitude divided by frequency of modulator) is not dependant on the frequency of the modulator, it is always equal to the amplitude of the modulator in radians. The benefit of solving the DC frequency shift problem, is that cascaded carrier-modulator pairs and feedback modulation are possible. The simpler calculation of modulation index makes it easier to have voices keep the same harmonic structure throughout all pitches.

The basic mathematics of phase modulation are available in any text on electronic communication theory.

Below is some C code for a digital oscillator that implements FM,PM,and AM. It illustrates the difference in implementation of FM & PM. It is only meant as an example, and not as an efficient implementation.

```
Code :
/* Example implementation of digital oscillator with FM, PM, & AM */

#define PI 3.14159265358979
#define RADIANS_TO_INDEX (512.0 / (2.0 * PI))

typedef struct{ /* oscillator data */
    double freq; /* oscillator frequency in radians per sample */
    double phase; /* accumulated oscillator phase in radians */
    double wavetable[512]; /* waveform lookup table */
} OscilRec;

/* oscil - compute 1 sample of oscillator output whose freq. phase and
 * wavetable are in the OscilRec structure pointed to by orec.
 */
double oscil(orec, fm, pm, am)
OscilRec *orec; /* pointer to the oscil's data */
double fm; /* frequency modulation input in radians per sample */
double pm; /* phase modulation input in radians */
double am; /* amplitude modulation input in any units you want */
{
    long tableindex; /* index into wavetable */
    double instantaneous_freq; /* oscillator freq + freq modulation */
    double instantaneous_phase; /* oscillator phase + phase modulation */
    double output; /* oscillator output */

    instantaneous_freq = orec->freq + fm; /* get instantaneous freq */
    orec->phase += instantaneous_freq; /* accumulate phase */
    instantaneous_phase = orec->phase + pm; /* get instantaneous phase */

    /* convert to lookup table index */
    tableindex = RADIANS_TO_INDEX * instantaneous_phase;
    tableindex &= 511; /* make it mod 512 === eliminate multiples of 2*k*PI */

    output = orec->wavetable[tableindex] * am; /* lookup and mult by am input */

    return (output); /* return oscillator output */
}
```


[Pseudo-Random generator](#) (click this to go back to the index)

Type : Linear Congruential, 32bit

References : Hal Chamberlain, "Musical Applications of Microprocessors" (Posted by Phil Burk)

Notes :

This can be used to generate random numeric sequences or to synthesise a white noise audio signal.
If you only use some of the bits, use the most significant bits by shifting right.
Do not just mask off the low bits.

Code :

```
/* Calculate pseudo-random 32 bit number based on linear congruential method. */
unsigned long GenerateRandomNumber( void )
{
    /* Change this for different random sequences. */
    static unsigned long randSeed = 22222;
    randSeed = (randSeed * 196314165) + 907633515;
    return randSeed;
}
```

[PulseQuad](#) (click this to go back to the index)

Type : Waveform

References : Posted by am[AT]andre-michelle[DOT]com

Notes :
This is written in Actionscript 3.0 (Flash9). You can listen to the example at <http://lab.andre-michelle.com/playing-with-pulse-harmonics>
It allows to morph between a sinus like quadratic function and an ordinary pulse width with adjustable pulse width. Note that the slope while morphing is always zero at the edge points of the waveform. It is not just distorsion.

Code :
<http://lab.andre-michelle.com/swf/f9/pulsequad/PulseQuad.as>

[Pulsewidth modulation](#) (click this to go back to the index)

Type : waveform generation

References : Steffan Diedrichsen

Notes :

Take an upramping sawtooth and its inverse, a downramping sawtooth. Adding these two waves with a well defined delay between 0 and period ($1/f$) results in a square wave with a duty cycle ranging from 0 to 100%.

Quick & Dirty Sine (click this to go back to the index)

Type : Sine Wave Synthesis

References : Posted by MisterToast

Notes :

This is proof of concept only (but code works--I have it in my synth now).

Note that x must come in as $0 < x \leq 4096$. If you want to scale it to something else (like $0 < x \leq 2 * M_PI$), do it in the call. Or do the math to scale the constants properly.

There's not much noise in here. A few little peaks here and there. When the signal is at -20dB, the worst noise is at around -90dB.

For speed, you can go all floats without much difference. You can get rid of that unitary negate pretty easily, as well. A couple other tricks can speed it up further--I went for clarity in the code.

The result comes out a bit shy of the range $-1 < x < 1$. That is, the peak is something like 0.999.

Where did this come from? I'm experimenting with getting rid of my waveform tables, which require huge amounts of memory. Once I had the Hamming anti-ringing code in, it looked like all my waveforms were smooth enough to approximate with curves. So I started with sine. Pulled my table data into Excel and then threw the data into a curve-fitting application.

This would be fine for a synth. The noise is low enough that you could easily get away with it. Ideal for a low-memory situation. My final code will be a bit harder to understand, as I'll break the curve up and curve-fit smaller sections.

Code :

```
float xSin(double x)
{
    //x is scaled 0<=x<4096
    const double A=-0.015959964859;
    const double B=217.68468676;
    const double C=0.000028716332164;
    const double D=-0.0030591066066;
    const double E=-7.3316892871734489e-005;
    double y;

    bool negate=false;
    if (x>2048)
    {
        negate=true;
        x-=2048;
    }
    if (x>1024)
        x=2048-x;
    if (negate)
        y=-((A+x)/(B+C*x*x)+D*x-E);
    else
        y=(A+x)/(B+C*x*x)+D*x-E;
    return (float)y;
}
```

Comments

from : toast [[a t]] somewhereyoucantfind.com

comment : Improved version:

```
float xSin(double x)
{
    //x is scaled 0<=x<4096
    const double A=-0.40319426317E-08;
    const double B=0.21683205691E+03;
    const double C=0.28463350538E-04;
    const double D=-0.30774648337E-02;
    double y;
```

```
    bool negate=false;
    if (x>2048)
    {
        negate=true;
        x-=2048;
    }
    if (x>1024)
        x=2048-x;
    y=(A+x)/(B+C*x*x)+D*x;
    if (negate)
        return (float)(-y);
    else
        return (float)y;
}
```

from : depinto1 [[a t]] oz.net

comment : %This is Matlab code. you can convert it to C

%All it take to make a high quality sine

%wave is 1 multiply and one subtract.

%You first have to initialize the 2 unit delays

% and the coefficient

```
Fs = 48000; %Sample rate
oscfreq = 1000.0; %Oscillator frequency in Hz
c1 = 2 * cos(2 * pi * oscfreq / Fs);
%Initialize the unit delays
d1 = sin(2 * pi * oscfreq / Fs);
d2 = 0;
%Initialization done here is the oscillator loop
% which generates a sinewave
for j=1:100
    output = d1; %This is the sine value
    fprintf(1, '%f\n', output);
    %one multiply and one subtract is all it takes
    d0 = d1 * c1 - d2;
    d2 = d1; %Shift the unit delays
    d1 = d0;
end
```

from : juuso.alasuutari [[a t]] gmail.com

comment : Hi,

Can I use this code in a GPL2 or GPL3 licensed program (a soft synth project called Snarl)? In other words, will you grant permission for me to re-license your code? And what name should I write down as copyright holder in the headers?

Thanks,
Juuso Alasuutari

[quick and dirty sine generator](#) (click this to go back to the index)

Type : sine generator

References : Posted by courierstv[AT]hotmail[DOT]com

Notes :

this is part of my library, although I've seen a lot of sine generators, I've never seen the simplest one, so I try to do it, tell me something, I've try it and work so tell me something about it

Code :

```
PSPsample PSPsin1::doOsc(int numCh)
{
    double x=0;
    double t=0;

    if(m_time[numCh]>m_sampleRate) //re-init cycle
        m_time[numCh]=0;

    if(m_time[numCh]>0)
    {
        t =(double)(((double)m_time[numCh])/(double)m_sampleRate);

        x=(m_2PI *(double)(t)*m_freq);
    }
    else
        x=0;

    PSPsample r=(PSPsample) sin(x+m_phase)*m_amp;

    m_time[numCh]++;

    return r;
}
```

Comments

from : pete [[a t]] bannister25.plus.com

comment : isn't the sin() function a little bit heavyweight? Since this is based upon slices of time, would it not be much more processor efficient to use a state variable filter that is self oscillating?

The operation:

```
t=(double)(((double)m_time[numCh])/(double)m_sampleRate);
```

also seems a little bit much, since t could be calculated by adding an interval value, which would eliminate the divide (needs more clocks). The divide would then only need to be done once.

An FDIV may take 39 clock cycles minimum(depending on the operands), whilst an FADD is far faster (3 clocks). An FMUL is comparable to an add, which would be a predominant instruction if using the SVF method.

FSIN may take between 16-126 clock cycles.

(clock cycle info nabbed from: <http://www.singlix.com/trdos/pentium.txt>)

from : rossb [[a t]] audiomulch.com

comment : See also the fun with sinusoids page:
<http://www.audiomulch.com/~rossb/code/sinusoids/>

[RBJ Wavetable 101](#) (click this to go back to the index)

References : Posted by Robert Bristow-Johnson

Linked file : [Wavetable-101.pdf](#)

Notes :
see linked file

Rosler and Lorenz Oscillators (click this to go back to the index)

Type : Chaotic LFO

References : Posted by kaleja[AT]estarcion[DOT]com

Notes :

The Rossler and Lorenz functions are iterated chaotic systems - they trace smooth curves that never repeat the same way twice. Lorenz is "unpitched", having no distinct peaks in its spectrum -- similar to pink noise. Rossler exhibits definite spectral peaks against a noisy broadband background.

Time-domain and frequency spectrum of these two functions, as well as other info, can be found at:

<http://www.physics.emory.edu/~weeks/research/tseries1.html>

These functions might be useful in simulating "analog drift."

Code :

Available on the web at:

<http://www.tinygod.com/code/BLorenzOsc.zip>

Comments

from : thaddy [[a t]] thaddy.com

comment : A Delphi/pascal version for VCL, KOL, Kylix and Freepascal on my website:
http://members.chello.nl/t.koning8/loro_sc.pas

Nice work!

[SawSin](#) (click this to go back to the index)

Type : Oscillator shape

References : Posted by Alexander Kritov

Code :

```
double sawsin(double x)
{
    double t = fmod(x/(2*M_PI),(double)1.0);
    if (t>0.5)
        return -sin(x);
    if (t<=0.5)
        return (double)2.0*t-1.0;
}
```

[Simple Time Stretching-Granular Synthesizer](#) (click this to go back to the index)

References : Posted by Harry-Chris

Notes :

Matlab function that implements crude time stretching - granulizing function, by overlap add in time domain.

Code :

```
function y = gran_func(x, w, H,H2, Fs, tr_amount)

% x -> input signal
% w -> Envelope - Window Vector
% H1 -> Original Hop Size
% H2 -> Synthesis Hop Size
% Fs -> Sample Rate
% str_amount -> time stretching factor

M = length(w);

pin = 1;
pend = length(x) - M;

y = zeros(1, floor( str_amount * length(x)) +M);

count = 1;
idx = 1;

while pin < pend
    input = x(pin : pin+M-1) .* w';

    y(idx : idx + M - 1) = y(idx : idx + M - 1) + input;

    pin = pin + H;
    count = count + 1;
    idx = idx + H2;
end
```

[Sine calculation](#) (click this to go back to the index)

Type : waveform generation, Taylor approximation of sin()

References : Posted by Phil Burk

Notes :
Code from JSyn for a sine wave generator based on a Taylor Expansion. It is not as efficient as the filter methods, but it has linear frequency control and is, therefore, suitable for FM or other time varying applications where accurate frequency is needed. The sine generated is accurate to at least 16 bits.

```
Code :
for(i=0; i < nSamples ; i++)
{
    //Generate sawtooth phasor to provide phase for sine generation
    IncrementWrapPhase(phase, freqPtr[i]);
    //Wrap phase back into region where results are more accurate

    if(phase > 0.5)
        yp = 1.0 - phase;
    else
    {
        if(phase < -0.5)
            yp = -1.0 - phase;
        else
            yp = phase;
    }

    x = yp * PI;
    x2 = x*x;

    //Taylor expansion out to x**9/9! factored into multiply-adds
    fastsin = x*(x2*(x2*(x2*(x2*(1.0/362880.0)
        - (1.0/5040.0))
        + (1.0/120.0))
        - (1.0/6.0))
        + 1.0);

    outPtr[i] = fastsin * amplPtr[i];
}
```

Square Waves (click this to go back to the index)

Type : waveform generation

References : Posted by Sean Costello

Notes :

One way to do a square wave:

You need two buzz generators (see Dodge & Jerse, or the Csound source code, for implementation details). One of the buzz generators runs at the desired square wave frequency, while the second buzz generator is exactly one octave above this pitch. Subtract the higher octave buzz generator's output from the lower buzz generator's output - the result should be a signal with all odd harmonics, all at equal amplitude. Filter the resultant signal (maybe integrate it). Voila, a bandlimited square wave! Well, I think it should work...

The one question I have with the above technique is whether it produces a waveform that truly resembles a square wave in the time domain. Even if the number of harmonics, and the relative ratio of the harmonics, is identical to an "ideal" bandwidth-limited square wave, it may have an entirely different waveshape. No big deal, unless the signal is processed by a nonlinearity, in which case the results of the nonlinear processing will be far different than the processing of a waveform that has a similar shape to a square wave.

Comments

from : dfl AT stanford. edu

comment : Actually, I don't think this would work...

The proper way to do it is subtract a phase shifted buzz (aka BLIT) at the same frequency. This is equivalent to comb filtering, which will notch out the even harmonics.

from : sean [[a t]] valhalladsp.com

comment : The above comment is correct, and my concept is inaccurate. My technique may have produced a signal with the proper harmonic structure, but it has been nearly 10 years since I wrote the post, so I can't remember what I was working with.

DFL's technique can be implemented with two buzz generators, or with a single buzz generator in conjunction with a fractional delay, where the delay controls the amount of phase shift.

Trammell Pink Noise (C++ class) (click this to go back to the index)

Type : pink noise generator

References : Posted by dfl at ccrma dot stanford dot edu

Code :

```
#ifndef _PinkNoise_H
#define _PinkNoise_H

// Technique by Larry "RidgeRat" Trammell 3/2006
// http://home.earthlink.net/~ltrammell/tech/pinkalg.htm
// implementation and optimization by David Lowenfels

#include <cstdlib>
#include <ctime>

#define PINK_NOISE_NUM_STAGES 3

class PinkNoise {
public:
    PinkNoise() {
        srand ( time(NULL) ); // initialize random generator
        clear();
    }

    void clear() {
        for( size_t i=0; i< PINK_NOISE_NUM_STAGES; i++ )
            state[ i ] = 0.0;
    }

    float tick() {
        static const float RMI2 = 2.0 / float(RAND_MAX); // + 1.0; // change for range [0,1)
        static const float offset = A[0] + A[1] + A[2];

        // unrolled loop
        float temp = float( rand() );
        state[0] = P[0] * (state[0] - temp) + temp;
        temp = float( rand() );
        state[1] = P[1] * (state[1] - temp) + temp;
        temp = float( rand() );
        state[2] = P[2] * (state[2] - temp) + temp;
        return ( A[0]*state[0] + A[1]*state[1] + A[2]*state[2] ) * RMI2 - offset;
    }

protected:
    float state[ PINK_NOISE_NUM_STAGES ];
    static const float A[ PINK_NOISE_NUM_STAGES ];
    static const float P[ PINK_NOISE_NUM_STAGES ];
};

const float PinkNoise::A[] = { 0.02109238, 0.07113478, 0.68873558 }; // rescaled by (1+P)/(1-P)
const float PinkNoise::P[] = { 0.3190, 0.7756, 0.9613 };

#endif
```

Comments

from : ltramme1476 [[a t]] earthlink.net

comment : Many thanks to David Lowenfels for posting this implementation of the early experimental version. I recommend switching to the new algorithm form described in 'newpink.htm' -- better range to 9+ octaves, better accuracy to +0.25 dB, and leveled computational loading. So where is MY submission to the archive? Um... well, it's coming... if he doesn't beat me to the punch again and post his code first! -- Larry Trammell (the RidgeRat)

Waveform generator using MinBLEPS (click this to go back to the index)

References : Posted by locke[AT]rpgfan.demon.co.uk

Linked file : [MinBLEPS.zip](#)

Notes :

C code and project file for MSVC6 for a bandwidth-limited saw/square (with PWM) generator using MinBLEPS.

This code is based on Eli's MATLAB MinBLEP code and uses his original minblep.mat file. Instead of keeping a list of all active MinBLEPS, the output of each MinBLEP is stored in a buffer, in which all consequent MinBLEPS and the waveform output are added together. This optimization makes it fast enough to be used realtime.

Produces slight aliasing when sweeping high frequencies. I don't know wether Eli's original code does the same, because I don't have MATLAB. Any help would be appreciated.

The project name is 'hardsync', because it's easy to generate hardsync using MinBLEPS.

Code :

Comments

from : info [[a t]] whitenoiseaudio.com

comment : <http://www.slack.net/~ant/bl-synth/windowed-impulse/>

This page also describes a similar algorithm for generating waves. Could the aliasing be due to the fact that the blep only occurs after the discontinuity? On this page the blep also occurs in the opposite direction as well, leading up to the discontinuity.

from : kernel[AT]audiospillage[DOT]com

comment : The sawtooth is a nice oscillator but I can't seem to get the square wave to work properly. Anyone else had any luck with this? Also, it's worth noting that the code assumes it is running on a little endian architecture.

Wavetable Synthesis (click this to go back to the index)

References : Robert Bristow-Johnson

Linked file : http://www.harmony-central.com/Synth/Articles/Wavetable_101/Wavetable-101.pdf

Notes :

Wavetable sythesis AES paper by RBJ.

[Weird synthesis](#) (click this to go back to the index)

[References](#) : Posted by Andy M00cho

[Notes](#) :
(quoted from Andy's mail...)
What I've done in a soft-synth I've been working on is used what I've termed Fooglers, no reason, just liked the name :) Anyway all I've done is use a *VERY* short delay line of 256 samples and then use 2 controllable taps into the delay with High Frequency Damping, and a feedback parameter.

Using a tiny fixed delay size of approx. 4.8ms (really 256 samples/1k memory with floats) means this costs, in terms of cpu consumption practically nothing, and the filter is a real simple 1 pole low-pass filter. Maybe not DSP'litically correct but all I wanted was to avoid the high frequencies trashing the delay line when high feedbacks (99%->99.9%) are used (when the fun starts ;).

I've been getting some really sexy sounds out of this idea, and of course you can have the delay line tuneable if you choose to use fractional taps, but I'm happy with it as it is.. 1 nice simple, yet powerful addition to the base oscillators.

In reality you don't need 2 taps, but I found that using 2 added that extra element of funkiness...

[Comments](#)

[from](#) : philmagnotta [[a t]] aol.com

[comment](#) : Andy:

I'm curious about your delay line. It's length is

4.8 m.sec.fixed. What are the variables in the two controllable taps and is the 6dB filter variable frequency wise?

Phil

[from](#) : electropop [[a t]] yahoo.com

[comment](#) : What you have there is the core of a physical modelling algorithm. I have done virtually the same thing to model plucked string instruments in Reaktor. It's amazingly realistic. See <http://joeorgren.com>

Beat Detector Class (click this to go back to the index)

References : Posted by DSPMaster[at]free[dot]fr

Notes :

This class was designed for a VST plugin. Basically, it's just a 2nd order LP filter, followed by an envelope detector (thanks Bram), feeding a Schmitt trigger. The rising edge detector provides a 1-sample pulse each time a beat is detected. Code is self documented...

Note : The class uses a fixed comparison level, you may need to change it.

Code :

```
// ***** BEATDETECTOR.H *****
#ifndef BeatDetectorH
#define BeatDetectorH

class TBeatDetector
{
private:
    float KBeatFilter;          // Filter coefficient
    float Filter1Out, Filter2Out;
    float BeatRelease;         // Release time coefficient
    float PeakEnv;             // Peak envelope follower
    bool BeatTrigger;          // Schmitt trigger output
    bool PrevBeatPulse;        // Rising edge memory
public:
    bool BeatPulse;            // Beat detector output

    TBeatDetector();
    ~TBeatDetector();
    virtual void setSampleRate(float SampleRate);
    virtual void AudioProcess (float input);
};

#endif

// ***** BEATDETECTOR.CPP *****
#include "BeatDetector.h"
#include "math.h"

#define FREQ_LP_BEAT 150.0f    // Low Pass filter frequency
#define T_FILTER 1.0f/(2.0f*M_PI*FREQ_LP_BEAT) // Low Pass filter time constant
#define BEAT_RUNTIME 0.02f    // Release time of envelope detector in second

TBeatDetector::TBeatDetector()
// Beat detector constructor
{
    Filter1Out=0.0;
    Filter2Out=0.0;
    PeakEnv=0.0;
    BeatTrigger=false;
    PrevBeatPulse=false;
    setSampleRate(44100);
}

TBeatDetector::~TBeatDetector()
{
    // Nothing specific to do...
}

void TBeatDetector::setSampleRate (float sampleRate)
// Compute all sample frequency related coeffs
{
    KBeatFilter=1.0/(sampleRate*T_FILTER);
    BeatRelease=(float)exp(-1.0f/(sampleRate*BEAT_RUNTIME));
}

void TBeatDetector::AudioProcess (float input)
// Process incoming signal
{
    float EnvIn;

    // Step 1 : 2nd order low pass filter (made of two 1st order RC filter)
    Filter1Out=Filter1Out+(KBeatFilter*(input-Filter1Out));
    Filter2Out=Filter2Out+(KBeatFilter*(Filter1Out-Filter2Out));

    // Step 2 : peak detector
    EnvIn=fabs(Filter2Out);
    if (EnvIn>PeakEnv) PeakEnv=EnvIn; // Attack time = 0
    else
    {
        PeakEnv*=BeatRelease;
        PeakEnv+=(1.0f-BeatRelease)*EnvIn;
    }

    // Step 3 : Schmitt trigger
    if (!BeatTrigger)
    {
        if (PeakEnv>0.3) BeatTrigger=true;
    }
    else
    {
        if (PeakEnv<0.15) BeatTrigger=false;
    }
}

```

```

}

// Step 4 : rising edge detector
BeatPulse=false;
if ((BeatTrigger)&&(!PrevBeatPulse))
    BeatPulse=true;
PrevBeatPulse=BeatTrigger;
}

```

Comments

```

from : thaddy [ [ a t ] ] thaddy.com
comment : // Nice work!
//Here's a Delphi and freepascal version:
unit beattrigger;

```

interface

```

type
TBeatDetector = class
private
    KBeatFilter,          // Filter coefficient
    Filter1Out,
    Filter2Out,
    BeatRelease,         // Release time coefficient
    PeakEnv:single;     // Peak envelope follower
    BeatTrigger,        // Schmitt trigger output
    PrevBeatPulse:Boolean; // Rising edge memory
public
    BeatPulse:Boolean;  // Beat detector output
    constructor Create;
    procedure setSampleRate(SampleRate:single);
    procedure AudioProcess (input:single);
end;

```

```

function fabs(value:single):Single;

```

implementation

```

const
    FREQ_LP_BEAT = 150.0;          // Low Pass filter frequency
    T_FILTER = 1.0/(2.0 * PI*FREQ_LP_BEAT); // Low Pass filter time constant
    BEAT_RTIME = 0.02; // Release time of envelope detector in second

```

```

constructor TBeatDetector.create;

```

```

// Beat detector constructor

```

```

begin
    inherited;
    Filter1Out:=0.0;
    Filter2Out:=0.0;
    PeakEnv:=0.0;
    BeatTrigger:=false;
    PrevBeatPulse:=false;
    setSampleRate(44100);
end;

```

```

procedure TBeatDetector.setSampleRate (sampleRate:single);

```

```

// Compute all sample frequency related coeffs

```

```

begin
    KBeatFilter:=1.0/(sampleRate*T_FILTER);
    BeatRelease:= exp(-1.0/(sampleRate*BEAT_RTIME));
end;

```

```

function fabs(value:single):Single;

```

```

asm
    fld value
    fabs
    fwait
end;

```

```

procedure TBeatDetector.AudioProcess (input:single);

```

```

var

```

```

    EnvIn:Single;

```

```

// Process incoming signal

```

```

begin

```

```

    // Step 1 : 2nd order low pass filter (made of two 1st order RC filter)

```

```

    Filter1Out:=Filter1Out+(KBeatFilter*(input-Filter1Out));

```

```

    Filter2Out:=Filter2Out+(KBeatFilter*(Filter1Out-Filter2Out));

```

```

    // Step 2 : peak detector

```

```

    EnvIn:=fabs(Filter2Out);

```

```

    if EnvIn>PeakEnv then PeakEnv:=EnvIn // Attack time = 0

```

```

    else

```

```

    begin

```

```

        PeakEnv:=PeakEnv*BeatRelease;

```

```
    PeakEnv:=PeakEnv+(1.0-BeatRelease)*EnvIn;
end;
// Step 3 : Schmitt trigger
if not BeatTrigger then
begin
    if PeakEnv>0.3 then BeatTrigger:=true;
end
else
begin
    if PeakEnv<0.15 then BeatTrigger:=false;
end;

// Step 4 : rising edge detector
BeatPulse:=false;
if (BeatTrigger = true ) and( not PrevBeatPulse) then
    BeatPulse:=true;
PrevBeatPulse:=BeatTrigger;
end;

end.
```

from : foo [[a t]] bar.baz

comment : If you have virtual methods the destructor should be virtual as well as otherwise the destruction of derived objects is undefined.

from : thaddy [[a t]] thaddy.com

comment : This already - implied - the case in the PAS version. The above comment holds only for the C++ version, but makes sense to me.

Coefficients for Daubechies wavelets 1-38 (click this to go back to the index)

Type : wavelet transform

References : Computed by Kazuo Hatano, Compiled and verified by Olli Niemitalo

Linked file : [daub.h](#) (this linked file is included below)

Linked files

```
/* Coefficients for Daubechies wavelets 1-38
* -----
* 2000.08.10 - Some more info added.
*
* Computed by Kazuo Hatano, Aichi Institute of Technology.
* ftp://phase.etl.go.jp/pub/phase/wavelet/index.html
*
* Compiled and verified by Olli Niemitalo.
*
* Discrete Wavelet Transformation (DWT) breaks a signal down into
* subbands distributed logarithmically in frequency, each sampled
* at a rate that has a natural proportion to the frequencies in that
* band. The traditional fourier transformation has no time domain
* resolution at all, or when done using many short windows on a
* longer data, equal resolution at all frequencies. The distribution
* of samples in the time and frequency domain by DWT is of form:
*
* log f
* |XXXXXXXXXXXXXXXXXXXXX X = a sample
* |X X X X X X X X f = frequency
* |X X X X t = time
* |X X
* |X
* |-----t
*
* Single
* subband decomposition and reconstruction:
*
*   -> high -> decimate -----> dilute -> high
*   | pass by 2 high subband by 2 pass \
* in | + out
*   | / =in
*   -> low -> decimate -----> dilute -> low
*   | pass by 2 low subband by 2 pass
*
* This creates two subbands from the input signal, both sampled at half
* the original frequency. The filters approximate halfband FIR filters
* and are determined by the choice of wavelet. Using Daubechies wavelets
* (and most others), the data can be reconstructed to the exact original
* even when the halfband filters are not perfect. Note that the amount
* of information (samples) stays the same throughout the operation.
*
* Decimation by 2: ABCDEFGHIJKLMNOPQR -> ACEGIKMOQ
* Dilution by 2: ACEGIKMOQ -> A0C0E0G0I0K0M0O0Q0
*
* To get the logarithmic resolution in frequency, the low subband is
* re-transformed, and again, the low subband from this transformation
* gets the same treatment etc.
*
* Decomposition:
*
*   -> high -> decimate -----> subband0
*   | pass by 2
* in | -> high -> decimate -----> subband1
*   | pass by 2
*   -> low -> decim | -> high -> decim -> subband2
*   | pass by 2 | pass by 2
*   | -> low -> decim |
*   | pass by 2 | . down to what suffices
*   | . -> . or if periodic data,
*   | . until short of data
*
* Reconstruction:
*
* subband0 -----> dilute -> high
* | by 2 pass \
* subband1 -----> dilute -> high + out
* | by 2 pass \ / =in
* subband2 -> dilute -> high + dilute -> low
* | by 2 pass \ / by 2 pass
* | + dilute -> low
* Start . / by 2 pass
* here! . -> dilute -> low
```

```

*           .   by 2       pass
*
* In a real-time application, the filters introduce delays, so you need
* to compensate them by adding additional delays to less-delayed higher
* bands, to get the summation work as intended.
*
* For periodic signals or windowed operation, this problem doesn't exist -
* a single subband transformation is a matrix multiplication, with wrapping
* implemented in the matrix:

```

```

* Decomposition:

```

```

* | L0 |   | C0 C1 C2 C3 |   | I0 |   L = lowpass output
* | H0 |   | C3 -C2 C1 -C0 |   | I1 |   H = highpass output
* | L1 |   |           C0 C1 C2 C3 |   | I2 |   I = input
* | H1 | = |           C3 -C2 C1 -C0 |   | I3 |   C = coefficients
* | L2 |   |           C0 C1 C2 C3 |   | I4 |
* | H2 |   |           C3 -C2 C1 -C0 |   | I5 |
* | L3 |   | C2 C3           C0 C1 |   | I6 |
* | H3 |   | C1 -C0           C3 -C2 |   | I7 |   Daubechies 4-coef:

```

```

*           1+sqrt(3)           3+sqrt(3)           3-sqrt(3)           1-sqrt(3)
* C0 = -----   C1 = -----   C2 = -----   C3 = -----
*           4 sqrt(2)           4 sqrt(2)           4 sqrt(2)           4 sqrt(2)

```

```

* Reconstruction:

```

```

* | I0 |   | C0 C3           C2 C1 |   | L0 |
* | I1 |   | C1 -C2           C3 -C0 |   | H0 |
* | I2 |   | C2 C1 C0 C3           |   | L1 |
* | I3 | = | C3 -C0 C1 -C2           |   | H1 |
* | I4 |   | C2 C1 C0 C3           |   | L2 |
* | I5 |   | C3 -C0 C1 -C2           |   | H2 |
* | I6 |   | C2 C1 C0 C3           |   | L3 |
* | I7 |   | C3 -C0 C1 -C2           |   | H3 |

```

```

* This file contains the lowpass FIR filter coefficients. Highpass
* coefficients you get by reversing tap order and multiplying by
* sequence 1,-1, 1,-1, ... Because these are orthogonal wavelets, the
* analysis and reconstruction coefficients are the same.

```

```

* A coefficient set convolved by its reverse is an ideal halfband lowpass
* filter multiplied by a symmetric windowing function. This creates the
* kind of symmetry in the frequency domain that enables aliasing-free
* reconstruction. Daubechies wavelets are the minimum-phase, minimum
* number of taps solutions for a number of vanishing moments (seven in
* Daub7 etc), which determines their frequency selectivity.
*/

```

```

const double Daub1[2] = {
  7.071067811865475244008443621048490392848359376884740365883398e-01,
  7.071067811865475244008443621048490392848359376884740365883398e-01};

```

```

const double Daub2[4] = {
  4.829629131445341433748715998644486838169524195042022752011715e-01,
  8.365163037378079055752937809168732034593703883484392934953414e-01,
  2.241438680420133810259727622404003554678835181842717613871683e-01,
  -1.294095225512603811744494188120241641745344506599652569070016e-01};

```

```

const double Daub3[6] = {
  3.326705529500826159985115891390056300129233992450683597084705e-01,
  8.068915093110925764944936040887134905192973949948236181650920e-01,
  4.598775021184915700951519421476167208081101774314923066433867e-01,
  -1.350110200102545886963899066993744805622198452237811919756862e-01,
  -8.544127388202666169281916918177331153619763898808662976351748e-02,
  3.522629188570953660274066471551002932775838791743161039893406e-02};

```

```

const double Daub4[8] = {
  2.303778133088965008632911830440708500016152482483092977910968e-01,
  7.148465705529156470899219552739926037076084010993081758450110e-01,
  6.308807679298589078817163383006152202032229226771951174057473e-01,
  -2.798376941685985421141374718007538541198732022449175284003358e-02,
  -1.870348117190930840795706727890814195845441743745800912057770e-01,
  3.084138183556076362721936253495905017031482172003403341821219e-02,
  3.288301166688519973540751354924438866454194113754971259727278e-02,
  -1.059740178506903210488320852402722918109996490637641983484974e-02};

```

```

const double Daub5[10] = {
  1.601023979741929144807237480204207336505441246250578327725699e-01,
  6.038292697971896705401193065250621075074221631016986987969283e-01,
  7.24308528437729277280712441022186407687562182320073725767335e-01,
  1.384281459013207315053971463390246973141057911739561022694652e-01,
  -2.422948870663820318625713794746163619914908080626185983913726e-01,

```

```
-3.224486958463837464847975506213492831356498416379847225434268e-02,  
7.757149384004571352313048938860181980623099452012527983210146e-02,  
-6.24149021279827427190519112920192970763557165687607323417435e-03,  
-1.258075199908199946850973993177579294920459162609785020169232e-02,  
3.335725285473771277998183415817355747636524742305315099706428e-03};
```

```
const double Daub6[12] = {  
1.115407433501094636213239172409234390425395919844216759082360e-01,  
4.946238903984530856772041768778555886377863828962743623531834e-01,  
7.511339080210953506789344984397316855802547833382612009730420e-01,  
3.152503517091976290859896548109263966495199235172945244404163e-01,  
-2.262646939654398200763145006609034656705401539728969940143487e-01,  
-1.297668675672619355622896058765854608452337492235814701599310e-01,  
9.750160558732304910234355253812534233983074749525514279893193e-02,  
2.752286553030572862554083950419321365738758783043454321494202e-02,  
-3.158203931748602956507908069984866905747953237314842337511464e-02,  
5.538422011614961392519183980465012206110262773864964295476524e-04,  
4.777257510945510639635975246820707050230501216581434297593254e-03,  
-1.077301085308479564852621609587200035235233609334419689818580e-03};
```

```
const double Daub7[14] = {  
7.785205408500917901996352195789374837918305292795568438702937e-02,  
3.965393194819173065390003909368428563587151149333287401110499e-01,  
7.291320908462351199169430703392820517179660611901363782697715e-01,  
4.697822874051931224715911609744517386817913056787359532392529e-01,  
-1.439060039285649754050683622130460017952735705499084834401753e-01,  
-2.240361849938749826381404202332509644757830896773246552665095e-01,  
7.130921926683026475087657050112904822711327451412314659575113e-02,  
8.061260915108307191292248035938190585823820965629489058139218e-02,  
-3.802993693501441357959206160185803585446196938467869898283122e-02,  
-1.657454163066688065410767489170265479204504394820713705239272e-02,  
1.255099855609984061298988603418777957289474046048710038411818e-02,  
4.295779729213665211321291228197322228235350396942409742946366e-04,  
-1.801640704047490915268262912739550962585651469641090625323864e-03,  
3.537137999745202484462958363064254310959060059520040012524275e-04};
```

```
const double Daub8[16] = {  
5.441584224310400995500940520299935503599554294733050397729280e-02,  
3.128715909142999706591623755057177219497319740370229185698712e-01,  
6.756307362972898068078007670471831499869115906336364227766759e-01,  
5.853546836542067127712655200450981944303266678053369055707175e-01,  
-1.582910525634930566738054787646630415774471154502826559735335e-02,  
-2.840155429615469265162031323741647324684350124871451793599204e-01,  
4.724845739132827703605900098258949861948011288770074644084096e-04,  
1.287474266204784588570292875097083843022601575556488795577000e-01,  
-1.736930100180754616961614886809598311413086529488394316977315e-02,  
-4.408825393079475150676372323896350189751839190110996472750391e-02,  
1.398102791739828164872293057263345144239559532934347169146368e-02,  
8.746094047405776716382743246475640180402147081140676742686747e-03,  
-4.870352993451574310422181557109824016634978512157003764736208e-03,  
-3.917403733769470462980803573237762675229350073890493724492694e-04,  
6.754494064505693663695475738792991218489630013558432103617077e-04,  
-1.174767841247695337306282316988909444086693950311503927620013e-04};
```

```
const double Daub9[18] = {  
3.807794736387834658869765887955118448771714496278417476647192e-02,  
2.438346746125903537320415816492844155263611085609231361429088e-01,  
6.048231236901111119030768674342361708959562711896117565333713e-01,  
6.572880780513005380782126390451732140305858669245918854436034e-01,  
1.331973858250075761909549458997955536921780768433661136154346e-01,  
-2.932737832791749088064031952421987310438961628589906825725112e-01,  
-9.684078322297646051350813353769660224825458104599099679471267e-02,  
1.485407493381063801350727175060423024791258577280603060771649e-01,  
3.072568147933337921231740072037882714105805024670744781503060e-02,  
-6.763282906132997367564227482971901592578790871353739900748331e-02,  
2.509471148314519575871897499885543315176271993709633321834164e-04,  
2.236166212367909720537378270269095241855646688308853754721816e-02,  
-4.723204757751397277925707848242465405729514912627938018758526e-03,  
-4.281503682463429834496795002314531876481181811463288374860455e-03,  
1.847646883056226476619129491125677051121081359600318160732515e-03,  
2.303857635231959672052163928245421692940662052463711972260006e-04,  
-2.519631889427101369749886842878606607282181543478028214134265e-04,  
3.934732031627159948068988306589150707782477055517013507359938e-05};
```

```
const double Daub10[20] = {  
2.66700579005555358661744877130858277192498290851289932779975e-02,  
1.881768000776914890208929736790939942702546758640393484348595e-01,  
5.272011889317255864817448279595081924981402680840223445318549e-01,  
6.884590394536035657418717825492358539771364042407339537279681e-01,  
2.811723436605774607487269984455892876243888859026150413831543e-01,  
-2.498464243273153794161018979207791000564669737132073715013121e-01,  
-1.959462743773770435042992543190981318766776476382778474396781e-01,  
1.273693403357932600826772332014009770786177480422245995563097e-01,
```

```
9.305736460357235116035228983545273226942917998946925868063974e-02,  
-7.139414716639708714533609307605064767292611983702150917523756e-02,  
-2.945753682187581285828323760141839199388200516064948779769654e-02,  
3.321267405934100173976365318215912897978337413267096043323351e-02,  
3.606553566956169655423291417133403299517350518618994762730612e-03,  
-1.073317548333057504431811410651364448111548781143923213370333e-02,  
1.395351747052901165789318447957707567660542855688552426721117e-03,  
1.992405295185056117158742242640643211762555365514105280067936e-03,  
-6.858566949597116265613709819265714196625043336786920516211903e-04,  
-1.164668551292854509514809710258991891527461854347597362819235e-04,  
9.358867032006959133405013034222854399688456215297276443521873e-05,  
-1.326420289452124481243667531226683305749240960605829756400674e-05};
```

```
const double Daub11[22] = {  
1.869429776147108402543572939561975728967774455921958543286692e-02,  
1.440670211506245127951915849361001143023718967556239604318852e-01,  
4.498997643560453347688940373853603677806895378648933474599655e-01,  
6.856867749162005111209386316963097935940204964567703495051589e-01,  
4.119643689479074629259396485710667307430400410187845315697242e-01,  
-1.622752450274903622405827269985511540744264324212130209649667e-01,  
-2.742308468179469612021009452835266628648089521775178221905778e-01,  
6.604358819668319190061457888126302656753142168940791541113457e-02,  
1.498120124663784964066562617044193298588272420267484653796909e-01,  
-4.647995511668418727161722589023744577223260966848260747450320e-02,  
-6.643878569502520527899215536971203191819566896079739622858574e-02,  
3.13350902190460763094798408303144536358105680880031964936445e-02,  
2.084090436018106302294811255656491015157761832734715691126692e-02,  
-1.536482090620159942619811609958822744014326495773000120205848e-02,  
-3.340858873014445606090808617982406101930658359499190845656731e-03,  
4.928417656059041123170739741708273690285547729915802418397458e-03,  
-3.085928588151431651754590726278953307180216605078488581921562e-04,  
-8.930232506662646133900824622648653989879519878620728793133358e-04,  
2.491525235528234988712216872666801088221199302855425381971392e-04,  
5.443907469936847167357856879576832191936678525600793978043688e-05,  
-3.463498418698499554128085159974043214506488048233458035943601e-05,  
4.494274277236510095415648282310130916410497987383753460571741e-06};
```

```
const double Daub12[24] = {  
1.311225795722951750674609088893328065665510641931325007748280e-02,  
1.095662728211851546057045050248905426075680503066774046383657e-01,  
3.773551352142126570928212604879206149010941706057526334705839e-01,  
6.571987225793070893027611286641169834250203289988412141394281e-01,  
5.158864784278156087560326480543032700677693087036090056127647e-01,  
-4.476388565377462666762747311540166529284543631505924139071704e-02,  
-3.161784537527855368648029353478031098508839032547364389574203e-01,  
-2.377925725606972768399754609133225784553366558331741152482612e-02,  
1.824786059275796798540436116189241710294771448096302698329011e-01,  
5.359569674352150328276276729768332288862665184192705821636342e-03,  
-9.643212009650708202650320534322484127430880143045220514346402e-02,  
1.084913025582218438089010237748152188661630567603334659322512e-02,  
4.154627749508444073927094681906574864513532221388374861287078e-02,  
-1.221864906974828071998798266471567712982466093116558175344811e-02,  
-1.284082519830068329466034471894728496206109832314097633275225e-02,  
6.711499008795509177767027068215672450648112185856456740379455e-03,  
2.248607240995273759950865211267234018343199786146177099262010e-03,  
-2.179503618627760471598903379584171187840075291860571264980942e-03,  
6.545128212509595566500430399327110729111770568897356630714552e-06,  
3.886530628209314435897288837795981791917488573420177523436096e-04,  
-8.850410920820432420821645961553726598738322151471932808015443e-05,  
-2.424154575703078402978915320531719580423778362664282239377532e-05,  
1.277695221937976658714046362616620887375960941439428756055353e-05,  
-1.529071758068510902712239164522901223197615439660340672602696e-06};
```

```
const double Daub13[26] = {  
9.202133538962367972970163475644184667534171916416562386009703e-03,  
8.286124387290277964432027131230466405208113332890135072514277e-02,  
3.119963221604380633960784112214049693946683528967180317160390e-01,  
6.110558511587876528211995136744180562073612676018239438526582e-01,  
5.888895704312189080710395347395333927665986382812836042235573e-01,  
8.698572617964723731023739838087494399231884076619701250882016e-02,  
-3.149729077113886329981698255932282582876888450678789025950306e-01,  
-1.245767307508152589413808336021260180792739295173634719572069e-01,  
1.794760794293398432348450072339369013581966256244133393042881e-01,  
7.29489336567771638092830610477661983325929026879873553627963e-02,  
-1.058076181879343264509667304196464849478860754801236658232360e-01,  
-2.648840647534369463963912248034785726419604844297697016264224e-02,  
5.613947710028342886214501998387331119988378792543100244737056e-02,  
2.379972254059078811465170958554208358094394612051934868475139e-02,  
-2.383142071032364903206403067757739134252922717636226274077298e-03,  
3.923941448797416243316370220815526558824746623451404043918407e-03,  
7.255589401617566194518393300502698898973529679646683695269828e-03,  
-2.761911234656862178014576266098445995350093330501818024966316e-03,  
-1.315673911892298936613835370593643376060412592653652307238124e-03,
```

```
9.323261308672633862226517802548514100918088299801952307991569e-04,  
4.925152512628946192140957387866596210103778299388823500840094e-05,  
-1.651289885565054894616687709238000755898548214659776703347801e-04,  
3.067853757932549346649483228575476236600428217237900563128230e-05,  
1.044193057140813708170714991080596951670706436217328169641474e-05,  
-4.700416479360868325650195165061771321650383582970958556568059e-06,  
5.220035098454864691736424354843176976747052155243557001531901e-07};
```

```
const double Daub14[28] = {  
6.461153460087947818166397448622814272327159419201199218101404e-03,  
6.236475884939889832798566758434877428305333693407667164602518e-02,  
2.548502677926213536659077886778286686187042416367137443780084e-01,  
5.543056179408938359926831449851154844078269830951634609683997e-01,  
6.311878491048567795576617135358172348623952456570017289788809e-01,  
2.186706877589065214917475918217517051765774321270432059030273e-01,  
-2.716885522787480414142192476181171094604882465683330814311896e-01,  
-2.180335299932760447555558812702311911975240669470604752747127e-01,  
1.383952138648065910739939690021573713989900463229686119059119e-01,  
1.399890165844607012492943162271163440328221555614326181333683e-01,  
-8.674841156816968904560822066727795382979149539517503657492964e-02,  
-7.15489555040461307358414511517380799095806967312953809990913e-02,  
5.523712625921604411618834060533403397913833632511672157671107e-02,  
2.698140830791291697399031403215193343375766595807274233284349e-02,  
-3.018535154039063518714822623489137573781575406658652624883756e-02,  
-5.615049530356959133218371367691498637457297203925810387698680e-03,  
1.278949326633340896157330705784079299374903861572058313481534e-02,  
-7.462189892683849371817160739181780971958187988813302900435487e-04,  
-3.849638868022187445786349316095551774096818508285700493058915e-03,  
1.061691085606761843032566749388411173033941582147830863893939e-03,  
7.080211542355278586442977697617128983471863464181595371670094e-04,  
-3.868319473129544821076663398057314427328902107842165379901468e-04,  
-4.177724577037259735267979539839258928389726590132730131054323e-05,  
6.875504252697509603873437021628031601890370687651875279882727e-05,  
-1.033720918457077394661407342594814586269272509490744850691443e-05,  
-4.389704901781394115254042561367169829323085360800825718151049e-06,  
1.724994675367812769885712692741798523587894709867356576910717e-06,  
-1.787139968311359076334192938470839343882990309976959446994022e-07};
```

```
const double Daub15[30] = {  
4.538537361578898881459394910211696346663671243788786997916513e-03,  
4.674339489276627189170969334843575776579151700214943513113197e-02,  
2.060238639869957315398915009476307219306138505641930902702047e-01,  
4.926317717081396236067757074029946372617221565130932402160160e-01,  
6.4581314035474243581764209120106917996432608287494046181071489e-01,  
3.390025354547315276912641143835773918756769491793554669336690e-01,  
-1.932041396091454287063990534321471746304090039142863827937754e-01,  
-2.888825965669656462484125009822332981311435630435342594971292e-01,  
6.528295284877281692283107919869574882039174285596144125965101e-02,  
1.901467140071229823484893116586020517959501258174336696878156e-01,  
-3.966617655579094448384366751896200668381742820683736805449745e-02,  
-1.111209360372316933656710324674058608858623762165914120505657e-01,  
3.387714392350768620854817844433523770864744687411265369463195e-02,  
5.478055058450761268913790312581879108609415997422768564244845e-02,  
-2.576700732843996258594525754269826392203641634825340138396836e-02,  
-2.081005016969308167788483424677000162054657951364899040996166e-02,  
1.508391802783590236329274460170322736244892823305627716233968e-02,  
5.101000360407543169708860185565314724801066527344222055526631e-03,  
-6.487734560315744995181683149218690816955845639388826407928967e-03,  
-2.417564907616242811667225326300179605229946995814535223329411e-04,  
1.943323980382211541764912332541087441011424865579531401452302e-03,  
-3.734823541376169920098094213645414611387630968030256625740226e-04,  
-3.595652443624688121649620075909808858194202454084090305627480e-04,  
1.558964899205997479471658241227108816255567059625495915228603e-04,  
2.579269915531893680925862417616855912944042368767340709160119e-05,  
-2.813329626604781364755324777078478665791443876293788904267255e-05,  
3.362987181737579803124845210420177472134846655864078187186304e-06,  
1.811270407940577083768510912285841160577085925337507850590290e-06,  
-6.316882325881664421201597299517657654166137915121195510416641e-07,  
6.133359913305752029056299460289788601989190450885396512173845e-08};
```

```
const double Daub16[32] = {  
3.189220925347738029769547564645958687067086750131428767875878e-03,  
3.490771432367334641030147224023020009218241430503984146140054e-02,  
1.650642834888531178991252730561134811584835002342723240213529e-01,  
4.303127228460038137403925424357684620633970478036986773924646e-01,  
6.373563320837888986319852412996030536498595940814198125967751e-01,  
4.402902568863569000390869163571679288527803035135272578789884e-01,  
-8.975108940248964285718718077442597430659247445582660149624718e-02,  
-3.270633105279177046462905675689119641757228918228812428141723e-01,  
-2.791820813302827668264519595026873204339971219174736041535479e-02,  
2.111906939471042887209680163268837900928491426167679439251042e-01,  
2.734026375271604136485245757201617965429027819507130220231500e-02,  
-1.323883055638103904500474147756493375092287817706027978798549e-01,
```


-6.239722752474871765674503394120025865444656311678760990761458e-03,
7.592423604427631582148498743941422461530405946100943351940313e-02,
-7.58897436885773763849489086463699579658697514499025400097160e-03,
-3.688839769173014233352666320894554314718748429706730831064068e-02,
1.029765964095596941165000580076616900528856265803662208854147e-02,
1.399376885982873102950451873670329726409840291727868988490100e-02,
-6.99001456341391667028429536517288338057856199646469078115759e-03,
-3.64427962149838932169000540933629387055333973353108668841215e-03,
3.128023381206268831661202559854678767821471906193608117450360e-03,
4.078969808497128362417470323406095782431952972310546715071397e-04,
-9.410217493595675889266453953635875407754747216734480509250273e-04,
1.1424152003872239264402280099555662945839684344936472652877091e-04,
1.747872452253381803801758637660746874986024728615399897971953e-04,
-6.103596621410935835162369150522212811957259981965919143961722e-05,
-1.394566898820889345199078311998401982325273569198675335408707e-05,
1.133660866127625858758848762886536997519471068203753661757843e-05,
-1.043571342311606501525454737262615404887478930635676471546032e-06,
-7.363656785451205512099695719725563646585445545841663327433569e-07,
2.30878408685754586640541273294200612130630673586665525372544e-07,
-2.109339630100743097000572623603489906836297584591605307745349e-08};

```
const double Daubl7[34] = {  
2.241807001037312853535962677074436914062191880560370733250531e-03,  
2.598539370360604338914864591720788315473944524878241294399948e-02,  
1.312149033078244065775506231859069960144293609259978530067004e-01,  
3.703507241526411504492548190721886449477078876896803823650425e-01,  
6.109966156846228181886678867679372082737093893358726291371783e-01,  
5.183157640569378393254538528085968046216817197718416402439904e-01,  
2.731497040329363500431250719147586480350469818964563003672942e-02,  
-3.283207483639617360909665340725061767581597698151558024679130e-01,  
-1.265997522158827028744679110933825505053966260104086162103728e-01,  
1.973105895650109927854047044781930142551422414135646917122284e-01,  
1.011354891774702721509699856433434802196622545499664876109437e-01,  
-1.268156917782863110948571128662331680384792185915017065732137e-01,  
-5.709141963167692728911239478651382324161160869845347053990144e-02,  
8.110598665416088507965885748555429201024364190954499194020678e-02,  
2.231233617810379595339136059534813756232242114093689244020869e-02,  
-4.692243838926973733300897059211400507138768125498030602878439e-02,  
-3.270955535819293781655360222177494452069525958061609392809275e-03,  
2.273367658394627031845616244788448969906713741338339498024864e-02,  
-3.0429899813563706859248263790720607863395457225096588287881e-03,  
-8.602921520322854831713706413243659917926736284271730611920986e-03,  
2.967996691526094872806485060008038269959463846548378995044195e-03,  
2.301205242153545624302059869038423604241976680189447476064764e-03,  
-1.436845304802976126222890402980384903503674530729935809561434e-03,  
-3.281325194098379713954444017520115075812402442728749700195651e-04,  
4.394654277686436778385677527317841632289249319738892179465910e-04,  
-2.561010956654845882729891210949920221664082061531909655178413e-05,  
-8.204803202453391839095482576282189866136273049636764338689593e-05,  
2.318681379874595084482068205706277572106695174091895338530734e-05,  
6.990600985076751273204549700855378627762758585902057964027481e-06,  
-4.505942477222988194102268206378312129713572600716499944918416e-06,  
3.016549609994557415605207594879939763476168705217646897702706e-07,  
2.957700933316856754979905258816151367870345628924317307354639e-07,  
-8.42394844600268017878707129692287706841031094222799622593133e-08,  
7.2674929685616081108079767441409035034158581719789791088892046e-09};
```

```
const double Daubl8[36] = {  
1.576310218440760431540744929939777747670753710991660363684429e-03,  
1.928853172414637705921391715829052419954667025288497572236714e-02,  
1.035884658224235962241910491937253596470696555220241672976224e-01,  
3.146789413370316990571998255652579931786706190489374509491307e-01,  
5.718268077666072234818589370900623419393673743130930561295324e-01,  
5.718016548886513352891119994065965025668047882818525060759395e-01,  
1.472231119699281415750977271081072312557864107355701387801677e-01,  
-2.936540407365587442479030994981150723935710729035053239661752e-01,  
-2.164809340051429711237678625668271471437937235669492408388692e-01,  
1.495339755653777893509301738913667208804816691893765610261943e-01,  
1.670813127632574045149318139950134745324205646353988083152250e-01,  
-9.233188415084628060429372558659459731431848000144569612074508e-02,  
-1.067522466598284855932200581614984861385266404624112083917702e-01,  
6.488721621190544281947577955141911463129382116634147846137149e-02,  
5.705124773853688412090768846499622260596226120431038524600676e-02,  
-4.452614190298232471556143559744653492971477891439833592755034e-02,  
-2.373321039586000103275209582665216110197519330713490233071565e-02,  
2.667070592647059029987908631672020343207895999936072813363471e-02,  
6.262167954305707485236093144497882501990325204745013190268052e-03,  
-1.305148094661200177277636447600807169755191054507571666606133e-02,  
1.186300338581174657301741592161819084544899417452317405185615e-04,  
4.943343605466738130665529516802974834299638313366477765295203e-03,  
-1.118732666992497072800658855238650182318060482584970145512687e-03,  
-1.340596298336106629517567228251583609823044524685986640323942e-03,  
6.284656829651457125619449885420838217551022796301582874349652e-04,
```

```
2.135815619103406884039052814341926025873200325996466522543440e-04,
-1.986485523117479485798245416362489554927797880264017876139605e-04,
-1.535917123534724675069770335876717193700472427021513236587288e-07,
3.741237880740038181092208138035393952304292615793985030731363e-05,
-8.520602537446695203919254911655523022437596956226376512305917e-06,
-3.332634478885821888782452033341036827311505907796498439829337e-06,
1.768712983627615455876328730755375176412501359114058815453100e-06,
-7.691632689885176146000152878539598405817397588156525116769908e-08,
-1.176098767028231698450982356561292561347579777695396953528141e-07,
3.068835863045174800935478294933975372450179787894574492930570e-08,
-2.507934454948598267195173183147126731806317144868275819941403e-09};
```

```
const double Daub19[38] = {
1.108669763181710571099154195209715164245299677773435932135455e-03,
1.428109845076439737439889152950199234745663442163665957870715e-02,
8.127811326545955065296306784901624839844979971028620366497726e-02,
2.643884317408967846748100380289426873862377807211920718417385e-01,
5.244363774646549153360575975484064626044633641048072116393160e-01,
6.017045491275378948867077135921802620536565639585963293313931e-01,
2.608949526510388292872456675310528324172673101301907739925213e-01,
-2.280913942154826463746325776054637207093787237086425909534822e-01,
-2.8538631755862418545975695028984237217356095588335149922119e-01,
7.465226970810326636763433111878819005865866149731909656365399e-02,
2.123497433062784888090608567059824197077074200878839448416908e-01,
-3.351854190230287868169388418785731506977845075238966819814032e-02,
-1.427856950387365749779602731626112812998497706152428508627562e-01,
2.75843506256286687501473520162198655374474596963423080762818e-02,
8.690675555581223248847645428808443034785208002468192759640352e-02,
-2.650123625012304089901835843676387361075068017686747808171345e-02,
-4.567422627723090805645444214295796017938935732115630050880109e-02,
1.62376740958504713032984257172372354318097067858752542571020e-02,
1.937554988917612764637094354457999814496885095875825546406963e-02,
-1.398838867853514163250401235248662521916813867453095836808366e-02,
-5.86692228101217472658449343605437373814608340808758177372765e-03,
7.040747367105243153014511207400620109401689897665383078229398e-03,
7.689543592575483559749139148673955163477947086039406129546422e-04,
-2.687551800701582003957363855070398636534038920982478290170267e-03,
3.418086534585957765651657290463808135214214848819517257794031e-04,
7.358025205054352070260481905397281875183175792779904858189494e-04,
-2.606761356786280057318315130897522790383939362073563408613547e-04,
-1.246007917341587753449784408901653990317341413341980904757592e-04,
8.711270467219922965416862388191128268412933893282083517729443e-05,
5.105950487073886053049222809934231573687367992106282669389264e-06,
-1.664017629715494454620677719899198630333675608812018108739144e-05,
3.010964316296526339695334454725943632645798938162427168851382e-06,
1.531931476691193063931832381086636031203123032723477463624141e-06,
-6.862755657769142701883554613486732854452740752771392411758418e-07,
1.447088298797844542078219863291615420551673574071367834316167e-08,
4.636937775782604223430857728210948898871748291085962296649320e-08,
-1.116402067035825816390504769142472586464975799284473682246076e-08,
8.666848838997619350323013540782124627289742190273059319122840e-10};
```

```
const double Daub20[40] = {
7.799536136668463215861994818889370970510722039232863880031127e-04,
1.054939462495039832454480973015641498231961468733236691299796e-02,
6.342378045908151497587346582668785136406523315729666353643372e-02,
2.199421135513970450080335972537209392121306761010882209298252e-01,
4.726961853109016963710241465101446230757804141171727845834637e-01,
6.104932389385938201631515660084201906858628924695448898824748e-01,
3.615022987393310629195602665268631744967084723079677894136358e-01,
-1.392120880114838725806970545155530518264944915437808314813582e-01,
-3.267868004340349674031122837905370666716645587480021744425550e-01,
-1.672708830907700757517174997304297054003744303620479394006890e-02,
2.282910568199163229728429126648223086437547237250290835639880e-01,
3.98502464577120219790581076522174181104027576954427684456660e-02,
-1.554587507072679559315307870562464374359996091752285157077477e-01,
-2.471682733861358401587992299169922262915151413349313513685587e-02,
1.022917191744425578861013681016866083888381385233081516583444e-01,
5.632246857307435506953246988215209861566800664402785938591145e-03,
-6.172289962468045973318658334083283558209278762007041823250642e-02,
5.874681811811826491300679742081997167209743446956901841959711e-03,
3.229429953076958175885440860617219117564558605035979601073235e-02,
-8.789324923901561348753650366700695916503030939283830968151332e-03,
-1.381052613715192007819606423860356590496904285724730356602106e-02,
6.721627302259456835336850521405425560520025237915708362002910e-03,
4.420542387045790963058229526673514088808999478115581153468068e-03,
-3.58149425960962277556169638358238375765194248623891034940330e-03,
-8.315621728225569192482585199373230956924484221135739973390038e-04,
1.392559619323136323905254999347967283760544147397530531142397e-03,
-5.349759843997695051759716377213680036185796059087353172073952e-05,
-3.851047486992176060650288501475716463266233035937022303649838e-04,
1.015328897367029050797488785306056522529979267572003990901472e-04,
6.77428082837729558011184406727978221295796652200819839464354e-05,
```

```
-3.710586183394712864227221271216408416958225264980612822617745e-05,  
-4.376143862183996810373095822528607606900620592585762190542483e-06,  
7.241248287673620102843105877497181565468725757387007139555885e-06,  
-1.011994010018886150340475413756849103197395069431085005709201e-06,  
-6.847079597000556894163334787575159759109091330092963990364192e-07,  
2.633924226270001084129057791994367121555769686616747162262697e-07,  
2.0143223550512694324757845944026047904414136633776958392681e-10,  
-1.814843248299695973210605258227024081458531110762083371310917e-08,  
4.056127055551832766099146230616888024627380574113178257963252e-09,  
-2.998836489619319566407767078372705385732460052685621923178375e-10};
```

```
const double Daub21[42] = {  
5.488225098526837086776336675992521426750673054588245523834775e-04,  
7.776639052354783754338787398088799862510779059555623704879234e-03,  
4.924777153817727491399853378340056968104483161598320693657954e-02,  
1.813596254403815156260378722764624190931951510708050516519181e-01,  
4.196879449393627730946850609089266339973601543036294871772653e-01,  
6.015060949350038975629880664020955953066542593896126705346122e-01,  
4.445904519276003403643290994523601016151342743089878478478962e-01,  
-3.572291961725529045922914178005307189036762547143966578066838e-02,  
-3.356640895305295094832978867114363069987575282256098351499731e-01,  
-1.123970715684509813515004981340306901641824212464197973490295e-01,  
2.115645276808723923846781645238468659430862736248896128529373e-01,  
1.152332984396871041993434411681730428103160016594558944687967e-01,  
-1.399404249325472249247758764839776903226503657502071670245304e-01,  
-8.177594298086382887387303634193790542522570670234556157566786e-02,  
9.660039032372422070232189700372539681627783322249829842275517e-02,  
4.572340574922879239251202944731235421034828710753381191345186e-02,  
-6.49775048937323206333231106008616685748929419452249544690967e-02,  
-1.865385920211851534093244412008141266131208093007217139232170e-02,  
3.972683542785044175197464400756126818299918992482587866999707e-02,  
3.357756390338110842532604766376200760791669954106679933144723e-03,  
-2.089205367797907948785235479746212371728219866525211135343707e-02,  
2.403470920805434762380632169785689545910525667396313550679652e-03,  
8.988824381971911875349463398395464114417817949738911101372312e-03,  
-2.891334348588901247375268718015882610844675931117463495551958e-03,  
-2.958374038932831280750770228215510959830170264176955719827510e-03,  
1.716607040630624138494506282569230126333308533535502799235333e-03,  
6.394185005120302146432543767052865436099994387647359452249347e-04,  
-6.906711170821016507268939228893784790518270744313525548714065e-04,  
-3.196406277680437193708834220804640347636984901270948088339102e-05,  
1.936646504165080615323696689856004910579777568504218782029027e-04,  
-3.635520250086338309442855006186370752206331429871136596927137e-05,  
-3.499665984987447953974079490046597240276268044409625722689849e-05,  
1.535482509276049283142233498646050472096482329299719141107128e-05,  
2.790330539814487046106169582691767916283793946025922387556917e-06,  
-3.090017164545699197158555936852697325985864588418167982685400e-06,  
3.16609544236703055660388900983395444005854535577781782000278e-07,  
2.992136630464852794401294607536813682771292352506328096125857e-07,  
-1.000400879030597332045460600516621971679363965166249211063755e-07,  
-2.254014974673301315631848514568259916717915549643308754828159e-09,  
7.058033541231121859020947976903904685464512825731230495144226e-09,  
-1.471954197650365265189549600816698778213247061389470277337173e-09,  
1.038805571023706553035373138760372703492942617518816122570050e-10};
```

```
const double Daub22[44] = {  
3.862632314910982158524358900615460368877852009576899680767316e-04,  
5.721854631334539120809783403484493333555361591386208129183833e-03,  
3.806993723641108494769873046391825574447727068953448390456335e-02,  
1.4836754089011142850114404448710249837385836373168215616427030e-01,  
3.677286834460374788614690818452372827430535649696462720334897e-01,  
5.784327310095244271421181831735444106385099957908657145590104e-01,  
5.079010906221639018391523325390716836568713192498711562711282e-01,  
7.372450118363015165570139016530653113725172412104955350368114e-02,  
-3.12726580428296191803322622621788537078452535993545440716988e-01,  
-2.005684061048870939324361244042200174132905844868237447130382e-01,  
1.640931881067664818606223226286885712554385317412228836705888e-01,  
1.7997318799289130372521424295313083168387840791424988422757762e-01,  
-9.711079840911470969274209179691733251456735137994201552926799e-02,  
-1.317681376866834107513648518146838345477875022352088357523838e-01,  
6.807631439273221556739202147004580559367442550641388181886023e-02,  
8.455737636682607503362813659356786494357635805197410905877078e-02,  
-5.136425429744413245727949984018884707909441768477091944584584e-02,  
-4.653081182750671347875833607846979997825771277976548080904423e-02,  
3.697084662069802057615318892988581825637896696876361343354380e-02,  
2.058670762756536044060249710676656807281671451609632981487139e-02,  
-2.348000134449318868560142854519364987363882333754753819791381e-02,  
-6.213782849364658499069336123807608293122900450508440420104462e-03,  
1.256472521834337406887017835495604463815382993214296088172221e-02,  
3.001373985076435951229129255588255746904937042979316054485183e-04,  
-5.455691986156717076595353163071679107868762395367234726592273e-03,  
1.044260739186025323350755659184734060807432172611689413745029e-03,  
1.827010495657279080112597436850157110235336772062961041154607e-03,
```

```
-7.706909881231196232880372722955519781655769913634565757339739e-04,  
-4.237873998391800799531947768003976978197438302533528661825758e-04,  
3.286094142136787341983758471405935405823323072829619248523697e-04,  
4.345899904532003379046992625575076092823809665933575578710696e-05,  
-9.405223634815760421845190098352673647881298980040512091599943e-05,  
1.137434966212593172736144274866639210339820203135670505287250e-05,  
1.737375695756189356163565074505405906859746605867772002320509e-05,  
-6.1667293164675783721522551668422979152169587307212708981768966e-06,  
-1.565179131995160159307426993578204733378112742579926503832095e-06,  
1.295182057318877573889711232345068147800395721925682566394936e-06,  
-8.779879873361286276888117046153049053917243760475816789226764e-08,  
-1.283336228751754417819693932114064887075096030264748079976736e-07,  
3.761228749337362366156711648187743399164239397803629022612862e-08,  
1.680171404922988885554331183691280245962290247654438114807112e-09,  
-2.729623146632976083449327361739104754443221903317745768938846e-09,  
5.335938821667489905169783227036804533253011117886586305435615e-10,  
-3.602113484339554703794807810939301847299106970237814334104274e-11};
```

```
const double Daub23[46] = {  
2.719041941282888414192673609703302357098336003920923958924757e-04,  
4.202748893183833538390034372523511472345215563611003407984701e-03,  
2.931000365788411514736204018929480427874317460676079959515131e-02,  
1.205155317839719336306053895611899089004274336891709067958035e-01,  
3.184508138528652363416527748460472152790575031409830417259640e-01,  
5.449311478735204282674240672421984387504149924834544495466793e-01,  
5.510185172419193913452724227212507720514144116478727269717859e-01,  
1.813926253638400136259098302138614937264260737638175539416540e-01,  
-2.613921480306441118856795735210118413900307577511142987337375e-01,  
-2.714020986078430556604069575184718123763697177381058877113471e-01,  
9.212540708241805260646030910734894258577648089100630012130261e-02,  
2.235736582420402317149513960822561717689875252792817094811874e-01,  
-3.303744709428937875006612792463031409461636228731285046551636e-02,  
-1.640113215318759250156057837165276039181451149292112929401186e-01,  
2.028307457564929974897286607551313323418860610791382310375731e-02,  
1.122970436181072886950734465075645977754665593869789965874572e-01,  
-2.112621235622724100704783293549467048999443844657058425212982e-02,  
-7.020739157490110946204219011957565343899895499962369353294028e-02,  
2.176585683449997560776882472168730165799461445156766923497545e-02,  
3.849533252256919901057154320407596073180564628069920893870768e-02,  
-1.852351365015615979794689960740674782817814176166333519597796e-02,  
-1.753710100303584537915846117408613551147985251726558719415169e-02,  
1.275194393152828646243157404474947115052750581861997731041018e-02,  
6.031840650024162816289878206037841640814102314209075233751820e-03,  
-7.075319273706152814194039481466556204493276773483821748740018e-03,  
-1.134865473356251691289337120013286756337393784110786907825400e-03,  
3.122876444981814499741914765125750522437659393621577492535411e-03,  
-2.465014005163512031940473100375377210862560761576109755841161e-04,  
-1.061231228886651321139357625683805642193648671030425010215075e-03,  
3.1942404927099011503676530359692366990929679170022583007683112e-04,  
2.567624520078737205563856675376636092314813400664190770435450e-04,  
-1.500218503490340967673163290447832236259277810659068637402668e-04,  
-3.378894834120903434270962452674534330903724108906662510305045e-05,  
4.42607120310924607762187530344093535701832843654692827539837e-05,  
-2.635207889249186237209225933170897825432335273771458456888097e-06,  
-8.347875567854625544366043748844183086765894974439245409223337e-06,  
2.397569546840240057403739507525641239509517148980849889986407e-06,  
8.147574834779447778085443041422881439860288287528356019216814e-07,  
-5.339005405209421154584783682848780965053642859373536945701365e-07,  
1.853091785633965019353699857864654181728710556702529908304185e-08,  
5.417549179539278736503176166323685597634496102979977037271945e-08,  
-1.399935495437998845130909687361847103274208993447892120341999e-08,  
-9.472885901812050535221582074673490573092096712822067564903012e-10,  
1.050446453696543404071105111096438573423068913105255997908040e-09,  
-1.932405111313417542192651899622541612314066389643607507706887e-10,  
1.250203302351040941433216718217504240541423430995137507404787e-11};
```

```
const double Daub24[48] = {  
1.914358009475513695026138336474115599435172088053846745168462e-04,  
3.082081714905494436206199424544404720984720556128685270556458e-03,  
2.248233994971641072358415157184825628226776692231940577581580e-02,  
9.726223583362519663806545734008355914527504417674578571164300e-02,  
2.729089160677263268706137134412557268751671263458895098625356e-01,  
5.043710408399249919771876890402814109246866444441814540282099e-01,  
5.749392210955419968460807901923407033144945935105622912839838e-01,  
2.80985532337118833442626085115402941842959475929278883281409e-01,  
-1.872714068851562376981887159775791469060265778441667840307934e-01,  
-3.179430789993627375453948489797707550898087789160025182664299e-01,  
4.776613684344728187950198323031360866349104994035553200788631e-03,  
2.392373887803108551973268291945824822214858134512317715815616e-01,  
4.252872964148383258147364472170645232684343235486951540533893e-02,  
-1.711753513703468896897638515080572393949165942335556397917666e-01,  
-3.877717357792001620177594726199572688446488033750771020190283e-02,  
1.210163034692242362312637311149062286659377039046006801523826e-01,
```

2.098011370914481534980883827326017063121637262728447783605518e-02,
-8.216165420800166702291466006164189460916816748629968198028898e-02,
-4.5784362418192216339797516339765068825260159169893967894877272e-03,
5.130162003998087915555334881398688958843078494595140394873884e-02,
-4.944709428125628299815920032649550811877887219282751174798211e-03,
-2.821310709490189098113895361900699228886900995412759197674058e-02,
7.66172188164658589732989940308764405384658404613669817843430e-03,
1.30499708710857383052494067883717533043101857128653233783396e-02,
-6.291435370018187780721843581169343900864298634085743861509767e-03,
-4.746568786323113800477796959513558401732252800905982385017245e-03,
3.736046178282523345179052160810332868725126356493155728625572e-03,
1.153764936839481504858282495202271984454410046682805375157566e-03,
-1.696456818974824394274534636412116243080312601322325642741589e-03,
-4.416184856141520063365958900079406737636243682138363561877750e-05,
5.861270593183109933716735450272894035425792347806515678695765e-04,
-1.181233237969554740613021227756568966806892308457221016257961e-04,
-1.460079817762616838924301818082729036314539476811023255670666e-04,
6.559388639305634085303738560455061974369354538271316071502698e-05,
2.183241460466558363365044032984257709791187640963509380549307e-05,
-2.022888292612697682860859987200455702614855595412267510558659e-05,
1.341157750809114719319937553186023660581084151828593222893663e-08,
3.901100338597702610409014129024223853127911530009766793352492e-06,
-8.980253143938407724149926669980791166378388013293887718404796e-07,
-4.032507756879971624098983247358983425236092110387724315244646e-07,
2.166339653278574639176393978510246335478946697396400359281412e-07,
-5.057645419792500308492508924343248979317507866520688417567606e-10,
-2.255740388176086107368821674947175804005323153443170526520277e-08,
5.157776789671999638950774266313208715015419699643333784626363e-09,
4.748375824256231118094453549799175824526559994333227456737433e-10,
-4.024658644584379774251499574468195118601698713554294941756559e-10,
6.991801157638230974132696433509625934021677793453732225542951e-11,
-4.342782503803710247259037552886749457951053124203814185811297e-12};

```
const double Daub25[50] = {  
1.348029793470188994578489247159356055370460656508881471268611e-04,  
2.256959591854779520121391049628056149270016860666661928130747e-03,  
1.718674125404015533817186914954848902241194002444696221013131e-02,  
7.803586287213267559750659320481403668422052199257139168386084e-02,  
2.316935078860218199900621518057089104946216881512075361624214e-01,  
4.596834151460945937896973864539659944010260858049947396093277e-01,  
5.816368967460577833534892038757085635755639698734580573323031e-01,  
3.678850748029466984371319740855532278670733841012809062966976e-01,  
-9.717464096463814276130048169040892607068486428294030952842447e-02,  
-3.364730796417461309562110148848845218930261030262170601615289e-01,  
-8.758761458765466140226687673880006154266689569025041229545538e-02,  
2.245378197451017129525176510409543155930843160711989062118482e-01,  
1.181552867199598604563067876819931882639429216001523151773895e-01,  
-1.505602137505796309518094206831433270850173484773520730186277e-01,  
-9.850861528996022153725952822686729410420350758543226219234795e-02,  
1.066338050184779528831274540522414711301747903916268438037723e-01,  
6.6752164449401860666895983072443984697329752470942906490126865e-02,  
-7.708411105657419356208567671699032054872853174701595359329826e-02,  
-3.717396286112250887598137324046870459877639250821705817221557e-02,  
5.361790939877949960629041419546536897037332284703545849594129e-02,  
1.554260592910229163981295854603203625062268043511894295387375e-02,  
-3.404232046065334099320628584033729153497903561399447916116575e-02,  
-3.079836794847036661636693963570288706232460663070983852354326e-03,  
1.892280447662762841086581178691039363674755753459524525886478e-02,  
-1.989425782202736944289461896386235348901617760816745484282494e-03,  
-8.860702618046368399013064252456556969199612331833605310278698e-03,  
2.726936258738495739871469244610042793734119359765762028996059e-03,  
3.322707773973191780118197357194829286271392998979276105842863e-03,  
-1.842484290203331280837780430014195744813667655929909114672154e-03,  
-8.999774237462950491085382524008429604309720852269895692000702e-04,  
8.77258193674827484388806190175921376284150686011179612908221e-04,  
1.153212440466300456460181455345639872216326644527860903202733e-04,  
-3.098800990984697989530544245356271119416614147098459162436317e-04,  
3.543714523276059005284289830559259809540337561365927850248007e-05,  
7.90464000396552825513749630316600173546310776264630487560e-05,  
-2.733048119960041746353244004225286857636045649642652816856524e-05,  
-1.277195293199783804144903848434605690990373526086311486716394e-05,  
8.990661393062588905369930197413951232059323587543226269327396e-06,  
5.232827708153076417963912065899772684403904504491727061662335e-07,  
-1.779201332653634562565948556039009149458987774189389221295909e-06,  
3.212037518862519094895005816661093988294166712919881121802831e-07,  
1.922806790142371601278104244711267420759978799176017569693322e-07,  
-8.656941732278507163388031517930974947984281611717187862530250e-08,  
-2.61159855611177086425984308915178220692284262717427427471722e-09,  
9.279224480081372372250073354726511359667401736947170444723772e-09,  
-1.880415755062155537197782595740975189878162661203102565611681e-09,  
-2.228474910228168899314793352064795957306403503495743572518755e-10,  
1.535901570162657197021927739530721955859277615795931442682785e-10,  
-2.527625163465644811048864286169758128142169484216932624854015e-11,
```

1.509692082823910867903367712096001664979004526477422347957324e-12};

```
const double Daub26[52] = {
  9.493795750710592117802731381148054398461637804818126397577999e-05,
  1.650520233532988247022384885622071050555268137055829216839523e-03,
  1.309755429255850082057770240106799154079932963479202407364818e-02,
  6.227474402514960484193581705107415937690538641013309745983962e-02,
  1.950394387167700994245891508369324694703820522489789125908612e-01,
  4.13292962278356368611610868666547082846741228042232731476147e-01,
  5.736690430342222603195557147853022060758392664086633396520345e-01,
  4.3915831178916623219314775565794105633815363384084590559889493e-01,
  1.774076780986685727823533562031556893226571319881417676492595e-03,
  -3.26384593691780021638534083005349953447745005769416287177497e-01,
  -1.748399612893925042664835683606584215248582345438816346170042e-01,
  1.812918323111226960705459766025430918716233584167982942044424e-01,
  1.827554095896723746537533832033286839689931924709760567945595e-01,
  -1.043239002859270439148009137202747658420968144330108510179290e-01,
  -1.479771932752544935782314546369458188243947772922980064071205e-01,
  6.982318611329236513756591683950208955110603212379412334701145e-02,
  1.064824052498086303236593797715344405836015002929319291715777e-01,
  -5.344856168148319149493577269390074213960237013099439431132086e-02,
  -6.85475960403591525454725258715351280947435823354011140858001e-02,
  4.223218579637203541206570902753288247790857760067894456114927e-02,
  3.853571597111186425832144567362328142994885395255438867968781e-02,
  -3.137811036306775484244644776337594435094096964336402798072360e-02,
  -1.776090356835818354094298625884058170354129044259951019182732e-02,
  2.073492017996382475887790073068984224515077665517103399898854e-02,
  5.829580555318887971939315747596613038479561943085291072787359e-03,
  -1.178549790619302893728624468402138072504226527540325463847390e-02,
  -5.287383992626814439198630765217969804966319971038003993984480e-04,
  5.601947239423804853206514239940474788977188460452053462770324e-03,
  -9.390582504738289646165698675070641765810790863514339205205998e-04,
  -2.145530281567620980305401403432221668847980295600748913748902e-03,
  8.383488056543616046381924054554052104937784379435436426690560e-04,
  6.161382204574344193703789012696411561214682388271673214197731e-04,
  -4.319557074261807466712901913481943478521991611607433971794602e-04,
  -1.060574748283803889966150803551837402553866816191659959347053e-04,
  1.574795238607493590547765666590811258087715699737771458390360e-04,
  -5.277795493037868976293566636015627609248847457646525246271036e-06,
  -4.109673996391477816326502438997466532822639385119090230965252e-05,
  1.0742219540872195031273584409245060623104931330938723936484593e-05,
  7.000078682964986734859102495210684809643657474253921074934684e-06,
  -3.887400161856795187587790410706550576033603097954065074023128e-06,
  -4.650463220640262639231145944536092973446596027469833860001618e-07,
  7.939210633709952088373459255067360793370284788682979065122810e-07,
  -1.079004237578671411922961583845716126060658213943840375162654e-07,
  -8.904466370168590769052983362721567202750591914741016835071257e-08,
  3.407795621290730008673832107214820587991557116806912418558069e-08,
  2.169328259850323106986222296525930099935873861026310788086221e-09,
  -3.776010478532324328184043667556576385639846460337894963138621e-09,
  6.78004724582863668305808192607091517605349478677442468580825e-10,
  1.002303191046526913509281844136258004034177309673269533418644e-10,
  -5.840408185341171468465492447799819262905317576847426970757700e-11,
  9.130510016371796243923232926650252570239054815939483900056681e-12,
  -5.251871224244435037810503452564279828539007071678724285717464e-13};
```

```
const double Daub27[54] = {
  6.687131385431931734918880680779563307675740731544063787599480e-05,
  1.205531231673213234251999812212394463872002561229330125152073e-03,
  9.952588780876619771874091297340545740163119816300838847749336e-03,
  4.945259998290488004302995584228917712171023349013386944893643e-02,
  1.629220275023933206396286389387812803673796872000118325233533e-01,
  3.671102141253898226423388094379126394383458407087000700420400e-01,
  5.538498609904800487605460395549044755068663194750017660900436e-01,
  4.934061226779989979265447084358038959373468583404767251300717e-01,
  1.028408550618229112710739475157388764479351682549490307668477e-01,
  -2.897168033145948463175311101489473923261698802610323264603418e-01,
  -2.482645819032605667810198368127693701263349361209208170092197e-01,
  1.148230195177853576326445213787661879970642975306605349249036e-01,
  2.2727328841417108265275037216925482827043581894357907763081103e-01,
  -3.878641863180231062443346843661817078060143110529946543683356e-02,
  -1.780317409590085821070366277249759321269342801053489323888575e-01,
  1.579939746024048431173907799261019471878724997312653292884660e-02,
  1.311979717171553289711406975836688896451835867594492827800969e-01,
  -1.406275155580876537026622167053147161846397735962817855782362e-02,
  -9.102290652956591798241345515773322449830692586525337562864481e-02,
  1.731101826549371089085675445961947677452358872325373949295769e-02,
  5.796940573471798814748840657698008349462526768238833307489106e-02,
  -1.851249356199807710545837861298826718763077900221574749342712e-02,
  -3.273906663102087145481936428049519742538150452785563039743756e-02,
  1.614696692239566682272152627542980896527822528487665111124260e-02,
  1.566559564892457873003263983940819950829497022298967052103291e-02,
  -1.157718645897628140054089958116866381056430680879332334217267e-02,
```

-5.862096345462925972966025215266179082657169806555503857975278e-03,
6.856635609684880675273182184141746359000591385833807880272568038e-03,
1.342626877303679609082208800217479591902967766971379107017011e-03,
-3.332854469520006162763300141047111065412307706449049389557931e-03,
1.457529625931728587128588244152604734177322144376309490881599e-04,
1.301177450244135139135787970279897042994109161268159963884641e-03,
-3.4183512269154276119465144737228006377896519777431057005796358e-04,
-3.879018574101327604369144470124819695479087900682219330965466e-04,
2.019719879690326857104208791272390315160018069955787875123234e-04,
7.660058387068576876674274961751262847965101108848090019821555e-05,
-7.711145517797584208411720507329584053382646435270054267102827e-05,
-3.517483614907445391752737841583832374184046409747387149129674e-06,
2.063442647736885318487206413360228908558806028468062177953960e-05,
-3.901164070638425528170558032557368703418425915665413541985623e-06,
-3.657500908187104997045760131046655906827644494899206692043298e-06,
1.634369624725637835424610743915128591988676092276368687669255e-06,
3.050880686251999094242671997731089918322345713516567387655763e-07,
-3.4724681473943892693264673179891460601330730511237974736379548e-07,
3.286558968055159530983261866450459360074591641809187825408848e-08,
4.026255052866908637178682747490340533992340623231336911661711e-08,
-1.321332273990056558848617809101876846857728483295631388083263e-08,
-1.309465606856955151282041809232358209226373823424148862843577e-09,
1.521614984778521740775073159445241799352681846880808663329946e-09,
-2.415526928011130660506395791946234018673860470542996426005750e-10,
-4.374986224293654395069947682013996351823060759948583134078918e-11,
2.213662088067662485181472969374945928903854605356443772873438e-11,
-3.2957901224765858076069953975043096139541415768606924980926275e-12,
1.828188352882424933624530026056448539377272017834175009418822e-13};

const double Daub28[56] = {
4.710807775014051101066545468288837625869263629358873937759173e-05,
8.794985159843870273564636742144073059158975665525081816488582e-04,
7.542650377646859177160195786201116927568410621050693986450538e-03,
3.909260811540534426092083794403768111329778710541126982205076e-02,
1.351379142536410450770749411679708279921694061092200363031937e-01,
3.225633612855224257318486139030596702170126503618082416187649e-01,
5.249982316303355562348293243640252929543774162151269406404636e-01,
5.305162934414858075256978195354516449402692654391295761050628e-01,
2.001761440459844380384404537971725815970574972480152145882083e-01,
-2.304989540475825257279397658067038304888129374484095837624889e-01,
-3.013278095326417816909366061441334075444383937588485826752087e-01,
3.285787916338710468450547883547348694255260871071954509422161e-02,
2.458081513737595535752949960866466132239832334168533456626848e-01,
3.690688531571127205290633425993077868843846977265847006108551e-02,
-1.828773307329849166920408764650763092868965221608724574218473e-01,
-4.683823374455167616514752420549419665215987106243491879971921e-02,
1.346275679102260877490923315484152662987698625205479167761416e-01,
3.447863127509970524678534595639646616244376966117385829345554e-02,
-9.768535580565244174963692133038973587005628990493154911133358e-02,
-1.734192283130589908795581592406238282930530566316914040035812e-02,
6.774789550190933956165341752699717255041141690153626336867769e-02,
3.448018955540951137600471926079622335842207388713342609755316e-03,
-4.33333686160862839386325498082828440376630920345380866688800e-02,
4.431732910062988320487418656322338284504389482966303454010563e-03,
2.468806001015186586264188361362046240243934625858343309818244e-02,
-6.815549764552309639259447104811254179605050667281644254737890e-03,
-1.206359196821849005842466619530619474644989878503490321948471e-02,
5.838816627748944864497370576838809711476027837762897602935327e-03,
4.784863112454241718009916669120329848973107781600157214960003e-03,
-3.725461247074254799171427871442937099025589672466088044410521e-03,
-1.360373845639692436577650137133777929659265166644839235882291e-03,
1.875998668202795626152766912508562385106168761893900192731562e-03,
1.415672393140464257573780581396205840941849282748250523509874e-04,
-7.486749559114629991320679819683227355746847370960399216568306e-04,
1.154656063658921251969297916771881248142872975490882572741198e-04,
2.295790982233456202366621544054366855729175050420515776344878e-04,
-8.903901490044488099517361247378396756893227855233897357882978e-05,
-4.907713416190250858324783990436748073854807494400738311968278e-05,
3.641401211050802781223450761733180188911730291497201507086247e-05,
4.638664981394294654002871426476885751050837817671843706915388e-06,
-1.004326041333422601781848560432120920634648692782357855473103e-05,
1.247900317574834146052381692752796047052443265982232422642017e-06,
1.840363734517769191684379309039277810350620305330900536404818e-06,
-6.670215479954892588747450458085225880096882699397256774967304e-07,
-1.757461173209842779903676264971918635870906983281392939812547e-07,
1.490660013535362170989340065033061951960933954388633507264360e-07,
-8.262387315626556965966429243600984899650039704831080988658278e-09,
-1.784138690875710077191713941441263246560738410213624546116655e-08,
5.04404705638343644463125284005786200226408772067808580373667e-09,
6.944540328946226952976704718677697525410051405055662575530111e-10,
-6.077041247229010224760245305596307803830053533836849384680534e-10,
8.492220011056382105461206077240377024404404638947591299761197e-11,
1.867367263783390418963879146175452376940453585791428841004699e-11,

-8.365490471258800799349289794397908900767054085216008197372193e-12,
1.188850533405901520842321749021089497203940688882364518455403e-12,
-6.367772354714857335632692092267254266368934590973693820942617e-14};

```
const double Daub29[58] = {  
3.318966279841524761813546359818075441349169975922439988843475e-05,  
6.409516803044434540833706729120596322083061716935004987374676e-04,  
5.70212651777337543476084398623507494914551464968126455168657e-03,  
3.077358022140837676716707336516751814713312018344719150923618e-02,  
1.113701169517405304762186166370327770191325772342190715118617e-01,  
2.80653455970982937696888126277048060650092398534229615289e-01,  
4.897588047621993143592705932993573539235839610055331620240518e-01,  
5.513744327583751951223746071670135992466984391233429663886536e-01,  
2.891052383358291634605691113586264061513180158354460952469246e-01,  
-1.540287344599000542466293779503370141731339982919280951230240e-01,  
-3.300409489175880520295083779487012611959310539629627124613719e-01,  
-5.570680007294085781514541931715795784309410235726214400350351e-02,  
2.361052361530259415983110734054626770649468357328362426830433e-01,  
1.124191748731883764769740670535880543076817816861518667898467e-01,  
-1.608779885941877360771615465531852333085159940159968393590303e-01,  
-1.078459499387214201077881957354707913786241153934264316589273e-01,  
1.144722958938182579734135930060053286267822797640393386903440e-01,  
8.322074716244975790297348835032537357891920536002627784941129e-02,  
-8.512549261563550232832311331420804581881235448862834507281486e-02,  
-5.502748952532572320924541450626650067707344725344841099873446e-02,  
6.347916458421186633577789314698972361081611994794140119302163e-02,  
3.053154327270413646637328212093941030592133225231728964047047e-02,  
-4.518798127778834515979704475304405691390090327474972089790857e-02,  
-1.291714255426679462966473962555410660387671182428076570686472e-02,  
2.947043187174764111028122319949903667638786379520519899154373e-02,  
2.648327307678167915542397563479749119673768286990136051577167e-03,  
-1.704122457360668969234196743407615179099529206118693044741086e-02,  
1.737880332720511164430027824345354801611373419264590068097416e-03,  
8.469725493560752287772961661104710791306496373354237126998903e-03,  
-2.550807127789472659145072247724735637183590942511858255354005e-03,  
-3.473798989681100630649790255076233970957721666820195620598374e-03,  
1.877120925723650133179338154344873477230567340668548016358682e-03,  
1.087053942226062966738944397844498417945523630053411148182206e-03,  
-1.000778327085680541055696707760062870925897014530348262794137e-03,  
-2.000711363076779808296301110796026470163110202848894744316755e-04,  
4.111283454742767033424740543004041500054889660665367490129376e-04,  
-2.292018041214499897382298271438084577065170236103859181134525e-05,  
-1.293044840080720609161466939678226852440475312744714379499074e-04,  
3.645026068562774967665464216602750761690984830805534178557146e-05,  
2.913344750169041218495787251929571015775436967652945386217480e-05,  
-1.657328395306616289863396387854880512976861409870690029695161e-05,  
-3.593644804025187638066915189731950450034629392522542962477168e-06,  
4.750609246452552850197117564759363194953518317428400241629683e-06,  
-3.0290545920528182864742228294307141792053791695855058563299597e-07,  
-8.975701750636280734511651941681818767895052287332471537510510e-07,  
2.633898386997696553900967704111473475368019612368922599394214e-07,  
9.387197411095863026484410601284876812292554863800653292318725e-08,  
-6.286156922010786166768503252870590953166867739448102804392389e-08,  
1.076591906619196137385201975028785139607670319821266803566785e-09,  
7.768978854770062238895964639391324551611701293594055935346266e-09,  
-1.893995386171984147774611076618946011337498790609031626697228e-09,  
-3.426800863263089001811012278889864200550342566386405676893537e-10,  
2.407099453509342962399811991929330725186626582891090462239366e-10,  
-2.9405892507645325828884739746382736644244682541297835986306504e-11,  
-7.8325097336278170323256556582819494794884131433810848844709881e-12,  
3.152762413370310423797539876893861621418382024668704492620948e-12,  
-4.285654870068344101898185073376307686875386259541180967347399e-13,  
2.219191311588302960934661700068023727737812918006011019184982e-14};
```

```
const double Daub30[60] = {  
2.338616172731421471474407279894891960011661146356580425400538e-05,  
4.666379504285509336662000111055365140848987563882199035322085e-04,  
4.300797165048069510045016757402827408493482974782286966500398e-03,  
2.413083267158837895194919987958311943976725005113561262334092e-02,  
9.123830406701570679321575555085899708564500191080751595642650e-02,  
2.420206709402140994467599658342919512318194032687898436229538e-01,  
4.504878218533178366981351802898336415314944375740699506554771e-01,  
5.575722329128364304078082520999850413492571645754785374629734e-01,  
3.662426833716279793144871151369089533016299234992584741629624e-01,  
-6.618367077593731501909741041813726474911212544474895441395148e-02,  
-3.329669750208556069196849320598850505877494561268613506392514e-01,  
-1.419685133300829310219026267403758254954270602825020111483505e-01,  
1.994621215806643032428990062111230223523226088131364328774921e-01,  
1.778298732448367361282050921330425046260289700971176750362566e-01,  
-1.145582194327077814891518778613672243404957549114393749173137e-01,  
-1.572368179599938126878197378886501553251711910617673398124611e-01,  
7.277865897036442699893544326605244235248713804556715604416632e-02,  
1.227477460450093778691578797698150091624353365248212907325446e-01,
```


-5.380646545825707676022015051837304300338645984615639237930800e-02,
-8.765869003638366048026572053699028353846982304851342479893827e-02,
4.380166467141773250305407710250135373016604593736480428415303e-02,
5.671236574473569492590636983030617493807140224924978946302257e-02,
-3.567339749675960965780819743176056734137251336781389369397564e-02,
-3.226375891935220815954913483392725682165778426411705216010280e-02,
2.707861959529418272206848318420006522973840949600186710327776e-02,
1.528796076985739546052896626042375110302102640936712142026221e-02,
-1.839974386811734118728169880549148389603890445324127330811811e-02,
-5.296859666131086629169938675330494864053932988161015674773617e-03,
1.091563165830488927536881480211929049886878831313700460017968e-02,
6.1967175649724438359253499284255315694546230739551683085460e-04,
-5.530730148192003288871383856487027893918513053091795443517653e-03,
8.433845866620933982126003584365932145598126087481400294999080e-04,
2.324520094060099304385756339638431339131122661576649123053845e-03,
-8.609276968110423879660725173525347077801305237644122054954659e-04,
-7.678782504380918697963922441514742758516706160788123977340073e-04,
5.050948239033467796256544554086554367969638627715114003635557e-04,
1.724825842351709725545759714374272164367933578194910678479473e-04,
-2.161718301169633804271038862087964094429005266172702380483361e-04,
-8.548305467584070994787824796256108217987765582429940610377190e-06,
6.982008370808327851082027193100914402221658444151889697045071e-05,
-1.339716863293971629296314599448901465078920406443516550195793e-05,
-1.636152478725426488654528710478856195004608401773950511915162e-05,
7.25214553589046901572340116993432790062289413069550273452916e-06,
2.327549098493686509557358103785598216688723737824121617676858e-06,
-2.187267676996166416699555236143059249832615777542412142603694e-06,
1.099474338526203304286307383463498542376432972308342428764576e-08,
4.261662326011572446469849114416378817419458434583398455985144e-07,
-1.000414682354500898864979332965559934104686157639553850670490e-07,
-4.764379965139453357729154748688006975561934425368712852985388e-08,
2.605442754977625431940885841950955928085338672381046225838880e-08,
5.553397861397053982967618072672572206490972606026556946910028e-10,
-3.331105680467578245901976412732595596538702049437802824373020e-09,
6.984862691832182584221096665570313611280449991512869846064780e-10,
1.613622978270904360610418704685783656905979134344922647926295e-10,
-9.461387997276802120884525814092001871993910062127702293573920e-11,
1.000105131393171192746337860330428369495110180346654025287492e-11,
3.239428638532286114355931428908079297696045600279108835760520e-12,
-1.185237592101582328254231496310584611948560976394420324137742e-12,
1.54399757084762004603616417646988780670333040868954794039905e-13,
-7.737942630954405708679963277418806436871098329050829841696327e-15};

const double Daub31[62] = {
1.648013386456140748122177817418358316441195236228590958603489e-05,
3.394122037769956699157160165352942212213928231154233571163033e-04,
3.236884068627721221829662672296912258338131668810067169630813e-03,
1.885369161298591269159568944275763468999829139547989648553486e-02,
7.433609301164788697908776495388047669378919816041031344650271e-02,
2.070128744852353286198055444111916450619762837756134323019573e-01,
4.091922000374278563928213235836188963704298775635493549519369e-01,
5.511398409142754983590484577074663132074992263886810324421617e-01,
4.294688082061372955430413148799008354573408538414331312236645e-01,
2.716921249736946422305354732634261873401679092095992827198308e-02,
-3.109551183195075186926560285811004715398678229333522634202008e-01,
-2.179784855235633521693544507220105631639547435903112747133934e-01,
1.401782887652732681656253206993073895422881511380152633441096e-01,
2.249667114737370933697297905066886078307490136415302624018330e-01,
-4.99263491604682397700579399730138693074543903234092797936484e-02,
-1.869623608957154494374577196258383009208655076187653847079167e-01,
1.543698842948893409652995335281236231845293548571166883219023e-02,
1.450895009319931981518942907854879059128872873116921504156674e-01,
-8.139832273469236863527708715566588550006680549152344840146851e-03,
-1.076127733234956326668605511648013952380301953590447106075614e-01,
1.094129745236496925725237900637802669504835743555466811796369e-02,
7.535361174328140695528289751109133941376701984419452638686226e-02,
-1.488002661810482202699555987503429289100801979910046913257306e-02,
-4.861907546485433003537603385831190109391263542044516048871113e-02,
1.615417156598591113619453864586701665635869166193865651960591e-02,
2.804761936675616906861927211659154977049392281479113764697785e-02,
-1.427627527776351943309800140756746087215016194775579070599004e-02,
-1.390055293926652880755898888934447671732373519028670201124816e-02,
1.051763948737184089128633441244991643331033825102031908858652e-02,
5.516163573310992566561289762241160214476622662764637181816550e-03,
-6.520852375874612553325469682628530079210293774541131381751695e-03,
-1.428264223218909891400516038687842292177211292295049238921068e-03,
3.393066776715931928419358796960612411097347419792355896915546e-03,
-6.397901106014600492881202314307290077992972755016494062875201e-05,
-1.459041741985160943114515221598080223845239255190055621901681e-03,
3.431398296904734438118401084929505912208229684629857530009147e-04,
4.998816175637222614896912406679513231966722440032799024979502e-04,
-2.396583469402949615285646688069476140260781708006174912535660e-04,
-1.243411617250228669409179807383399199879641177993453588807726e-04,

1.089584350416766882738651833752634206358441308880869184416670e-04,
1.501335727444532997071651937630983442758297688087711521441229e-05,
-3.631255157860086164261313773172162991107348698083164489165837e-05,
4.034520235184278839752741499546098778993926344831736074409765e-06,
8.795301342692987765440618030678349427367022581211855857458220e-06,
-3.035142365891509630069007852947057220760887215249503512783023e-06,
-1.3690602309429407820504897519871239550744404782177163471279285e-06,
9.810015422044371573950976088058064384946146188110905321673802e-07,
5.327250656974915426977440959783080593776012130063170688309127e-08,
-1.975925129170206248152121156696590501303803187231928513867046e-07,
3.616826517331004805247567218405798591329788122337274956172315e-08,
2.32830971382140964430853888589329921141948539678106680777082e-08,
-1.061529602150252306500404266150823962402673780484965538270541e-08,
-6.474311687959861398702581539341954438747926255671605657095807e-10,
1.408568151025177427076547804944585301332087108125727813194374e-09,
-2.524043954153353306183643702933218308617979467184848456565837e-10,
-7.348930032486263904766913919653624379586487437915175106407348e-11,
3.692108808871129411604189196259677640440919369478263728899602e-11,
-3.327008967125979929910636246337150851642079794871116041187279e-12,
-1.324334917243963163878274345609465717294426628053460151843705e-12,
4.445467096291932163298411852093011459626037560439178917611592e-13,
-5.559442050579014337641375730083534521513818164827556763756543e-14,
2.699382879762665647295493928801387173921314576598505507855504e-15};

const double Daub32[64] = {
1.161463302135014885567464100760659332951431420121048996305591e-05,
2.466566906380903352739104211274667134470169443886449124673996e-04,
2.431261919572266100780423071905958127811969678055971488060574e-03,
1.468104638141913563547809006402194831107662001343421893488086e-02,
6.025749912033537081745451975527967031851677384078997261920024e-02,
1.757507836394388988189299915753348505208376399651864661397588e-01,
3.675096285973496361995340339143234125206079560406868595968025e-01,
5.343179193409538322901117858552186425529774700290587495921679e-01,
4.778091637339484033555130814414794130354053753675509287934741e-01,
1.206305382656178269538098710665261299391507308342013788891222e-01,
-2.666981814766755535489784087869865024226542605534080371507405e-01,
-2.774215815584272153338153320303401666681294506143291967655666e-01,
6.471335480551623831000090095167664918448659157720155321560811e-02,
2.483106423568801736064852157222867588791898170114101300999760e-01,
2.466244483969740441701479334808723214802614938081258920635302e-02,
-1.921023447085468984341365278247990525863123891147783426068990e-01,
-4.899511718467173853355943225576377418394280156945986899417475e-02,
1.452320794752866460838830744051944832326998342053148426312341e-01,
4.440490819993974022640619534046603571086531544468421519143629e-02,
-1.094561131160893831027722774343269232755171130623890041619420e-01,
-2.962787250844770491204452379051215505049068645551070779367843e-02,
8.087414063848395744090831590426327690818854671836423275412813e-02,
1.410615151610660772869738802931740150275269382463799031013905e-02,
-5.692631406247843550478416271158537960555270097953330567652364e-02,
-2.380264464932573834443178362086503847328134994591954135879789e-03,
3.705145792354468010437633458013030898015496905609424004450953e-02,
-4.145907660827218781460700428862611061267328108653649653634276e-03,
-2.166282283639119347634778516947485598599029367518033869601702e-02,
6.167527310685675112579058689520105004744367282412921739811164e-03,
1.101740071540688116532806119564345712473051769079712407908648e-02,
-5.411568257275791208581502410752383050600045942275647685361370e-03,
-4.649216751184411528658094984504900172989190128905887602541396e-03,
3.627224640687864960122122984391704782343548385375321260251988e-03,
1.468955100468467772528811782840480639166582822577191079260543e-03,
-1.964740555821778254183647540656746450092725858126595984907304e-03,
-2.211678729579097916278097586914956834196749138610403102772710e-04,
8.673058518450555343925662389563539890596549655683386287799624e-04,
-1.024537310607396186949656796812972062290796122915930356634122e-04,
-3.059654423826911750479261161552574500739091332121504634422577e-04,
1.053915461739828114700905192091104141076083602686374410146603e-04,
8.103678329134838389828091896334156224227821362491626044950428e-05,
-5.259809282684322782648914338377962890245975842272425408122506e-05,
-1.294045779405512723950480259110995722517019870286295908085366e-05,
1.824268401980691220603850117995712615809177092802967489081228e-05,
-6.361781532260254953363913076575914206506177493714496098327288e-07,
-4.558309576264423135123964145585288808181431652781253437738445e-06,
1.202889036321620990296134494079846952404216422923750605507047e-06,
7.560047625595947819392627283726711361273296630256477108501994e-07,
-4.285970693151457255418342315045357407199066350632593899896712e-07,
-5.003361868748230293692887222336390314786090450819216035110269e-08,
8.965966311957728376981484572655177545054433542721057470726361e-08,
-1.219924359483373093110396748985081720383992859961285213840740e-08,
-1.104383021722648979552131128575075255513372249283096583736746e-08,
4.250422311980592983740943309197245384991941251563471671065543e-09,
4.384387799940474369553236949848427579687147486892033587998023e-10,
-5.881091462634605628881794361152305108432139465417759716875076e-10,
8.904723796221605490455387579189371137903330749397374037644960e-11,
3.26327074133290787598184498010494837595551273115386408552080e-11,

```
-1.430918765169202320188022211739750594608742928641485026836608e-11,  
1.0756106535010622115165734990153347111902874668945095034791947e-12,  
5.361482229611801638107331379599434078296259332654994508124989e-13,  
-1.663800489433402369889818192962259823988673359967722467427927e-13,  
2.000715303810524954375796020597627467104635766752154321244151e-14,  
-9.421019139535078421314655362291088223782497046057523323473331e-16};
```

```
const double Daub33[66] = {  
8.186358314175091939858945975190102731733968885547217619434602e-06,  
1.791016153702791479424389068736094134247294413108336017758506e-04,  
1.8227094335164084208084617771787691709255513374281497713580568e-03,  
1.139594337458160925830840619716397130445853638888472948832932e-02,  
4.861466653171619508385707681587366397164931431125053574327899e-02,  
1.481863131800528081784673514426737436792606299953305691300616e-01,  
3.267181301177075783930752787756046348844272437670999719562429e-01,  
5.093761725149396552227892926384090200953139820961482931291482e-01,  
5.112547705832674655425831875568453973369927971748064975152374e-01,  
2.095823507130554216526494469993023406452629154801126958766008e-01,  
-2.042026223985421049629055102642279430174095014493415546881477e-01,  
-3.159974107665602561905181464284910961862968513875028980451424e-01,  
-1.927833943695275915600583425408664108893845271616240406358226e-02,  
2.454206121192791114179964351253140999836791489738418857473689e-01,  
9.985155868033815698139640215477639365289384281516885362929979e-02,  
-1.714280990518593279308738113273443832545615219650436927029674e-01,  
-1.108441331167107910806084983056783194189909198734302929909672e-01,  
1.21967856403734614938913458437100977759176392114812695272220e-01,  
9.478808805061595889263191779090571160237408179346345390888721e-02,  
-9.114696835133148913093153757138373418923462847746880902676089e-02,  
-7.030248505405615921453280814171665167171986608963193275084895e-02,  
7.019114394099653254998935842432841393915841096633514680190145e-02,  
4.573456189389667743139040427641638967843459421665709740086516e-02,  
-5.347125133582228919431110824663168583260050383336359554980188e-02,  
-2.524858297747649929258392207837724793937727346177294684700378e-02,  
3.868706076024496481748675031852528047303323816250150793091832e-02,  
1.070326582001954942654534968137727769698168853186071888736311e-02,  
-2.572876175473297336123211392278301875687760837710204579628265e-02,  
-2.167758617353607324783298657172830203896433848418061622436727e-03,  
1.531695411585766548347442266431874060229304787191589430967538e-02,  
-1.594288782414604768637856446111392724059836934455189837500244e-03,  
-7.953540387057939240459305406538116220678495240302592677582773e-03,  
2.389062408165908575935815973439728988151836094753689966108405e-03,  
3.480800953405711999411461002429227385937942254778524257436278e-03,  
-1.860718214455795912074482150710567824317228203897000129729967e-03,  
-1.204309257604658876916644980097327372892008586047095719636829e-03,  
1.074380696351291355073899234941719080473877020595209197706651e-03,  
2.727305847336937211749282358350196461733595290569540045817329e-04,  
-4.908329007590351474487792254066540683724948757382104652497458e-04,  
4.393166251766185755059005296958129844094063524324718175254673e-06,  
1.780431898251245351831728023200069586928513661382622116969992e-04,  
-4.1604385162737093062334368807933932360567787692918883118883736e-05,  
-4.929564423417301834310231482621574127409950921583062559483686e-05,  
2.423335398816890365621188379922041046073808819182024026589770e-05,  
9.070805757828453800203677464921508178468256685438211818575040e-06,  
-8.866121366757736169176034432364298134186929098274651022820760e-06,  
-3.607516102879771631230351118595069330196155459105589342866625e-07,  
2.288371276141527305481395545993763010565968667577768164201792e-06,  
-4.426923407952870147984002129341809185622768353983550670755106e-07,  
-3.985791291985944076942626511739220753169387460984290019185514e-07,  
1.8224433325710534377467128998002798233969112236553215291639303e-07,  
3.377972703730854377516206663481869099376154259897212784144779e-08,  
-3.987838198518880722819502850814936369197384392561970319349663e-08,  
3.672863576838181340505563759379169099717712645283448779390320e-09,  
5.111211857347453839549366593998758891130921028374576213256027e-09,  
-1.671392677251932495173219614104411841891545601521784559793012e-09,  
-2.496402105246193648073519269370197331176405371538404298745013e-10,  
2.426833102305682309891302883361232297664099485514601790344279e-10,  
-3.049574453945863430361296931455141500128170151643206937547928e-11,  
-1.420236859889936792437077844940412749343225644487770840543290e-11,  
5.509414720765524548752673631197714447818740985929081064907524e-12,  
-3.343481218953278765982537222689984725170758193566174566492199e-13,  
-2.152488386833302618520603545685994753329478275805993737095214e-13,  
6.214740247174398315576214699577230693021307854673557214652751e-14,  
-7.196510545363322414033654470779070592316600780697558361083151e-15,  
3.289373678416306368625564108782095644036415401902518812978798e-16};
```

```
const double Daub34[68] = {  
5.770510632730285627466067796809329117324708919047900817738025e-06,  
1.299476200679530037833484815390569400369432658207722720405084e-04,  
1.364061390059049998200014449396877439591680435610837369411339e-03,  
8.819889403884978803182764563095879335330977939541630862804757e-03,  
3.904884135178594138905026219591569204043816577941517019631916e-02,  
1.241524821113768081954449898210969172708199672428635378051285e-01,  
2.877650592337145629334256618087718872558560120999651277991839e-01,
```

4.784787462793710621468610706120519466268010329031345843336104e-01,
5.3055509965646317731033260223990794445605699030503652382795600e-01,
2.903663295072749510455945186199530115755664977934564128822650e-01,
-1.282468421744371672912377747048558427612774932943748628650824e-01,
-3.315253015083869417715548463087537345035828886426345397256876e-01,
-1.038919155156404718287260506925867970596448618647006698388596e-01,
2.169072201874275950610018667099322465619408030256534197819784e-01,
1.666017504122074437311574334509261366682993700573488534577890e-01,
-1.273373582238011562843862636988693890108793629966541695807247e-01,
-1.609249271778668063014799490429649196614628857267382976958607e-01,
7.799184693794810738265349531832015087096882277333968473726399e-02,
1.341259602711361284802399913977387999358280900708582462625539e-01,
-5.448296806413904636632671383140642554265865948686157271017286e-02,
-1.029475969928140852342073823689090498245496056845473569066667e-01,
4.357609464963129726428486610925800727137724136370669421246609e-02,
7.31852354367956055546221335452045680757998947493883124934567e-02,
-3.701283841786244960356402125554190040750079009127461655784927e-02,
-4.743855964527776247220681410983851377889756018716427358008296e-02,
3.073974657395934459931226513844134346305562928466993208164603e-02,
2.722835075635419610095839895805858855202745897718117731496534e-02,
-2.367173792282636485046786438094940427456079528043555566867110e-02,
-1.314398001665716086105827506126287041342680578404007359439612e-02,
1.640937419986519252112261495537409592363156309874473310057471e-02,
4.713649260999809905918876125437488856235874027077755004539205e-03,
-1.004550670836151917439146861146431000364858401181337134891421e-02,
-6.194748845153872839014356621835501857322345445234809347431098e-04,
5.334950768759936032170270195983921511565539100791906952901398e-03,
-7.692127975067836975989490900561029844887285335804349474993607e-04,
-2.399453943537055863933124827688081952701780599883067560501870e-03,
8.589959874363661955444898475746536583497522107459291718900058e-04,
8.751999064078688732610570055224339733760304773327228476255647e-04,
-5.52735576214419797516415296735124460550632283763688359649888e-04,
-2.326732140233531635428863212833942245597361085708567528230733e-04,
2.650772397558057819755811309071002543822145660933016957735937e-04,
2.660050018453441903046828468025589086403126180798464347801678e-05,
-9.914697770780134603580350758869378471802751837608461971022567e-05,
1.353117227249649581251887376414486225127346352042209141315562e-05,
2.844951419697807376503080001943765930601242225183893658540032e-05,
-1.057657494257950623848316304755218120233253479317574337409622e-05,
-5.710826510998303938275050074333400305512451419983646591762318e-06,
4.169871758547028398316761659984928804362023643629741358799744e-06,
4.979718101421307748081857636471761057429219265531618602960147e-07,
-1.116306534817008428597995070751765080383261658112656948526954e-06,
1.448195708333185127061180618150009526758658641231104901703561e-07,
2.025990666667859216690536885693725545344933235432307649205497e-07,
-7.526701740412589411177481797841044281662555785969415398369019e-08,
-1.990346501531736915866180448337614967570744211158241514589121e-08,
1.740423332936068076497051274445147160190783847854409836489662e-08,
-8.665744261368722215864741166245385888818567571145958531936939e-10,
-2.316501946995482751582294240136010067415084499025753117941001e-09,
6.446378210323402313101214894500231181606520211579581132442548e-10,
1.300410318609415248880403259300467720631189120978928377152233e-10,
-9.904774537632409015479530333979124540183199174591377762845227e-11,
1.004208735461769864836516428998306778031143650101842361622330e-11,
6.08012535400016725405025929915591291115751734288584563131636e-12,
-2.107879108915301546285370395443778864676275235126044599683271e-12,
9.799451158211597727901178520526388692140586041163624252991805e-14,
8.579194051799733179793112298652600511486581216528683482143106e-14,
-2.317083703906408481078257081903089523234020423092175261925515e-14,
2.587338381935699555813538163144986688834142571207152879144731e-15,
-1.148944754480590128244815794312606245147888158018823490936280e-16};

```
const double Daub35[70] = {  
  9.067934061148559026665247110206084571051201477121972612218005e-06,  
  9.421469475576740631603027533116630224451049736050903361458759e-05,  
  1.019122680375098109319314672751485080202557607467199213778085e-03,  
  6.807292884319132011971333979015625113494050642797397817625326e-03,  
  3.123628851149071453063391210769353068187088999495893257051179e-02,  
  1.034044558614783789938787754929279183985553322796063517049140e-01,  
  2.513073789944933128513251971488905042866779761014740192816902e-01,  
  4.435927392240354378183910489448494594782039032807956294826105e-01,  
  5.370084275091661028670690231716974547580034932361053607723887e-01,  
  3.603456405180473278744458573988718422538114217890792270621563e-01,  
  -4.388388187393404111343479394097224312100349011932028865098625e-02,  
  -3.238228649121161212147302807993176715625480327235512530593160e-01,  
  -1.817869767667278325788350264528191676841493369460849123538616e-01,  
  1.660413574907809195438433327470947940538097914525298064477785e-01,  
  2.172992893210892977675493456199559114036326358517672106972956e-01,  
  -6.52628713106753892154895911331108284007380738865652420304233e-02,  
  -1.919195892985939528760786800798636198516495957924798820500876e-01,  
  1.930954466601835091947734585938109944647435243484967057775110e-02,  
  1.552924803962371144206753760712566993987319378965231186477630e-01,  
  -4.752680834111350445288110998030979143710864689041902167119118e-03,
```

-1.205855226433935545076589480704957722635324456812322150437989e-01,
4.734229172641948763293980314992213293971770695480616789828384e-03,
8.991354757072954417865374195261962983644048998218233900481856e-02,
-9.318558949903924837875002823617504227246562152671894579504378e-03,
-6.335603744044346612098887534020545705731671718057964802006671e-02,
1.322854958503655524455929847605110719648746890497356808289302e-02,
4.125469306470509212749750814299126656151504805845417994651417e-02,
-1.436683978422007182104025173214012797788904894291716373493525e-02,
-2.416949780166026740294880681731084091264533168816746227537030e-02,
1.276645671565674419403918018742432714973656598227939824940035e-02,
1.228943600811871086161967625814297050611100200023898377949151e-02,
-9.577797899235709998147309703713518608283233882793489733491642e-03,
-5.085991649233429881797630583578921194675393807761154549733547e-03,
6.137754586740521089596801883631921221145712545042519987641234e-03,
1.428088794070762107355585870669842132609159040625895090070111e-03,
-3.357644380922383229567732565298665639037348585961127075507937e-03,
7.615969435172736546769649923895317451534703066016116257300160e-06,
1.549637469702362975561719246539787717204438637997824935787688e-03,
-3.346692164250854961608526121524596908041109918361306282201310e-04,
-5.864810318991817532175809224131456738367101035694188223408841e-04,
2.6483288199612890393028610122699710966048565368047575218693134e-04,
1.700012283661249043584690194716767771204207742625746308522935e-04,
-1.365883072261161602559926714744746422567509177443594045709653e-04,
-2.976995962848509743944225866488519668585242655980656646544319e-05,
5.304143122913310222538317980686374696005605533475685587486683e-05,
-2.437001526827789860990429478540556752694389693432668831073769e-05,
-1.5724420772720281693663288966405861215692805972737981986121447e-05,
4.308047861716731191350493437937513220737450410132878032163179e-06,
3.353345862871309889390877168046133657377105681618708355266688e-06,
-1.895929617693153288493891051875444439753318548105998166574535e-06,
-3.903931733287306166657519468494511920760767388397825775326745e-07,
5.302368616904760917074352633915743250769600635829229600812520e-07,
-3.700308378205124537986402644918879149894035910106489082512364e-08,
-9.990396944534900755781728477561240762191443422318249128866740e-08,
3.008188650719066928230268918661718274504955045022550217051301e-08,
1.0849027337899346825266560240100449884720749303326571747323086e-08,
-7.458116552893037631192407611262788593505988638365840409367117e-09,
5.897951310384361575470355861162022501172491937837712969865619e-11,
1.030823345485433383811700481488557422005210168069163779730908e-09,
-2.43354557375167293616887250405940817227367937230289801251648e-10,
-6.407938256501889018430608323235974406219193176918284664973727e-11,
4.000536627253744510742788201354093006471710416671002244302586e-11,
-3.125639357108557540598098228678150768528121565391376265627294e-12,
-2.567065476155081449204643852428401530283519685638256074752850e-12,
8.015088533687900921948605418789324826115616416343391081288979e-13,
-2.59795432889384808431519820509438914570668012920898638802995e-14,
-3.397720856796267431956783825659069596940335130100871912329556e-14,
8.624037434720089202680337663692777682810714650060805832406135e-15,
-9.298012529324185420921555664719863501848315099116725184370339e-16,
4.014628712333488654318569164614220308046021091178184654250982e-17};

```
const double Daub36[72] = {  
2.867925182755946334630479473029238615535511775894262711054705e-06,  
6.8260286785463586917486291022096053622403442665053035981791715e-05,  
7.60215109966848828586972677106082100141275054892389379198545e-04,  
5.240297377409884366201603524392995696042174937194435235003941e-03,  
2.489056564482796484885927333959115579403023347044729739255255e-02,  
8.565209259526409083864716995521111486437594750377856524772704e-02,  
2.177569530979008149637945915719999746248969705650625533415876e-01,  
4.064336977082553467407793990250384445903151630768558142125382e-01,  
5.322668952607286914777444748641462027213554723153906901129337e-01,  
4.17875335600969786362063455937423645522275302996931178265919e-01,  
4.397519752934862993862182898358763783110745559238982179690132e-02,  
-2.944210395891145711100715969898758940722458887377844633443675e-01,  
-2.4680703697812552710524798278622698446566520718230313889086016e-01,  
9.811420416311477050518401371401568038943437322299913514049728e-02,  
2.465372776089742110529709111809595434656418762898152706621356e-01,  
7.278515095792229009687682299460382878643139026668958884429641e-03,  
-1.993372056086496198603363400094784142714162256792182570541036e-01,  
-4.586140074639271639145126228774831743002971373998329604574394e-02,  
1.541062366276428841776316300420654875883842819413623395358262e-01,  
5.027618007353842862036816972809884096761706036019748316890913e-02,  
-1.18803754310135631680181693138354744607315295104444224449501e-01,  
-3.988085357551317584091699967924044034100374257075864260934102e-02,  
9.115678225801654406336059281306715151058903055370522031843771e-02,  
2.503872144956848989919484296709846860569180993040383621980546e-02,  
-6.820901663681751124880436344265538690580358108714540763125119e-02,  
-1.131910031681742794381808082173695022123056280821611354577883e-02,  
4.851308354780908538616267662315735632292989749013261207046367e-02,  
1.424972661765391603147802607378542396323429657660009755652404e-03,  
-3.198072067763969654470293513742344601172739688274251641873778e-02,  
3.984040198717004857397179486790082321314291366656151213429068e-03,  
1.906359478062535932877576164368198274858108513696832728889209e-02,
```

-5.657813245058818380424016973516714570499161434975761798379020e-03,
-9.990263473281372348001743806489172665465685056975652497503772e-03,
5.022989106665829004699819220796538830393945994687289792465541e-03,
4.413484835350575251918616780287775585471012556848037301025999e-03,
-3.484541445404883311209541395428535732697661971818727286003028e-03,
-1.50307406629664374954936355363411879858070202740814054964603e-03,
1.99079377185173270404293245701878186600899439513475823305914e-03,
2.776812795712026068152384207605140383490242756921936501940389e-04,
-9.463403823261101964604918059447913047725482130063492242779878e-04,
8.614565758992702032613879159402330909634737204578606399403107e-05,
3.693507284967510502620040341882236687749563414433432842567511e-04,
-1.155118895843527096848076999413102395191976350936666573818799e-04,
-1.131899468084665671727391922924411467938450743565106978099456e-04,
6.694741196930590257104231749283786251555566773398199990337698e-05,
2.375106683660860777161950832380341362257503761490580896617678e-05,
-2.731390824654337912922346414722045404779935825834384250023192e-05,
-1.183471059985615942783182762352360917304348034947412986608322e-06,
8.372218198160788432628056043217491552198857358432112275253310e-06,
-1.586145782434577495502614631566211839722879492827911790709498e-06,
-1.870811602859180713762972281154953528056257451900381097476968e-06,
8.311421279707778528163597405935375886855029592150424544500718e-07,
2.548423522556577831218519052844387478819866531902854523544709e-07,
-2.455377658434232699135878286794578515387138194247693201846263e-07,
2.753249073339512254085076456700241929492720457889076058451072e-09,
4.799043465450992009934526867650497683545716858606119786327559e-08,
-1.156093688817008406756913949175208452083765368825442482226093e-08,
-5.612784343327791397474114357094368557982413895802980814813369e-09,
3.138841695782424018351567952158415003571380699236147752239001e-09,
1.090815553713751810964713058800448676068475673611349566405716e-10,
-4.512545778563249634425200856088490195004077806062978067796020e-10,
8.962418203859611987065968320295929679774693465791367610044773e-11,
3.037429098112535221800013609576297196061786927734556635696416e-11,
-1.599716689261357143200396922409448515398648489795044468046420e-11,
8.876846287217374213524399682895564055949886050748321818411161e-13,
1.070969357114017002424433471621197579059927261727846375968378e-12,
-3.029285026974877268896134589769473854669758797446795757329862e-13,
5.542263182639804235231685861028995158694397223907295269180336e-15,
1.338071386299105896025578761458472955294763310766371178363783e-14,
-3.204628543401749860439316638848579711789176444320134355253750e-15,
3.339971984818693213132578777712503670014459411167839211495237e-16,
-1.403274175373190617489823209168013922564353495443487431242610e-17};

```
const double Daub37[74] = {  
    2.022060862498392121815038335333633351464174415618614893795880e-06,  
    4.942343750628132004714286117434454499485737947791397867195910e-05,  
    5.662418377066724013768394373249439163518654840493603575144737e-04,  
    4.024140368257286770702140124893772447952256842478891548092703e-03,  
    1.976228615387959153244055502205017461538589475705618414896893e-02,  
    7.058482597718160832030361890793007659963483925312132741868671e-02,  
    1.873263318620649448028843491747601576761901656888288838192023e-01,  
    3.684409724003061409445838616964941132670287724754729425204047e-01,  
    5.181670408556228873104519667534437205387109579265718071174178e-01,  
    4.622075536616057145505448401528172070050768534504278694229363e-01,  
    1.308789632330201726057701201017649601034381070893275586898075e-01,  
    -2.46180429761083132869018581145720710365433914584680691693717e-01,  
    -2.943759152626617722808219575932673733674290772235644691367427e-01,  
    1.967150045235938977077768648740052380288156507222647187301894e-02,  
    2.515232543602686933435224095078166291442923992611593827552710e-01,  
    8.180602838721862339029076982652411696000045533716726027662147e-02,  
    -1.8196229177860800740882425652522521644443143868752611284260e-01,  
    -1.084517138233017845554078812341876568514835176341639783558543e-01,  
    1.299296469598537527842528895259188653120602318620944502979726e-01,  
    1.017802968388141797470948228505865617480048287983176581607964e-01,  
    -9.660754061668439030915405045955772715988585374771282291315496e-02,  
    -8.233021190655740867404073660920379414988302492018783774702028e-02,  
    7.504761994836017933579005072594245435071674452882148228583865e-02,  
    5.956741087152995245435589042520108066877114768216272503684398e-02,  
    -5.925681563265897095153806724965924334077555174281436189512239e-02,  
    -3.825382947938424882011108885090442116802994193611884738133373e-02,  
    4.580794415126833246633256156110381805848138158784734496981778e-02,  
    2.097280059259754883313769469036393294461497749083921162354229e-02,  
    -3.352358406410096994358662875913243067234786296009238949920582e-02,  
    -8.833493890410232394064187990625563257107429109130726291528648e-03,  
    2.261865154459947356571431658958802912061105608212828675323452e-02,  
    1.690472383484423743663952859090705636512807161536954018400081e-03,  
    -1.376398196289478433857985486097070339786225136728067000591187e-02,  
    1.519305778833399218481261844599507408563295102235964076544334e-03,  
    7.387757452855583640107787619408806919082115520707105052944171e-03,  
    -2.248053187003824706127276829147166466869908326245810952521710e-03,  
    -3.394523276408398601988475786247462646314228994098320665709345e-03,  
    1.816871343801423525477184531347879515909226877688306010517914e-03,  
    1.263934258117477182626760951047019242187910977671449470318766e-03,  
    -1.111484865318630197259018233162929628309920117691177260742614e-03,
```

-3.2807884708801984194071864551908995357062322295554613820907245e-04,
5.490532773373631230219769273898345809368332716288071475378651e-04,
1.534439023195503211083338679106161291342621676983096723309776e-05,
-2.208944032455493852493630802748509781675182699536797043565515e-04,
4.336726125945695214852398433524024058216834313839357806404424e-05,
7.055138782065465075838703109997365141906130284669094131032488e-05,
-3.098662927619930052417611453170793938796310141219293329658062e-05,
-1.639162496160583099236044020495877311072716199713679670940295e-05,
1.354327718416781810683349121150634031343717637827354228989989e-05,
1.849945003115590390789683032647334516600314304175482456338006e-06,
-4.309941556597092389020622638271988877959028012481278949268461e-06,
4.85473139696411681769911684430785681028852413859386141424933e-07,
1.002121399297177629772998172241869405763288457224082581829033e-06,
-3.494948603445727645895194867933547164628229076947330682199174e-07,
-1.509885388671583553484927666148474078148724554849968758642331e-07,
1.10903123221643938999036327867142640916239658806376290861690e-07,
5.350657515461434290618742656970344024396382191417247602674540e-09,
-2.252193836724805775389816424695618411834716065179297102428180e-08,
4.224485706362419268050011630338101126995607958955688879525896e-09,
2.793974465953982659829387370821677112004867350709951380622807e-09,
-1.297205001469435139867868007585972538983682739297235604327668e-09,
-1.031411129096974965677950646498153071722880698222864687038596e-10,
1.946164894082315021308714557636277980079559327508927751052218e-10,
-3.203398244123241367987902201268363088933939831689591684670080e-11,
-1.398415715537641487959551682557483348661602836709278513081908e-11,
6.334955440973913249611879065201632922100533284261000819747915e-12,
-2.09636319423480054161477574275555713279549381264881030843258e-13,
-4.421612409872105367333572734854401373201808896976552663098518e-13,
1.138052830921439682522395208295427884729893377395129205716662e-13,
-4.518889607463726394454509623712773172513778367070839294449849e-16,
-5.243025691884205832260354503748325334301994904062750850180233e-15,
1.189012387508252879928637969242590755033933791160383262132698e-15,
-1.199280335852879554967035114674445327319437557227036460257649e-16,
4.906615064935203694857690087429901193139905690549533773201453e-18};

```
const double Daub38[76] = {  
1.425776641674131672055420247567865803211784397464191115245081e-06,  
3.576251994264023012742569014888876217958307227940126418281357e-05,  
4.211702664727116432247014444906469155300573201130549739553848e-04,  
3.083088119253751774288740090262741910177322520624582862578292e-03,  
1.56372493475215617277490102724080070486270026632620664785632e-02,  
5.788994361285925649727664279317241952513246287766481213301801e-02,  
1.600719935641106973482800861166599685169395465055048951307626e-01,  
3.307757814110146511493637534404611754800768677041577030757306e-01,  
4.965911753117180976599171147718708939352414838951726087564419e-01,  
4.933560785171007975728485346997317064969513623594359091115804e-01,  
2.130505713555785138286743353458562451255624665951160445122307e-01,  
-1.828676677083358907975548507946239135218223185041410632924815e-01,  
-3.216756378089978628483471725406916361929841940528189059002548e-01,  
-6.226650604782432226643360160478765847565862101045597180310490e-02,  
2.321259638353531085028708104285994998671615563662858079262996e-01,  
1.499851196187170199586403453788927307298226028262603028635758e-01,  
-1.417956859730596216710053144522330276392591055375830654519080e-01,  
-1.599125651582443618288533214523534937804208844386102639177693e-01,  
8.563812155615105741612217814369165313487129645536001850276987e-02,  
1.41414734073382680068468123119379170594092606174915755283496153e-01,  
-5.658645863072738145681787657843320646815509410635114234947902e-02,  
-1.147311707107443752394144019458942779715665489230169950201022e-01,  
4.309589543304764288137871223616030624246568683595408792078602e-02,  
8.720439826203975011910714164154456762073786124233088471855868e-02,  
-3.660510340287429567372071039506772372567938710943432838908247e-02,  
-6.176620870841315993604736705613246241897497782373337911398117e-02,  
3.198987753153780630818381136366859026137035450576631134176875e-02,  
4.005498110511594820952087086241114309038577379366732959648548e-02,  
-2.689149388089451438550851767715967313417890393287236700072071e-02,  
-2.311413402054931680856913553585621248925303865540203357180768e-02,  
2.090464525565524340215982365351342094670261491526831672682244e-02,  
1.129049727868596484270081487761544232851115891449843967151657e-02,  
-1.470188206539868213708986402816605045648481224662435114088245e-02,  
-4.131306656031089274123231103326745723188134548520938157995702e-03,  
9.214785032197180512031534870181734003522861645903894504302286e-03,  
5.625715748403532005741565594881148757066703437214522101740941e-04,  
-5.071314509218348093935061417505663002006821323958752649640329e-03,  
7.169821821064019257784165364894915621888541496773370435889585e-04,  
2.4006977818909731838923069140825921439884140550210130139535193e-03,  
-8.448626665537775009068937851465856973251363010924003314643612e-04,  
-9.424614077227377964015942271780098283910230639908018778588910e-04,  
5.810759750532863662020321063678196633409555706981476723988312e-04,  
2.817639250380670746018048967535608190123523180612961062603672e-04,  
-3.031020460726611993600629020329784682496477106470427787747855e-04,  
-4.555682696668420274688683005987764360677217149927938344795290e-05,  
1.262043350166170705382346537131817701361522387904917335958705e-04,  
-1.155409103833717192628479047983460953381959342642374175822863e-05,
```

-4.175141648540397797296325065775711309197411926289412468280801e-05,
1.334176149921350382547503457286060922218070031330137601427324e-05,
1.037359184045599795632258335010065103524959844966094870217687e-05,
-6.456730428469619160379910439617575420986972394137121953806236e-06,
-1.550844350118602575853380148525912999401292473185534395740371e-06,
2.149960269939665207789548199790770596890252405076394885606038e-06,
-8.487087586072593071869805266089426629606479876982221840833098e-08,
-5.187733738874144426008474683378542368066310000602823096009187e-07,
1.396377545508355481227961581059961184519872502493462010264633e-07,
8.400351046895965526933587176781279507953080669259318722910523e-08,
-4.884757937459286762082185411608763964041010392101914854918157e-08,
-5.424274800287298511126684174854414928447521710664476410973981e-09,
1.034704539274858480924046490952803937328239537222908159451039e-08,
-1.436329487795135706854539856979275911183628476521636251660849e-09,
-1.349197753983448821850381770889786301246741304307934955997111e-09,
5.261132557357598494535766638772624572100332209198979659077082e-10,
6.732336490189308685740626964182623159759767536724844030164551e-11,
-8.278256522538134727330692938158991115335384611795874767521731e-11,
1.101692934599454551150832622160224231280195362919498540913658e-11,
6.291537317039508581580913620859140835852886308989584198166174e-12,
-2.484789237563642857043361214502760723611468591833262675852242e-12,
2.626496504065252070488282876470525379851429538389481576454618e-14,
1.808661236274530582267084846343959377085922019067808145635263e-13,
-4.249817819571463006966616371554206572863122562744916796556474e-14,
-4.563397162127373109101691643047923747796563449194075621854491e-16,
2.045099676788988907802272564402310095398641092819367167252952e-15,
-4.405307042483461342449027139838301611006835285455050155842865e-16,
4.304596839558790016251867477122791508849697688058169053134463e-17,
-1.716152451088744188732404281737964277713026087224248235541071e-18};

[DFT](#) (click this to go back to the index)

Type : fourier transform

References : Posted by Andy Mucho

Code :

```
AnalyseWaveform(float *waveform, int framesize)
{
    float aa[MaxPartials];
    float bb[MaxPartials];
    for(int i=0;i<partials;i++)
    {
        aa[i]=0;
        bb[i]=0;
    }

    int hfs=framesize/2;
    float pd=pi/hfs;
    for (i=0;i<framesize;i++)
    {
        float w=waveform[i];
        int im = i-hfs;
        for(int h=0;h<partials;h++)
        {
            float th=(pd*(h+1))*im;
            aa[h]+=w*cos(th);
            bb[h]+=w*sin(th);
        }
    }
    for (int h=0;h<partials;h++)
        amp[h]= sqrt(aa[h]*aa[h]+bb[h]*bb[h])/hfs;
}
```

Envelope detector (click this to go back to the index)

References : Posted by Bram

Notes :
Basically a one-pole LP filter with different coefficients for attack and release fed by the abs() of the signal. If you don't need different attack and decay settings, just use in->abs()->LP

```
Code :  
//attack and release in milliseconds  
float ga = (float) exp(-1/(SampleRate*attack));  
float gr = (float) exp(-1/(SampleRate*release));  
  
float envelope=0;  
  
for(...)  
{  
    //get your data into 'input'  
    EnvIn = abs(input);  
  
    if(envelope < EnvIn)  
    {  
        envelope *= ga;  
        envelope += (1-ga)*EnvIn;  
    }  
    else  
    {  
        envelope *= gr;  
        envelope += (1-gr)*EnvIn;  
    }  
    //envelope now contains.....the envelope ;)  
}
```

Comments

from : arguru [[a t]] smartelectronix

comment : Nice , just a typo: //attack and release is entered in SECONDS actually in this code ;)

from : antiprosynthesis [[a t]] gmail.com

comment : // Slightly faster version of the envelope follower using one multiply form.

// attTime and relTime is in seconds

```
float ga = exp(-1.0f/(sampleRate*attTime));  
float gr = exp(-1.0f/(sampleRate*relTime));
```

```
float envOut = 0.0f;
```

```
for( ... )  
{  
    // get your data into 'input'  
  
    envIn = fabs(input);  
  
    if( envOut < envIn )  
        envOut = envIn + ga * (envOut - envIn);  
    else  
        envOut = envIn + gr * (envOut - envIn);  
  
    // envOut now contains the envelope  
}
```

from : madgel79 [[a t]] nate.com

comment : in my code , attack_coef and release_coef are always '0'.

If I use only 'abs()' , it also work well.
why can it be possible?

Would you please give me some information about this problem.

Thanks.

Envelope Detector class (C++) (click this to go back to the index)

Type : envelope detector

References : Posted by Citizen Chunk

Linked file : <http://www.chunkware.com/opensource/EnvelopeDetector.zip>

Notes :

This is a C++ implementation of a simple envelope detector. The time constant (ms) represents the time it takes for the envelope to charge/discharge 63% (RC time constant).

(see linked files)

Comments

from : citizenchunk [at] chunkware [dot] com

comment : due to popular demand, i have added an AttRelEnvelope class to this source, implementing a typical attack/release envelope.

for my own taste, i prefer to keep the state variable separate from the envelope detector. however, if you prefer to have it as a member variable, you can easily inherit these classes and add them. (but please remember to add a function for initializing the state before runtime.)

from : citizenchunk [at] chunkware [dot] com

comment : link moved:

<http://www.chunkware.com/downloads/simpleSource.zip>

Envelope follower with different attack and release (click this to go back to the index)

References : Posted by Bram

Notes :

xxxx_in_ms is xxxx in milliseconds ;-)

Code :

```
init::  
  
attack_coef = exp(log(0.01)/( attack_in_ms * samplerate * 0.001));  
release_coef = exp(log(0.01)/( release_in_ms * samplerate * 0.001));  
envelope = 0.0;  
  
loop::  
  
tmp = fabs(in);  
if(tmp > envelope)  
    envelope = attack_coef * (envelope - tmp) + tmp;  
else  
    envelope = release_coef * (envelope - tmp) + tmp;
```

Comments

from : jm [[a t]] kampsax.dtu.dk

comment : the expressions of the form:

```
xxxx_coef = exp(log(0.01)/( xxxx_in_ms * samplerate * 0.001));
```

can be simplified a little bit to:

```
xxxx_coef = pow(0.01, 1.0/( xxxx_in_ms * samplerate * 0.001));
```

from : kainhart [[a t]] hotmail.com

comment : Excuse me if I'm asking a lame question but is the envelope variable the output for the given input sample? Also would this algorithm apply to each channel independently for a stereo signal? One more question what is an Envelope Follower, what does it sound like?

from : yanyuqiang [[a t]] hotmail.com

comment : What's the difference between this one and the one you posted named 'Envelope detector'? Different definition? What's the exact definition of release time and attack time?

from : scoofy [[a t]] inf.elte.hu

comment : Here the definition of the attack/release time is the time for the envelope to fall from 100% to 1%. In the other version, the definition is for the envelope to fall from 100% to 36.7%. So in this one the envelope is about 4.6 times faster.

Fast in-place Walsh-Hadamard Transform (click this to go back to the index)

Type : wavelet transform

References : Posted by Timo H Tossavainen

Notes :
IIRC, They're also called walsh-hadamard transforms.
Basically like Fourier, but the basis functions are squarewaves with different sequencies.
I did this for a transform data compression study a while back.
Here's some code to do a walsh hadamard transform on long ints in-place (you need to divide by n to get transform) the order is bit-reversed at output, IIRC.
The inverse transform is the same as the forward transform (expects bit-reversed input). i.e. $x = 1/n * FWHT(FWHT(x))$ (x is a vector)

```
Code :
void inline wht_bfly (long& a, long& b)
{
    long tmp = a;
    a += b;
    b = tmp - b;
}

// just a integer log2
int inline l2 (long x)
{
    int l2;
    for (l2 = 0; x > 0; x >>=1)
    {
        ++ l2;
    }

    return (l2);
}

////////////////////////////////////
// Fast in-place Walsh-Hadamard Transform //
////////////////////////////////////

void FWHT (std::vector& data)
{
    const int log2 = l2 (data.size()) - 1;
    for (int i = 0; i < log2; ++i)
    {
        for (int j = 0; j < (1 << log2); j += 1 << (i+1))
        {
            for (int k = 0; k < (1<<i); ++k)
            {
                wht_bfly (data [j + k], data [j + k + (1<<i)]);
            }
        }
    }
}
```

Comments

from : i_arслан [[a t]] gmx.net

comment : How can i implement this code in matlab ?

from : Noor98z [[a t]] hotmail.com

comment : Need an Implementation of Walsh Transform Matlab Code

from : ambadarneh [[a t]] hotmail.com

comment : well sir nice work ,but how can i get the code in MATLAB .
could you help me please.

thank you

yours sicerly

Ala' Badarneh

Jordan University of Science & Technology

Biomedical Engineering Department

from : msalharbi [[a t]] hotmail.com

comment : Noor98z@hotmail.com

need implementation of walsh transform Matlab code
and me need.

from : fahira10 [[a t]] hotmail.com

comment : Dear sir,

your job is amazing!

I was just searching for a WHT program but I did not find any Matlab command for that. Is there a way to implement that code in Matlab?

Best regards

from : z_xf [[a t]] sdu.edu.cn

comment : nice work ,but how can i get the code in MATLAB .
could you help me please.thank you!

yours sicerly zhang

from : (1< - ????????????????????????)

comment : What is symbols "(1<" in C++; I dont know this symbols. Please help me.

from : beeka at beeka dot org

comment : You might want to look at the page source. This shows the "(1<" was trying to be "(1 << i)" (I hope the browser leaves that one alone):

```
for (int k = 0; k < (1 << i); ++k)
{
    wht_bfly (data [j + k], data [j + k + (1 << i)]);
}
```

from : bcreado [[a t]] hotmail.com

comment : well sir nice work ,but how can i get the code in MATLAB .
could you help me please

from : anonymous

comment : Hello,
I just thought I would let you know that I do not want MatLab code for this.
Thank you.

from : tired_of_morons [[a t]] gmail.com

comment : FOR CRYING OUT LOUD!! IMPLEMENT IT YOURSELF IN MATLAB! OR MAYBE, JUST MAYBE, IF YOU REQUEST FOR MATLAB CODE TEN TIMES MORE, YOU'LL GET IT... NOT!

"Most people are amazing. Just not in a positive sense..." - unknown

from : nobody [[a t]] nowhere.com

comment : What the hell is MATLAB anyway?

from : scoofy [[a t]] inf.elte.hu

comment : MATLAB guys: Have you ever tried the google search "fast walsh hadamard matlab"? If yes, look at the first link.

from : debonair_swathi [[a t]] yahoo.co.in

comment : i need program in c to generate a hadamard matrix of any order

from : debonair_swathi [[a t]] yahoo.co.in

comment : sir i need program in c to get the walsh matrix of any order to use as a coding tecchnique in CDMA .can u email it to me

from : anonymous

comment : Lend a hand and they grab your shoulder...

[FFT](#) (click this to go back to the index)

References : Toth Laszlo

Linked file : [rvfft.ps](#)

Linked file : [rvfft.cpp](#) (this linked file is included below)

Notes :
A paper (postscript) and some C++ source for 4 different fft algorithms, compiled by Toth Laszlo from the Hungarian Academy of Sciences Research Group on Artificial Intelligence.
Toth says: "I've found that Sorensen's split-radix algorithm was the fastest, so I use this since then (this means that you may as well delete the other routines in my source - if you believe my results)."

Linked files

```
//
//          FFT library
//
// (one-dimensional complex and real FFTs for array
// lengths of 2^n)
//
// Author: Toth Laszlo (tothl@inf.u-szeged.hu)
//
// Research Group on Artificial Intelligence
// H-6720 Szeged, Aradi vertanuk tere 1, Hungary
//
// Last modified: 97.05.29
////////////////////////////////////

#include <math.h>
#include <stdlib.h>
#include "pi.h"

////////////////////////////////////
//Gives back "i" bit-reversed where "size" is the array
//length
//currently none of the routines call it

long bitreverse(long i, long size){

    long result,mask;

    result=0;
    for(;size>1;size>>=1){
        mask=i&1;
        i>>=1;
        result<<=1;
        result|=mask;
    }

/*    __asm{          the same in asseibly
        mov eax,i
        mov ecx,size
        mov ebx,0
        l:shr eax,1
        rcl ebx,1
        shr ecx,1
        cmp ecx,1
        jnz l
        mov result,ebx
    }*/
    return result;
}

////////////////////////////////////
//Bit-reverser for the Bruun FFT
//Parameters as of "bitreverse()"

long bruun_reverse(long i, long sizze){

    long result, add;

    result=0;
    add=sizze;

    while(1){
        if(i!=0) {
            while((i&1)==0) { i>>=1; add>>=1;}
            i>>=1; add>>=1;
            result+=add;
        }
        else {result<<=1;result+=add; return result;}
    }
}
```

```

if(i!=0) {
  while((i&1)==0) { i>>=1; add>>=1;}
  i>>=1; add>>=1;
  result-=add;
}
else {result<<=1;result-=add; return result;}
}
}

/*assembly version
long bruun_reverse(long i, long sizze){

  long result;

  result=0;

  __asm{
  mov edx,sizze
  mov eax,i
  mov ebx,0
  l: bsf cx,eax
  jz kesz1
  inc cx
  shr edx,cl
  add ebx,edx
  shr eax,cl
  bsf cx,eax
  jz kesz2
  inc cx
  shr edx,cl
  sub ebx,edx
  shr eax,cl
  jmp l
kesz1:
  shl ebx,1

  add ebx,edx
  jmp vege
kesz2:
  shl ebx,1
  sub ebx,edx

  vege: mov result,ebx
  }
return result;
}*/

////////////////////////////////////
// Decimation-in-freq radix-2 in-place butterfly
// data: array of doubles:
// re(0),im(0),re(1),im(1),...,re(size-1),im(size-1)
// it means that size=array_length/2 !!
//
// suggested use:
// input in normal order
// output in bit-reversed order
//
// Source: Rabiner-Gold: Theory and Application of DSP,
// Prentice Hall,1978

void dif_butterfly(double *data,long size){

  long angle,astep,d1;
  double xr,yr,xi,yi,wr,wi,dr,di,ang;
  double *l1, *l2, *end, *o12;

  astep=1;
  end=data+size+size;
  for(d1=size;d1>1;d1>>=1,astep+=astep){
    l1=data;
    l2=data+d1;

    o12=l2;
    for(;l2<end;l1=l2,l2=l2+d1){
      for(angle=0;l1<o12;l1+=2,l2+=2){
        ang=2*pi*angle/size;
        wr=cos(ang);
        wi=-sin(ang);
        xr=*l1+*l2;
        xi=(l1+1)+*(l2+1);
        dr=*l1-*l2;
        di=(l1+1)-*(l2+1);
        yr=dr*wr-di*wi;
        yi=dr*wi+di*wr;

```



```

                *(l1)=xr;*(l1+1)=xi;
                *(l2)=yr;*(l2+1)=yi;
                angle+=astep;
            }
        }
    }
}

```

```

////////////////////////////////////
// Decimation-in-time radix-2 in-place inverse butterfly
// data: array of doubles:
// re(0),im(0),re(1),im(1),...,re(size-1),im(size-1)
// it means that size=array_length/2 !!
//
// suggested use:
// input in bit-reversed order
// output in normal order
//
// Source: Rabiner-Gold: Theory and Application of DSP,
// Prentice Hall,1978

```

```

void inverse_dit_butterfly(double *data,long size){

    long angle,astep,d1;
    double xr,yr,xi,yi,wr,wi,dr,di,ang;
    double *l1, *l2, *end, *ol2;

    astep=size>>1;
    end=data+size+size;
    for(d1=2;astep>0;d1+=d1,astep>>=1){
        l1=data;
        l2=data+d1;
        for(;l2<end;l1=l2,l2=l2+d1){
            ol2=l2;
            for(angle=0;l1<ol2;l1+=2,l2+=2){
                ang=2*pi*angle/size;
                wr=cos(ang);
                wi=sin(ang);
                xr=*l1;
                xi=*(l1+1);
                yr=*l2;
                yi=*(l2+1);
                dr=yr*wr-yi*wi;
                di=yr*wi+yi*wr;
                *(l1)=xr+dr;*(l1+1)=xi+di;
                *(l2)=xr-dr;*(l2+1)=xi-di;
                angle+=astep;
            }
        }
    }
}

```

```

////////////////////////////////////
// data shuffling into bit-reversed order
// data: array of doubles:
// re(0),im(0),re(1),im(1),...,re(size-1),im(size-1)
// it means that size=array_length/2 !!
//
// Source: Rabiner-Gold: Theory and Application of DSP,
// Prentice Hall,1978

```

```

void unshuffle(double *data, long size){

    long i,j,k,l,m;
    double re,im;

    //old version - turned out to be a bit slower
    /*for(i=0;i<size-1;i++){
        j=bitreverse(i,size);
        if (j>i){ //swap
            re=data[i+1];im=data[i+1+1];
            data[i+1]=data[j+1];data[i+1+1]=data[j+1+1];
            data[j+1]=re;data[j+1+1]=im;
        }
    }*/

    l=size-1;
    m=size>>1;
    for (i=0,j=0; i<l ; i++){
        if (i<j){
            re=data[j+1]; im=data[j+1+1];
            data[j+1]=data[i+1]; data[j+1+1]=data[i+1+1];
            data[i+1]=re; data[i+1+1]=im;
        }
    }
}

```

```

    }
    k=m;
    while (k<=j){
        j-=k;
        k>>=1;
    }
    j+=k;
}

////////////////////////////////////
// used by reallfft
// parameters as above
//
// Source: Brigham: The Fast Fourier Transform
// Prentice Hall, ?

void realize(double *data, long size){

    double xr,yr,xi,yi,wr,wi,dr,di,ang,astep;
    double *l1, *l2;

    l1=data;
    l2=data+size+size-2;
    xr=*l1;
    xi=*(l1+1);
    *l1=xr+xi;
    *(l1+1)=xr-xi;
    l1+=2;
    astep=pi/size;
    for(ang=astep;l1<=l2;l1+=2,l2-=2,ang+=astep){
        xr=(*l1+*l2)/2;
        yi=(-(*l1)+(*l2))/2;
        yr=*(l1+1)+*(l2+1))/2;
        xi=*(l1+1)-*(l2+1))/2;
        wr=cos(ang);
        wi=-sin(ang);
        dr=yr*wr-yi*wi;
        di=yr*wi+yi*wr;
        *l1=xr+dr;
        *(l1+1)=xi+di;
        *l2=xr-dr;
        *(l2+1)=-xi+di;
    }
}

////////////////////////////////////
// used by inverse reallfft
// parameters as above
//
// Source: Brigham: The Fast Fourier Transform
// Prentice Hall, ?

void unrealize(double *data, long size){

    double xr,yr,xi,yi,wr,wi,dr,di,ang,astep;
    double *l1, *l2;

    l1=data;
    l2=data+size+size-2;
    xr=(*l1)/2;
    xi=*(l1+1))/2;
    *l1=xr+xi;
    *(l1+1)=xr-xi;
    l1+=2;
    astep=pi/size;
    for(ang=astep;l1<=l2;l1+=2,l2-=2,ang+=astep){
        xr=(*l1+*l2)/2;
        yi=-(-(*l1)+(*l2))/2;
        yr=*(l1+1)+*(l2+1))/2;
        xi=*(l1+1)-*(l2+1))/2;
        wr=cos(ang);
        wi=-sin(ang);
        dr=yr*wr-yi*wi;
        di=yr*wi+yi*wr;
        *l2=xr+dr;
        *(l1+1)=xi+di;
        *l1=xr-dr;
        *(l2+1)=-xi+di;
    }
}

////////////////////////////////////

```

```

// in-place Radix-2 FFT for complex values
// data: array of doubles:
// re(0),im(0),re(1),im(1),...,re(size-1),im(size-1)
// it means that size=array_length/2 !!
//
// output is in similar order, normalized by array length
//
// Source: see the routines it calls ...

void fft(double *data, long size){

    double *l, *end;

    dif_butterfly(data,size);
    unshuffle(data,size);

    end=data+size+size;
    for(l=data;l<end;l++){*l=*l/size;};
}

////////////////////////////////////
// in-place Radix-2 inverse FFT for complex values
// data: array of doubles:
// re(0),im(0),re(1),im(1),...,re(size-1),im(size-1)
// it means that size=array_length/2 !!
//
// output is in similar order, NOT normalized by
// array length
//
// Source: see the routines it calls ...

void ifft(double* data, long size){

    unshuffle(data,size);
    inverse_dif_butterfly(data,size);
}

////////////////////////////////////
// in-place Radix-2 FFT for real values
// (by the so-called "packing method")
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),re(1),im(1),re(2),im(2),...,re(size/2-1),im(size/2-1)
// normalized by array length
//
// Source: see the routines it calls ...

void realfft_packed(double *data, long size){

    double *l, *end;
    double div;

    size>>=1;
    dif_butterfly(data,size);
    unshuffle(data,size);
    realize(data,size);

    end=data+size+size;
    div=size+size;
    for(l=data;l<end;l++){*l=*l/div;};
}

////////////////////////////////////
// in-place Radix-2 inverse FFT for real values
// (by the so-called "packing method")
// data: array of doubles:
// re(0),re(size/2),re(1),im(1),re(2),im(2),...,re(size/2-1),im(size/2-1)
//
// output:
// re(0),re(1),re(2),...,re(size-1)
// NOT normalized by array length
//
//Source: see the routines it calls ...

void irealfft_packed(double *data, long size){

    double *l, *end;

    size>>=1;

```

```

unrealize(data,size);
unshuffle(data,size);
inverse_dit_butterfly(data,size);

end=data+size+size;
for(l=data;l<end;l++){*l=(*l)*2;};
}

////////////////////////////////////
// Bruun FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),...,re(i),im(i)... pairs in
// "bruun-reversed" order
// normalized by array length
//
// Source:
// Bruun: z-Transform DFT Filters and FFT's
// IEEE Trans. ASSP, ASSP-26, No. 1, February 1978
//
// Comments:
// (this version is implemented in a manner that every
// cosine is calculated only once;
// faster than the other version (see next)

void realfft_bruun(double *data, long size){

double *end, *l0, *l1, *l2, *l3;
long dl, dl2, dl_o, dl2_o, i, j, k, kk;
double d0,d1,d2,d3,c,c2,p4;

end=data+size;
//first filterings, when there're only two taps
size>>=1;
dl=size;
dl2=dl/2;
for(;dl>1;dl>>=1,dl2>>=1){
l0=data;
l3=data+dl;
for(i=0;i<dl;i++){
d0=*l0;
d2=*l3;
*l0=d0+d2;
*l3=d0-d2;
l0++;
l3++;
}
}
l0=data;l1=data+1;
d0=*l0;d1=*l1;
*l0=d0+d1;*l1=d0-d1;
l1+=2;
*l1=-(*l1);

//the remaining filterings

p4=pi/(2*size);
j=1;
kk=1;
dl_o=size/2;
dl2_o=size/4;
while(dl_o>1){
for(k=0;k<kk;k++){
c2=p4*bruun_reverse(j,size);
c=2*cos(c2);
c2=2*sin(c2);
dl=dl_o;
dl2=dl2_o;
for(;dl>1;dl>>=1,dl2>>=1){
l0=data+((dl*j)<<1);
l1=l0+dl2;l2=l0+dl;l3=l1+dl;
for(i=0;i<dl2;i++){
d1=(*l1)*c;
d2=(*l2)*c;
d3=*l3+(*l1);
d0=*l0+(*l2);
*l0=d0+d1;
*l1=d3+d2;
*l2=d0-d1;
*l3=d3-d2;

```

```

    l0++;
    l1++;
    l2++;
    l3++;
}
}
//real conversion
l3-=4;
*l3=*l3-c*(l0)/2;
*l0=-c2*(l0)/2;
*l1=*l1+c*(l2)/2;
*l2=-c2*(l2)/2;
j++;
}
dl_o>>=1;
dl2_o>>=1;
kk<<=1;
}

//division with array length
size<<=1;
for(i=0;i<size;i++) data[i]=data[i]/size;
}

/////////////////////////////////////////////////////////////////
// Bruun FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),re(1),im(1),re(2),im(2),...,re(size/2-1),im(size/2-1)
// normalized by array length
//
// Source: see the routines it calls ...
void realfft_bruun_unshuffled(double *data, long size){

double *data2;
long i,j,k;

realfft_bruun(data,size);
//unshuffling - cannot be done in-place (?)
data2=(double *)malloc(size*sizeof(double));
for(i=1,k=size>>1;i<k;i++){
    j=bruun_reverse(i,k);
    data2[j+j]=data[i+i];
    data2[j+j+1]=data[i+i+1];
}
for(i=2;i<size;i++) data[i]=data2[i];
free(data2);
}

/////////////////////////////////////////////////////////////////
// Bruun FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),...,re(i),im(i)... pairs in
// "bruun-reversed" order
// normalized by array length
//
// Source:
// Bruun: z-Transform DFT Filters and FFT's
// IEEE Trans. ASSP, ASSP-26, No. 1, February 1978
//
// Comments:
// (this version is implemented in a row-by-row manner;
// control structure is simpler, but there are too
// much cosine calls - with a cosine lookup table
// probably this would be slightly faster than bruun_fft

/*void realfft_bruun2(double *data, long size){

double *end, *l0, *l1, *l2, *l3;
long dl, dl2, i, j;
double d0,d1,d2,d3,c,c2,p4;

end=data+size;
p4=pi/(size);
size>>=1;
dl=size;
dl2=dl/2;

```

```

//first filtering, when there're only two taps
for(;d1>1;d1>=>=1,d12>=>=1){
    l0=data;
    l3=data+d1;
    for(i=0;i<d1;i++){
        d0=*l0;
        d2=*l3;
        *l0=d0+d2;
        *l3=d0-d2;
        l0++;
        l3++;
    }
    //the remaining filterings
    j=1;
    while(l3<end){
        l0=l3;l1=l0+d12;l2=l0+d1;l3=l1+d1;
        c=2*cos(p4*bruun_reverse(j,size));
        for(i=0;i<d12;i++){
            d0=*l0;
            d1=*l1;
            d2=*l2;
            d3=*l3;
            *l0=d0+c*d1+d2;
            *l2=d0-c*d1+d2;
            *l1=d1+c*d2+d3;
            *l3=d1-c*d2+d3;
            l0++;
            l1++;
            l2++;
            l3++;
        }
        j++;
    }
}

//the last row: transform of real data
//the first two cells
l0=data;l1=data+1;
d0=*l0;d1=*l1;
*l0=d0+d1;*l1=d0-d1;
l1+=2;
*l1=-(*l1);
l0+=4;l1+=2;
//the remaining cells
j=1;
while(l0<end){
    c=p4*bruun_reverse(j,size);
    c2=sin(c);
    c=cos(c);
    *l0=*l0-c*( *l1);
    *l1=-c2*( *l1);
    l0+=2;
    l1+=2;
    *l0=*l0+c*( *l1);
    *l1=-c2*( *l1);
    l0+=2;
    l1+=2;
    j++;
}
//division with array length
for(i=0;i<size;i++) data[i]=data[i]/size;
}
*/

//the same as realfft_bruun_unshuffled,
//but calls realfft_bruun2
/*void realfft_bruun_unshuffled2(double *data, long size){

double *data2;
long i,j,k;

realfft_bruun2(data,size);
//unshuffling - cannot be done in-place (?)
data2=(double *)malloc(size*sizeof(double));
for(i=1,k=size>>1;i<k;i++){
    j=bruun_reverse(i,k);
    data2[j+j]=data[i+i];
    data2[j+j+1]=data[i+i+1];
}
for(i=2;i<size;i++) data[i]=data2[i];
free(data2);
}*/

```

```

////////////////////////////////////
// Sorensen in-place split-radix FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(1),re(2),...,re(size/2),im(size/2-1),...,im(1)
// normalized by array length
//
// Source:
// Sorensen et al: Real-Valued Fast Fourier Transform Algorithms,
// IEEE Trans. ASSP, ASSP-35, No. 6, June 1987

void realfft_split(double *data,long n){

    long i,j,k,i5,i6,i7,i8,i0,id,i1,i2,i3,i4,n2,n4,n8;
    double t1,t2,t3,t4,t5,t6,a3,ss1,ss3,cc1,cc3,a,e,sqrt2;

    sqrt2=sqrt(2.0);
    n4=n-1;

    //data shuffling
    for (i=0,j=0,n2=n/2; i<n4 ; i++){
        if (i<j){
            t1=data[j];
            data[j]=data[i];
            data[i]=t1;
        }
        k=n2;
        while (k<=j){
            j-=k;
            k>>=1;
        }
        j+=k;
    }

    /*-----*/

    //length two butterflies
    i0=0;
    id=4;
    do{
        for (; i0<n4; i0+=id){
            i1=i0+1;
            t1=data[i0];
            data[i0]=t1+data[i1];
            data[i1]=t1-data[i1];
        }
        id<<=1;
        i0=id-2;
        id<<=1;
    } while ( i0<n4 );

    /*-----*/
    //L shaped butterflies
    n2=2;
    for(k=n;k>2;k>>=1){
        n2<<=1;
        n4=n2>>2;
        n8=n2>>3;
        e = 2*pi/(n2);
        i1=0;
        id=n2<<1;
        do{
            for (; i1<n; i1+=id){
                i2=i1+n4;
                i3=i2+n4;
                i4=i3+n4;
                t1=data[i4]+data[i3];
                data[i4]-=data[i3];
                data[i3]=data[i1]-t1;
                data[i1]+=t1;
                if (n4!=1){
                    i0=i1+n8;
                    i2+=n8;
                    i3+=n8;
                    i4+=n8;
                    t1=(data[i3]+data[i4])/sqrt2;
                    t2=(data[i3]-data[i4])/sqrt2;
                    data[i4]=data[i2]-t1;
                    data[i3]=-data[i2]-t1;
                    data[i2]=data[i0]-t2;
                    data[i0]+=t2;
                }
            }
        } while (i1<n);
    }
}

```

```

    }
    }
    id<<=1;
    i1=id-n2;
    id<<=1;
    } while ( i1<n );
a=e;
for (j=2; j<=n8; j++){
    a3=3*a;
    cc1=cos(a);
    ss1=sin(a);
    cc3=cos(a3);
    ss3=sin(a3);
    a=j*e;
    i=0;
    id=n2<<1;
    do{
    for (; i<n; i+=id){
    i1=i+j-1;
    i2=i1+n4;
    i3=i2+n4;
    i4=i3+n4;
    i5=i+n4-j+1;
    i6=i5+n4;
    i7=i6+n4;
    i8=i7+n4;
    t1=data[i3]*cc1+data[i7]*ss1;
    t2=data[i7]*cc1-data[i3]*ss1;
    t3=data[i4]*cc3+data[i8]*ss3;
    t4=data[i8]*cc3-data[i4]*ss3;
    t5=t1+t3;
    t6=t2+t4;
    t3=t1-t3;
    t4=t2-t4;
    t2=data[i6]+t6;
    data[i3]=t6-data[i6];
    data[i8]=t2;
    t2=data[i2]-t3;
    data[i7]=-data[i2]-t3;
    data[i4]=t2;
    t1=data[i1]+t5;
    data[i6]=data[i1]-t5;
    data[i1]=t1;
    t1=data[i5]+t4;
    data[i5]=-t4;
    data[i2]=t1;
    }
    id<<=1;
    i=id-n2;
    id<<=1;
    } while(i<n);
    }
}

//division with array length
for(i=0;i<n;i++) data[i]/=n;
}

////////////////////////////////////
// Sorensen in-place split-radix FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),re(1),im(1),re(2),im(2),...,re(size/2-1),im(size/2-1)
// normalized by array length
//
// Source:
// Source: see the routines it calls ...

void realfft_split_unshuffled(double *data,long n){
    double *data2;
    long i,j;

    realfft_split(data,n);
    //unshuffling - not in-place
    data2=(double *)malloc(n*sizeof(double));
    j=n/2;
    data2[0]=data[0];
    data2[1]=data[j];
    for(i=1;i<j;i++) {data2[i+i]=data[i];data2[i+i+1]=data[n-i];}

```



```

for(i=0;i<n;i++) data[i]=data2[i];
free(data2);
}

////////////////////////////////////
// Sorensen in-place inverse split-radix FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size/2),im(size/2-1),...,im(1)
//
// output:
// re(0),re(1),re(2),...,re(size-1)
// NOT normalized by array length
//
// Source:
// Sorensen et al: Real-Valued Fast Fourier Transform Algorithms,
// IEEE Trans. ASSP, ASSP-35, No. 6, June 1987

void irealfft_split(double *data,long n){

    long i,j,k,i5,i6,i7,i8,i0,id,i1,i2,i3,i4,n2,n4,n8,n1;
    double t1,t2,t3,t4,t5,a3,ss1,ss3,cc1,cc3,a,e,sqrt2;

    sqrt2=sqrt(2.0);

n1=n-1;
n2=n<<1;
for(k=n;k>2;k>>=1){
    id=n2;
    n2>>=1;
    n4=n2>>2;
    n8=n2>>3;
    e = 2*pi/(n2);
    i1=0;
    do{
        for (; i1<n; i1+=id){
            i2=i1+n4;
            i3=i2+n4;
            i4=i3+n4;
            t1=data[i1]-data[i3];
            data[i1]+=data[i3];
            data[i2]*=2;
            data[i3]=t1-2*data[i4];
            data[i4]=t1+2*data[i4];
            if (n4!=1){
                i0=i1+n8;
                i2+=n8;
                i3+=n8;
                i4+=n8;
                t1=(data[i2]-data[i0])/sqrt2;
                t2=(data[i4]+data[i3])/sqrt2;
                data[i0]+=data[i2];
                data[i2]=data[i4]-data[i3];
                data[i3]=2*(-t2-t1);
                data[i4]=2*(-t2+t1);
            }
        }
        id<<=1;
        i1=id-n2;
        id<<=1;
    } while ( i1<n1 );
a=e;
for (j=2; j<=n8; j++){
    a3=3*a;
    cc1=cos(a);
    ss1=sin(a);
    cc3=cos(a3);
    ss3=sin(a3);
    a=j*e;
    i=0;
    id=n2<<1;
    do{
        for (; i<n; i+=id){
            i1=i+j-1;
            i2=i1+n4;
            i3=i2+n4;
            i4=i3+n4;
            i5=i+n4-j+1;
            i6=i5+n4;
            i7=i6+n4;
            i8=i7+n4;
            t1=data[i1]-data[i6];
            data[i1]+=data[i6];
            t2=data[i5]-data[i2];

```

```

    data[i5]+=data[i2];
    t3=data[i8]+data[i3];
    data[i6]=data[i8]-data[i3];
    t4=data[i4]+data[i7];
    data[i2]=data[i4]-data[i7];
    t5=t1-t4;
    t1+=t4;
    t4=t2-t3;
    t2+=t3;
    data[i3]=t5*cc1+t4*ss1;
    data[i7]=-t4*cc1+t5*ss1;
    data[i4]=t1*cc3-t2*ss3;
    data[i8]=t2*cc3+t1*ss3;
    }
    id<=1;
    i=id-n2;
    id<=1;
} while(i<n1);
}

/*-----*/
i0=0;
id=4;
do{
    for (; i0<n1; i0+=id){
        i1=i0+1;
        t1=data[i0];
        data[i0]=t1+data[i1];
        data[i1]=t1-data[i1];
    }
    id<=1;
    i0=id-2;
    id<=1;
} while ( i0<n1 );

/*-----*/

//data shuffling
for (i=0,j=0,n2=n/2; i<n1 ; i++){
    if (i<j){
        t1=data[j];
        data[j]=data[i];
        data[i]=t1;
    }
    k=n2;
    while (k<=j){
        j-=k;
        k>>=1;
    }
    j+=k;
}

////////////////////////////////////
// Sorensen in-place radix-2 FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(1),re(2),...,re(size/2),im(size/2-1),...,im(1)
// normalized by array length
//
// Source:
// Sorensen et al: Real-Valued Fast Fourier Transform Algorithms,
// IEEE Trans. ASSP, ASSP-35, No. 6, June 1987

void realfft_radix2(double *data,long n){

    double xt,a,e, t1, t2, cc, ss;
    long i, j, k, n1, n2, n3, n4, i1, i2, i3, i4;

n4=n-1;
//data shuffling
for (i=0,j=0,n2=n/2; i<n4 ; i++){
    if (i<j){
        xt=data[j];
        data[j]=data[i];
        data[i]=xt;
    }
    k=n2;
    while (k<=j){

```

```

        j-=k;
        k>>=1;
    }
    j+=k;
}

/* ----- */
    for (i=0; i<n; i += 2)
    {
        xt = data[i];
        data[i] = xt + data[i+1];
        data[i+1] = xt - data[i+1];
    }
/* ----- */
    n2 = 1;
    for (k=n;k>2;k>>=1){
        n4 = n2;
        n2 = n4 << 1;
        n1 = n2 << 1;
        e = 2*pi/(n1);
        for (i=0; i<n; i+=n1){
            xt = data[i];
            data[i] = xt + data[i+n2];
            data[i+n2] = xt-data[i+n2];
            data[i+n4+n2] = -data[i+n4+n2];
            a = e;
            n3=n4-1;
            for (j = 1; j <=n3; j++){
                i1 = i+j;
                i2 = i - j + n2;
                i3 = i1 + n2;
                i4 = i - j + n1;
                cc = cos(a);
                ss = sin(a);
                a += e;
                t1 = data[i3] * cc + data[i4] * ss;
                t2 = data[i3] * ss - data[i4] * cc;
                data[i4] = data[i2] - t2;
                data[i3] = -data[i2] - t2;
                data[i2] = data[i1] - t1;
                data[i1] += t1;
            }
        }
    }

//division with array length
    for(i=0;i<n;i++) data[i]/=n;
}

////////////////////////////////////
// Sorensen in-place split-radix FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),re(1),im(1),re(2),im(2),...,re(size/2-1),im(size/2-1)
// normalized by array length
//
// Source:
// Source: see the routines it calls ...

void realfft_radix2_unshuffled(double *data,long n){

    double *data2;
    long i,j;

    //unshuffling - not in-place
    realfft_radix2(data,n);
    data2=(double *)malloc(n*sizeof(double));
    j=n/2;
    data2[0]=data[0];
    data2[1]=data[j];
    for(i=1;i<j;i++) {data2[i+i]=data[i];data2[i+i+1]=data[n-i];}
    for(i=0;i<n;i++) data[i]=data2[i];
    free(data2);
}

```

[FFT classes in C++ and Object Pascal](#) (click this to go back to the index)

Type : Real-to-Complex FFT and Complex-to-Real IFFT

References : Laurent de Soras (Object Pascal translation by Frederic Vanmol)

Linked file : [FFTReal.zip](#)

Notes :
(see linkfile)

Comments

from : ms_shirbiny [[a t]] hotmail.com

comment : the file doesn't exist

Frequency response from biquad coefficients (click this to go back to the index)

Type : biquad

References : Posted by peter[AT]sonicreef[DOT]com

Notes :

Here is a formula for plotting the frequency response of a biquad filter. Depending on the coefficients that you have, you might have to use negative values for the b- coefficients.

Code :

```
//w = frequency (0 < w < PI)
//square(x) = x*x

y = 20*log( (sqrt( square(a0*square(cos(w))-
a0*square(sin(w))+a1*cos(w)+a2)+square(2*a0*cos(w)*sin(w)+a1*(sin(w))) ) /
sqrt( square( square(cos(w))- square(sin(w))+b1*cos(w)+b2)+square(2* cos(w)*sin(w)+b1*(sin(w))) ) ) );
```

Comments

from : Sergey

comment : There is a better formula of a higher order by George Yohng available on this site

from : dfl [[a t]] ccrma.stanford.edu

comment : this formula can have roundoff errors with frequencies close to zero... (especially a problem with high samplerate filters)

here is a better formula:

from RBJ @ http://groups.google.com/group/comp.dsp/browse_frm/thread/8c0fa8d396aeb444/a1bc5b63ac56b686

```
20*log10[|H(e^jw)|] =
10*log10[ (b0+b1+b2)^2 - 4*(b0*b1 + 4*b0*b2 + b1*b2)*phi + 16*b0*b2*phi^2 ]
-10*log10[ (a0+a1+a2)^2 - 4*(a0*a1 + 4*a0*a2 + a1*a2)*phi + 16*a0*a2*phi^2 ]
```

where $\phi = \sin^2(w/2)$

Java FFT (click this to go back to the index)

Type : FFT Analysis

References : Posted by Lorenzo Heer

Notes :

May not work correctly ;-)

Code :

```
// WTest.java
/*
   Copyright (C) 2003 Lorenzo Heer, (helohe at bluewin dot ch)

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/
public class WTest{

    private static double[] sin(double step, int size){
        double f = 0;
        double[] ret = new double[size];
        for(int i = 0; i < size; i++){
            ret[i] = Math.sin(f);
            f += step;
        }
        return ret;
    }

    private static double[] add(double[] a, double[] b){
        double[] c = new double[a.length];
        for(int i = 0; i < a.length; i++){
            c[i] = a[i] + b[i];
        }
        return c;
    }

    private static double[] sub(double[] a, double[] b){
        double[] c = new double[a.length];
        for(int i = 0; i < a.length; i++){
            c[i] = a[i] - b[i];
        }
        return c;
    }

    private static double[] add(double[] a, double b){
        double[] c = new double[a.length];
        for(int i = 0; i < a.length; i++){
            c[i] = a[i] + b;
        }
        return c;
    }

    private static double[] cp(double[] a, int size){
        double[] c = new double[size];
        for(int i = 0; i < size; i++){
            c[i] = a[i];
        }
        return c;
    }

    private static double[] mul(double[] a, double b){
        double[] c = new double[a.length];
        for(int i = 0; i < a.length; i++){
            c[i] = a[i] * b;
        }
        return c;
    }

    private static void print(double[] value){
        for(int i = 0; i < value.length; i++){
            System.out.print(i + ", " + value[i] + "\n");
        }
        System.out.println();
    }

    private static double abs(double[] a){
        double c = 0;
        for(int i = 0; i < a.length; i++){
```

```

    c = ((c * i) + Math.abs(a[i])) / (i + 1);
}
return c;
}

private static double[] fft(double[] a, int min, int max, int step){
double[] ret = new double[(max - min) / step];
int i = 0;
for(int d = min; d < max; d = d + step){
double[] f = sin(fc(d), a.length);
double[] dif = sub(a, f);
ret[i] = 1 - abs(dif);
i++;
}
return ret;
}

private static double[] fft_log(double[] a){
double[] ret = new double[1551];
int i = 0;
for(double d = 0; d < 15.5; d = d + 0.01){
double[] f = sin(fc(Math.pow(2,d)), a.length);
double[] dif = sub(a, f);
ret[i] = Math.abs(1 - abs(dif));
i++;
}
return ret;
}

private static double fc(double d){
return d * Math.PI / res;
}

private static void print_log(double[] value){
for(int i = 0; i < value.length; i++){
System.out.print(Math.pow(2,((double)i/100d)) + "," + value[i] + "\n");
}
System.out.println();
}

public static void main(String[] args){
double[] f_0 = sin(fc(440), sample_length); // res / pi =>14005
//double[] f_1 = sin(.02, sample_length);
double[] f_2 = sin(fc(520), sample_length);
//double[] f_3 = sin(.25, sample_length);

//double[] f = add( add( add(f_0, f_1), f_2), f_3);

double[] f = add(f_0, f_2);

//print(f);

double[] d = cp(f,1000);
print_log(fft_log(d));
}

static double length = .2; // sec
static int res = 44000; // resoultion (pro sec)
static int sample_length = res; // resoultion
}

```

Comments

from : eamonk2 [[a t]] yahoo.co.uk

comment : How does this work? Where can I enter an array of bytes/doubles and where does it return the fft of that array? Also if I have an array of bytes of sound data captured from the microphone(say plucking a guitar string), and perform fft on the array, is the average of that array the frequency of the note? Please reply as soon as you can. Thanks

from : anon

comment : For a comprehensive suite of Java transforms with FFTW-level performance see <http://piotr.wendykier.googlepages.com/jtransforms>

Look ahead limiting (click this to go back to the index)

References : Posted by Wilfried Welti

Notes :

use `add_value` with all values which enter the look-ahead area, and `remove_value` with all value which leave this area. to get the maximum value in the look-ahead area, use `get_max_value`. in the very beginning initialize the table with zeroes.

If you always want to know the maximum amplitude in your look-ahead area, the thing becomes a sorting problem. very primitive approach using a look-up table

Code :

```
void lookup_add(unsigned section, unsigned size, unsigned value)
{
    if (section==value)
        lookup[section]++;
    else
    {
        size >>= 1;
        if (value>section)
        {
            lookup[section]++;
            lookup_add(section+size,size,value);
        }
        else
            lookup_add(section-size,size,value);
    }
}

void lookup_remove(unsigned section, unsigned size, unsigned value)
{
    if (section==value)
        lookup[section]--;
    else
    {
        size >>= 1;
        if (value>section)
        {
            lookup[section]--;
            lookup_remove(section+size,size,value);
        }
        else
            lookup_remove(section-size,size,value);
    }
}

unsigned lookup_getmax(unsigned section, unsigned size)
{
    unsigned max = lookup[section] ? section : 0;
    size >>= 1;
    if (size)
        if (max)
        {
            max = lookup_getmax((section+size),size);
            if (!max) max=section;
        }
        else
            max = lookup_getmax((section-size),size);
    return max;
}

void add_value(unsigned value)
{
    lookup_add(LOOKUP_VALUES>>1, LOOKUP_VALUES>>1, value);
}

void remove_value(unsigned value)
{
    lookup_remove(LOOKUP_VALUES>>1, LOOKUP_VALUES>>1, value);
}

unsigned get_max_value()
{
    return lookup_getmax(LOOKUP_VALUES>>1, LOOKUP_VALUES>>1);
}
```


LPC analysis (autocorrelation + Levinson-Durbin recursion) (click this to go back to the index)

References : Posted by mail[AT]mutagene[DOT]net

Notes :
The autocorrelation function implements a warped autocorrelation, so that frequency resolution can be specified by the variable 'lambda'. Levinson-Durbin recursion calculates autoregression coefficients a and reflection coefficients (for lattice filter implementation) K. Comments for Levinson-Durbin function implement matlab version of the same function.

No optimizations.

```
Code :
//find the order-P autocorrelation array, R, for the sequence x of length L and warping of lambda
//wAutocorrelate(&pfSrc[stIndex],siglen,R,P,0);
wAutocorrelate(float * x, unsigned int L, float * R, unsigned int P, float lambda)
{
    double * dl = new double [L];
    double * Rt = new double [L];
    double r1,r2,r1t;
    R[0]=0;
    Rt[0]=0;
    r1=0;
    r2=0;
    r1t=0;
    for(unsigned int k=0; k<L;k++)
    {
        Rt[0]+=double(x[k])*double(x[k]);

        dl[k]=r1-double(lambda)*double(x[k]-r2);
        r1 = x[k];
        r2 = dl[k];
    }
    for(unsigned int i=1; i<=P; i++)
    {
        Rt[i]=0;
        r1=0;
        r2=0;
        for(unsigned int k=0; k<L;k++)
        {
            Rt[i]+=double(dl[k])*double(x[k]);

            r1t = dl[k];
            dl[k]=r1-double(lambda)*double(r1t-r2);
            r1 = r1t;
            r2 = dl[k];
        }
    }
    for(i=0; i<=P; i++)
        R[i]=float(Rt[i]);
    delete[] dl;
    delete[] Rt;
}

// Calculate the Levinson-Durbin recursion for the autocorrelation sequence R of length P+1 and return the
autocorrelation coefficients a and reflection coefficients K
LevinsonRecursion(unsigned int P, float *R, float *A, float *K)
{
    double Aml[62];

    if(R[0]==0.0) {
        for(unsigned int i=1; i<=P; i++)
        {
            K[i]=0.0;
            A[i]=0.0;
        }
        else {
            double km,Em1,Em;
            unsigned int k,s,m;
            for (k=0;k<=P;k++){
                A[0]=0;
                Aml[0]=0; }
            A[0]=1;
            Aml[0]=1;
            km=0;
            Em1=R[0];
            for (m=1;m<=P;m++) //m=2:N+1
            {
                double err=0.0f; //err = 0;
                for (k=1;k<=m-1;k++) //for k=2:m-1
                    err += Aml[k]*R[m-k]; // err = err + aml(k)*R(m-k+1);
                km = (R[m]-err)/Em1; //km=(R(m)-err)/Em1;
                K[m-1] = -float(km);
                A[m]=(float)km; //am(m)=km;
                for (k=1;k<=m-1;k++) //for k=2:m-1
                    A[k]=float(Aml[k]-km*Aml[m-k]); // am(k)=aml(k)-km*aml(m-k+1);
                Em=(1-km*km)*Em1; //Em=(1-km*km)*Em1;
                for(s=0;s<=P;s++) //for s=1:N+1
                    Aml[s] = A[s]; // aml(s) = am(s)
                Em1 = Em; //Em1 = Em;
            }
        }
    }
}
```

```

return 0;
}

```

Comments

from : abhimj [[a t]] lycos.com
comment : Hi, I the LPC coeffs are to be converted to cepstral coeffs, the formula for cepstral coeffs requires some 'gain term' that is calculated in the LPC analysis phase. Can you please tell how to get this gain term?

from : Christian [[a t]] savioursofsoul.de
comment : Blind Object Pascal Translation:

```

unit Levinson;

```

```

interface

```

```

type

```

```

TDoubleArray = array of Double;
TSingleArray = array of Single;

```

```

implementation

```

```

//find the P-order autocorrelation array, R, for the sequence x of length L and warping of lambda

```

```

procedure Autocorrelate(x,R : TSingleArray; P : Integer; lambda : Single; l: Integer = -1);

```

```

var dl,Rt : TDoubleArray;

```

```

r1,r2,r1t : Double;

```

```

k,i : Integer;

```

```

begin

```

```

// Initialization

```

```

if l=-1 then l:=Length(x);

```

```

SetLength(dl,l);

```

```

SetLength(Rt,l);

```

```

R[0]:=0;

```

```

Rt[0]:=0;

```

```

r1:=0;

```

```

r2:=0;

```

```

r1t:=0;

```

```

for k:=0 to l-1 do

```

```

begin

```

```

Rt[0]:=Rt[0]+x[k]*x[k];

```

```

dl[k]:=r1-lambda*(x[k]-r2);

```

```

r1:= x[k];

```

```

r2:= dl[k];

```

```

end;

```

```

for i:=1 to P do

```

```

begin

```

```

Rt[i]:=0;

```

```

r1:=0;

```

```

r2:=0;

```

```

for k:=0 to l-1 do

```

```

begin

```

```

Rt[i]:=Rt[i]+dl[k]*x[k];

```

```

r1t:= dl[k];

```

```

dl[k]:=r1-lambda*(r1t-r2);

```

```

r1:=r1t;

```

```

r2:=dl[k];

```

```

end;

```

```

end;

```

```

for i:=1 to P do R[i]:=Rt[i];

```

```

setlength(Rt,0);

```

```

setlength(dl,0);

```

```

end;

```

```

// Calculate the Levinson-Durbin recursion for the autocorrelation sequence

```

```

// R of length P+1 and return the autocorrelation coefficients a and reflection coefficients K

```

```

procedure LevinsonRecursion(P : Integer; R,A,K : TSingleArray);

```

```

var Am1 : TDoubleArray;

```

```

i,j,s,m : Integer;

```

```

km,Em1,Em : Double;

```

```

err : Double;

```

```

begin

```

```

SetLength(Am1,62);

```

```

if (R[0]=0.0) then

```

```

begin

```

```

for i:=1 to P do

```

```

begin

```

```

K[i]:=0.0;

```

```

A[i]:=0.0;

```

```

end;

```

```

end

```

```

else

```

```
begin
for j:=0 to P do
begin
A[0]:=0;
Am1[0]:=0;
end;
A[0]:=1;
Am1[0]:=1;
km:=0;
Em1:=R[0];
for m:=1 to P do
begin
err:=0.0;
for j:=1 to m-1 do err:=err+Am1[j]*R[m-j];
km:=(R[m]-err)/Em1;
K[m-1]:=-km;
A[m]:=km;
for j:=1 to m-1 do A[j]:=Am1[j]-km*Am1[m-j];
Em:=(1-km*km)*Em1;
for s:=0 to P do Am1[s]:=A[s];
Em1:=Em;
end;
end;
end;
```

from : jamie [[a t]] postlude.co.uk
comment : Hi,

This loop:

```
for (k=0;k<=P;k++){
A[0]=0;
Am1[0]=0; }
```

Looks like it shouldn't be there?

Magnitude and phase plot of arbitrary IIR function, up to 5th order (click this to go back to the index)

Type : magnitude and phase at any frequency

References : Posted by George Yohng

Notes :

Amplitude and phase calculation of IIR equation run at sample rate "sampleRate" at frequency "F".

AMPLITUDE

```
-----  
cf_mag(F,sampleRate,  
       a0,a1,a2,a3,a4,a5,  
       b0,b1,b2,b3,b4,b5)  
-----
```

PHASE

```
-----  
cf_phi(F,sampleRate,  
       a0,a1,a2,a3,a4,a5,  
       b0,b1,b2,b3,b4,b5)  
-----
```

If you need a frequency diagram, draw a plot for
F=0...sampleRate/2

If you need amplitude in dB, use cf_lin2db(cf_mag(.....))

Set b0=-1 if you have such function:

$$y[n] = a0*x[n] + a1*x[n-1] + a2*x[n-2] + a3*x[n-3] + a4*x[n-4] + a5*x[n-5] + b1*y[n-1] + b2*y[n-2] + b3*y[n-3] + b4*y[n-4] + b5*y[n-5];$$

Set b0=1 if you have such function:

$$y[n] = a0*x[n] + a1*x[n-1] + a2*x[n-2] + a3*x[n-3] + a4*x[n-4] + a5*x[n-5] - b1*y[n-1] - b2*y[n-2] - b3*y[n-3] - b4*y[n-4] - b5*y[n-5];$$

Do not try to reverse engineer these formulae - they don't give any sense other than they are derived from transfer function, and they work. :)

Code :

```
/*  
 C file can be downloaded from  
 http://www.yohng.com/dsp/cfsmp.c  
*/  
  
#define C_PI 3.14159265358979323846264  
  
double cf_mag(double f,double rate,  
             double a0,double a1,double a2,double a3,double a4,double a5,  
             double b0,double b1,double b2,double b3,double b4,double b5)  
{  
    return  
    sqrt((a0*a0 + a1*a1 + a2*a2 + a3*a3 + a4*a4 + a5*a5 +  
          2*(a0*a1 + a1*a2 + a2*a3 + a3*a4 + a4*a5)*cos((2*f*C_PI)/rate) +  
          2*(a0*a2 + a1*a3 + a2*a4 + a3*a5)*cos((4*f*C_PI)/rate) +  
          2*a0*a3*cos((6*f*C_PI)/rate) + 2*a1*a4*cos((6*f*C_PI)/rate) +  
          2*a2*a5*cos((6*f*C_PI)/rate) + 2*a0*a4*cos((8*f*C_PI)/rate) +  
          2*a1*a5*cos((8*f*C_PI)/rate) + 2*a0*a5*cos((10*f*C_PI)/rate))/  
          (b0*b0 + b1*b1 + b2*b2 + b3*b3 + b4*b4 + b5*b5 +  
          2*(b0*b1 + b1*b2 + b2*b3 + b3*b4 + b4*b5)*cos((2*f*C_PI)/rate) +  
          2*(b0*b2 + b1*b3 + b2*b4 + b3*b5)*cos((4*f*C_PI)/rate) +  
          2*b0*b3*cos((6*f*C_PI)/rate) + 2*b1*b4*cos((6*f*C_PI)/rate) +  
          2*b2*b5*cos((6*f*C_PI)/rate) + 2*b0*b4*cos((8*f*C_PI)/rate) +  
          2*b1*b5*cos((8*f*C_PI)/rate) + 2*b0*b5*cos((10*f*C_PI)/rate)));  
}  
  
double cf_phi(double f,double rate,  
            double a0,double a1,double a2,double a3,double a4,double a5,  
            double b0,double b1,double b2,double b3,double b4,double b5)  
{  
    atan2((a0*b0 + a1*b1 + a2*b2 + a3*b3 + a4*b4 + a5*b5 +  
           (a0*b1 + a1*(b0 + b2) + a2*(b1 + b3) + a5*b4 + a3*(b2 + b4) +  
           a4*(b3 + b5))*cos((2*f*C_PI)/rate) +  
           ((a0 + a4)*b2 + (a1 + a5)*b3 + a2*(b0 + b4) +  
           a3*(b1 + b5))*cos((4*f*C_PI)/rate) + a3*b0*cos((6*f*C_PI)/rate) +  
           a4*b1*cos((6*f*C_PI)/rate) + a5*b2*cos((6*f*C_PI)/rate) +  
           a0*b3*cos((6*f*C_PI)/rate) + a1*b4*cos((6*f*C_PI)/rate) +  
           a2*b5*cos((6*f*C_PI)/rate) + a4*b0*cos((8*f*C_PI)/rate) +  
           a5*b1*cos((8*f*C_PI)/rate) + a0*b4*cos((8*f*C_PI)/rate) +  
           a1*b5*cos((8*f*C_PI)/rate) +  
           (a5*b0 + a0*b5)*cos((10*f*C_PI)/rate))/  
          (a0*a0 + a1*a1 + a2*a2 + a3*a3 + a4*a4 + a5*a5 +  
          2*(a0*a1 + a1*a2 + a2*a3 + a3*a4 + a4*a5)*cos((2*f*C_PI)/rate) +  
          2*(a0*a2 + a1*a3 + a2*a4 + a3*a5)*cos((4*f*C_PI)/rate) +  
          2*a0*a3*cos((6*f*C_PI)/rate) + 2*a1*a4*cos((6*f*C_PI)/rate) +  
          2*a2*a5*cos((6*f*C_PI)/rate) + 2*a0*a4*cos((8*f*C_PI)/rate) +  
          2*a1*a5*cos((8*f*C_PI)/rate) + 2*a0*a5*cos((10*f*C_PI)/rate));  
}
```

```

(b0*b0 + b1*b1 + b2*b2 + b3*b3 + b4*b4 + b5*b5 +
2*((b0*b1 + b1*b2 + b2*(b2 + b4) + b4*b5)*cos((2*f*C_PI)/rate) +
(b2*(b0 + b4) + b3*(b1 + b5))*cos((4*f*C_PI)/rate) +
(b0*b3 + b1*b4 + b2*b5)*cos((6*f*C_PI)/rate) +
(b0*b4 + b1*b5)*cos((8*f*C_PI)/rate) +
b0*b5*cos((10*f*C_PI)/rate)),

((a1*b0 + a3*b0 + a5*b0 - a0*b1 + a2*b1 + a4*b1 - a1*b2 +
a3*b2 + a5*b2 - a0*b3 - a2*b3 + a4*b3 -
a1*b4 - a3*b4 + a5*b4 - a0*b5 - a2*b5 - a4*b5 +
2*(a3*b1 + a5*b1 - a0*b2 + a4*(b0 + b2) - a1*b3 + a5*b3 +
a2*(b0 - b4) - a0*b4 - a1*b5 - a3*b5)*cos((2*f*C_PI)/rate) +
2*(a3*b0 + a4*b1 + a5*(b0 + b2) - a0*b3 - a1*b4 - a0*b5 - a2*b5)*
cos((4*f*C_PI)/rate) + 2*a4*b0*cos((6*f*C_PI)/rate) +
2*a5*b1*cos((6*f*C_PI)/rate) - 2*a0*b4*cos((6*f*C_PI)/rate) -
2*a1*b5*cos((6*f*C_PI)/rate) + 2*a5*b0*cos((8*f*C_PI)/rate) -
2*a0*b5*cos((8*f*C_PI)/rate))*sin((2*f*C_PI)/rate))/
(b0*b0 + b1*b1 + b2*b2 + b3*b3 + b4*b4 + b5*b5 +
2*(b0*b1 + b1*b2 + b2*b3 + b3*b4 + b4*b5)*cos((2*f*C_PI)/rate) +
2*(b0*b2 + b1*b3 + b2*b4 + b3*b5)*cos((4*f*C_PI)/rate) +
2*b0*b3*cos((6*f*C_PI)/rate) + 2*b1*b4*cos((6*f*C_PI)/rate) +
2*b2*b5*cos((6*f*C_PI)/rate) + 2*b0*b4*cos((8*f*C_PI)/rate) +
2*b1*b5*cos((8*f*C_PI)/rate) + 2*b0*b5*cos((10*f*C_PI)/rate));
}

double cf_lin2db(double lin)
{
    if (lin<9e-51) return -1000; /* prevent invalid operation */
    return 20*log10(lin);
}

```

Comments

from :

comment : They don't appear to make any sense at all.

from : Rob

comment : Actually it is simpler to simply take the zero-padded b and a coefficients and do real->complex FFT like this (matlab code):

```

H_complex=fft(b,N)/fft(a,N);
phase=angle(H_complex);
Magn=abs(H_complex);

```

This will give you N/2 points from 0 to pi angle freq (or 0 to nyquist freq).

/Rob

from : Christian [[a t]] savioursofsoul.de

comment : Here are the formulas if you only have a biquad. But i am not sure, if maybe the phase is shifted with pi/2...

```

20*Log10(
    sqrt(
        (a0*a0+a1*a1+a2*a2+
        2*(a0*a1+a1*a2)*cos(w)+
        2*(a0*a2)*cos(2*w)
        )
    /
    (
        1 + b1*b1 + b2*b2 +
        2*(b1 + b1*b2)*cos(w)+
        2*b2*cos(2*w)
        )
    )
)

ArcTan2(
    (
        a0+a1*b1+a2*b2+
        (a0*b1+a1*(1+b2)+a2*b1)*cos(w)+
        (a0*b2+a2)*cos(2*w)
        )
    /
    (
        1+b1*b1+b2*b2+
        2*
        (
            (b1+b1*b2)*cos(w)+ b2*cos(2*w)
            )
        )
    )
    ,
    (
        (
            a1-a0*b1+a2*b1-a1*b2+
            2*(-a0*b2+a2)*cos(w)
            )
        )
    )
)*sin(w)

```

```

/
(
  1+b1*b1+b2*b2+
  2*(b1 + b1*b2)*cos(w)+
  2*b2*cos(2*w)
)
)
)

```

from : Christian [[a t]] savioursofsoul.de
comment : Same code, but (hopefully) better layout...

```

20*Log10(
  sqrt(
    (a0*a0+a1*a1+a2*a2+
      2*(a0*a1+a1*a2)*cos(w)+
      2*(a0*a2)*cos(2*w)
    )
  )
  /
  (
    1 + b1*b1 + b2*b2 +
    2*(b1 + b1*b2)*cos(w)+
    2*b2*cos(2*w)
  )
)
)

```

from : Christian [[a t]] savioursofsoul.de
comment : Recursive Delphi Code with arbitrary order:

```

unit Plot;

interface

type TArrayOfDouble = Array of Double;

function MagnitudeCalc(f,rate : Double; a,b : TArrayOfDouble): Double;

implementation

uses Math;

function MulVectCalc(const v: TArrayOfDouble; const Z, N : Integer) : Double;
begin
  if N=0
  then result:=0
  else result:=(v[N-1]*v[N-1+Z])+MulVectCalc(v,Z,N-1);
end;

function MagCascadeCalc(const v: TArrayOfDouble; const w : double; N, Order : Integer) : Double;
begin
  if N=1
  then result:=(MulVectCalc(v,0,Order))
  else result:=((MulVectCalc(v,N-1,1+Order-N)*(2*cos((N-1)*w))+MagCascadeCalc(v, w, N-1, Order)));
end;

function MagnitudeCalc(f,rate : Double; a,b : TArrayOfDouble): Double;
var w : Double;
begin
  w:=(2*f*pi)/rate;
  result:=sqrt(MagCascadeCalc(a, w, Length(a),Length(a))/MagCascadeCalc(b, w, Length(b),Length(b)));
end;

end.

```

from : John
comment : Surely no-one here would actually code 20*log10(sqrt(x)) instead of using 10*log10(x) ... :-)

```

from : Christian [ [ a t ] ] savioursofsoul.de
comment : function CalcMagPart(w: Double; C : TDoubleArray):Double;
var i,j,l: Integer;
    temp : Double;
begin
  l:=Length(C);
  temp:=0;
  for j:=0 to l-1
  do temp:=temp+C[j]*C[j];
  result:=temp;
  for i:=1 to l-1 do
  begin
    temp:=0;
    for j:=0 to l-i-1
    do temp:=temp+C[j]*C[j+i];
    result:=Result+2*temp*cos(i*w);
  end;
end;

```

end;

```
function CalcMagnitude_dB(const f,rate: Double; const A,B: TDoubleArray): Double;
var w : Double;
begin
  w:=(2*f*pi)/rate;
  result:=10*log10(CalcMagPart(w,A)/CalcMagPart(w,B));
end;
```

Here's a really fast function for an arbitrary IIR with high order without stack overflows or recursion.
And specially for John without sqrt.

from : nobody [[a t]] nowhere.com

comment : I wonder who are creating these comments and why, and why can't they be deleted.

from : notme [[a t]] somewhere.com

comment : Sometimes the nonsense spam will have links in it, but who knows, some kids might just do it for kicks when they're writing their silly little net spider-bots or whatever you call 'em.

Either way, someone with the PHP password for this site should do a quick half and hour and delete this things. (Including this message!)

:)

Measuring interpolation noise (click this to go back to the index)

References : Posted by Jon Watte

Notes :

You can easily estimate the error by evaluating the actual function and evaluating your interpolator at each of the mid-points between your samples. The absolute difference between these values, over the absolute value of the "correct" value, is your relative error. \log_{10} of your relative error times 20 is an estimate of your quantization noise in dB. Example:

You have a table for every 0.5 "index units". The value at index unit 72.0 is 0.995 and the value at index unit 72.5 is 0.999. The interpolated value at index 72.25 is 0.997. Suppose the actual function value at that point was 0.998; you would have an error of 0.001 which is a relative error of 0.001002004.. $\log_{10}(\text{error})$ is about -2.99913, which times 20 is about -59.98. Thus, that's your quantization noise at that position in the table. Repeat for each pair of samples in the table.

Note: I said "quantization noise" not "aliasing noise". The aliasing noise will, as far as I know, only happen when you start up-sampling without band-limiting and get frequency aliasing (wrap-around), and thus is mostly independent of what specific interpolation mechanism you're using.

QFT and DQFT (double precision) classes (click this to go back to the index)

References : Posted by Joshua Scholar

Linked file : [qft.tar_1.gz](#)

Notes :

Since it's a Visual C++ project (though it has relatively portable C++) I guess the main audience are PC users. As such I'm including a zip file. Some PC users wouldn't know what to do with a tgz file.

The QFT and DQFT (double precision) classes supply the following functions:

1. Real valued FFT and inverse FFT functions. Note that separate arrays are used for real and imaginary component of the resulting spectrum.
2. Decomposition of a spectrum into a separate spectrum of the even samples and a spectrum of the odd samples. This can be useful for building filter banks.
3. Reconstituting a spectrum from separate spectrums of the even samples and odd samples. This can be useful for building filter banks.
4. A discrete Sin transform (a QFT decomposes an FFT into a DST and DCT).
5. A discrete Cos transform.

6. Since a QFT does its last stage calculating from the outside in the last part can be left unpacked and only calculated as needed in the case where the entire spectrum isn't needed (I used this for calculating correlations and convolutions where I only needed half of the results).

ReverseNoUnpack()
UnpackStep()
and NegUnpackStep()
implement this functionality

NOTE Reverse() normalizes its results (divides by one half the blocklength), but ReverseNoUnpack() does not.

7. Also if you only want the first half of the results you can call ReverseHalf()

NOTE Reverse() normalizes its results (divides by one half the blocklength), but ReverseHalf() does not.

8. QFT is less numerically stable than regular FFTs. With single precision calculations, a block length of 2^{15} brings the accuracy down to being barely accurate enough. At that size, single precision calculations tested sound files would occasionally have a sample off by 2, and a couple off by 1 per block. Full volume white noise would generate a few samples off by as much as 6 per block at the end, beginning and middle.

No matter what the inputs the errors are always at the same positions in the block. There is some sort of cancellation that gets more delicate as the block size gets bigger.

For the sake of doing convolutions and the like where the forward transform is done only once for one of the inputs, I created an AccurateForward() function. It uses a regular FFT algorithm for blocks larger than 2^{12} , and decomposes into even and odd FFTs recursively.

In any case you can always use the double precision routines to get more accuracy. DQFT even has routines that take floats as inputs and return double precision spectrum outputs.

As for portability:

1. The files qft.cpp and dqft.cpp start with defines:
`#define _USE_ASM`

If you comment those define out, then what's left is C++ with no assembly language.

2. There is unnecessary windows specific code in "criticalSection.h"
I used a critical section because objects are not reentrant (each object has permanent scratch pad memory), but obviously critical sections are operating system specific. In any case that code can easily be taken out.

If you look at my code and see that there's a test built in the examples that makes sure that the results are in the ballpark of being right. It wasn't that I expected the answers to be far off, it was that I un-commented the "no assembly language" versions of some routines and I wanted to make sure that they weren't broken.

[Simple peak follower](#) (click this to go back to the index)

Type : amplitude analysis

References : Posted by Phil Burk

Notes :
This simple peak follower will give track the peaks of a signal. It will rise rapidly when the input is rising, and then decay exponentially when the input drops. It can be used to drive VU meters, or used in an automatic gain control circuit.

```
Code :  
// halfLife = time in seconds for output to decay to half value after an impulse  
  
static float output = 0.0;  
  
float scalar = pow( 0.5, 1.0/(halfLife * sampleRate));  
  
if( input < 0.0 )  
    input = -input; /* Absolute value. */  
  
if ( input >= output )  
{  
    /* When we hit a peak, ride the peak to the top. */  
    output = input;  
}  
else  
{  
    /* Exponential decay of output when signal is low. */  
    output = output * scalar;  
    /*  
    ** When current gets close to 0.0, set current to 0.0 to prevent FP underflow  
    ** which can cause a severe performance degradation due to a flood  
    ** of interrupts.  
    */  
    if( output < VERY_SMALL_FLOAT ) output = 0.0;  
}
```

tone detection with Goertzel (click this to go back to the index)

Type : Goertzel

References : Posted by espenr[AT]ii[DOT]uib[DOT]no

Linked file : <http://www.ii.uib.no/~espenr/tonedetect.zip>

Notes :

Goertzel is basically DFT of parts of a spectrum not the total spectrum as you normally do with FFT. So if you just want to check out the power for some frequencies this could be better. Is good for DTFM detection I've heard.

The WNK isn't calculated 100% correctly, but it seems to work so ;) Yeah and the code is C++ so you might have to do some small adjustment to compile it as C.

Code :

```
/** Tone detect by Goertzel algorithm
 *
 * This program basically searches for tones (sines) in a sample and reports the different dB it finds for
 * different frequencies. Can easily be extended with some thresholding to report true/false on detection.
 * I'm far from certain goertzel it implemented 100% correct, but it works :)
 *
 * Hint, the SAMPLERATE, BUFFERSIZE, FREQUENCY, NOISE and SIGNALVOLUME all affects the outcome of the reported
 * dB. Tweak
 * em to find the settings best for your application. Also, seems to be pretty sensitive to noise (whitenoise
 * anyway) which
 * is a bit sad. Also I don't know if the goertzel really likes float values for the frequency ... And using
 * 44100 as
 * samplerate for detecting 6000 Hz tone is kinda silly I know :)
 *
 * Written by: Espen Riskedal, espenr@ii.uib.no, july-2002
 */

#include <iostream>
#include <cmath>
#include <cstdlib>

using std::rand;
// math stuff
using std::cos;
using std::abs;
using std::exp;
using std::log10;
// iostream stuff
using std::cout;
using std::endl;

#define PI 3.14159265358979323844
// change the defines if you want to
#define SAMPLERATE 44100
#define BUFFERSIZE 8820
#define FREQUENCY 6000
#define NOISE 0.05
#define SIGNALVOLUME 0.8

/** The Goertzel algorithm computes the k-th DFT coefficient of the input signal using a second-order filter.
 * http://ptolemy.eecs.berkeley.edu/papers/96/dtmf_ict/www/node3.html.
 * Basically it just does a DFT of the frequency we want to check, and none of the others (FFT calculates for
 * all frequencies).
 */
float goertzel(float *x, int N, float frequency, int samplerate) {
    float Skn, Skn1, Skn2;
    Skn = Skn1 = Skn2 = 0;

    for (int i=0; i<N; i++) {
        Skn2 = Skn1;
        Skn1 = Skn;
        Skn = 2*cos(2*PI*frequency/samplerate)*Skn1 - Skn2 + x[i];
    }

    float Wnk = exp(-2*PI*frequency/samplerate); // this one ignores complex stuff
    //float Wnk = exp(-2*j*PI*k/N);
    return (Skn - Wnk*Skn1);
}

/** Generates a tone of the specified frequency
 * Gotten from: http://groups.google.com/groups?hl=en&lr=&ie=UTF-8&oe=UTF-
 * 8&safe=off&selm=3c641e%243jn%40uicsl.csl.uiuc.edu
 */
float *makeTone(int samplerate, float frequency, int length, float gain=1.0) {
    //y(n) = 2 * cos(A) * y(n-1) - y(n-2)
    //A= (frequency of interest) * 2 * PI / (sampling frequency)
    //A is in radians.
    // frequency of interest MUST be <= 1/2 the sampling frequency.
    float *tone = new float[length];
    float A = frequency*2*PI/samplerate;

    for (int i=0; i<length; i++) {
        if (i > 1) tone[i]= 2*cos(A)*tone[i-1] - tone[i-2];
        else if (i > 0) tone[i] = 2*cos(A)*tone[i-1] - (cos(A));
    }
}
```

```

else tone[i] = 2*cos(A)*cos(A) - cos(2*A);
}

for (int i=0; i<length; i++) tone[i] = tone[i]*gain;

return tone;
}

/** adds whitenoise to a sample */
void *addNoise(float *sample, int length, float gain=1.0) {
for (int i=0; i<length; i++) sample[i] += (2*(rand()/(float)RAND_MAX)-1)*gain;
}

/** returns the signal power/dB */
float power(float value) {
return 20*log10(abs(value));
}

int main(int argc, const char* argv) {
cout << "Samplerate: " << SAMPLERATE << "Hz\n";
cout << "Buffersize: " << BUFFERSIZE << " samples\n";
cout << "Correct frequency is: " << FREQUENCY << "Hz\n";
cout << " - signal volume: " << SIGNALVOLUME*100 << "%\n";
cout << " - white noise: " << NOISE*100 << "%\n";

float *tone = makeTone(SAMPLERATE, FREQUENCY, BUFFERSIZE, SIGNALVOLUME);
addNoise(tone, BUFFERSIZE,NOISE);

int stepsize = FREQUENCY/5;

for (int i=0; i<10; i++) {
int freq = stepsize*i;
cout << "Trying freq: " << freq << "Hz -> dB: " << power(goertzel(tone, BUFFERSIZE, freq, SAMPLERATE)) <<
endl;
}
delete tone;

return 0;
}

```

Comments

from : ashg [[a t]] eth.net
comment : Hello!

I am interested in knowing that could we implement the Goertzel algorithm using only integer variables and not using floa at all. Please let me know. I need it urgently.
regards,
Ashish

from : yunusk [[a t]] telesis.com.tr
comment : Hello,

I will implement DTMF/MFR1/MFR2 generation/detection function using DSP. I have found goertzel algorithm for DTMF. Can I use this algorithm for MFR1 and MFR2? Could you please let me know?
Best Regards
Yunus

from : asaeiaf [[a t]] hotmail.com
comment : Hello

I'm going to implement a Goertzel algorithm on a Fixed_point DSP .
Could you lead me to the changes I should consider?
I really appreciate your helping me.

regards,
Afsaneh Asaei

from : jfishman [[a t]] umsis.miamil.edu
comment : Does anybody know if and how this can be done in real time?

Thanks,
JF

from : no
comment : It can.

from : pabitra_mohan208 [[a t]] yahoo.com
comment : sir.

i m pabitra.
please help me to understand this program.

from : Christian [[a t]] savioursofsoul.de
comment : yet untested Delphi translation of the algorithm:

```

function Goertzel(Buffer:array of Single; frequency, samplerate: single):single;
var Skn, Skn1, Skn2 : Single;
i : Integer;
temp1, temp2 : Single;

```

```
begin
skn:=0;
skn1:=0;
skn2:=0;
temp1:=2*PI*frequency/samplerate;
temp2:=Cos(temp1);
for i:=0 to Length(Buffer) do
begin
  Skn2 = Skn1;
  Skn1 = Skn;
  Skn = 2*temp2*Skn1 - Skn2 + Buffer[i];
end;
Result:=(Skn - exp(-temp1)*Skn1);
end;
```

Maybe someone can use it...

Christian

from : GPOLAT51 [[a t]] YAHOO.COM
comment : HI . I NEED DSP C62X CODE FOR DTMF DETECTÝON WÝTH GOERTZEL. ÝF YOU HAVE SEND ÝT GPOLAT51@YAHOO.COM
TAHANKSSS..

from : kevin [[a t]] subatomicglue.com
comment : why's the $2*\cos(2*PI*frequency/samplerate)$ in that inner loop? It's all constants... I'd precompute this outside the for loop.

also. use the float version of cos and exp... cosf expf... that way you're not wasting time converting and using the more expensive functions.

from : net_day2004 [[a t]] hotmail.com
comment : if anyone please can help me, i need a c++ code about the fft.the code should input and array of N rows and 1 column, all set to 1, and also it should input the length of fft(interval from f1 to f2) and output the results.

thank you,
i would appreciate your help,
bassem

Tone detection with Goertzel (x86 ASM) (click this to go back to the index)

Type : Tone detection with Goertzel in x86 assembly

References : Posted by Christian[AT]savioursofsoul[DOT]de

Notes :

This is an "assembled" version of the Goertzel Tone Detector. It is about 2 times faster than the original code.

The code has been tested and it works fine.

Hope you can use it. I'm gonna try to build a Tuner (as VST-Plugin). I hope, that this will work :-\ If anyone is intrested, please let me know.

Christian

Code :

```
function Goertzel_x87(Buffer :Psingle; BLength:Integer; frequency: Single; samplerate: Single):Single;
asm
    mov ecx,BLength
    mov eax,Buffer
    fld x2
    fldpi
    fmulp
    fmul frequency
    fdiv samplerate
    fld st(0)
    fcos
    fld x2
    fmulp
    fxch st(1)
    fldz
    fsub st(0),st(1)
    fstp st(1)

    fld12e
    fmul
    fld st(0)
    frndint
    fsub st(1),st(0)
    fxch st(1)
    f2xm1
    fld1
    fadd
    fscale
    fstp st(1)

    fldz
    fldz
    fldz
@loopStart:
    fxch st(1)
    fxch st(2)
    fstp st(0)
    fld st(3)
    fmul st(0),st(1)
    fsub st(0),st(2)
    fld [eax].Single
    faddp
    add eax,4
    loop @loopStart
@loopEnd:

    fxch st(3)
    fmulp st(2), st(0)
    fsub st(0),st(1)
    fstp result
    ffree st(2)
    ffree st(1)
    ffree st(0)
end;
```

Comments

from : thaddy [[a t]] thaddy.com

comment : Here's a variant on the theme that compensates for harmonics:

[CODE]

Function Goertzel(.Buffer: array of double; frequency, samplerate: double):.double;

var

Qkn, Qkn1, Qkn2, Wkn, Mk: double;

i: integer;

begin

Qkn:=0; Qkn1:=0;

Wkn:=2*.PI*.frequency/samplerate;

Mk:=2*.Cos(.Wkn);

for i:=0 to High(.Buffer) do begin

 Qkn2:= Qkn1; Qkn1:= Qkn;

 Qkn := Buffer[i] + Mk*.Qkn1 - Qkn2;

end;

```
Result := sqrt(Qkn*Qkn + Qkn1*Qkn1 - Mk*Qkn*Qkn1);
end;
[/CODE]
```

Posted on www.delphimaster.ru by Jeer

from : thaddy [[a t]] thaddy.com
comment : Here's what I ment ;)

```
<code>
function Goertzel(Buffer: array of double; frequency, samplerate: double):double;
var
Qkn, Qkn1, Qkn2, Wkn, Mk : double;
i : integer;
begin
Qkn:=0; Qkn1:=0;
Wkn:=2*PI*frequency/samplerate;
Mk:=2*cos(Wkn);
for i:=0 to High(Buffer) do begin
  Qkn2 := Qkn1; Qkn1 := Qkn;
  Qkn := Buffer[i] + Mk*Qkn1 - Qkn2;
end;
Result := sqrt(Qkn*Qkn + Qkn1*Qkn1 - Mk*Qkn*Qkn1);
end;
</code>
```

[1 pole LPF for smooth parameter changes \(click this to go back to the index\)](#)

Type : 1-pole LPF class

References : Posted by zioguido@gmail.com

Notes :

This is a very simple class that I'm using in my plugins for smoothing parameter changes that directly affect audio stream. It's a 1-pole LPF, very easy on CPU.

Change the value of variable "a" (0~1) for slower or a faster response.

Of course you can also use it as a lowpass filter for audio signals.

Code :

```
class CParamSmooth
{
public:
    CParamSmooth() { a = 0.99f; b = 1.f - a; z = 0; };
    ~CParamSmooth();
    inline float Process(float in) { z = (in * b) + (z * a); return z; }
private:
    float a, b, z;
};
```


1-RC and C filter (click this to go back to the index)

Type : Simple 2-pole LP

References : Posted by madbrain[AT]videotron[DOT]ca

Notes :
This filter is called 1-RC and C since it uses these two parameters. C and R correspond to raw cutoff and inverted resonance, and have a range from 0 to 1.

```
Code :
//Parameter calculation
//cutoff and resonance are from 0 to 127

c = pow(0.5, (128-cutoff) / 16.0);
r = pow(0.5, (resonance+24) / 16.0);

//Loop:

v0 = (1-r*c)*v0 - (c)*v1 + (c)*input;
v1 = (1-r*c)*v1 + (c)*v0;

output = v1;
```

Comments

from : yes

comment : input is not in 0 - 1 range.

for cutoff i guess 128.

for reso the same ?

from : scoofy [[a t]] inf.elte.hu

comment : Nice. This is very similar to a state variable filter in many ways. Relationship between c and frequency:

$$c = 2 * \sin(\pi * \text{freq} / \text{samplerate})$$

You can approximate this (tuning error towards nyquist):

$$c = 2 * \pi * \text{freq} / \text{samplerate}$$

Relationship between r and q factor:

$$r = 1/q$$

This filter has stability issues for high r values. State variable filter stability limits seem to work fine here. It can also be oversampled for better stability and wider frequency range (use 0.5*original frequency):

```
//Loop:

v0 = (1-r*c)*v0 - c*v1 + c*input;
v1 = (1-r*c)*v1 + c*v0;
tmp = v1;

v0 = (1-r*c)*v0 - c*v1 + c*input;
v1 = (1-r*c)*v1 + c*v0;
output = (tmp+v1)*0.5;

-- peter schoffhauzer
```

[18dB/oct resonant 3 pole LPF with tanh\(\) dist](#) (click this to go back to the index)

References : Posted by Josep M Comajuncosas

Linked file : [lpf18.zip](#)

Linked file : [lpf18.sme](#)

Notes :

Implementation in CSound and Sync Modular...

[1st and 2nd order pink noise filters](#) (click this to go back to the index)

Type : Pink noise

References : Posted by umminger[AT]umminger[DOT]com

Notes :

Here are some new lower-order pink noise filter coefficients.

These have approximately equiripple error in decibels from 20hz to 20khz at a 44.1khz sampling rate.

1st order, ~ +/- 3 dB error (not recommended!)

num = [0.05338071119116 -0.03752455712906]

den = [1.00000000000000 -0.97712493947102]

2nd order, ~ +/- 0.9 dB error

num = [0.04957526213389 -0.06305581334498 0.01483220320740]

den = [1.00000000000000 -1.80116083982126 0.80257737639225]

3 Band Equaliser (click this to go back to the index)

References : Posted by Neil C

Notes :

Simple 3 band equaliser with adjustable low and high frequencies ...

Fairly fast algo, good quality output (seems to be accoustically transparent with all gains set to 1.0)

How to use ...

1. First you need to declare a state for your eq

```
EQSTATE eq;
```

2. Now initialise the state (we'll assume your output frequency is 48Khz)

```
set_3band_state(eq,880,5000,480000);
```

Your EQ bands are now as follows (approximatley!)

low band = 0Hz to 880Hz

mid band = 880Hz to 5000Hz

high band = 5000Hz to 24000Hz

3. Set the gains to some values ...

```
eq.lg = 1.5; // Boost bass by 50%
```

```
eq.mg = 0.75; // Cut mid by 25%
```

```
eq.hg = 1.0; // Leave high band alone
```

4. You can now EQ some samples

```
out_sample = do_3band(eq,in_sample)
```

Have fun and mail me if any problems ... etanza at lycos dot co dot uk

Neil C / Etanza Systems, 2006 :)

Code :

First the header file ...

```
//-----  
//  
//          3 Band EQ :)  
//  
// EQ.H - Header file for 3 band EQ  
//  
// (c) Neil C / Etanza Systems / 2K6  
//  
// Shouts / Loves / Moans = etanza at lycos dot co dot uk  
//  
// This work is hereby placed in the public domain for all purposes, including  
// use in commercial applications.  
//  
// The author assumes NO RESPONSIBILITY for any problems caused by the use of  
// this software.  
//  
//-----
```

```
#ifndef __EQ3BAND__  
#define __EQ3BAND__
```

```
// -----  
//| Structures |  
// -----
```

```
typedef struct
```

```
{  
    // Filter #1 (Low band)  
  
    double lf;        // Frequency  
    double flp0;     // Poles ...  
    double flp1;  
    double flp2;  
    double flp3;  
  
    // Filter #2 (High band)  
  
    double hf;        // Frequency  
    double f2p0;     // Poles ...  
    double f2p1;  
    double f2p2;
```

```

double f2p3;

// Sample history buffer

double  sdm1;    // Sample data minus 1
double  sdm2;    //                2
double  sdm3;    //                3

// Gain Controls

double  lg;      // low gain
double  mg;      // mid gain
double  hg;      // high gain

} EQSTATE;

// -----
//| Exports |
// -----

extern void  init_3band_state(EQSTATE* es, int lowfreq, int highfreq, int mixfreq);
extern double do_3band(EQSTATE* es, double sample);

#endif // #ifndef __EQ3BAND__
//-----

Now the source ...
//-----
//
//                               3 Band EQ :)
//
// EQ.C - Main Source file for 3 band EQ
//
// (c) Neil C / Etanza Systems / 2K6
//
// Shouts / Loves / Moans = etanza at lycos dot co dot uk
//
// This work is hereby placed in the public domain for all purposes, including
// use in commercial applications.
//
// The author assumes NO RESPONSIBILITY for any problems caused by the use of
// this software.
//
//-----

// NOTES :
//
// - Original filter code by Paul Kellet (musicdsp.pdf)
//
// - Uses 4 first order filters in series, should give 24dB per octave
//
// - Now with P4 Denormal fix :)

//-----

// -----
//| Includes |
// -----

#include <math.h>
#include "eq.h"

// -----
//| Constants |
// -----

static double vsa = (1.0 / 4294967295.0); // Very small amount (Denormal Fix)

// -----
//| Initialise EQ |
// -----

// Recommended frequencies are ...
//
// lowfreq = 880 Hz
// highfreq = 5000 Hz
//
// Set mixfreq to whatever rate your system is using (eg 48Khz)

void init_3band_state(EQSTATE* es, int lowfreq, int highfreq, int mixfreq)
{
// Clear state

memset(es,0,sizeof(EQSTATE));

// Set Low/Mid/High gains to unity

es->lg = 1.0;

```

```

es->mg = 1.0;
es->hg = 1.0;

// Calculate filter cutoff frequencies

es->lf = 2 * sin(M_PI * ((double)lowfreq / (double)mixfreq));
es->hf = 2 * sin(M_PI * ((double)highfreq / (double)mixfreq));
}

// -----
//| EQ one sample |
// -----

// - sample can be any range you like :)
//
// Note that the output will depend on the gain settings for each band
// (especially the bass) so may require clipping before output, but you
// knew that anyway :)

double do_3band(EQSTATE* es, double sample)
{
    // Locals

    double l,m,h;          // Low / Mid / High - Sample Values

    // Filter #1 (lowpass)

    es->f1p0 += (es->lf * (sample - es->f1p0)) + vsa;
    es->f1p1 += (es->lf * (es->f1p0 - es->f1p1));
    es->f1p2 += (es->lf * (es->f1p1 - es->f1p2));
    es->f1p3 += (es->lf * (es->f1p2 - es->f1p3));

    l          = es->f1p3;

    // Filter #2 (highpass)

    es->f2p0 += (es->hf * (sample - es->f2p0)) + vsa;
    es->f2p1 += (es->hf * (es->f2p0 - es->f2p1));
    es->f2p2 += (es->hf * (es->f2p1 - es->f2p2));
    es->f2p3 += (es->hf * (es->f2p2 - es->f2p3));

    h          = es->sdm3 - es->f2p3;

    // Calculate midrange (signal - (low + high))

    m          = es->sdm3 - (l + h);

    // Scale, Combine and store

    l          *= es->lg;
    m          *= es->mg;
    h          *= es->hg;

    // Shuffle history buffer

    es->sdm3    = es->sdm2;
    es->sdm2    = es->sdm1;
    es->sdm1    = sample;

    // Return result

    return(l + m + h);
}

```

//-----

Comments

from : yuri_xl [[a t]] tom.com
comment : Great Thanks!

I have one problem the below:

```

double f2p0; // Poles ...
double f2p1;
double f2p2;
double f2p3;

```

that I want to know the starting value
about f2p0,f2p1,...!

from : james_braun_gottvater_der_funk [[a t]] yahoo.com
comment : yuri:

The invocation of memset() during the initialization method sets all the the members of the struct to zero.

from : hellmanc [[a t]] hotmail.com

comment : This is great -- I want to develop a compressor/limiter/expander and have been looking long and hard for bandpass / eq filtering code.
Here it is!

I am sure we could easily expand this into an x band eq.

Thanks!

from : tom tom
comment : Hi !

I've just transposed your code under Delphi.

It works well if the gain is under 1, but if i put gain > 1 i get clipping (annoying sound clips), even at 1.1;

Is it normal ?

I convert my smallint (44100 16 bits) to double before process, and convert the obtained value back to smallint with clipping (if < -32768 i set it to -32768, and if > 32768 i set it to 32768).

What did i do wrong ?

Regards

Tom

from : herbert7 [[a t]] gmx.de
comment : Hi.

Maybe the answer is quite easy. The upper limit is 32767 not 32768.

Regards

Herbert

from : angga0017163 [[a t]] yahoo.com
comment : Hi, Can U send me a full source code for this 3 band state eq from start to end ??
Please !!!!

I really need it for my study in school.
I hope you can send me, to my email.

thanks you.
regard

angga

303 type filter with saturation (click this to go back to the index)

Type : Runge-Kutta Filters

References : Posted by Hans Mikelson

Linked file : [filters001.txt](#) (this linked file is included below)

Notes :

I posted a filter to the Csound mailing list a couple of weeks ago that has a 303 flavor to it. It basically does wacky distortions to the sound. I used Runge-Kutta for the diff eq. simulation though which makes it somewhat sluggish.

This is a CSound score!!

Comments

from : nobody [[a t]] nowhere.com

comment : Anyone do this in C or C++ yet?

Linked files

```
; ORCHESTRA
;-----
; Runge-Kutta Filters
; Coded by Hans Mikelson June, 2000
;-----
sr      =      44100          ; Sample rate
kr      =      44100          ; Kontrol rate
ksmps   =      1             ; Samples/Kontrol period
nchnls  =      2             ; Normal stereo
      zakinit 50, 50

;-----
; Envelope (Knob twisting simulation)
;-----
instr 1

idur    =      p3            ; Duration
iamp    =      p4            ; Amplitude
ilps    =      p5            ; Loops
iofst   =      p6            ; Offset
itabl   =      p7            ; Table
ioutch  =      p8            ; Output channel
iphase  =      p9            ; Phase

kout    oscili iamp, ilps/idur, itabl, iphase ; Create the envelope
zkw     kout+iofst, ioutch      ; Send out to the zak channel

endin

;-----
; Runge-Kutta Freaky Filter
;-----
instr 7

idur    =      p3            ; Duration
iamp    =      p4            ; Amplitude
kfco2   zkr  p5            ; Filter cutoff
kq1     zkr  p6            ; Q
ih      =      .001          ; Diff eq step size
ipanl   =      sqrt(p8)      ; Pan left
ipanr   =      sqrt(1-p8)    ; Pan right
ifqc    =      cpspch(p9)    ; Pitch to frequency
kpb     zkr  p10           ; Pentic bounce frequency
kpa     zkr  p11           ; Pentic bounce amount
kasym   zkr  p12           ; Q assymmetry amount
kasep   zkr  p13           ; Q assymmetry separation

kdclck  linseg 0, .02, 1, idur-.04, 1, .02, 0 ; Declick envelope
kfco1   expseg 1, idur, .1
kfco    =      kfco1*kfco2
kq      =      kq1*kfco^1.2*.1
kfc     =      kfco/8/sr*44100

ay      init 0
ay1     init 0
ay2     init 0
ay3     init 0
axs     init 0
```



```

avxs  init  0
ax    vco   1, ifqc, 2, 1, 1, 1      ; Square wave

; R-K Section 1
afdbk =      kq*ay/(1+exp(-ay*3*kasep)*kasym)  ; Only oscillate in one
direction
ak11 =      ih*((ax-ay1)*kfc-afdbk)
ak21 =      ih*((ax-(ay1+.5*ak11))*kfc-afdbk)
ak31 =      ih*((ax-(ay1+.5*ak21))*kfc-afdbk)
ak41 =      ih*((ax-(ay1+ak31))*kfc-afdbk)
ay1  =      ay1+(ak11+2*ak21+2*ak31+ak41)/6

; R-K Section 2
ak12 =      ih*((ay1-ay2)*kfc)
ak22 =      ih*((ay1-(ay2+.5*ak12))*kfc)
ak32 =      ih*((ay1-(ay2+.5*ak22))*kfc)
ak42 =      ih*((ay1-(ay2+ak32))*kfc)
ay2  =      ay2+(ak12+2*ak22+2*ak32+ak42)/6

; Pentic bounce equation
ax3   =      -.1*ay*kpb
aaxs  =      (ax3*ax3*ax3*ax3*ay2)*1000*kpa    ; Update acceleration

; R-K Section 3
ak13 =      ih*((ay2-ay3)*kfc+aaxs)
ak23 =      ih*((ay2-(ay3+.5*ak13))*kfc+aaxs)
ak33 =      ih*((ay2-(ay3+.5*ak23))*kfc+aaxs)
ak43 =      ih*((ay2-(ay3+ak33))*kfc+aaxs)
ay3  =      ay3+(ak13+2*ak23+2*ak33+ak43)/6

; R-K Section 4
ak14 =      ih*((ay3-ay)*kfc)
ak24 =      ih*((ay3-(ay+.5*ak14))*kfc)
ak34 =      ih*((ay3-(ay+.5*ak24))*kfc)
ak44 =      ih*((ay3-(ay+ak34))*kfc)
ay    =      ay+(ak14+2*ak24+2*ak34+ak44)/6

aout  =      ay*iamp*kdclck*.07 ; Apply amp envelope and declck

outs  aout*ipanl, aout*ipanr ; Output the sound

endin

-- Hans Mikelson <hljmm@charter.net>

```

All-Pass Filters, a good explanation (click this to go back to the index)

Type : information

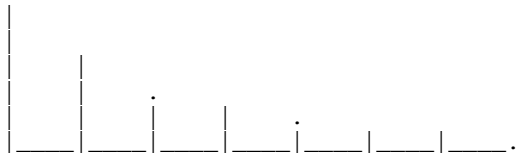
References : Posted by Olli Niemitalo

Linked file : [filters002.txt](#) (this linked file is included below)

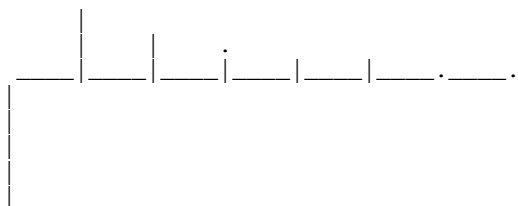
Linked files

All-Pass Filters

A true allpass is not just:



Instead, the first peak is negative and has a carefully chosen height.



This can be achieved by adding scaled $in(T)$ to the scaled output of:

$$out(T) = in(T) + out(T-delay)*gain$$

Choosing the scalings right... Looking at the critical frequencies, those that go A) a number of full cycle per peak, and those that go B) a number of full cycles and a half-cycle per peak, we can set the constraints to eliminate cancellation:

A) The sum of all peaks must be 1.

B) The sum of odd peaks minus the sum of even peaks (including the negative peak which we give number zero) must be 1.

Let's get into business now that we know what we want. We call the amplitude of the negative peak "a" and the amplitude of the first positive peak "b". The ratio between adjacent positive peaks, the feedback gain, is denoted by "g".

A) The sum of positive peaks is a geometric series and simplifies to $b/(1-g)$. Our first constraint becomes:

$$a + b/(1-g) = 1.$$

B) Using similar math... The sum of odd peaks is $b/(1-g^2)$. The sum of even peaks is $a + b*g/(1-g^2)$. So the second constraint is formed:

$$b/(1-g^2) - (a + b*g/(1-g^2)) = 1.$$

Solving the two equations we get:

$$\begin{aligned} a &= -g \\ b &= 1-g^2 \end{aligned}$$

Here i had a GREAT phewwww feeling of relief looking at <http://harmony-central.com/Effects/Articles/Reverb/allpass.html>

Choosing g is up to you. You can even make it negative, just remember to keep $|g| < 1$. Gosh, perhaps complex g could be used to create a pulsating response (forget i said that)!

We can write the allpass routine in a programmer-wise pleasant way, still preserving ease of mathematical analysis:

$$\begin{aligned} mid(T) &= in(T) + g*mid(T-delay); \\ out(T) &= (1/g-g)*mid(T) - (1/g)*in(T); \end{aligned}$$

That can be thought of as two filters put in serial and a third one in parallel with them. The first of the two serial ones is the delay with

feedback. The other "filters" are just different gains.

For the first code line, the frequency response is the usual set peaks:

$$\frac{1}{1 - g e^{-i w \text{ delay}}}$$

Adding the mixing on the second code line gives the flat-magnitude frequency response of the whole allpass system:

$$\left| \frac{1/g - g}{1 - g e^{-i w \text{ delay}}} - 1/g \right| = 1$$

The phase response is a wavy one: (formula not double-checked!)

$$\text{atan} \frac{(g^2-1) \sin(\text{delay } w)}{(g^2+1) \cos(\text{delay } w) - 2g}$$

I hope this cleared things out - and that there aren't fatal errors! :)

-- Olli Niemitalo <oniemita@mail.student.oulu.fi>

Reprise des calculs (T = delay) :

$$H = \frac{1/g - g}{1 - g \exp(-iwT)} - \frac{1}{g}$$

$$H = \frac{1 - g^2 - (1 - g \exp(-iwT))}{(1 - g \exp(-iwT)) * g}$$

$$H = \frac{-g + \exp(-iwT)}{1 - g \exp(-iwT)}$$

$$H = \exp(-iwT) * \frac{1 - g \exp(iwT)}{1 - g \exp(-iwT)}$$

Le numerateur et le denominateur sont conjugues. d'ou :

$$|H| = 1$$

et

$$\arg(H) = -wT - 2 * \arctan \frac{g * \sin(wT)}{1 - g * \cos(wT)}$$

Ce dephasage est le meme que celui trouve par Olli mais a l'avantage de separer le retard global et le dephasage.

-- Laurent de Soras <ldesoras@club-internet.fr>

More generally (in fact, maximally generally) you will get an all-pass response with any transfer function of the form :

$$H(z) = b * \frac{z^{-n} * \sum (a(i) * z^i)}{\sum (\text{conj}(a(i)) * z^{-i})}$$

where $|b| = 1$ and of course n should be large enough that your filter is causal.

-- Frederick Umminger <fumminger@my-Deja.com>

Another 4-pole lowpass... (click this to go back to the index)

Type : 4-pole LP/HP

References : Posted by fuzzipliz [AT] gmx [DOT] net

Notes :

Vaguely based on the Stilson/Smith Moog paper, but going in a rather different direction from others I've seen here.

The parameters are peak frequency and peak magnitude (g below); both are reasonably accurate for magnitudes above 1. DC gain is 1.

The filter has some undesirable properties - e.g. it's unstable for low peak freqs if implemented in single precision (haven't been able to cleanly separate it into biquads or one-poles to see if that helps), and it responds so strongly to parameter changes that it's not advisable to update the coefficients much more rarely than, say, every eight samples during sweeps, which makes it somewhat expensive.

I like the sound, however, and the accuracy is nice to have, since many filters are not very strong in that respect.

I haven't looked at the HP again for a while, but IIRC it had approximately the same good and bad sides.

Code :

```
double coef[9];
double d[4];
double omega; //peak freq
double g;      //peak mag

// calculating coefficients:

double k,p,q,a;
double a0,a1,a2,a3,a4;

k=(4.0*g-3.0)/(g+1.0);
p=1.0-0.25*k;p*=p;

// LP:
a=1.0/(tan(0.5*omega)*(1.0+p));
p=1.0+a;
q=1.0-a;

a0=1.0/(k+p*p*p*p);
a1=4.0*(k+p*p*p*q);
a2=6.0*(k+p*p*q*q);
a3=4.0*(k+p*q*q*q);
a4= (k+q*q*q*q);
p=a0*(k+1.0);

coef[0]=p;
coef[1]=4.0*p;
coef[2]=6.0*p;
coef[3]=4.0*p;
coef[4]=p;
coef[5]=-a1*a0;
coef[6]=-a2*a0;
coef[7]=-a3*a0;
coef[8]=-a4*a0;

// or HP:
a=tan(0.5*omega)/(1.0+p);
p=a+1.0;
q=a-1.0;

a0=1.0/(p*p*p*p+k);
a1=4.0*(p*p*p*q-k);
a2=6.0*(p*p*q*q+k);
a3=4.0*(p*q*q*q-k);
a4= (q*q*q*q+k);
p=a0*(k+1.0);

coef[0]=p;
coef[1]=-4.0*p;
coef[2]=6.0*p;
coef[3]=-4.0*p;
coef[4]=p;
coef[5]=-a1*a0;
coef[6]=-a2*a0;
coef[7]=-a3*a0;
coef[8]=-a4*a0;

// per sample:

out=coef[0]*in+d[0];
d[0]=coef[1]*in+coef[5]*out+d[1];
d[1]=coef[2]*in+coef[6]*out+d[2];
d[2]=coef[3]*in+coef[7]*out+d[3];
d[3]=coef[4]*in+coef[8]*out;
```

Comments

from : Christian [[a t]] savioursofsoul.de

comment : Yet untested object pascal translation:

```
unit T4PoleUnit;
```

```
interface
```

```
type TFilterType=(ftLowPass, ftHighPass);
T4Pole=class(TObject)
private
fGain : Double;
fFreq : Double;
fSR : Single;
protected
fCoeffs : array[0..8] of Double;
d : array[0..3] of Double;
fFilterType : TFilterType;
procedure SetGain(s:Double);
procedure SetFrequency(s:Double);
procedure SetFilterType(v:TFilterType);
procedure Calc;
public
constructor Create;
function Process(s:single):single;
published
property Gain: Double read fGain write SetGain;
property Frequency: Double read fFreq write SetFrequency;
property SampleRate: Single read fSR write fSR;
property FilterType: TFilterType read fFilterType write SetFilterType;
end;
```

```
implementation
```

```
uses math;
```

```
const kDenorm = 1.0e-25;
```

```
constructor T4Pole.Create;
```

```
begin
inherited create;
fFreq:=1000;
fSR:=44100;
Calc;
end;
```

```
procedure T4Pole.SetFrequency(s:Double);
```

```
begin
fFreq:=s;
Calc;
end;
```

```
procedure T4Pole.SetGain(s:Double);
```

```
begin
fGain:=s;
Calc;
end;
```

```
procedure T4Pole.SetFilterType(v:TFilterType);
```

```
begin
fFilterType:=v;
Calc;
end;
```

```
procedure T4Pole.Calc;
```

```
var k,p,q,b,s : Double;
a : array[0..4] of Double;
begin
fGain:=1;
if fFilterType=ftLowPass
then s:=1
else s:=-1;
// calculating coefficients:
k:=(4.0*fGain-3.0)/(fGain+1.0);
p:=1.0-0.25*k;
q:=p*p;
```

```
if fFilterType=ftLowPass
then b:=1.0/(tan(pi*fFreq/fSR)*(1.0+p))
else b:=tan(pi*fFreq/fSR)/(1.0+p);
p:=1.0+b;
q:=s*(1.0-b);
```

```
a[0] := 1.0/( k+p*p*p*p);
a[1] := 4.0*(s*k+p*p*p*q);
a[2] := 6.0*( k+p*p*q*q);
a[3] := 4.0*(s*k+p*q*q*q);
a[4] := ( k+q*q*q*q);
p := a[0]*(k+1.0);
```

```
fCoeffs[0]=p;  
fCoeffs[1]=4.0*p*s;  
fCoeffs[2]=6.0*p;  
fCoeffs[3]=4.0*p*s;  
fCoeffs[4]=p;  
fCoeffs[5]=-a[1]*a[0];  
fCoeffs[6]=-a[2]*a[0];  
fCoeffs[7]=-a[3]*a[0];  
fCoeffs[8]=-a[4]*a[0];  
end;
```

```
function T4Pole.Process(s:single):single;  
begin  
Result:=fCoeffs[0]*s+d[0];  
d[0]:=fCoeffs[1]*s+fCoeffs[5]*Result+d[1];  
d[1]:=fCoeffs[2]*s+fCoeffs[6]*Result+d[2];  
d[2]:=fCoeffs[3]*s+fCoeffs[7]*Result+d[3];  
d[3]:=fCoeffs[4]*s+fCoeffs[8]*Result;  
end;  
  
end.
```

Bass Booster (click this to go back to the index)

Type : LP and SUM

References : Posted by Johny Dupej

Notes :

This function adds a low-passed signal to the original signal. The low-pass has a quite wide response.

Params:

selectivity - frequency response of the LP (higher value gives a steeper one) [70.0 to 140.0 sounds good]

ratio - how much of the filtered signal is mixed to the original

gain2 - adjusts the final volume to handle cut-offs (might be good to set dynamically)

Code :

```
#define saturate(x) __min(__max(-1.0,x),1.0)

float BassBoosta(float sample)
{
static float selectivity, gain1, gain2, ratio, cap;
gain1 = 1.0/(selectivity + 1.0);

cap= (sample + cap*selectivity )*gain1;
sample = saturate((sample + cap*ratio)*gain2);

return sample;
}
```

Comments

from : sashaslonmailer [[a t]] mail.ru

comment : Can you say more about ratio ,gain2 and gain1

ratio is from 0..1, isn't is?

gain1 ?

gain2 ?

[Biquad C code](#) (click this to go back to the index)

[References](#) : Posted by Tom St Denis

[Linked file](#) : [biquad.c](#) (this linked file is included below)

[Notes](#) :

Implementation of the RBJ cookbook, in C.

[Comments](#)

[from](#) : s_olivani [[a t]] yahoo.com

[comment](#) : Hi Tom St Denis,

Can you help us in understanding how lower cut off and higher cut off frequencies are calculated for a given centre frequency and bandwidth in octave in the code.

Thanks in advance.

[Linked files](#)

```
/* Simple implementation of Biquad filters -- Tom St Denis
```

```
*
```

```
* Based on the work
```

```
Cookbook formulae for audio EQ biquad filter coefficients
```

```
-----
```

```
by Robert Bristow-Johnson, pbjrbj@viconet.com a.k.a. robert@audioheads.com
```

```
* Available on the web at
```

```
http://www.smartelectronix.com/musicdsp/text/filters005.txt
```

```
* Enjoy.
```

```
*
```

```
* This work is hereby placed in the public domain for all purposes, whether  
* commercial, free [as in speech] or educational, etc. Use the code and please  
* give me credit if you wish.
```

```
*
```

```
* Tom St Denis -- http://tomstdenis.home.dhs.org
```

```
*/
```

```
/* this would be biquad.h */
```

```
#include <math.h>
```

```
#include <stdlib.h>
```

```
#ifndef M_LN2
```

```
#define M_LN2 0.69314718055994530942
```

```
#endif
```

```
#ifndef M_PI
```

```
#define M_PI 3.14159265358979323846
```

```
#endif
```

```
/* whatever sample type you want */
```

```
typedef double smp_type;
```

```
/* this holds the data required to update samples thru a filter */
```

```
typedef struct {
```

```
    smp_type a0, a1, a2, a3, a4;
```

```
    smp_type x1, x2, y1, y2;
```

```
}
```

```
biquad;
```

```
extern smp_type BiQuad(smp_type sample, biquad * b);
```

```
extern biquad *BiQuad_new(int type, smp_type dbGain, /* gain of filter */  
                          smp_type freq, /* center frequency */  
                          smp_type srate, /* sampling rate */  
                          smp_type bandwidth); /* bandwidth in octaves */
```

```
/* filter types */
```

```
enum {
```

```
    LPF, /* low pass filter */
```

```
    HPF, /* High pass filter */
```

```
    BPF, /* band pass filter */
```

```
    NOTCH, /* Notch Filter */
```

```
    PEQ, /* Peaking band EQ filter */
```

```
    LSH, /* Low shelf filter */
```

```
    HSH /* High shelf filter */
```

```
};
```

```
/* Below this would be biquad.c */
```



```

/* Computes a BiQuad filter on a sample */
smp_type BiQuad(smp_type sample, biquad * b)
{
    smp_type result;

    /* compute result */
    result = b->a0 * sample + b->a1 * b->x1 + b->a2 * b->x2 -
        b->a3 * b->y1 - b->a4 * b->y2;

    /* shift x1 to x2, sample to x1 */
    b->x2 = b->x1;
    b->x1 = sample;

    /* shift y1 to y2, result to y1 */
    b->y2 = b->y1;
    b->y1 = result;

    return result;
}

/* sets up a BiQuad Filter */
biquad *BiQuad_new(int type, smp_type dbGain, smp_type freq,
smp_type srate, smp_type bandwidth)
{
    biquad *b;
    smp_type A, omega, sn, cs, alpha, beta;
    smp_type a0, a1, a2, b0, b1, b2;

    b = malloc(sizeof(biquad));
    if (b == NULL)
        return NULL;

    /* setup variables */
    A = pow(10, dbGain / 40);
    omega = 2 * M_PI * freq /srate;
    sn = sin(omega);
    cs = cos(omega);
    alpha = sn * sinh(M_LN2 / 2 * bandwidth * omega /sn);
    beta = sqrt(A + A);

    switch (type) {
    case LPF:
        b0 = (1 - cs) / 2;
        b1 = 1 - cs;
        b2 = (1 - cs) / 2;
        a0 = 1 + alpha;
        a1 = -2 * cs;
        a2 = 1 - alpha;
        break;
    case HPF:
        b0 = (1 + cs) / 2;
        b1 = -(1 + cs);
        b2 = (1 + cs) / 2;
        a0 = 1 + alpha;
        a1 = -2 * cs;
        a2 = 1 - alpha;
        break;
    case BPF:
        b0 = alpha;
        b1 = 0;
        b2 = -alpha;
        a0 = 1 + alpha;
        a1 = -2 * cs;
        a2 = 1 - alpha;
        break;
    case NOTCH:
        b0 = 1;
        b1 = -2 * cs;
        b2 = 1;
        a0 = 1 + alpha;
        a1 = -2 * cs;
        a2 = 1 - alpha;
        break;
    case PEQ:
        b0 = 1 + (alpha * A);
        b1 = -2 * cs;
        b2 = 1 - (alpha * A);
        a0 = 1 + (alpha /A);
        a1 = -2 * cs;
        a2 = 1 - (alpha /A);
        break;
    case LSH:
        b0 = A * ((A + 1) - (A - 1) * cs + beta * sn);

```

```

    b1 = 2 * A * ((A - 1) - (A + 1) * cs);
    b2 = A * ((A + 1) - (A - 1) * cs - beta * sn);
    a0 = (A + 1) + (A - 1) * cs + beta * sn;
    a1 = -2 * ((A - 1) + (A + 1) * cs);
    a2 = (A + 1) + (A - 1) * cs - beta * sn;
    break;
case HSH:
    b0 = A * ((A + 1) + (A - 1) * cs + beta * sn);
    b1 = -2 * A * ((A - 1) + (A + 1) * cs);
    b2 = A * ((A + 1) + (A - 1) * cs - beta * sn);
    a0 = (A + 1) - (A - 1) * cs + beta * sn;
    a1 = 2 * ((A - 1) - (A + 1) * cs);
    a2 = (A + 1) - (A - 1) * cs - beta * sn;
    break;
default:
    free(b);
    return NULL;
}

/* precompute the coefficients */
b->a0 = b0 /a0;
b->a1 = b1 /a0;
b->a2 = b2 /a0;
b->a3 = a1 /a0;
b->a4 = a2 /a0;

/* zero initial samples */
b->x1 = b->x2 = 0;
b->y1 = b->y2 = 0;

return b;
}
/* crc==3062280887, version==4, Sat Jul 7 00:03:23 2001 */

```

Butterworth Optimized C++ Class (click this to go back to the index)

Type : 24db Resonant Lowpass

References : Posted by neotec

Notes :

This is exactly the same as posted by "Zxform" (filters004.txt). The only difference is, that this version is an optimized one.

Parameters:

Cutoff [0.f -> Nyquist.f]

Resonance [0.f -> 1.f]

There are some minima and maxima defined, to make it sound nice in all situations. This class is part of some of my VST Plugins, and works well and executes fast.

Code :

```
// FilterButterworth24db.h

#pragma once

class CFilterButterworth24db
{
public:
    CFilterButterworth24db(void);
    ~CFilterButterworth24db(void);
    void SetSampleRate(float fs);
    void Set(float cutoff, float q);
    float Run(float input);

private:
    float t0, t1, t2, t3;
    float coef0, coef1, coef2, coef3;
    float history1, history2, history3, history4;
    float gain;
    float min_cutoff, max_cutoff;
};

// FilterButterworth24db.cpp

#include <math.h>

#define BUDDA_Q_SCALE 6.f

#include "FilterButterworth24db.h"

CFilterButterworth24db::CFilterButterworth24db(void)
{
    this->history1 = 0.f;
    this->history2 = 0.f;
    this->history3 = 0.f;
    this->history4 = 0.f;

    this->SetSampleRate(44100.f);
    this->Set(22050.f, 0.0);
}

CFilterButterworth24db::~CFilterButterworth24db(void)
{
}

void CFilterButterworth24db::SetSampleRate(float fs)
{
    float pi = 4.f * atanf(1.f);

    this->t0 = 4.f * fs * fs;
    this->t1 = 8.f * fs * fs;
    this->t2 = 2.f * fs;
    this->t3 = pi / fs;

    this->min_cutoff = fs * 0.01f;
    this->max_cutoff = fs * 0.45f;
}

void CFilterButterworth24db::Set(float cutoff, float q)
{
    if (cutoff < this->min_cutoff)
        cutoff = this->min_cutoff;
    else if (cutoff > this->max_cutoff)
        cutoff = this->max_cutoff;

    if (q < 0.f)
        q = 0.f;
    else if (q > 1.f)
        q = 1.f;

    float wp = this->t2 * tanf(this->t3 * cutoff);
    float bd, bd_tmp, b1, b2;

    q *= BUDDA_Q_SCALE;
```

```

q += 1.f;

b1 = (0.765367f / q) / wp;
b2 = 1.f / (wp * wp);

bd_tmp = this->t0 * b2 + 1.f;

bd = 1.f / (bd_tmp + this->t2 * b1);

this->gain = bd * 0.5f;

this->coef2 = (2.f - this->t1 * b2);

this->coef0 = this->coef2 * bd;
this->coef1 = (bd_tmp - this->t2 * b1) * bd;

b1 = (1.847759f / q) / wp;

bd = 1.f / (bd_tmp + this->t2 * b1);

this->gain *= bd;
this->coef2 *= bd;
this->coef3 = (bd_tmp - this->t2 * b1) * bd;
}

float CFilterButterworth24db::Run(float input)
{
float output = input * this->gain;
float new_hist;

output -= this->history1 * this->coef0;
new_hist = output - this->history2 * this->coef1;

output = new_hist + this->history1 * 2.f;
output += this->history2;

this->history2 = this->history1;
this->history1 = new_hist;

output -= this->history3 * this->coef2;
new_hist = output - this->history4 * this->coef3;

output = new_hist + this->history3 * 2.f;
output += this->history4;

this->history4 = this->history3;
this->history3 = new_hist;

return output;
}

```

Comments

from : neotec

comment : I have checked the peak output of this filter and especially for low frequencies ... there is a simple fix, which makes it sound better with low frequencies: change the line in Set(...) that reads 'this->gain = bd * 0.5f;' to 'this->gain = bd;'

from : bob [[a t]] yahoo.com

comment : Thanks for the quick reply. I've tried your change and it's made a slight tonal difference here, but the tests were not particularly scientific. I've discovered more detail in the problem, and it's one that has been commented on with other filters: If I sweep the filter quickly up or down the low frequencies it blows out really badly, even with zero Q. I'm new to filter math, so excuse my ignorance if this is a common thing with Butterworth.

from : neotec

comment : Yep ... this filter reacts very extreme on fast cutoff changes. I've added a function to my VST Synthesizer, which 'fades' the cutoff value from actual value to the desired one in about 0.05 seconds. My modulation envelopes do have similar restrictions concerning speed.

from : neotec

comment : If you want to know how this filter sounds, visit the kvraudio forum, and search here: "KVR Forum » Instruments" for "Cetone VST Plugins".

from : nobody [[a t]] nowhere.com

comment : I'm wondering about that tanh in the "Set."

Could replace with a pade approximation, maybe. What is the range of inputs going into it?

In other words, how small and big does this get?...

this->t3 * cutoff

from : toast [[a t]] somewhereyoucantfind.com

comment : Possible small optimization. It depends on how smart your compiler is, but sections like this...

```

output = new_hist + this->history3 * 2.f;
output += this->history4;

```

can be changed to this to change the multiply to an addition:

```
output = this->history3;
output += output+new_Hist+this->history4;
```

from : toast [[a t]] somewhereyoucantfind.com

comment : While I'm at it, one of these divisions can easily be switched to a multiply...

```
b1 = (1.847759f / q) / wp;
```

```
b1=(1.847759f/(q*wp));
```

from : bob [[a t]] yahoo.com

comment : Four times oversampling removes the problems with fast cut-off sweeps at low values. This filter has the same shape as a normal biquad filter, with a more pronounced resonance boost.

from : bob [[a t]] yahoo.com

comment : This sounds really nice, especially with resonance. Although it becomes unstable below 4K (at 44100 s/r), which explains why the min_cutoff value has been set quite high. Would using doubles help stabilise it? Also, I can't figure out how to get a high pass out of this, can anybody help? Cheers.

from : musicdsp.org [[a t]] mindcontrol.org

comment : Why would oversampling solve the problem? If you over-sample, the poles have to reach even further into the relative frequencies, and stability would become more of a problem AFAICT.

from : bob [[a t]] yahoo.com

comment : It just seems to. If you 4X over-sample, then it gives it a 4X chance to recover from each sweep change, presuming you're not changing the filter cut-off at 4X also.

from : musicdsp [[a t]] dsparsons.co.uk.nowhere

comment : thing is with 4X oversampling on this is that you'll be reducing precision on omega (wp here), and so should probably shift to double rather than float to help accuracy.

C++ class implementation of RBJ Filters (click this to go back to the index)

References : Posted by arguru[AT]smartelectronic[DOT]com

Linked file : [CFxRbjFilter.h](#) (this linked file is included below)

Notes :

[WARNING: This code is not FPU undernormalization safe!]

Linked files

```
class CFxRbjFilter
{
public:

CFxRbjFilter()
{
// reset filter coeffs
b0a0=b1a0=b2a0=a1a0=a2a0=0.0;

// reset in/out history
oul=ou2=in1=in2=0.0f;
};

float filter(float in0)
{
// filter
float const yn = b0a0*in0 + b1a0*in1 + b2a0*in2 - a1a0*oul - a2a0*ou2;

// push in/out buffers
in2=in1;
in1=in0;
ou2=oul;
oul=yn;

// return output
return yn;
};

void calc_filter_coeffs(int const type,double const frequency,double const sample_rate,double
const q,double const db_gain,bool q_is_bandwidth)
{
// temp pi
double const temp_pi=3.1415926535897932384626433832795;

// temp coef vars
double alpha,a0,a1,a2,b0,b1,b2;

// peaking, lowshelf and hishelp
if(type>=6)
{
double const A = pow(10.0,(db_gain/40.0));
double const omega = 2.0*temp_pi*frequency/sample_rate;
double const tsin = sin(omega);
double const tcos = cos(omega);

if(q_is_bandwidth)
alpha=tsin*sinh(log(2.0)/2.0*q*omega/tsin);
else
alpha=tsin/(2.0*q);

double const beta = sqrt(A)/q;

// peaking
if(type==6)
{
b0=float(1.0+alpha*A);
b1=float(-2.0*tcos);
b2=float(1.0-alpha*A);
a0=float(1.0+alpha/A);
a1=float(-2.0*tcos);
a2=float(1.0-alpha/A);
}

// lowshelf
if(type==7)
{
b0=float(A*((A+1.0)-(A-1.0)*tcos+beta*tsin));
b1=float(2.0*A*((A-1.0)-(A+1.0)*tcos));
b2=float(A*((A+1.0)-(A-1.0)*tcos-beta*tsin));
a0=float((A+1.0)+(A-1.0)*tcos+beta*tsin);
a1=float(-2.0*((A-1.0)+(A+1.0)*tcos));
a2=float((A+1.0)+(A-1.0)*tcos-beta*tsin);
}
}
};
```

```

}

// hishelf
if(type==8)
{
    b0=float(A*((A+1.0)+(A-1.0)*tcos+beta*tsin));
    b1=float(-2.0*A*((A-1.0)+(A+1.0)*tcos));
    b2=float(A*((A+1.0)+(A-1.0)*tcos-beta*tsin));
    a0=float((A+1.0)-(A-1.0)*tcos+beta*tsin);
    a1=float(2.0*((A-1.0)-(A+1.0)*tcos));
    a2=float((A+1.0)-(A-1.0)*tcos-beta*tsin);
}
}
else
{
    // other filters
    double const omega = 2.0*temp_pi*frequency/sample_rate;
    double const tsin = sin(omega);
    double const tcos = cos(omega);

    if(q_is_bandwidth)
        alpha=tsin*sinh(log(2.0)/2.0*q*omega/tsin);
    else
        alpha=tsin/(2.0*q);

    // lowpass
    if(type==0)
    {
        b0=(1.0-tcos)/2.0;
        b1=1.0-tcos;
        b2=(1.0-tcos)/2.0;
        a0=1.0+alpha;
        a1=-2.0*tcos;
        a2=1.0-alpha;
    }

    // hipass
    if(type==1)
    {
        b0=(1.0+tcos)/2.0;
        b1=-(1.0+tcos);
        b2=(1.0+tcos)/2.0;
        a0=1.0+ alpha;
        a1=-2.0*tcos;
        a2=1.0-alpha;
    }

    // bandpass csg
    if(type==2)
    {
        b0=tsin/2.0;
        b1=0.0;
        b2=-tsin/2;
        a0=1.0+alpha;
        a1=-2.0*tcos;
        a2=1.0-alpha;
    }

    // bandpass czpg
    if(type==3)
    {
        b0=alpha;
        b1=0.0;
        b2=-alpha;
        a0=1.0+alpha;
        a1=-2.0*tcos;
        a2=1.0-alpha;
    }

    // notch
    if(type==4)
    {
        b0=1.0;
        b1=-2.0*tcos;
        b2=1.0;
        a0=1.0+alpha;
        a1=-2.0*tcos;
        a2=1.0-alpha;
    }

    // allpass
    if(type==5)

```

```
{
  b0=1.0-alpha;
  b1=-2.0*tcos;
  b2=1.0+alpha;
  a0=1.0+alpha;
  a1=-2.0*tcos;
  a2=1.0-alpha;
}

// set filter coeffs
b0a0=float(b0/a0);
b1a0=float(b1/a0);
b2a0=float(b2/a0);
a1a0=float(a1/a0);
a2a0=float(a2/a0);
};

private:

// filter coeffs
float b0a0,b1a0,b2a0,a1a0,a2a0;

// in/out history
float ou1,ou2,in1,in2;
};
```


C-Weighed Filter (click this to go back to the index)

Type : digital implementation (after bilinear transform)

References : Posted by Christian@savioursofsoul.de

Notes :

unoptimized version!

Code :

First prewarp the frequency of both poles:

```
K1 = tan(0.5*Pi*20.6 / SampleRate) // for 20.6Hz
K2 = tan(0.5*Pi*12200 / SampleRate) // for 12200Hz
```

Then calculate the both biquads:

```
b0 = 1
b1 = 0
b2 = -1
a0 = ((K1+1)*(K1+1)*(K2+1)*(K2+1));
a1 = -4*(K1*K1*K2*K2+K1*K1*K2+K1*K2*K2-K1-K2-1)*t;
a2 = - ((K1-1)*(K1-1)*(K2-1)*(K2-1))*t;
```

and:

```
b3 = 1
b4 = 0
b5 = -1
a3 = ((K1+1)*(K1+1)*(K2+1)*(K2+1));
a4 = -4*(K1*K1*K2*K2+K1*K1*K2+K1*K2*K2-K1-K2-1)*t;
a5 = - ((K1-1)*(K1-1)*(K2-1)*(K2-1))*t;
```

Now use an equation for calculating the biquads like this:

```
Stage1 = b0*Input + State0;
State0 = + a1/a0*Stage1 + State1;
State1 = b2*Input + a2/a0*Stage1;

Output = b3*Stage1 + State2;
State2 = + a4/a3*Output + State2;
State3 = b5*Stage1 + a5/a3*Output;
```

Comments

from : Christian [[a t]] savioursofsoul.de

comment : You might still need to normalize the filter output. You can do this easily by multipliing either the b0 and b2 or the b3 and b5 with a constant.

Typically the filter is normalized to have a gain of 0dB at 1kHz

Also oversampling of this filter might be useful.

Cascaded resonant lp/hp filter (click this to go back to the index)

Type : lp+hp

References : Posted by tobybear[AT]web[DOT]de

Notes :

```
// Cascaded resonant lowpass/hipass combi-filter
// The original source for this filter is from Paul Kellet from
// the archive. This is a cascaded version in Delphi where the
// output of the lowpass is fed into the highpass filter.
// Cutoff frequencies are in the range of  $0 \leq x < 1$  which maps to
// 0..nyquist frequency
```

// input variables are:

```
// cut_lp: cutoff frequency of the lowpass (0..1)
// cut_hp: cutoff frequency of the hipass (0..1)
// res_lp: resonance of the lowpass (0..1)
// res_hp: resonance of the hipass (0..1)
```

Code :

```
var n1,n2,n3,n4:single; // filter delay, init these with 0!
    fb_lp,fb_hp:single; // storage for calculated feedback
const p4=1.0e-24; // Pentium 4 denormal problem elimination

function dofilter(inp,cut_lp,res_lp,cut_hp,res_hp:single):single;
begin
    fb_lp:=res_lp+res_lp/(1-cut_lp);
    fb_hp:=res_hp+res_hp/(1-cut_hp);
    n1:=n1+cut_lp*(inp-n1+fb_lp*(n1-n2))+p4;
    n2:=n2+cut_lp*(n1-n2);
    n3:=n3+cut_hp*(n2-n3+fb_hp*(n3-n4))+p4;
    n4:=n4+cut_hp*(n3-n4);
    result:=i-n4;
end;
```

Comments

from : office [[a t]] hermannseib.com

comment : I guess the last line should read
result:=inp-n4;

Right?

Bye,

Hermann

from : courierfst [[a t]] hotmail.com

comment : excuse me which type is? 6db/oct or 12 or what?

thanks

from : christianalthaus [[a t]] gmx.de

comment : result := n2-n4

:)

Cool Sounding Lowpass With Decibel Measured Resonance (click this to go back to the index)

Type : LP 2-pole resonant tweaked butterworth

References : Posted by daniel_jacob_werner [AT] yaho [DOT] com [DOT] au

Notes :
This algorithm is a modified version of the tweaked butterworth lowpass filter by Patrice Tarrabia posted on musicdsp.org's archives. It calculates the coefficients for a second order IIR filter. The resonance is specified in decibels above the DC gain. It can be made suitable to use as a SoundFont 2.0 filter by scaling the output so the overall gain matches the specification (i.e. if resonance is 6dB then you should scale the output by -3dB). Note that you can replace the sqrt(2) values in the standard butterworth highpass algorithm with my "q =" line of code to get a highpass also. How it works: normally q is the constant sqrt(2), and this value controls resonance. At sqrt(2) resonance is 0dB, smaller values increase resonance. By multiplying sqrt(2) by a power ratio we can specify the resonant gain at the cutoff frequency. The resonance power ratio is calculated with a standard formula to convert between decibels and power ratios (the powf statement...).

Good Luck,
Daniel Werner
<http://experimentalscene.com/>

Code :
float c, csq, resonance, q, a0, a1, a2, b1, b2;

```
c = 1.0f / (tanf(pi * (cutoff / samplerate)));
csq = c * c;
resonance = powf(10.0f, -(resonancedB * 0.1f));
q = sqrt(2.0f) * resonance;
a0 = 1.0f / (1.0f + (q * c) + (csq));
a1 = 2.0f * a0;
a2 = a0;
b1 = (2.0f * a0) * (1.0f - csq);
b2 = a0 * (1.0f - (q * c) + csq);
```

Comments

from : acid_mutant[aa]yahoo[doot]com

comment :

For some reason when I tested this algorithm, even though the frequency response looked OK in my graphs - i.e. it should resonate the output didn't seem to be very resonant - it could be a phase issue, I'll keep checking.

(BTW: I use an impulse, then FFT, then display the power bands returned)

from : dans [[a t]] dans.com

comment : shouldn't it be

```
resonance = powf(10.0f, -(resonancedB * 0.05f));
```

instead of

```
resonance = powf(10.0f, -(resonancedB * 0.1f));
```

to get correct dB gain?

... since gain = $10^{(dB/20)}$...

from : scoofy [[a t]] inf.elte.hu

comment : Agree with the last post.

from : dwerner.spam.me.not [[a t]] bounce.experimentalscene.com

comment : The algorithm was developed with a digital signal of 32-bit floating point pseudo-random white noise running through it. The level of resonance was measured by visually plotting the output of the FFT of the signal. I half agree with the second last post, i.e. dB in acoustics is not the same as dB in digital audio. Correct me if I am wrong, it is a long time since I thought about this.

DC filter (click this to go back to the index)

Type : 1-pole/1-zero DC filter

References : Posted by andy[DOT]rossol[AT]bluewin[DOT]ch

Notes :

This is based on code found in the document:

"Introduction to Digital Filters (DRAFT)"

Julius O. Smith III (jos@ccrma.stanford.edu)

(<http://www-ccrma.stanford.edu/~jos/filters/>)

Some audio algorithms (asymmetric waveshaping, cascaded filters, ...) can produce DC offset. This offset can accumulate and reduce the signal/noise ratio.

So, how to fix it? The example code from Julius O. Smith's document is:

...

```
y(n) = x(n) - x(n-1) + R * y(n-1)
```

```
// "R" between 0.9 .. 1
```

```
// n=current (n-1)=previous in/out value
```

...

"R" depends on sampling rate and the low frequency point. Do not set "R" to a fixed value (e.g. 0.99) if you don't know the sample rate. Instead set R to:

(-3dB @ 40Hz): $R = 1 - (250/\text{samplerate})$

(-3dB @ 30Hz): $R = 1 - (190/\text{samplerate})$

(-3dB @ 20Hz): $R = 1 - (126/\text{samplerate})$

Comments

from : andy[DOT]rossol[AT]bluewin[DOT]ch

comment : I just received a mail from a musicdsp reader:

'How to calculate "R" for a given (-3dB) low frequency point?'

$$R = 1 - (\pi^2 * \text{frequency} / \text{samplerate})$$

($\pi=3.14159265358979$)

from : rbj [[a t]] surfglobal.net

comment : particularly if fixed-point arithmetic is used, this simple high-pass filter can create it's own DC offset because of limit-cycles. to cure that look at

http://www.dspguru.com/comp.dsp/tricks/alg/dc_block.htm

this trick uses the concept of "noise-shaping" to prevent DC in any limit-cycles.

r b-j

[Delphi Class implementation of the RBJ filters](#) (click this to go back to the index)

Type : Delphi class implementation of the RBJ filters

References : Posted by veryangrymobster@hotmail.com

Notes :

I haven't tested this code thoroughly as it's pretty much a straight conversion from Arguru c++ implementation.

Code :

```
{
RBJ Audio EQ Cookbook Filters
A pascal conversion of arguru[AT]smartelectronic[DOT]com's
c++ implementation.

WARNING:This code is not FPU undernormalization safe.

Filter Types
0-LowPass
1-HiPass
2-BandPass CSG
3-BandPass CZPG
4-Notch
5-AllPass
6-Peaking
7-LowShelf
8-HiShelf
}
unit uRbjEqFilters;

interface

uses math;

type
  TRbjEqFilter=class
  private
    b0a0,b1a0,b2a0,a1a0,a2a0:single;
    in1,in2,ou1,ou2:single;
    fSampleRate:single;
    fMaxBlockSize:integer;
    fFilterType:integer;
    fFreq,fQ,fDBGain:single;
    fQIsBandWidth:boolean;
    procedure SetQ(NewQ:single);
  public
    out1:array of single;
    constructor create(SampleRate:single;MaxBlockSize:integer);
    procedure CalcFilterCoeffs(pFilterType:integer;pFreq,pQ,pDBGain:single;pQIsBandWidth:boolean);overload;
    procedure CalcFilterCoeffs;overload;
    function Process(input:single):single; overload;
    procedure Process(Input:psingle;sampleframes:integer); overload;
    property FilterType:integer read fFilterType write fFilterType;
    property Freq:single read fFreq write fFreq;
    property q:single read fQ write SetQ;
    property DBGain:single read fDBGain write fDBGain;
    property QIsBandWidth:boolean read fQIsBandWidth write fQIsBandWidth;
  end;
implementation

constructor TRbjEqFilter.create(SampleRate:single;MaxBlockSize:integer);
begin
  fMaxBlockSize:=MaxBlockSize;
  setLength(out1,fMaxBlockSize);
  fSampleRate:=SampleRate;

  fFilterType:=0;
  fFreq:=500;
  fQ:=0.3;
  fDBGain:=0;
  fQIsBandWidth:=true;

  in1:=0;
  in2:=0;
  ou1:=0;
  ou2:=0;
end;

procedure TRbjEqFilter.SetQ(NewQ:single);
begin
  fQ:=(1-NewQ)*0.98;
end;

procedure TRbjEqFilter.CalcFilterCoeffs(pFilterType:integer;pFreq,pQ,pDBGain:single;pQIsBandWidth:boolean);
begin
  FilterType:=pFilterType;
  Freq:=pFreq;
  Q:=pQ;
  DBGain:=pDBGain;
  QIsBandWidth:=pQIsBandWidth;
```

```

CalcFilterCoeffs;
end;

procedure TRbjEqFilter.CalcFilterCoeffs;
var
  alpha,a0,a1,a2,b0,b1,b2:single;
  A,beta,omega,tsin,tcos:single;
begin
  //peaking, LowShelf or HiShelf
  if fFilterType>=6 then
  begin
    A:=power(10.0,(DBGain/40.0));
    omega:=2*pi*fFreq/fSampleRate;
    tsin:=sin(omega);
    tcos:=cos(omega);

    if fQIsBandWidth then
      alpha:=tsin*sinh(log2(2.0)/2.0*fQ*omega/tsin)
    else
      alpha:=tsin/(2.0*fQ);

    beta:=sqrt(A)/fQ;

    // peaking
    if fFilterType=6 then
    begin
      b0:=1.0+alpha*A;
      b1:=-2.0*tcos;
      b2:=1.0-alpha*A;
      a0:=1.0+alpha/A;
      a1:=-2.0*tcos;
      a2:=1.0-alpha/A;
    end else
    // lowshelf
    if fFilterType=7 then
    begin
      b0:=(A*((A+1.0)-(A-1.0)*tcos+beta*tsin));
      b1:=(2.0*A*((A-1.0)-(A+1.0)*tcos));
      b2:=(A*((A+1.0)-(A-1.0)*tcos-beta*tsin));
      a0:=((A+1.0)+(A-1.0)*tcos+beta*tsin);
      a1:=(-2.0*((A-1.0)+(A+1.0)*tcos));
      a2:=((A+1.0)+(A-1.0)*tcos-beta*tsin);
    end;
    // hishelf
    if fFilterType=8 then
    begin
      b0:=(A*((A+1.0)+(A-1.0)*tcos+beta*tsin));
      b1:=(-2.0*A*((A-1.0)+(A+1.0)*tcos));
      b2:=(A*((A+1.0)+(A-1.0)*tcos-beta*tsin));
      a0:=((A+1.0)-(A-1.0)*tcos+beta*tsin);
      a1:=((2.0*((A-1.0)-(A+1.0)*tcos));
      a2:=((A+1.0)-(A-1.0)*tcos-beta*tsin);
    end;
  end else //other filter types
  begin
    omega:=2*pi*fFreq/fSampleRate;
    tsin:=sin(omega);
    tcos:=cos(omega);
    if fQIsBandWidth then
      alpha:=tsin*sinh(log2(2)/2*fQ*omega/tsin)
    else
      alpha:=tsin/(2*fQ);
    //lowpass
    if fFilterType=0 then
    begin
      b0:=(1-tcos)/2;
      b1:=1-tcos;
      b2:=(1-tcos)/2;
      a0:=1+alpha;
      a1:=-2*tcos;
      a2:=1-alpha;
    end else //hipass
    if fFilterType=1 then
    begin
      b0:=(1+tcos)/2;
      b1:=-(1+tcos);
      b2:=(1+tcos)/2;
      a0:=1+alpha;
      a1:=-2*tcos;
      a2:=1-alpha;
    end else //bandpass CSG
    if fFilterType=2 then
    begin
      b0:=tsin/2;
      b1:=0;
      b2:=-tsin/2;
      a0:=1+alpha;
      a1:=-1*tcos;
      a2:=1-alpha;
    end else //bandpass CZPG
    if fFilterType=3 then
    begin

```

```

    b0:=alpha;
    b1:=0.0;
    b2:=-alpha;
    a0:=1.0+alpha;
    a1:=-2.0*tcos;
    a2:=1.0-alpha;
end else //notch
if fFilterType=4 then
begin
    b0:=1.0;
    b1:=-2.0*tcos;
    b2:=1.0;
    a0:=1.0+alpha;
    a1:=-2.0*tcos;
    a2:=1.0-alpha;
end else //allpass
if fFilterType=5 then
begin
    b0:=1.0-alpha;
    b1:=-2.0*tcos;
    b2:=1.0+alpha;
    a0:=1.0+alpha;
    a1:=-2.0*tcos;
    a2:=1.0-alpha;
end;
end;

b0a0:=b0/a0;
b1a0:=b1/a0;
b2a0:=b2/a0;
a1a0:=a1/a0;
a2a0:=a2/a0;
end;

```

```
function TRbjEqFilter.Process(input:single):single;
```

```

var
    LastOut:single;
begin
    // filter
    LastOut:= b0a0*input + b1a0*in1 + b2a0*in2 - a1a0*ou1 - a2a0*ou2;

    // push in/out buffers
    in2:=in1;
    in1:=input;
    ou2:=ou1;
    ou1:=LastOut;

    // return output
    result:=LastOut;
end;

```

```

{
the process method is overloaded.
use Process(input:single):single;
for per sample processing
use Process(Input:psingle;sampleframes:integer);
for block processing. The input is a pointer to
the start of an array of single which contains
the audio data.
i.e.
RBJFilter.Process(@WaveData[0],256);
}

```

```

procedure TRbjEqFilter.Process(Input:psingle;sampleframes:integer);
var
    i:integer;
    LastOut:single;
begin
    for i:=0 to SampleFrames-1 do
    begin
        // filter
        LastOut:= b0a0*(input^)+ b1a0*in1 + b2a0*in2 - a1a0*ou1 - a2a0*ou2;
        //LastOut:=input^;
        // push in/out buffers
        in2:=in1;
        in1:=input^;
        ou2:=ou1;
        ou1:=LastOut;

        Out1[i]:=LastOut;

        inc(input);
    end;
end;
end.

```

Digital RIAA equalization filter coefficients (click this to go back to the index)

Type : RIAA

References : Posted by Frederick Umminger

Notes :

Use at your own risk. Confirm correctness before using. Don't assume I didn't goof something up.

-Frederick Umminger

Code :

The "turntable-input software" thread inspired me to generate some coefficients for a digital RIAA equalization filter. These coefficients were found by matching the magnitude response of the s-domain transfer function using some proprietary Matlab scripts. The phase response may or may not be totally whacked.

The s-domain transfer function is

$$R3(1+R1*C1*s)(1+R2*C2*s)/(R1(1+R2*C2*s) + R2(1+R1*C1*s) + R3(1+R1*C1*s)(1+R2*C2*s))$$

where

R1 = 883.3k
R2 = 75k
R3 = 604
C1 = 3.6n
C2 = 1n

This is based on the reference circuit found in <http://www.hagtech.com/pdf/riaa.pdf>

The coefficients of the digital transfer function $b(z^{-1})/a(z^{-1})$ in descending powers of z , are:

44.1kHz

```
b = [ 0.02675918611906 -0.04592084787595 0.01921229297239 ]
a = [ 1.00000000000000 -0.73845850035973 -0.17951755477430 ]
error +/- 0.25dB
```

48kHz

```
b = [ 0.02675918611906 -0.04592084787595 0.01921229297239 ]
a = [ 1.00000000000000 -0.73845850035973 -0.17951755477430 ]
error +/- 0.15dB
```

88.2kHz

```
b = [ 0.04872204977233 -0.09076930609195 0.04202280710877 ]
a = [ 1.00000000000000 -0.85197860443215 -0.10921171201431 ]
error +/- 0.01dB
```

96kHz

```
b = [ 0.05265477122714 -0.09864197097385 0.04596474352090 ]
a = [ 1.00000000000000 -0.85835597216218 -0.10600020417219 ]
error +/- 0.006dB
```

Comments

from : jtp_1960 [[a t]] hotmail.com

comment : Hmm... since I'm having lack in knowledge of utilizing this type of 'data' in programming, could someone be kind and give a short code example of its usage (@ some samplerate), lets say, using Basic/VB language (though, C-C++/Pascal-Delphi/Java goes as well)?

JT

from : musicdsp [[a t]] TAKEMEOUTdsparsons.co.uk

comment : they are coefficients to plug into a std biquad. look through the filters section of musicdsp you'll find a load of examples of biquads (essentially two quadratic equations which are solved together to do the DSP stuff).

It's of the form

$$\text{out} = b_0 \cdot \text{in}[0] + b_1 \cdot \text{in}[-1] + b_2 \cdot \text{in}[-2] - a_1 \cdot \text{out}[-1] - a_2 \cdot \text{out}[-2]$$

where $\text{in}[0,-1,-2]$ are the current input and the previous 2; and $\text{out}[-1,-2]$ are the last two outputs.

Generally the previous output coefficients are subtracted, but sometimes the signs are swapped, and they are added like the inputs.

some algorithms use a for ins and b for outs, others use them the other way around. Generally (but not always) there are 3 input and 2 output coeffs, so you can work out which is which.

HTH
DSP

from : nobody [[a t]] nowhere.com

comment : I don't get it. How do you set the frequency, Kenneth?

What frequencies are being passed?

from : jtp_1960 [[a t]] hotmail.com

comment : Hmm...

Since no links allowed here, I have started a topic on this matter @ KVR

topic number: 170235

topic name: "Coefficients of the digital transfer function ... How to ?"

I tried the 44.1/48kHz version and it produced quite 'bad' results .. lots of rattle in audio and the RIAA curve form is not as it should be (should be: 20Hz; ±19.27dB ... ~1kHz; ±0dB ... 20kHz; ±19.62dB). (couple of pictures linked in KVR topic).

Also, .. if this is the result in anyway, this is the 'production curve' used in mastering process ... how can it be changed to 'opposite' ...

JT

from : jtp_1960 [[a t]] hotmail.com

comment : Thanks to all so far.

I found this quote from another forum:

QUOTE:

"All you should need to do to get the complementary curve is swap the a and b vectors, and then multiply both vectors by 1/a(0) to normalize. That will give the coefficients for the inverse filter."

/QUOTE

w/ a note that it was taken from one of those OPs (Frederick Umminger's) postings ... but the reference link was dead so I couldn't read the whole story. If OP or anyone else can give some light in this matter of how to make that swap w/ normalization (fully) so I could try w/ higher SR data. I did try and got values like -20.1287341287123, etc..

I actually got the 44.1/48kHz curve managed w/ help from a post in another forum. But there were nothing explained fully.

QUOTE:

"

; Filter coefficients (48kHz) for RIAA curve from Frederick

; Umminger; see

;

; b = [0.02675918611906 -0.04592084787595 0.01921229297239]

; a = [1.00000000000000 -0.73845850035973 -0.17951755477430]

; error +/- 0.15dB

; inverted filter for phono playback (48kHz):

;

; b = [0.2275882473429072 -0.1680644758323426 -0.0408560856583673]

; a = [1.0000000000000000 -1.7160778983199925 0.7179700042784745]

;

; since a[1] is too large, it must be splitted into a11 and a12

static b0=0.2275882473429072, b1=-0.1680644758323426, b2=-0.0408560856583673

static a1=.85803894915999625, a2=-0.7179700042784745

"

/QUOTE

just lots of numbers

JT

from : jtp_1960 [[a t]] hotmail.com

comment : This seem to become a monologue but, ... I'm still having issues w/ those 88.2kHz and 96kHz filter coefficients when inverted.

Noticed that when those coefficients for 88.2kHz and 96kHz are inverted, in both cases, a1 and a2 gets values which maybe are not good in equation

$y[i] = b0x[i] + b1x[i-1] + b2x[i-2] - a1y[i-1] - a2y[i-2]$

because of, a1 gets a negative value and its decimal part is bigger than a2 is --> "--a1y[i-1] - a2y[i-2]" --> looks like y[i] starts growing after every sample calculation. This is not an issue w/ data for 44.1kHz and 48kHz. When I change those a1/a2 decimal parts so that the abs(a1)-a2 =< 1 becomes true then filter works well (though not right results). Also, while analyzing the VST plugin, using C.W.Buddes VST PluginAnalyzer, Delphi tracer (Watch) shows y[i] become over 1.0 after ~830 sampleframes and after 8192 sampleframes, y[i] has value of 2.488847401e+11 already (i.e. 248884740100). This shouldn't be a coding problem since a friend of mine tested these w/ SynthMaker (no coding needed) and the results were equal.

If this "-a1x[i-1]-a2x[i-2] > 1" is an issue, are there any methods to get it fixed w/o loosing the accuracy OP got into those original coefficients?

jtp

from : Christian [[a t]] savioursofsoul.de

comment : Try to plot the poles and zeroes. If there are poles outside the unit circle, your filter will be unstable!

To eliminate poles outside the unit circle, construct an allpass filter which has zeroes at the same position as the unwanted poles. They are now canceling out themselves, so that you only have poles inside the unit circle. Your filter should be stable now!

All you need to know now is how to transform the filter coefficients into poles and zeroes and vice versa. If you're using delphi, you might want to have a look into the DFilter class of the open source project 'Delphi ASIO & VST Packages'.

from : jtp_1960 [[a t]] hotmail.com

comment : Thanks for your suggestion Christian.
I didn't try this allpass method because of

- I managed to get this issue rounded through another way (I have now 3rd-4th order filters working here as VST and standalone for all those four samplerates mentioned here and I'm also considering to add ones for 174.6 kHz and 192 kHz as well)

- as I'm learning these filter matters and delphi programming, I would have needed some good examples to do this

My final thoughts over those coefficients listed in F. Ummingers post:

As those coefficients needs to be inversed before getting the RIAA reproduction done, I can't say 100% sure if any of those works properly then (maybe one set does).

When inversion is done as was suggested elsewhere:

- swap a/b vectors,

- multiply all with 1/a0 and

- optional: 'normalize' b's by dividing every b with sum of b's

, only coefficients for 44.1kHz and 48kHz seem to become stable but, which one is the right one then since, those original coefficients are same for both? I suppose those can't be equal coefficients because this is sample accurate filter in question, or can those?. If not then, which one is the correct one ... you can find it out by trying (least the resulting sound quality should tell this). Maybe Hannes Rohde (quote in my 3rd post) went through this and found the right ones or just used those given for 48kHz (SoundBlaster DSP is internally 48kHz).

What's wrong with those others? It seems that both, 88.2kHz and 96kHz coefficients as inversed, produces unstable filter which won't work (see my previous post)

jtp

from : jtp_1960 [[a t]] hotmail.com

comment : FYI, here are working filter coefficients for biquad implementation of RIAA EQ Reproduction filters:

44.1kHz:

a = [1.0000000000 -1.7007240000 0.7029381524]

b = [1.0000000000 -0.7218922000 -0.1860520545]

error ~-0.23dB

48kHz:

a = [1.0000000000 -1.7327655000 0.7345534436]

b = [1.0000000000 -0.7555521000 -0.1646257113]

error ~-0.14dB

88.2kHz:

a = [1.0000000000 -1.8554648000 0.8559721393]

b = [1.0000000000 -0.8479577000 -0.1127631993]

error 0.008dB

and 96kHz:

a = [1.0000000000 -1.8666083000 0.8670382873]

b = [1.0000000000 -0.8535331000 -0.1104595113]

error ~-0.006dB

NOTES:

- By swapping the a1<->b1 and a2<->b2 you'll get the production filter.

- All these given filter coefficients produces a bit gained filter (~+12.5dB or so) so, if you like to adjust the 1 kHz = 0dB, it can be done quite accurately by finding linear difference using software like Tobybear's FilterExplorer. Enter coefficients into FilterExplorer, by moving mouse cursor over the plotted magnitude curve in magnitude plot window, find/point the ~1kHz position and then check the magnitude value (value inside the brackets) found in info field. Use this value as divider for b coefficients.

jtp

jiiteepie@yahoo.se

Direct form II (click this to go back to the index)

Type : generic

References : Posted by Fuzzpitz

Notes :

I've noticed there's no code for direct form II filters in general here, though probably many of the filter examples use it. I haven't looked at them all to verify that, but there certainly doesn't seem to be a snippet describing this.

This is a simple direct form II implementation of a k-pole, k-zero filter. It's a little faster than (a naive, real-time implementation of) direct form I, as well as more numerically accurate.

Code :

Direct form I pseudocode:

```
y[n] = a[0]*x[n] + a[1]*x[n-1] + .. + a[k]*x[n-k]
      - b[1]*y[n-1] - .. - b[k]*y[n-k];
```

Simple equivalent direct form II pseudocode:

```
y[n] = a[0]*x[n] + d[0];
d[0] = a[1]*x[n] - b[1]*y[n] + d[1];
d[1] = a[2]*x[n] - b[2]*y[n] + d[2];
.
.
d[k-2] = a[k-1]*x[n] - b[k-1]*y[n] + d[k-1];
d[k-1] = a[k]*x[n] - b[k]*y[n];
```

For example, a biquad:

```
out = a0*in + a1*h0 + a2*h1 - b1*h2 - b2*h3;
h1 = h0;
h0 = in;
h3 = h2;
h2 = out;
```

becomes

```
out = a0*in + d0;
d0 = a1*in - b1*out + d1;
d1 = a2*in - b2*out;
```

Comments

from : scoofy [[a t]] inf.elte.hu

comment : I think the per sample denormal number elimination on x87 FPU's is more difficult, since you need to check for denormals at 3 places instead of one (if I'm right).

from : gtjennings [[a t]] gmail.com

comment : Are the constants (a and b) wrong here. Don't they need to be switched? If you look at like wikipedia that's the case and it makes more sense. I'm trying to implement a low pass filter at 25mhz passband edge. I'm getting alot of fluctuation in my output more that expect. Any suggestions?

```
int main(int argc, char *argv[])
{
    double b[3] = {1,2,1};
    double a1[3] = {1,-1.9995181705254206,0.99952100328066507};
    //double a1[3] = {1,-1.9252217796690612,0.95315661147483732};
    double a2[3] = {1,-1.9985996261556458,0.99860245760957123};
    double a3[3] = {1,-1.9977949691405856,0.99779779945453828};
    double a4[3] = {1,-1.9971690447494761,0.99717187417666975};
    double a5[3] = {1,-1.9967721889631873,0.9967750178281477};
    double a6[2] = {1, -0.99831813425055116};
    double d[3] = {0};
    double y[5][3] = {0};
    double out[2] = {0};
    double x[3]={0}, x1,x2,in;
    double i=0;
    char wait;
    while(i<10000000)
    {
        x1 = sin(2*10000*3.14159265*i);
        x2 = sin(2*10000*3.14159265*i-3.14159265);
        in = x1 * x2;

        x[0] = in * 7.0818881108085789e-7;

        y[0][0] = b[0]*x[0] + b[1]*x[1] + b[2]*x[2] - a1[1]*y[0][1] - a1[2]*y[0][1];
        y[0][1] = y[0][0];
        x[2] = x[1];
        x[1] = x[0];
        ///////////////////////////////////////////////////////////////////
        y[0][0] = y[0][0] * 7.0786348128153693e-7;
        y[1][0] = b[0]*y[0][0] + b[1]*y[0][1] + b[2]*y[0][2] - a2[1]*y[1][1] - a2[2]*y[1][1];
        y[0][2] = y[0][1];
```

```

y[0][1] = y[0][0];
y[1][1] = y[1][0];
////////////////////////////////////
y[1][0] = y[1][0] * 7.0757848807506174e-7;
y[2][0] = b[0]*y[1][0] + b[1]*y[1][1] + b[2]*y[1][2] - a3[1]*y[2][1] - a3[2]*y[2][1];
y[1][2] = y[1][1];
y[1][1] = y[1][0];
y[2][1] = y[2][0];
////////////////////////////////////
y[2][0] = y[2][0] * 7.0735679834155469e-7;
y[3][0] = b[0]*y[2][0] + b[1]*y[2][1] + b[2]*y[2][2] - a4[1]*y[3][1] - a4[2]*y[3][1];
y[2][2] = y[2][1];
y[2][1] = y[2][0];
y[3][1] = y[3][0];
////////////////////////////////////
y[3][0] = y[3][0] * 7.0721624006526327e-007;
y[4][0] = b[0]*y[3][0] + b[1]*y[3][1] + b[2]*y[3][2] - a5[1]*y[4][1] - a5[2]*y[4][1];
y[3][2] = y[3][1];
y[3][1] = y[3][0];
y[4][1] = y[4][0];
////////////////////////////////////
/*y[4][0] = y[4][0]* 0.000840932874724457;
out[0] = 1*y[4][0] + 1*y[4][1] - a6[1]*out[1];
y[4][1] = y[4][0];
out[1] = out[0];*/

cout<<y[4][0]<<"\n";

i+=.1;
}

```

Fast Downsampling With Antialiasing (click this to go back to the index)

References : Posted by mumart[AT]gmail[DOT]com

Notes :
A quick and simple method of downsampling a signal by a factor of two with a useful amount of antialiasing. Each source sample is convolved with { 0.25, 0.5, 0.25 } before downsampling.

Code :

```
int filter_state;

/* input_buf can be equal to output_buf */
void downsample( int *input_buf, int *output_buf, int output_count ) {
    int input_idx, input_end, output_idx, output_sam;
    input_idx = output_idx = 0;
    input_end = output_count * 2;
    while( input_idx < input_end ) {
        output_sam = filter_state + ( input_buf[ input_idx++ ] >> 1 );
        filter_state = input_buf[ input_idx++ ] >> 2;
        output_buf[ output_idx++ ] = output_sam + filter_state;
    }
}
```

Comments

from : dsp [[a t]] dsparsons.nospam.co.uk

comment : I see this is designed for integers; what are your thoughts on altering it to floats and doing simple division rather than bit shifts?

from : mumart [[a t]] gmail.com

comment : It will work fine in floating point. I would probably use multiplication rather than division though, as I would expect that to be faster (ie. >> 1 --> *0.5, >>2 --> *0.25).

from : dfl [[a t]] ccrma.stanford.edu

comment : this triangular window is still not the greatest antialiaser... but it's probably fine for something like an oversampled lowpass filter!

from : mumart [[a t]] gmail.com

comment : For my purposes(modelling a first-order-hold dac) it was fine. The counterpart to it I suppose is this one - a classic exponential decay, which gives a lovely warm sound. Each sample is convolved with { 0.5, 0.25, 0.125, ...etc }

```
int filter_state;
```

```
void downsample( int *input_buf, int *output_buf, int output_count ) {
    int input_idx, output_idx, input_ep1;
    output_idx = 0;
    input_idx = 0;
    input_ep1 = output_count * 2;
    while( input_idx < input_ep1 ) {
        filter_state = ( filter_state + input_buf[ input_idx ] ) >> 1;
        output_buf[ output_idx ] = filter_state;
        filter_state = ( filter_state + input_buf[ input_idx + 1 ] ) >> 1;
        input_idx += 2;
        output_idx += 1;
    }
}
```

I'm not a great fan of all these high-order filters, the mathematics are more than I can cope with :)

Cheers,
Martin

from : k-asche [[a t]] web.de

comment : Hi @ all,

what is a good initialization value of filter_state?

Greetings

Karsten

Formant filter (click this to go back to the index)

References : Posted by Alex

```
Code :
/*
Public source code by alex@smartelectronix.com
Simple example of implementation of formant filter
Vowelnum can be 0,1,2,3,4 <=> A,E,I,O,U
Good for spectral rich input like saw or square
*/
//-----VOWEL COEFFICIENTS
const double coeff[5][11]= {
{ 8.11044e-06,
8.943665402, -36.83889529, 92.01697887, -154.337906, 181.6233289,
-151.8651235, 89.09614114, -35.10298511, 8.388101016, -0.923313471  ///A
},
{ 4.36215e-06,
8.90438318, -36.55179099, 91.05750846, -152.422234, 179.1170248,  ///E
-149.6496211,87.78352223, -34.60687431, 8.282228154, -0.914150747
},
{ 3.33819e-06,
8.893102966, -36.49532826, 90.96543286, -152.4545478, 179.4835618,
-150.315433, 88.43409371, -34.98612086, 8.407803364, -0.932568035  ///I
},
{ 1.13572e-06,
8.994734087, -37.2084849, 93.22900521, -156.6929844, 184.596544,  ///O
-154.3755513, 90.49663749, -35.58964535, 8.478996281, -0.929252233
},
{ 4.09431e-07,
8.997322763, -37.20218544, 93.11385476, -156.2530937, 183.7080141,  ///U
-153.2631681, 89.59539726, -35.12454591, 8.338655623, -0.910251753
}
};
//-----
static double memory[10]={0,0,0,0,0,0,0,0,0,0};
//-----
float formant_filter(float *in, int vowelnum)
{
    res= (float) ( coeff[vowelnum][0] *in +
    coeff[vowelnum][1] *memory[0] +
    coeff[vowelnum][2] *memory[1] +
    coeff[vowelnum][3] *memory[2] +
    coeff[vowelnum][4] *memory[3] +
    coeff[vowelnum][5] *memory[4] +
    coeff[vowelnum][6] *memory[5] +
    coeff[vowelnum][7] *memory[6] +
    coeff[vowelnum][8] *memory[7] +
    coeff[vowelnum][9] *memory[8] +
    coeff[vowelnum][10] *memory[9] );

    memory[9]= memory[8];
    memory[8]= memory[7];
    memory[7]= memory[6];
    memory[6]= memory[5];
    memory[5]= memory[4];
    memory[4]= memory[3];
    memory[3]= memory[2];
    memory[2]= memory[1];
    memory[1]= memory[0];
    memory[0]=(double) res;
    return res;
}
}
```

Comments

from : rhettanderson [[a t]] yahoo.com

comment : Where did the coefficients come from? Do they relate to frequencies somehow? Are they male or female? Etc.

from : el98shn [[a t]] ing.umu.se

comment : And are the coefficients for 44k1hz?

/stefancrs

from : meeloo [[a t]] meeloo.net

comment : It seem to be ok at 44KHz although I get quite lot of distortion with this filter.

There are typos in the given code too, the correct version looks like this i think:

```
float formant_filter(float *in, int vowelnum)
{
    float res= (float) ( coeff[vowelnum][0]* (*in) +
    coeff[vowelnum][1] *memory[0] +
    coeff[vowelnum][2] *memory[1] +
    coeff[vowelnum][3] *memory[2] +
    coeff[vowelnum][4] *memory[3] +
    coeff[vowelnum][5] *memory[4] +
    coeff[vowelnum][6] *memory[5] +
    coeff[vowelnum][7] *memory[6] +
    coeff[vowelnum][8] *memory[7] +
    coeff[vowelnum][9] *memory[8] +
    coeff[vowelnum][10] *memory[9] );
```

...

(missing type and asterisk in the first calc line ;).

I tried morphing from one vowel to another and it works ok except in between 'A' and 'U' as I get a lot of distortion and sometime (depending on the signal) the filter goes into auto-oscillation.

Sebastien Metrot

from : larsby [[a t]] elak.org

comment : How did you get the coefficients?

Did I miss something?

/Larsby

from : stefan.hallen [[a t]] dice.se

comment : Yeah, morphing linearly between the coefficients works just fine. The distortion I only get when not lowering the amplitude of the input. So I lower it :)

Larsby, you can approximate filter curves quite easily, check your dsp literature :)

from : alex [[a t]] smartelectronix.com

comment : Correct, it is for sampling rate of 44kHz.

It supposed to be female (soprano), approximated with its five formants.

--Alex.

from : ruiner33 [[a t]] hotmail.com

comment : Can you tell us how you calculated the coefficients?

from : antiprosynthesis [[a t]] hotmail.com

comment : The distorting/sharp A vowel can be toned down easy by just changing the first coeff from 8.11044e-06 to 3.11044e-06. Sounds much better that way.

from : jnorberg [AT] gmail [DOT] com

comment : Hi, I get the last formant (U) to self-oscillate and distort out of control whatever I feed it with. all the other ones sound fine...

any suggestions?

Thanks,
Jonas

from : texmex [[a t]] iki.fi

comment : I was playing around this filter, and after hours of debugging finally noticed that converting those coefficients to float just won't do it. The resulting filter is not stable anymore. Doh...

I don't have any idea how to convert them, though.

from : jayman_21 [[a t]] hotmail.com

comment : How do you go about calculating the coefficients???

from : mysterious T

comment : Fantastic, it's all I can say! Done the linear blending and open blending matrix (a-e, a-i, a-o, a-u, e-i, e-o...etc..etc..). Too much fun!

Thanks a lot, Alex!

from : Thiyana.Maitriya [[a t]] honeywell.com

comment : Could you tell us how you calculated the coefficients?

[frequency warped FIR lattice](#) (click this to go back to the index)

Type : FIR using allpass chain

References : Posted by mail[AT]mutagene[DOT]net

Notes :
Not at all optimized and pretty hungry in terms of arrays and overhead (function requires two arrays containing lattice filter's internal state and outputs to another two arrays with their next states). In this implementation I think you'll have to juggle taps1/newtaps in your processing loop, alternating between one set of arrays and the other for which to send to wfirlattice).

A frequency-warped lattice filter is just a lattice filter where every delay has been replaced with an allpass filter. By adjusting the allpass filters, the frequency response of the filter can be adjusted (e.g., design an FIR that approximates some filter. Play with with warping coefficient to "sweep" the FIR up and down without changing any other coefficients). Much more on warped filters can be found on Aki Harma's website (<http://www.acoustics.hut.fi/~aqi/>)

```
Code :
float wfirlattice(float input, float *taps1, float *taps2, float *reflcof, float lambda, float *newtaps1,
float *newtaps2, int P)
// input is filter input
// taps1,taps2 are previous filter states (init to 0)
// reflcof are reflection coefficients. abs(reflcof) < 1 for stable filter
// lambda is warping (0 = no warping, 0.75 is close to bark scale at 44.1 kHz)
// newtaps1, newtaps2 are new filter states
// P is the order of the filter
{
float forward;
float topline;

forward = input;
topline = forward;

for (int i=0;i<P;i++)
{
newtaps2[i] = topline;
newtaps1[i] = float(lambda)*(-topline + taps1[i]) + taps2[i];
topline = newtaps1[i]+forward*(reflcof[i]);
forward += newtaps1[i]*(reflcof[i]);
taps1[i]=newtaps1[i];
taps2[i]=newtaps2[i];
}
return forward;
}
```

Comments

from : fuzzpilz [[a t]] gmx.net

comment : Couldn't you easily do away with newtaps entirely? As in:

```
for(int i=0;i<P;i++)
{
taps1[i]=lambda*(taps1[i]-topline)+taps2[i];
taps2[i]=topline;
topline=taps1[i]+forward*reflcof[i];
forward+=taps1[i]*reflcof[i];
}
```

I haven't had time to try this in a plugin yet, but if Maple is to be trusted at all, that works.

(2WarpDelay is nice, by the way)

from : mail [[a t]] mutagene.net

comment : haha, thanks, that's awesome! how embarrassing ;)

(glad you like 2warpdelay! the warped IIR lattice is up on harma's site too, though you might save yourself time if you read the errata: <http://www.acoustics.hut.fi/~aqi/papers/oops.html> : ()

from : davett [[a t]] yahoot.com

comment : This looks really interesting.

How do I get the coeffs for it, and how do I invert it to get back to the original signal?

Thanks,

DaveT

Hilbert Filter Coefficient Calculation (click this to go back to the index)

Type : Uncle Hilbert

References : Posted by Christian[at]savioursofsoul[dot]de

Notes :

This is the delphi code to create the filter coefficients, which are needed to phaseshift a signal by 90°
This may be useful for an evelope detector...

By windowing the filter coefficients you can trade phase response flatness with magnitude response flatness.

I had problems checking its response by using a dirac impulse. White noise works fine.

Also this introduces a latency of N/2!

Code :

```
type TSingleArray = Array of Single;

procedure UncleHilbert(var FilterCoefficients: TSingleArray; N : Integer);
var i,j : Integer;
begin
  SetLength(FilterCoefficients,N);
  for i:=0 to (N div 4) do
    begin
      FilterCoefficients[(N div 2)+(2*i-1)]:=+2/(PI*(2*i-1));
      FilterCoefficients[(N div 2)-(2*i-1)]:=-2/(PI*(2*i-1));
    end;
  end;
```

[High quality /2 decimators \(click this to go back to the index\)](#)

Type : Decimators

References : Posted by Paul Sernine

Notes :

These are /2 decimators,

Just instantiate one of them and use the Calc method to obtain one sample while inputing two. There is 5,7 and 9 tap versions.

They are extracted/adapted from a tutorial code by Thierry Rochebois. The optimal coefficients are excerpts of Traitement numérique du signal, 5eme edition, M Bellanger, Masson pp. 339-346.

Code :

```
//Filtres décimateurs
// T.Rochebois
// Based on
//Traitement numérique du signal, 5eme edition, M Bellanger, Masson pp. 339-346
class Decimateur5
{
private:
float R1,R2,R3,R4,R5;
const float h0;
const float h1;
const float h3;
const float h5;
public:
Decimateur5():h0(346/692.0f),h1(208/692.0f),h3(-44/692.0f),h5(9/692.0f)
{
R1=R2=R3=R4=R5=0.0f;
}
float Calc(const float x0,const float x1)
{
float h5x0=h5*x0;
float h3x0=h3*x0;
float h1x0=h1*x0;
float R6=R5+h5x0;
R5=R4+h3x0;
R4=R3+h1x0;
R3=R2+h1x0+h0*x1;
R2=R1+h3x0;
R1=h5x0;
return R6;
}
};
class Decimateur7
{
private:
float R1,R2,R3,R4,R5,R6,R7;
const float h0,h1,h3,h5,h7;
public:
Decimateur7():h0(802/1604.0f),h1(490/1604.0f),h3(-116/1604.0f),h5(33/1604.0f),h7(-6/1604.0f)
{
R1=R2=R3=R4=R5=R6=R7=0.0f;
}
float Calc(const float x0,const float x1)
{
float h7x0=h7*x0;
float h5x0=h5*x0;
float h3x0=h3*x0;
float h1x0=h1*x0;
float R8=R7+h7x0;
R7=R6+h5x0;
R6=R5+h3x0;
R5=R4+h1x0;
R4=R3+h1x0+h0*x1;
R3=R2+h3x0;
R2=R1+h5x0;
R1=h7x0;
return R8;
}
};
class Decimateur9
{
private:
float R1,R2,R3,R4,R5,R6,R7,R8,R9;
const float h0,h1,h3,h5,h7,h9;
public:
Decimateur9():h0(8192/16384.0f),h1(5042/16384.0f),h3(-1277/16384.0f),h5(429/16384.0f),h7(-
116/16384.0f),h9(18/16384.0f)
{
R1=R2=R3=R4=R5=R6=R7=R8=R9=0.0f;
}
float Calc(const float x0,const float x1)
{
float h9x0=h9*x0;
float h7x0=h7*x0;
float h5x0=h5*x0;
float h3x0=h3*x0;
float h1x0=h1*x0;
```

```
float R10=R9+h9*x0;  
R9=R8+h7*x0;  
R8=R7+h5*x0;  
R7=R6+h3*x0;  
R6=R5+h1*x0;  
R5=R4+h1*x0+h0*x1;  
R4=R3+h3*x0;  
R3=R2+h5*x0;  
R2=R1+h7*x0;  
R1=h9*x0;  
return R10;  
}  
};
```

Comments

from : qwernath

comment : Works fine. Thanks.

from : safsf

comment : cool, now how would Thierry Rochebois go about making a high quality *2 interpolator?

from : phoenix-69

comment : I have a copy of Traitement numérique du signal, 5eme edition, I can dig out the interpolator ;)

[Karlsen](#) (click this to go back to the index)

Type : 24-dB (4-pole) lowpass

References : Posted by Best Regards,Ove Karlsen

Notes :

There's really not much voodoo going on in the filter itself, it's as simple as possible:

```
pole1 = (in * frequency) + (pole1 * (1 - frequency));
```

Most of you can probably understand that math, it's very similar to how an analog condenser works.

Although, I did have to do some JuJu to add resonance to it.

While studying the other filters, I found that the feedback phase is very important to how the overall resonance level will be, and so I made a dynamic feedback path, and constant Q approximation by manipulation of the feedback phase.

A bonus with this filter, is that you can "overdrive" it... Try high input levels..

Code :

```
// Karlsen 24dB Filter by Ove Karlsen / Synergy-7 in the year 2003.
// b_f = frequency 0..1
// b_q = resonance 0..50
// b_in = input
// to do bandpass, subtract poles from eachother, highpass subtract with input.

float b_inSH = b_in // before the while statement.

while (b_oversample < 2) { //2x oversampling (@44.1khz)
float prevfp;
prevfp = b_f;
if (prevfp > 1) {prevfp = 1;} // Q-limiter

b_f = (b_f * 0.418) + ((b_q * pole4) * 0.582); // dynamic feedback
float intfp;
intfp = (b_f * 0.36) + (prevfp * 0.64); // feedback phase
b_in = b_inSH - intfp; // inverted feedback

pole1 = (b_in * b_f) + (pole1 * (1 - b_f)); // pole 1
if (pole1 > 1) {pole1 = 1;} else if (pole1 < -1) {pole1 = -1;} // pole 1 clipping
pole2 = (pole1 * b_f) + (pole2 * (1 - b_f)); // pole 2
pole3 = (pole2 * b_f) + (pole3 * (1 - b_f)); // pole 3
pole4 = (pole3 * b_f) + (pole4 * (1 - b_f)); // pole 4

b_oversample++;
}
lowpassout = b_in;
```

Comments

from : ove [[a t]] synergy-7.com

comment : Hi.

Seems to be a slight typo in my code.

lowpassout = pole4; // ofcourse :)

Best Regards,
Ove Karlsen

from : matt at ahsodit dot com

comment : Hi Ove, we spoke once on the #AROS IRC channel... I'm trying to put this code into a filter object, but I'm wandering what datatype the input and output should be?

I'm processing my audio data in packets of 8000 signed words (16 bits) at a time. can I put one audio sample words into this function? Since it seems to require a floating point input!

Thanks

from : ove_code [[a t]] ihsan-vst.com

comment : Hi Matt.

Yes, it does indeed need float inputs.

Best Regards,
Ove Karlsen.

from : unkargherth [[a t]] terra.es

comment : Can somebody explain exactly how to make the band Pass and high pass, i tried as explained and don't work exactly as expected

highpass = in - pole4

make "some kind of highpass", but not as expected
cut frequency

and for band pass, how we subtract the poles between them ?

pole4-pole3-pole2-pole1 ?
pole1-pole2-pole3-pole4 ?

Also, is there a way to get a Notch ?

from : Christian [[a t]] savioursofsoul.de

comment : Below you will find an object pascal version of the filter.

L=Lowpass
H=Highpass
N=Notch
B=Bandpass

Regards,

Christian

--
unit KarlsenUnit;

interface

type

```
TKarsen = class
private
  fQ   : Single;
  fF1,fF : Single;
  fFS  : Single;
  fTmp  : Double;
  fOS   : Byte;
  fPole : Array[1..4] of Single;
  procedure SetFrequency(v:Single);
  procedure SetQ(v:Single);
public
  constructor Create;
  destructor Destroy; override;
  procedure Process(const l : Single; var L,B,N,H: Single);
  property Frequency: Single read fF write SetFrequency;
  property SampleRate: Single read fFS write fFS;
  property Q: Single read fQ write SetQ;
  property OverSample: Byte read fOS write fOS;
end;
```

implementation

uses sysutils;

const kDenorm = 1.0e-24;

constructor TKarsen.Create;

```
begin
  inherited;
  fFS:=44100;
  Frequency:=1000;
  fOS:=2;
  Q:=1;
end;
```

destructor TKarsen.Destroy;

```
begin
  inherited;
end;
```

procedure TKarsen.SetFrequency(v:Single);

```
begin
  if fFS<=0 then raise exception.create('Sample Rate Error!');
  if v<>fF then
  begin
    fF:=v;
    fF1:=fF/fFS; // fF1 range from 0..1
  end;
```

```

end;

procedure TKarlsen.SetQ(v:Single);
begin
if v<>fQ then
begin
if v<0 then fQ:=0 else
if v>50 then fQ:=50 else
fQ:=v;
end;
end;

procedure TKarlsen.Process(const I : Single; var L,B,N,H: Single);
var prevfp : Single;
intfp : Single;
o : Integer;
begin
for o:=0 to fOS-1 do
begin
prevfp:=fTmp;
if (prevfp > 1) then prevfp:=1; // Q-limiter
fTmp:=(fTmp*0.418)+((fQ*fPole[4])*0.582); // dynamic feedback
intfp:=(fTmp*0.36)+(prevfp*0.64); // feedback phase
fPole[1]:= (((1+kDenorm)-intfp) * fF1) + (fPole[1] * (1 - fF1));
if (fPole[1] > 1)
then fPole[1]:= 1
else if fPole[1] < -1
then fPole[1]:= -1;
fPole[2]:= (fPole[1]*fF1)+(fPole[2]*(1-fF1)); // pole 2
fPole[3]:= (fPole[2]*fF1)+(fPole[3]*(1-fF1)); // pole 3
fPole[4]:= (fPole[3]*fF1)+(fPole[4]*(1-fF1)); // pole 4
end;
L:=fPole[4];
B:=fPole[4]-fPole[1];
N:=1-fPole[1];
H:=1-fPole[4]-fPole[1];
end;
end.

```

from : unkargherth [[a t]] terras.es
comment : Thanks Christian!!

Anyway, i tried something similar and seems that what you call Notch is really a Bandpass and the bandpass makes something really strange

Anyway i'm having other problems with this filter too. It seems to cut too Low for low pass and too high for high pass. Also, resonance sets a peak far away from the cut frequency. And last but not least, the slope isn't 24 db/oct, really is much lesser, but not in a consistent way: sometimes is 6, sometimes 12, sometimes 20, etc

Any ideas ?

from : mail [[a t]] ihsan-dsp.com
comment : Your problem sounds a bit strange, maybe you should check your implementation.

Nice to see a pascal version too, Christian!

Although I really recommend one set a lower denormal threshold, maybe a 1/100, it really affects the sound of the filter. The best is probably tweaking that value in realtime to see what sounds best.
Also, doubles for the buffers.. :)

Very Best Regards,
Ove Karlsen

from : musicdsp [[a t]] dsparsons.co.uk
comment : Christian, shouldn't your code end:

```

L:=fPole[4];
B:=fPole[4]-fPole[1];
//CWB posted
//N:=1-fPole[1];
//B:=1-fPole[4]-fPole[1];

```

```

//DSP posted
H:=1-fPole[4]; //Surely pole 4 would give a 24dB/Oct HP, rather than the 6dB version posted
N:=1-fPole[4]-fPole[1]; //Inverse of BP

```

Any thoughts, anyone?

DSP

from : mail [[a t]] ihsan-dsp.com


```
float b_cut = ((fvar1 * fvar1) + ((fvar1 / (b_slope)) * (1 - fvar1)))
/ ((1 * fvar1) + ((1 / (b_slope)) * (1 - fvar1)));
```

Rename for convenience and clarity

```
fvar1=co
b_slope=sl
```

```
=> (co^2+(co(1-co)))
-----
      sl
-----
(1*co)+(1-co)
-----
      sl
```

multiply numerator & denominator by sl to even things up

```
=> (sl*co^2+(co(1-co)))
-----
      (sl*co)+(1-co)
```

expand brackets

```
=> sl*co^2+co-co^2
-----
      sl*co+1-co
```

refactor

```
=> co(sl*co+1-co)
-----
      sl*co+1-co
```

(sl*co+1-co) cancels out, leaving..

```
=> co
```

if I've got anything wrong here, please pipe up..

Duncan

from : musicdsp [[a t]] dsparsons.co.uk
comment : (actually, typing the assignment into Excel reveals the same as my proof..)

from : mail [[a t]] ihsan-dsp.com
comment : Final version, Stenseth, 17. february, 2006.

```
// Fast differential amplifier approximation
```

```
double b_inr = b_in * b_filterdrive;
if (b_inr < 0) {b_inr = -b_inr;}
double b_inrns = b_inr;
if (b_inr > 1) {b_inr = 1;}
double b_dax = b_inr - ((b_inr * b_inr) * 0.5);
b_dax = b_dax - b_inr;
b_inr = b_inr + b_dax;

b_inr = b_inr * 0.24;

if (b_inr > 1) {b_inr = 1;}
b_dax = b_inr - ((b_inr * 0.33333333) * (b_inr * b_inr));
b_dax = b_dax - b_inr;
b_inr = b_inr + b_dax;

b_inr = b_inr / 0.24;

double b_mul = b_inrns / b_inr; // beware of zero
b_sbuf1 = ((b_sbuf1 - (b_sbuf1 * 0.4300)) + (b_mul * 0.4300));

b_mul = b_sbuf1 + ((b_mul - b_sbuf1) * 0.6910);
b_in = b_in / b_mul;
```

```
// This method sounds the best here..
```

// About denormals, it does not seem to be much of an issue here, probably because I input the filters with oscillators, and not samples, or other, where the level may drop below the denormal threshold for extended periods of time. However, if you do, you probably want to quantize out the information below the threshold, in the buffers, and raise/lower the inputlevel before/after the filter. Adding low levels of noise may be effective aswell. This is described somewhere else on this site.

```
double b_cutsc = pow(1024,b_cut) / 1024; // perfect tracking..
```

```
b_fbuf1 = ((b_fbuf1 - (b_fbuf1 * b_cutsc)) + (b_in * b_cutsc));
b_in = b_fbuf1;
b_fbuf2 = ((b_fbuf2 - (b_fbuf2 * b_cutsc)) + (b_in * b_cutsc));
b_in = b_fbuf2;
```



```
b_fb3 = ((b_fb3 - (b_fb3 * b_cutsc)) + (b_in * b_cutsc));
b_in = b_fb3;
b_fb4 = ((b_fb4 - (b_fb4 * b_cutsc)) + (b_in * b_cutsc));
b_in = b_fb4;
```

Soundwise, it's somewhere between a transistor ladder, and a diode ladder.
Enjoy!

Ove Karlsen.

PS: I prefer IRL communication these days, so if you need to reach me, please dial my cellphone, +047 928 50 803.

from : read [[a t]] bw

comment : Another iteration, please delete all other posts than this.

Arif Ove Karlsen's 24dB Ladder Approximation, 3.nov 2007

As you may know, The traditional 4-pole Ladder found in vintage hardware synths, had a particular sound. The nonlinearities inherent in the suboptimal components, often added a particular flavour to the sound.

Digital does mathematical calculations much better than any analog solution, and therefore, when the filter was emulated by digital filter types, some of the character got lost.

I believe this mainly boils down to the resonance limiting occurring in the analog version.

Therefore I have written a very fast ladder approximation, not emulating any of what may seem necessary, such as pole saturation, which in turn results in nonlinear cutoff frequency, and loss of volume at lower cutoffs. However this can be implemented, if wanted, by putting the necessary saturation functions inside the code. If you seek the true analog sound, you may want to do a full differential amplifier emulation as well.

But - I believe in the end, you would end up wanting a perfect filter, with just the touch that makes it sound analog, resonance limiting.

So here it is, Karlsen Ladder, v4. A very resource efficient ladder. Can furthermore be optimized with asm.

```
rez = pole4 * rezamount; if (rez > 1) {rez = 1;}
input = input - rez;
pole1 = pole1 + ((-pole1 + input) * cutofffreq);
pole2 = pole2 + ((-pole2 + pole1) * cutofffreq);
pole3 = pole3 + ((-pole3 + pole2) * cutofffreq);
pole4 = pole4 + ((-pole4 + pole3) * cutofffreq);
output = pole4;
```

--

I can be reached by email @ 1a2r4i54f5o5v2ek1a1r5ls6en@3ho2tm6ail1.c5o6m!no!nums

from : pissed [[a t]] off.com

comment : I modeled some instruments using this filter... everything was fine until... i tried to change sample rate.

CUTOFFS CHANGED, RESOS CHANGED!

(my implementation is 100% as Ove's)

This filter is good only if You want to use one sample rate.

Joe

from : whygivemyemail [[a t]] oklwill.com

comment : Samplerate? Do you mean cutoff frequency?

Dave

from : the [[a t]] same.as.above

comment : I mean sampling frequency. Everything is ok with only one SR eg. 44100. When You change it to eg. 192000 the filter gives completely different cutoff frequency and resonance.

[Karlsten Fast Ladder](#) (click this to go back to the index)

Type : 4 pole ladder emulation

References : Posted by arifovekarlsen[AT]hotmail[DOT]com

Notes :

ATTN Admin: You should remove the old version named "Karlsten" on your website, and rather include this one instead.

Code :

```
// An updated version of "Karlsten 24dB Filter"  
// This time, the fastest incarnation possible.  
// The very best greetings, Arif Ove Karlsten.  
// arifovekarlsen->hotmail.com
```

```
b_rscl = b_buf4; if (b_rscl > 1) {b_rscl = 1;}  
b_in = (-b_rscl * b_rez) + b_in;  
b_buf1 = ((-b_buf1 + b_in1) * b_cut) + b_buf1;  
b_buf2 = ((-b_buf2 + b_buf1) * b_cut) + b_buf2;  
b_buf3 = ((-b_buf3 + b_buf2) * b_cut) + b_buf3;  
b_buf4 = ((-b_buf4 + b_buf3) * b_cut) + b_buf4;  
b_lpout = b_buf4;
```

Comments

from : nobody [[a t]] nowhere.com

comment : Where are the coefficients? How do I set the cutoff frequency?

from : scoofy [[a t]] inf.elte.hu

comment : The parameters are:

b_cut - cutoff freq
b_rez - resonance
b_in1 - input

Cutoff is normalized frequency in rads ($2\pi \cdot \text{cutoff}/\text{samplerate}$). Stability limit for b_cut is around 0.7-0.8.

There's a typo, the input is sometimes b_in, sometimes b_in1. Anyways why do you use a b_ prefix for all your variables? Wouldn't it be more easy to read like this:

```
resoclip = buf4; if (resoclip > 1) resoclip = 1;  
in = in - (resoclip * res);  
buf1 = ((in - buf1) * cut) + buf1;  
buf2 = ((buf1 - buf2) * cut) + buf2;  
buf3 = ((buf2 - buf3) * cut) + buf3;  
buf4 = ((buf3 - buf4) * cut) + buf4;  
lpout = buf4;
```

Also note that asymmetrical clipping gives you DC offset (at least that's what I get), so symmetrical clipping is better (and gives a much smoother sound).

-- peter schoffhauzer

from : arif [[a t]] str8dsp.com

comment : Tee b_ prefix is simply a procedure I began using when I started programming C. Influenced by the BEOS operating system. However it seemed to also make my code more readable, atleast to me. So I started using various prefixes for various things, making the variables easily recognizable. Peter, everyone, I am now reachable on www.str8dsp.com - Do also check out the plugin offers there!

from : aok [[a t]] str8dsp.com

comment : Here's even another filter, I will probably never get around to making any product with this one so here it is, pseudo-vintage diode ladder.

Diode Ladder, (unbuffered)

```
// limit resonance, rzl, tweak smearing with fltw, 0.3230 seems to be a good vintage sound.  
in = in - rzl;  
in = in + ((-in + kbuf1) * cutoff);  
kbuf1 = in + ((-in + kbuf1) * fltw);  
in = in + ((-in + kbuf2) * cutoff);  
kbuf2 = in + ((-in + kbuf2) * fltw);  
etc..
```

from : dev [[a t]] fxpointaudio.com

comment : "Cutoff is normalized frequency in rads ($2\pi \cdot \text{cutoff}/\text{samplerate}$):

This seems to be valid for very low (< 200 Hz) frequencies - higher sample rates seem to be "Closer"

thanks

Lowpass filter for parameter edge filtering (click this to go back to the index)

References : Olli Niemitalo

Linked file : [filter001.gif](#)

Notes :

use this filter to smooth sudden parameter changes
(see linkfile!)

Code :

```
/* - Three one-poles combined in parallel
 * - Output stays within input limits
 * - 18 dB/oct (approx) frequency response rolloff
 * - Quite fast, 2x3 parallel multiplications/sample, no internal buffers
 * - Time-scalable, allowing use with different samplerates
 * - Impulse and edge responses have continuous differential
 * - Requires high internal numerical precision
 */
{
    /* Parameters */
    // Number of samples from start of edge to halfway to new value
    const double scale = 100;
    // 0 < Smoothness < 1. High is better, but may cause precision problems
    const double smoothness = 0.999;

    /* Precalc variables */
    double a = 1.0-(2.4/scale); // Could also be set directly
    double b = smoothness; // -"-
    double acoef = a;
    double bcoef = a*b;
    double ccoef = a*b*b;
    double mastergain = 1.0 / (-1.0/(log(a)+2.0*log(b))+2.0/
        (log(a)+log(b))-1.0/log(a));
    double again = mastergain;
    double bgain = mastergain * (log(a*b*b)*(log(a)-log(a*b)) /
        ((log(a*b*b)-log(a*b))*log(a*b))
        - log(a)/log(a*b));
    double cgain = mastergain * (-log(a)-log(a*b)) /
        (log(a*b*b)-log(a*b));

    /* Runtime variables */
    long streamofs;
    double areg = 0;
    double breg = 0;
    double creg = 0;

    /* Main loop */
    for (streamofs = 0; streamofs < streamsize; streamofs++)
    {
        /* Update filters */
        areg = acoef * areg + fromstream [streamofs];
        breg = bcoef * breg + fromstream [streamofs];
        creg = ccoef * creg + fromstream [streamofs];

        /* Combine filters in parallel */
        long temp = again * areg
            + bgain * breg
            + cgain * creg;

        /* Check clipping */
        if (temp > 32767)
        {
            temp = 32767;
        }
        else if (temp < -32768)
        {
            temp = -32768;
        }

        /* Store new value */
        tostream [streamofs] = temp;
    }
}
```

Comments

from : scoofy [[a t]] inf.elte.hu

comment : Wouldn't just one pole with a low cutoff suit this purpose? At least that's what I usually do for smoothing parameter changes, and it works fine.

LP and HP filter (click this to go back to the index)

Type : biquad, tweaked butterworth

References : Posted by Patrice Tarrabia

Code :

```
r = rez amount, from sqrt(2) to ~ 0.1
f = cutoff frequency
(from ~0 Hz to SampleRate/2 - though many
synths seem to filter only up to SampleRate/4)
```

The filter algo:

```
out(n) = a1 * in + a2 * in(n-1) + a3 * in(n-2) - b1*out(n-1) - b2*out(n-2)
```

Lowpass:

```
c = 1.0 / tan(pi * f / sample_rate);

a1 = 1.0 / ( 1.0 + r * c + c * c );
a2 = 2 * a1;
a3 = a1;
b1 = 2.0 * ( 1.0 - c*c ) * a1;
b2 = ( 1.0 - r * c + c * c ) * a1;
```

Hipass:

```
c = tan(pi * f / sample_rate);

a1 = 1.0 / ( 1.0 + r * c + c * c );
a2 = -2 * a1;
a3 = a1;
b1 = 2.0 * ( c*c - 1.0 ) * a1;
b2 = ( 1.0 - r * c + c * c ) * a1;
```

Comments

from : andy_rossol [[a t]] hotmail.com

comment : Ok, the filter works, but how to use the resonance parameter (r)? The range from sqrt(2)-lowest to 0.1 (highest res.) is Ok for a LP with Cutoff > 3 or 4 KHz, but for lower cutoff frequencies and higher res you will get values much greater than 1! (And this means clipping like hell)

So, has anybody calculated better parameters (for r, b1, b2)?

from : kainhart [[a t]] hotmail.com

comment : Below is my attempt to implement the above lowpass filter in c#. I'm just a beginner at this so it's probably something that I've messed up. If anybody can offer a suggestion of what I may be doing wrong please help. I'm getting a bunch of stable staticky noise as my output of this filter currently.

from : kainhart [[a t]] hotmail.com

```
comment : public class LowPassFilter
{
    /// <summary>
    /// rez amount, from sqrt(2) to ~ 0.1
    /// </summary>
    float r;
    /// <summary>
    /// cutoff frequency
    /// (from ~0 Hz to SampleRate/2 - though many
    /// synths seem to filter only up to SampleRate/4)
    /// </summary>
    float f;
    float c;

    float a1;
    float a2;
    float a3;
    float b1;
    float b2;

//    float in0 = 0;
//    float in1 = 0;
//    float in2 = 0;

//    float out0;
//    float out1 = 0;
//    float out2 = 0;

    private int _SampleRate;

    public LowPassFilter(int sampleRate)
    {
        _SampleRate = sampleRate;

//        SetParams(_SampleRate / 2f, 0.1f);
//        SetParams(_SampleRate / 8f, 1f);
    }
}
```

```

public float Process(float input)
{
    float output = a1 * input +
                  a2 * in1 +
                  a3 * in2 -
                  b1 * out1 -
                  b2 * out2;

    in2 = in1;
    in1 = input;

    out2 = out1;
    out1 = output;

    Console.WriteLine(input + ", " + output);

    return output;
}

```

from : kainhart [[a t]] hotmail.com

comment : `/// <summary>`

```

///
/// </summary>
public float CutoffFrequency
{
    set
    {
        f = value;
        c = (float) (1.0f / Math.Tan(Math.PI * f / _SampleRate));
        SetParams();
    }
    get
    {
        return f;
    }
}

```

`/// <summary>`

`///`

`/// </summary>`

`public float Resonance`

```

{
    set
    {
        r = value;
        SetParams();
    }
    get
    {
        return r;
    }
}

```

`public void SetParams(float cutoffFrequency, float resonance)`

```

{
    r = resonance;
    CutoffFrequency = cutoffFrequency;
}

```

`/// <summary>`

`/// TODO rename`

`/// </summary>`

`/// <param name="c"></param>`

`/// <param name="resonance"></param>`

`private void SetParams()`

```

{
    a1 = 1f / (1f + r*c + c*c);
    a2 = 2 * a1;
    a3 = a1;
    b1 = 2f * (1f - c*c) * a1;
    b2 = (1f - r*c + c*c) * a1;
}

```

}

from : kainhart [[a t]] hotmail.com

comment : Nevermind I think I solved my problem. I was missing parens around the coefficients and the variables ... $(a1 * input)$...

from : kainhart[AT]hotmail.com

comment : After implementing the lowpass algorithm I get a loud ringing noise on some frequencies both high and low. Any ideas?

from : ldahl [[a t]] gmx.de

comment : hi,
since this is the best filter i found on the net, i really need bandpass and bandstop!!! can anyone help me with the coefficients?

from : scoofy [[a t]] inf.elte.hu
comment : AFAIK there's no separate bandpass and bandstop version of Butterworth filters. Instead, bandpass is usually done by cascading a HP and a LP filter, and bandstop is the mixed output of a HP and a LP filter. However, there's bandpass biquad code (for example RBJ biquad filters). Cheers Peter

from : scoofy [[a t]] inf.elte.hu
comment : You can save two divisions for lowpass using
`c = tan((0.5 - (f * inv_samplerate))*pi);`
instead of
`c = 1.0 / tan(pi * f / sample_rate);`
where `inv_samplerate` is `1.0/sample_rate` precalculated. (mul is faster than div)

However, the latter form can be approximated very well below 4kHz (at 44kHz samplerate) with
`c = 1.0 / (pi * f * inv_sample_rate);`
which is far better than both of the previous two equations, because it does not use any transcendental functions. So, an optimized form is:

```
f0 = f * inv_sample_rate;  
if (f0 < 0.1) c = 1.0 / (f0 * pi); // below 4.4k  
else c = tan((0.5 - f0) * pi);
```

This needs only about ~60% CPU below 4.4kHz. Probably using lookup tables could make it even faster...

Mapping resonance range 0..1 to 0..self-osc:
`float const sqrt_two = 1.41421356;`
`r = sqrt_two - resonance * sqrt_two;`

Setting resonance in the conventional q form (like in RBJ biquads):
`r = 1.0/q;`

Cheers, Peter

from : scoofy [[a t]] elte.hu
comment : However I find that this algorithm has a slight tuning error regardless of using approximation or not. '`inv_samplerate = 0.95 * samplerate`' seems to give a more accurate frequency tuning.

from : scoofy [[a t]] inf.elte.hu
comment : You can use the same trick for highpass:

precalc when setting up the filter:
`inv_samplerate = 1.0 / samplerate * 0.957;`
(multiplying by 0.957 seems to give the most precise tuning)

and then calculating c:

```
f0 = f * inv_samplerate;  
if (f0 < 0.05) c = (f0 * pi);  
else c = tan(f0 * pi);
```

Now I used 0.05 instead of 0.1, thats `0.05 * 44100 = 2.2k` instead of 4.4k. So, this is a bit more precise than 0.1, because around 3-4k it had a slight error, however, only noticeable on the analyzer when compared to the original version. This is still about two third of the logarithmic frequency scale, so it's quite a bit of a speed improvement. You can use either precision for both lowpass and highpass.

For calculating `tan()`, you can take some quick `sin()` approximation, and use:
`tan(x)=sin(x)/sin(half_pi-x)`

There are many good pieces of code for that in the archive.

I tried to make some `1/x` based approximations for `1.0/tan(x)`, here is one:

```
inline float tan_inv_approx(float x)  
{  
    float const two_div_pi = 2.0f/3.141592654f;  
    if (x<0.5f) return 1.0f/x;  
    else return 1.467f*(1.0f/x-two_div_pi);  
}
```

This one is pretty fast, however it is a quite rough estimate; it has some 1-2 semitones frequency tuning error around 5-8 kHz and above 10kHz. Might be usable for synths, however, or somewhere where scientific precision is not needed.

Cheers, Peter

from : scoofy [[a t]] inf.elte.hu
comment : Sorry, forget the `* 0.957` tuning, this algorithm is precise without that, the mistake was in my program. Everything else is valid, I hope.

from : foxes [[a t]] bk.ru
comment : Optimization for Hipass:

```
c = tan(pi * f / sample_rate);
```

```
c = ( c + r ) * c;
```

```
a1 = 1.0 / ( 1.0 + c );
```

```
b1 = ( 1.0 - c );
```

```
out(n) = ( a1 * out(n-1) + in - in(n-1) ) * b1;
```

[LPF 24dB/Oct](#) (click this to go back to the index)

Type : Chebyshev

References : Posted by Christian[AT]savioursofsoul[DOT]de

Code :

First calculate the prewarped digital frequency:

```
K = tan(Pi * Frequency / Samplerate);
```

Now we calc some Coefficients:

```
sg = Sinh(PassbandRipple);
cg = Cosh(PassbandRipple);
cg *= cg;
```

```
Coeff[0] = 1 / (cg-0.85355339059327376220042218105097);
Coeff[1] = K * Coeff[0]*sg*1.847759065022573512256366378792;
Coeff[2] = 1 / (cg-0.14644660940672623779957781894758);
Coeff[3] = K * Coeff[2]*sg*0.76536686473017954345691996806;
```

```
K *= K; // (just to optimize it a little bit)
```

Calculate the first biquad:

```
A0 = (Coeff[1]+K+Coeff[0]);
A1 = 2*(Coeff[0]-K)*t;
A2 = (Coeff[1]-K-Coeff[0])*t;
B0 = t*K;
B1 = 2*B0;
B2 = B0;
```

Calculate the second biquad:

```
A3 = (Coeff[3]+K+Coeff[2]);
A4 = 2*(Coeff[2]-K)*t;
A5 = (Coeff[3]-K-Coeff[2])*t;
B3 = t*K;
B4 = 2*B3;
B5 = B3;
```

Then calculate the output as follows:

```
Stage1 = B0*Input + State0;
State0 = B1*Input + A1/A0*Stage1 + State1;
State1 = B2*Input + A2/A0*Stage1;

Output = B3*Stage1 + State2;
State2 = B4*Stage1 + A4/A3*Output + State2;
State3 = B5*Stage1 + A5/A3*Output;
```

Comments

[from](#) : musicdsp[at] Nospam dsparsons[dot]co[dot]uk

[comment](#) : You've used two notations here (as admitted on KVR!)..

Updated calculation code reads:

```
==== Start ====
```

Calculate the first biquad:

```
//A0 = (Coeff[1]+K+Coeff[0]);
t = 1/(Coeff[1]+K+Coeff[0]);
A1 = 2*(Coeff[0]-K)*t;
A2 = (Coeff[1]-K-Coeff[0])*t;
B0 = t*K;
B1 = 2*B0;
B2 = B0;
```

Calculate the second biquad:

```
//A3 = (Coeff[3]+K+Coeff[2]);
t = 1/(Coeff[3]+K+Coeff[2]);
A4 = 2*(Coeff[2]-K)*t;
A5 = (Coeff[3]-K-Coeff[2])*t;
B3 = t*K;
B4 = 2*B3;
B5 = B3;
```

Then calculate the output as follows:

```
Stage1 = B0*Input + State0;
State0 = B1*Input + A1*Stage1 + State1;
State1 = B2*Input + A2*Stage1;

Output = B3*Stage1 + State2;
State2 = B4*Stage1 + A4*Output + State2;
```


State3 = B5*Stage1 + A5*Output;

==--== End ==--==

Hope that clears up any confusion for future readers :-)

from : neolit123 [[a t]] gmail.com

comment : Just ported this into Reaper's native JesuSonic.

There are errors in both of the codes above :D

Use this:

//start

A0 = 1/(Coeff[1]+K+Coeff[0]);

A1 = 2*(Coeff[0]-K)*A0;

A2 = (Coeff[1]-K-Coeff[0])*A0;

B0 = A0*K;

B1 = 2*B0;

B2 = B0;

A3 = 1/(Coeff[3]+K+Coeff[2]);

A4 = 2*(Coeff[2]-K)*A3;

A5 = (Coeff[3]-K-Coeff[2])*A3;

B3 = A3*K;

B4 = 2*B3;

B5 = B3;

Stage1 = B0*Input + State0;

State0 = B1*Input + A1*Stage1 + State1;

State1 = B2*Input + A2*Stage1;

Output = B3*Stage1 + State2;

State2 = B4*Stage1 + A4*Output + State3;

State3 = B5*Stage1 + A5*Output;

//end

@RossClement[AT]gmail[DOT]com

'State3' should be added in this line

-> State2 = B4*Stage1 + A4*Output + State3;

from : RossClement [[a t]] gmail.com

comment : The variable State3 is assigned a value, but is never used anywhere. Is there a reason for this?

Moog Filter (click this to go back to the index)

Type : Antti's version (nonlinearities)

References : Posted by Christian[at]savioursofsoul[dot]de

Notes :

Here is a Delphi/Object Pascal translation of Antti's Moog Filter.

Antti wrote:

"At last DAFX I published a paper presenting a non-linear model of the Moog ladder. For that, see http://dafx04.na.infn.it/WebProc/Proc/P_061.pdf

I used quite different approach in that one. A half-sample delay ([0.5 0.5] FIR filter basically) is inserted in the feedback loop. The remaining tuning and resonance error are corrected with polynomials. This approach depends on using at least 2X oversampling - the response after nyquist/2 is abysmal but that's taken care of by the oversampling.

Victor Lazzarini has implemented my model in CSound:
http://www.csounds.com/udo/displayOpcode.php?opcode_id=32

In summary: You can use various methods, but you will need some numerically derived correction to realize exact tuning and resonance control. If you can afford 2X oversampling, use Victor's CSound code - the tuning has been tested to be very close ideal.

Ps. Remember to use real oversampling instead of the "double sampling" the CSound code uses."

I did not implemented real oversampling, but i inserted additional noise, which simulates the resistance noise and also avoids denormal problems...

Code :

<http://www.savioursofsoul.de/Christian/MoogFilter.pas>

Comments

from : Christian [[a t]] savioursofsoul.de

comment : You can also listen to it (Windows-VST) here: <http://www.savioursofsoul.de/Christian/VST/MoogVST.zip>

from : rlindner at gmx ..dot.. net

comment : and here is the same thing written in C. It was written while translating the CSound Code into code for the synthmaker code module as an intermediate step to enable debugging thru gdb. The code was written to be easy adoptable for the synthmaker code module (funny defines, static vars, single sample tick function,...) Has some room for improvements, but nothing fancy for seasoned C programmers.

```
#include <memory.h>
#include <stdio.h>
#include <math.h>
```

```
#define polyin float
#define polyout float
```

```
#define BUFSIZE 64
```

```
float delta_func [BUFSIZE];
float out_buffer [BUFSIZE];
```

```
void tick ( float in, float cf, float reso, float *out ) {
```

```
// start of sm code
```

```
// filter based on the text "Non linear digital implementation of the moog ladder filter" by Antti Houvilainen
// adopted from Csound code at http://www.kunstmusik.com/udo/cache/moogladder.udo
polyin input;
polyin cutoff;
polyin resonance;

polyout sigout;
```

```
// remove this line in sm
input = in; cutoff = cf; resonance = reso;
```

```
// resonance [0..1]
// cutoff from 0 (0Hz) to 1 (nyquist)
```

```
float pi; pi = 3.1415926535;
```

```

float v2; v2 = 40000; // twice the 'thermal voltage of a transistor'
float sr; sr = 22100;

float cutoff_hz;
cutoff_hz = cutoff * sr;

static float az1;
static float az2;
static float az3;
static float az4;
static float az5;
static float ay1;
static float ay2;
static float ay3;
static float ay4;
static float amf;

float x; // temp var: input for taylor approximations
float xabs;
float exp_out;
float tanh1_out, tanh2_out;
float kfc;
float kf;
float kfc_r;
float kac_r;
float k2vg;

kfc = cutoff_hz/sr; // sr is half the actual filter sampling rate
kf = cutoff_hz/(sr*2);
// frequency & amplitude correction
kfc_r = 1.8730*(kfc*kfc*kfc) + 0.4955*(kfc*kfc) - 0.6490*kfc + 0.9988;
kac_r = -3.9364*(kfc*kfc) + 1.8409*kfc + 0.9968;

x = -2.0 * pi * kfc_r * kf;
exp_out = expf(x);

k2vg = v2*(1-exp_out); // filter tuning

// cascade of 4 1st order sections
float x1 = (input - 4*resonance*amf*kac_r) / v2;
float tanh1 = tanhf (x1);
float x2 = az1/v2;
float tanh2 = tanhf (x2);
ay1 = az1 + k2vg * ( tanh1 - tanh2);

// ay1 = az1 + k2vg * ( tanh( (input - 4*resonance*amf*kac_r) / v2) - tanh(az1/v2) );
az1 = ay1;

ay2 = az2 + k2vg * ( tanh(ay1/v2) - tanh(az2/v2) );
az2 = ay2;

ay3 = az3 + k2vg * ( tanh(ay2/v2) - tanh(az3/v2) );
az3 = ay3;

ay4 = az4 + k2vg * ( tanh(ay3/v2) - tanh(az4/v2) );
az4 = ay4;

// 1/2-sample delay for phase compensation
amf = (ay4+az5)*0.5;
az5 = ay4;

// oversampling (repeat same block)
ay1 = az1 + k2vg * ( tanh( (input - 4*resonance*amf*kac_r) / v2) - tanh(az1/v2) );
az1 = ay1;

ay2 = az2 + k2vg * ( tanh(ay1/v2) - tanh(az2/v2) );
az2 = ay2;

ay3 = az3 + k2vg * ( tanh(ay2/v2) - tanh(az3/v2) );
az3 = ay3;

ay4 = az4 + k2vg * ( tanh(ay3/v2) - tanh(az4/v2) );
az4 = ay4;

// 1/2-sample delay for phase compensation
amf = (ay4+az5)*0.5;
az5 = ay4;

sigout = amf;

```

```

// end of sm code

*out = sigout;
} // tick

int main ( int argc, char *argv[] ) {

// set delta function
memset ( delta_func, 0, sizeof(delta_func));
delta_func[0] = 1.0;

int i = 0;
for ( i = 0; i < BUFSIZE; i++ ) {
tick ( delta_func[i], 0.6, 0.7, out_buffer+i );
}
for ( i = 0; i < BUFSIZE; i++ ) {
printf ("%f;", out_buffer[i] );
}
printf ( "\n" );

} // main

```

from : didid [[a t]] skynet.be
comment : I think that a better speed optimization of Tanh2 would be to extract the sign bit (using integer) instead of abs, and add it back to the final result, to avoid FABS, FCOMP and the branching

from : Christian [[a t]] savioursofsoul.de
comment : After reading some more assembler documents for university, i had the same idea...
Now: Coding!

from : didid [[a t]] skynet.be
comment : Btw1, is the idea to get rid of the "*0.5" in the "1/2 sample delay" block by using *2 instead of *4 in the first filter?

Btw2, following the same simplification, you can also precompute "-2*fQ*fAcr" outside.

from : didid [[a t]] skynet.be
comment : Forget the sign bit thing, actually your Tanh could already have been much faster at the source:

```

a:=f_abs(x);
a:=a*(6+a*(3+a));
if (x<0)
then Result:=-a/(a+12)
else Result:= a/(a+12);

```

can be written as the much simpler:

Result:=x*(6+Abs(x)*(3+Abs(x)))/(Abs(x)+12)

..so in asm:

```

function Tanh2(x:Single):Single;
const c3 :Single=3;
      c6 :Single=6;
      c12:Single=12;

```

```

Asm
  FLD  x
  FLD  ST(0)
  FABS
  FLD  c3
  FADD ST(0),ST(1)
  FMUL ST(0),ST(1)
  FADD c6
  FMUL ST(0),ST(2)
  FXCH ST(1)
  FADD c12
  FDIVP ST(1),ST(0)
  FSTP ST(1)

```

End;

..but it won't be much faster than your code, because:

-of the slow FDIV

-it's still a function call. Since our dumb Delphi doesn't support assembler macro's, you waste a lot in the function call. You can still try to inline a plain pascal code, but since our dumb Delphi isn't good at compiling float code neither..

Solutions:

-a lookup table for the TanH

-you write the filter processing in asm as well, and you put the TanH code in a separate file (without the header, and assuming in & out are ST(0)). You then \$! to insert that file when the function call is needed. Poorman's macro's in Delphi :)

Still, that's a lot of FDIV for a filter..

from : didid [[a t]] skynet.be
comment : forget it, I was all wrong :)
gonna re-post a working version later

still I think that most of the CPU will always be wasted by the division.

from : didid [[a t]] skynet.be
comment : Ignore the above, here it is working (for this code, assuming a premultiplied x):

```
function Tanh2(x:Single):Single;
```

```
const c3 :Single=3;
```

```
      c6 :Single=6;
```

```
      c12 :Single=12;
```

```
Asm
```

```
  FLD  x  
  FLD  ST(0)  
  FABS           // a  
  FLD  c3  
  FADD  ST(0),ST(1)  
  FMUL  ST(0),ST(1)  
  FADD  c6           // b  
  FMUL  ST(2),ST(0) // x*b  
  FMULP ST(1),ST(0) // a*b  
  FADD  c12  
  FDIVP ST(1),ST(0)
```

```
End;
```

from : Christian [[a t]] savioursofsoul.de
comment : That code is nice and very fast! But it's not that accurate. But indeed very fast! Thanks for the code. My approach was a lot slower.

from : didid [[a t]] skynet.be
comment : But it should be as accurate as the one in your pascal code: it's the same approximation as Tanh2_pas2. Of course we're still talking about a Tanh approximation. You can still take it up to /24 for cheap.
Of course, don't try the first one I posted, it's completely wrong.

Btw it's also more accurate than pascal code, since it keeps values in the 80bit FPU registers, while the Delphi compiler will put them back to the 32bit variables in-between.

Btw2 I implemented the {\$! macro.inc} trick, works pretty well. The whole thing speeded up the code by almost 2. For more you could 3DNow 2 Tanh at once, it'd be easy in that filter.

from : Christian [[a t]] savioursofsoul.de
comment : OK, it's indeed my tanh2_pas2 approximation, but i've plotted both functions once and i found out, that the /24 version is much more accurate and that it is worth to calculate the additional macc operation.

But of course the assembler version is much more accurate indeed.

After assembler optimization, i could only speedup the whole thing to a factor of 1.5 the speed (measured with an impulse) and up to a factor of 1.7 measured with noise and the initial version with the branch.

I will now do the 3DNow/SSE optimisation, let's see how it can be speeded up further more...

from : didid [[a t]] skynet.be
comment : something bugs me about this filter.. I was assuming that it was made for a standard -1..1 normalized input. But looking at the 1/40000 drive gain, isn't it made for a -32768..32767 input? Otherwise I don't see what the Tanh drive is doing, it's basically linear for such low values, and I can't hear any difference with or without it.

from : Christian [[a t]] savioursofsoul.de
comment : Usually the tanh component wasn't a desired feature in the analog filter design. They try to keep the input of a differential amplifier very low to retain the linearity.

I have uploaded some plots from my VST Plugin Analyser Pro:

<http://www.savioursofsoul.de/Christian/VST/filter4.png> (with -1..+1)

<http://www.savioursofsoul.de/Christian/VST/filter5.png> (with -32768..+32767)

<http://www.savioursofsoul.de/Christian/VST/filter1.png> (other...)

<http://www.savioursofsoul.de/Christian/VST/filter2.png> (other...)

<http://www.savioursofsoul.de/Christian/VST/filter3.png> (other...)

Additionally i have updated the VST with a new slider for a gain multiplication (<http://www.savioursofsoul.de/Christian/VST/MoogVST.zip>)

from : fintain_dowd [[a t]] gmail.com

comment : There is an error in the C implementation above,
sr = 44100Hz, half the rate of the filter which is oversampled at a rate of 88200Hz. So the 22100 needs to be changed.
Christians MoogFilter.pas implements it correctly.

Moog VCF (click this to go back to the index)

Type : 24db resonant lowpass

References : CSound source code, Stilson/Smith CCRMA paper.

Notes :

Digital approximation of Moog VCF. Fairly easy to calculate coefficients, fairly easy to process algorithm, good sound.

```
Code :
//Init
cutoff = cutoff freq in Hz
fs = sampling frequency //(e.g. 44100Hz)
res = resonance [0 - 1] //(minimum - maximum)

f = 2 * cutoff / fs; //[0 - 1]
k = 3.6*f - 1.6*f*f -1; //(Empirical tuning)
p = (k+1)*0.5;
scale = e^((1-p)*1.386249);
r = res*scale;
y4 = output;

y1=y2=y3=y4=oldx=oldy1=oldy2=oldy3=0;

//Loop
/--Inverted feed back for corner peaking
x = input - r*y4;

//Four cascaded onepole filters (bilinear transform)
y1=x*p + oldx*p - k*y1;
y2=y1*p+oldy1*p - k*y2;
y3=y2*p+oldy2*p - k*y3;
y4=y3*p+oldy3*p - k*y4;

//Clipper band limited sigmoid
y4 = y4 - (y4^3)/6;

oldx = x;
oldy1 = y1;
oldy2 = y2;
oldy3 = y3;
```

Comments

from : hagenkaiser [[a t]] gmx.de

comment : I guess the input is supposed to be between -1..1

from : toast [[a t]] nowhere.com

comment : OK. Can't guarantee this is a 100% translation to real C++, but it does work. Aside from possible mistakes, I mucked hard with the coefficient translation, so that there is a slight difference in the numbers.

1. What kind of filter ends up with exp() in the coefficient calculation instead of the usual sin(), cos(), tan() transcendentals? If someone can explain where the e to the x comes from, I'd appreciate it.

2. Since I didn't understand the origin of the coefficients, I saw the whole section as an exercise in algebra. There were some superfluous multiplications and additions.

First I implemented the e to the x with pow(). That was stupid. I switched to exp(). Hated that too. Checked out the range of inputs and decided on a common approximation for exp().

I think it's now one of the fastest filter coefficient calcs you'll see for a 4 pole. Go ahead--put the cutoff on an envelope and the q on an LFO. Hell, FM the Q if you want!

A pretty good filter. Watch the res (aka Q). Above 9, squeals like a pig. Not in a good way.

Needs more work, i think, to stand up with the better LPs in the real VST world. But an awfully cool starting place.

If I did something awful here in the translation, or if you have a question about how to use it, best to ask me (mistertoast) over at KvRadio.com in the dev section.

I'm toying with the idea of wedging this into TobyBear's filter designer as a flt file. If you manage first, let me know. I'm mistertoast.

from : nobody [[a t]] nowhere.com

comment : Still working on this one. Anyone notice it's got a few syntax problems? Makes me wonder if it's even been tried.

Missing parenthesis. Uses ^ twice. If I get it to work, I'll post usable code.

from : toast [[a t]] nowhere.com

comment : MoogFilter.h:

```
class MoogFilter
{
public:
MoogFilter();
```

```

void init();
void calc();
float process(float x);
~MoogFilter();
float getCutoff();
void setCutoff(float c);
float getRes();
void setRes(float r);
protected:
float cutoff;
float res;
float fs;
float y1,y2,y3,y4;
float oldx;
float oldy1,oldy2,oldy3;
float x;
float r;
float p;
float k;
};

```

```

from : toast [ [ a t ] ] nowhere.com
comment :
MoogFilter.cpp:

```

```

#include "MoogFilter.h"

```

```

MoogFilter::MoogFilter()
{
fs=44100.0;

```

```

init();
}

```

```

MoogFilter::~MoogFilter()
{
}

```

```

void MoogFilter::init()
{
// initialize values
y1=y2=y3=y4=oldx=oldy1=oldy2=oldy3=0;
calc();
};

```

```

void MoogFilter::calc()
{
float f = (cutoff+cutoff) / fs; //[0 - 1]
p=f*(1.8f-0.8*f);
k=p+p-1.f;

```

```

float t=(1.f-p)*1.386249f;
float t2=12.f+t*t;
r = res*(t2+6.f*t)/(t2-6.f*t);
};

```

```

float MoogFilter::process(float input)
{
// process input
x = input - r*y4;

```

```

//Four cascaded onepole filters (bilinear transform)
y1= x*p + oldx*p - k*y1;
y2=y1*p + oldy1*p - k*y2;
y3=y2*p + oldy2*p - k*y3;
y4=y3*p + oldy3*p - k*y4;

```

```

//Clipper band limited sigmoid
y4=(y4*y4*y4)/6.f;

```

```

oldx = x; oldy1 = y1; oldy2 = y2; oldy3 = y3;
return y4;
}

```

```

float MoogFilter::getCutoff()
{ return cutoff; }

```

```

void MoogFilter::setCutoff(float c)
{ cutoff=c; calc(); }

```

```

float MoogFilter::getRes()
{ return res; }

```



```
void MoogFilter::setRes(float r)
{ res=r; calc(); }
```

from : toast [[a t]] somewhereyoucantfind.com

comment : I see where the exp() comes from. It just models the resonance. I think it needs more work. At high frequencies it goes into self-oscillation much more quickly than at low frequencies.

Moog VCF, variation 1 (click this to go back to the index)

Type : 24db resonant lowpass

References : CSound source code, Stilson/Smith CCRMA paper., Paul Kellett version

Notes :

The second "q =" line previously used exp() - I'm not sure if what I've done is any faster, but this line needs playing with anyway as it controls which frequencies will self-oscillate. I think it could be tweaked to sound better than it currently does.

Highpass / Bandpass :

They are only 6dB/oct, but still seem musically useful - the 'fruity' sound of the 24dB/oct lowpass is retained.

Code :

```
// Moog 24 dB/oct resonant lowpass VCF
// References: CSound source code, Stilson/Smith CCRMA paper.
// Modified by paul.kellett@maxim.abel.co.uk July 2000

float f, p, q;          //filter coefficients
float b0, b1, b2, b3, b4; //filter buffers (beware denormals!)
float t1, t2;          //temporary buffers

// Set coefficients given frequency & resonance [0.0...1.0]

q = 1.0f - frequency;
p = frequency + 0.8f * frequency * q;
f = p + p - 1.0f;
q = resonance * (1.0f + 0.5f * q * (1.0f - q + 5.6f * q * q));

// Filter (in [-1.0...+1.0])

in -= q * b4;          //feedback
t1 = b1;  b1 = (in + b0) * p - b1 * f;
t2 = b2;  b2 = (b1 + t1) * p - b2 * f;
t1 = b3;  b3 = (b2 + t2) * p - b3 * f;
         b4 = (b3 + t1) * p - b4 * f;
b4 = b4 - b4 * b4 * b4 * 0.166667f; //clipping
b0 = in;

// Lowpass output:  b4
// Highpass output: in - b4;
// Bandpass output: 3.0f * (b3 - b4);
```

Comments

from : daniel.martin24 [[a t]] gmx.net

comment : I just tried the filter code and it seems like the highpass output is the same as the lowpass output, or at least another lowpass...

But i'm still testing the filter code...

from : daniel.martin24 [[a t]] gmx.net

comment : Sorry for the Confusion, it works....

I just had a typo in my code.

One thing i did to get the HP sound nicer was

HP output: (in - 3.0f * (b3 - b4))-b4

But I'm a newbie to DSP Filters...

from : windowsucks2000 [[a t]] netscape.net

comment : Hey, thanks for this code. I'm a bit confused as to the range to the frequency and resonance. Is it really 0.0-1.0? If so, how so I specify a certain frequency, such as... 400Hz? THANKS!

from : Christian [[a t]] savioursofsoul.de

comment : frequency * nyquist

or

frequency * samplerate

don't know the exact implementation.

from : nobody [[a t]] nowhere.com

comment : >>Hey, thanks for this code. I'm a bit confused as to the range to the frequency and resonance. Is it really 0.0-1.0? If so, how so I specify a certain frequency, such as... 400Hz? THANKS!

I'd guess it would be:

frequency/(samplerate/2.f)

Moog VCF, variation 2 (click this to go back to the index)

Type : 24db resonant lowpass

References : CSound source code, Stilson/Smith CCRMA paper., Timo Tossavainen (?) version

Notes :

in[x] and out[x] are member variables, init to 0.0 the controls:

fc = cutoff, nearly linear [0,1] -> [0, fs/2]

res = resonance [0, 4] -> [no resonance, self-oscillation]

Code :

```
Tdouble MoogVCF::run(double input, double fc, double res)
{
    double f = fc * 1.16;
    double fb = res * (1.0 - 0.15 * f * f);
    input -= out4 * fb;
    input *= 0.35013 * (f*f)*(f*f);
    out1 = input + 0.3 * in1 + (1 - f) * out1; // Pole 1
    in1 = input;
    out2 = out1 + 0.3 * in2 + (1 - f) * out2; // Pole 2
    in2 = out1;
    out3 = out2 + 0.3 * in3 + (1 - f) * out3; // Pole 3
    in3 = out2;
    out4 = out3 + 0.3 * in4 + (1 - f) * out4; // Pole 4
    in4 = out3;
    return out4;
}
```

Comments

from : askywhale [[a t]] yahoo.fr

comment : This one works pretty well, thanks !

from : rdupres[AT]hotmail.com

comment : could somebody explain, what means this

```
input -= out4 * fb;
input *= 0.35013 * (f*f)*(f*f);
```

is "input-" and "input*" the name of a variable ??

or is this an Csound specific parameter ??

I want to translate this piece to Assemblercode

Robert Dupres

from : johanc [[a t]] popbandetleif.cjb.net

comment : input is name of a variable with type double.

```
input -= out4 * fb;
```

is just a shorter way for writing:

```
input = input - out4 * fb;
```

and the *= operator is works similar:

```
input *= 0.35013 * (f*f)*(f*f);
```

is equal to

```
input = input * 0.35013 * (f*f)*(f*f);
```

/ Johan

from : dfl [[a t]] ccrma.stanford.edu

comment : I've found this filter is unstable at low frequencies, namely when changing quickly from high to low frequencies...

from : williamk [[a t]] wusik.com

comment : I'm trying to double-sample this filter, like the Variable-State one. But so far no success, any tips?

Wk

from : mail [[a t]] mutagene.net

comment : What do you mean no success? What happens? Have you tried doing the usual oversampling tricks (sinc/hermite/mix-with-zeros-and-filter), call the moogVCF twice (with fc = fc*0.5) and then filter and decimate afterwards?

I'm been trying to find a good waveshaper to put in the feedback path but haven't found a good sounding stable one yet. I had one version of the filter that tracked the envelope of out4 and used it to control the degree to which values below some threshold (say 0.08) would get squashed towards zero.

That sounded ok (actually quite good for very high inputs), but wasn't entirely stable and was glitching for low frequencies. Then I tried a $out4 = (1+d) * (out4) / (1 + d * (out4))$ waveshaper, but that just aliased horribly and made the filter sound mushy and noisy.

Plain old polynomial ($x = x-x*x*x$) saturation sounds dull. There must be something better out there, though... and I'd much prefer not to have to oversample to get it, though I guess that might be unavoidable.

from : seezelf [[a t]] gmail.com

comment : Excuse me but just a basic question from a young developer

in line " input -= out4 * fb; "

i don't understand when and how "out4" is initialised

is it the "out4" return by the previous execution?

which initialisation for the first execution?

from : musicdsp [[a t]] [remove this]dparsons.co.uk

comment : all the outs should be initialised to zero, so first time around, nothing is subtracted. However, thereafter, the previous output is multiplied and subtracted from the input.

HTH

Notch filter (click this to go back to the index)

Type : 2 poles 2 zeros IIR

References : Posted by Olli Niemitalo

Notes :
Creates a muted spot in the spectrum with adjustable steepness. A complex conjugate pair of zeros on the z- plane unit circle and neutralizing poles approaching at the same angles from inside the unit circle.

```
Code :
Parameters:
0 =< freq =< samplerate/2
0 =< q < 1 (The higher, the narrower)

AlgoAlgo=double pi = 3.141592654;
double sqrt2 = sqrt(2.0);

double freq = 2050; // Change! (zero & pole angle)
double q = 0.4;     // Change! (pole magnitude)

double z1x = cos(2*pi*freq/samplerate);
double a0a2 = (1-q)*(1-q)/(2*(fabs(z1x)+1)) + q;
double a1 = -2*z1x*a0a2;
double b1 = -2*z1x*q;
double b2 = q*q;
double reg0, reg1, reg2;

unsigned int streamofs;
reg1 = 0;
reg2 = 0;

/* Main loop */
for (streamofs = 0; streamofs < streamsize; streamofs++)
{
    reg0 = a0a2 * ((double)fromstream[streamofs]
                  + fromstream[streamofs+2])
          + a1 * fromstream[streamofs+1]
          - b1 * reg1
          - b2 * reg2;

    reg2 = reg1;
    reg1 = reg0;

    int temp = reg0;

    /* Check clipping */
    if (temp > 32767) {
        temp = 32767;
    } else if (temp < -32768) temp = -32768;

    /* Store new value */
    tostream[streamofs] = temp;
}
}
```

Comments

from : rouncer81 [[a t]] hotmail.com
comment : i tried implementing it, failed,
and its wierd how it looks further in time
instead of backwards, you cant use it in
a running rendered stream cause the end of
it stuffs up....

from : musicdsp[at] Nospam dsparsons[dot]co[dot]uk
comment : I think it's a type, and should be

```
==-=
reg0 = a0a2 * ((double)fromstream[streamofs]
              + fromstream[streamofs-2])
      + a1 * fromstream[streamofs-1]
      - b1 * reg1
      - b2 * reg2;
==-=
```

In which case there is some rangechecking to be done when streamofs<2

You could just use the coeffs, and stick them into whatever biquad code you have hanging around :)

DSP

from : alfaulk165 [[a t]] aol.com

comment : Still doesn't work. Either way its looking one each side of a centre value, so it doesn't change the maths. Not sure what the code should be. Ideas???

One pole filter, LP and HP (click this to go back to the index)

Type : Simple 1 pole LP and HP filter

References : Posted by scoofy[AT]inf[DOT]elte[DOT]hu

Notes :

Slope: 6dB/Oct

Reference: www.dspguide.com

Code :

```
Process loop (lowpass):  
out = a0*in - b1*tmp;  
tmp = out;
```

Simple HP version: subtract lowpass output from the input (has strange behaviour towards nyquist):

```
out = a0*in - b1*tmp;  
tmp = out;  
hp = in-out;
```

Coefficient calculation:

```
x = exp(-2.0*pi*freq/samplerate);  
a0 = 1.0-x;  
b1 = -x;
```

Comments

from : nobody [[a t]] nowhere.com

comment : Of course, you don't need tmp.

Process loop (lowpass):

```
out = a0*in + b1*out;
```

from : dj_rejo [[a t]] hotmail.com

comment : Why don't you just say:

Process loop (lowpass):

```
out = a0*in + b1*tmp;  
tmp = out;
```

Simple HP version: subtract lowpass output from the input (has strange behaviour towards nyquist):

```
out = a0*in + b1*tmp;  
tmp = out;  
hp = in-out;
```

Coefficient calculation:

```
x = exp(-2.0*pi*freq/samplerate);  
a0 = 1.0-x;  
b1 = x;
```

from : scoofy [[a t]] inf.elte.hu

comment : There's a tradition among digital filter designers that the pole coefficients have a negative sign. Of course the other one is also valid, and sometimes these notations are mixed up.

If you're worried about the extra negation operation, then you could say

```
b1 = -x;  
a0 = 1.0+b1;
```

so that there's no additional operation overhead.

-- peter schoffhauzer

from : scoofy [[a t]] inf.elte.hu

comment : Indeed.

One pole LP and HP (click this to go back to the index)

References : Posted by Bram

Code :

LP:
recursion: $tmp = (1-p)*in + p*tmp$ with output = tmp
coefficient: $p = (2-\cos(x)) - \sqrt{(2-\cos(x))^2 - 1}$ with $x = 2*\pi*cutoff/samplerate$
coefficient approximation: $p = (1 - 2*cutoff/samplerate)^2$

HP:
recursion: $tmp = (p-1)*in - p*tmp$ with output = tmp
coefficient: $p = (2+\cos(x)) - \sqrt{(2+\cos(x))^2 - 1}$ with $x = 2*\pi*cutoff/samplerate$
coefficient approximation: $p = (2*cutoff/samplerate)^2$

Comments

from : skyphos_atw [[a t]] hotmail.com

comment : coefficient: $p = (2-\cos(x)) - \sqrt{(2-\cos(x))^2 - 1}$ with $x = 2*\pi*cutoff/samplerate$

p is always -1 using the formula above. The square eliminates the squareroot and $(2-\cos(x)) - (2-\cos(x))$ is 0.

from : q [[a t]] q

comment : Look again. The -1 is inside the sqrt.

from : batlord[A.T.]o2[D.O.T.]pl

comment : skyphos:
 $\sqrt{(2-\cos(x))^2 - 1}$ doesn't equal
 $\sqrt{(2-\cos(x))^2} + \sqrt{-1}$

so

-1 can be inside the sqrt, because $(2-\cos(x))^2$ will be always \geq one.

[One pole, one zero LP/HP](#) (click this to go back to the index)

[References](#) : Posted by mistert[AT]inwind[DOT]it

```
Code :
void SetLPF(float fCut, float fSampling)
{
    float w = 2.0 * fSampling;
    float Norm;

    fCut *= 2.0F * PI;
    Norm = 1.0 / (fCut + w);
    b1 = (w - fCut) * Norm;
    a0 = a1 = fCut * Norm;
}

void SetHPF(float fCut, float fSampling)
{
    float w = 2.0 * fSampling;
    float Norm;

    fCut *= 2.0F * PI;
    Norm = 1.0 / (fCut + w);
    a0 = w * Norm;
    a1 = -a0;
    b1 = (w - fCut) * Norm;
}
```

Where
out[n] = in[n]*a0 + in[n-1]*a1 + out[n-1]*b1;

Comments

[from](#) : petersteiner [[a t]] hypemart.net
[comment](#) : what is n? lol...sorry but i mean this seriously! ;)

[from](#) : musicdsp [[a t]] dsparsons.co.uk
[comment](#) : n is the index of sample being considered.

out[] is an array of samples being output, and in[] is the input array. you would construct a loop such that:

```
[Pseudocode]
loop n{0..numsamples-1}
    out[n] = in[n]*a0 + in[n-1]*a1 + out[n-1]*b1;
end loop;
[/Pseudocode]
```

You will need some cleverness so that [n-1] doesn't cause an index error when n=0, but I'll leave that to you :)

[from](#) : petersteiner [[a t]] hypemart.net
[comment](#) : whoops - sorry, of course n = number... stupid me ;)
interesting code, i will see if can adapt that to delphi, shouldn't be a big deal ;)

i assume i dont need to place either setHPF or LPF into the samples loop, just the block itself?

[from](#) : musicdsp [[a t]] dsparsons.co.uk
[comment](#) : absolutley - set the coefficients outside of the loop. There is the case of changes being made whilst the loop is running, depends what platform/host you are writing for.

I'm a delphi code as well. Feel free to use my posted address if you need to :) DSP

[from](#) : rouncer81 [[a t]] hotmail.com
[comment](#) : implemented first try worked.
excellent for what it is.

[from](#) : jenaleek[AT]yahoo[DOT]com
[comment](#) : Shouldn't that be float w = 2*PI*fSampling; ???

In which case we can simplify:

```
void SetLPF(float fCut, float fSampling)
{
    a0 = fCut/(fSampling+fCut);
    a1 = a0;
    b1 = (fSampling-fCut)/(fSampling+fCut);
}

void SetHPF(float fCut, float fSampling)
{
    a0 = fSampling/(fSampling+fCut);
    a1 = -a0;
    b1 = (fSampling-fCut)/(fSampling+fCut);
}
```

You can keep the norm = 1/(fSampling+fCut) if you like.

[One zero, LP/HP](#) (click this to go back to the index)

[References](#) : Posted by Bram

[Notes](#) :

LP is only 'valid' for cutoffs > samplerate/4

HP is only 'valid' for cutoffs < samplerate/4

[Code](#) :

```
theta = cutoff*2*pi / samplerate
```

LP:

```
H(z) = (1+p*z^(-1)) / (1+p)
```

```
out[i] = 1/(1+p) * in[i] + p/(1+p) * in[i-1];
```

```
p = (1-2*cos(theta)) - sqrt((1-2*cos(theta))^2 - 1)
```

```
Pi/2 < theta < Pi
```

HP:

```
H(z) = (1-p*z^(-1)) / (1+p)
```

```
out[i] = 1/(1+p) * in[i] - p/(1+p) * in[i-1];
```

```
p = (1+2*cos(theta)) - sqrt((1+2*cos(theta))^2 - 1)
```

```
0 < theta < Pi/2
```

[Comments](#)

[from](#) : newbie attack

[comment](#) : What is the implementation of z^{-1} ?

[from](#) : spam [[a t]] hell.no

[comment](#) : $z^{-1} = 1/z$

[from](#) : cam

[comment](#) : z^{-1} is a single sample delay. The second line of code is the actual filter code. First line is a transfer function.

[from](#) : Gorgr

[comment](#) : wt's the meaning of 'p'?

One-Liner IIR Filters (1st order) (click this to go back to the index)

Type : IIR 1-pole

References : Posted by chris at ariescode dot com

Notes :

Here is a collection of one liner IIR filters.

Each filter has been transformed into a single C++ expression.

The filter parameter is f or g, and the state variable that needs to be kept around between iterations is s.

- Christian

Code :

101 Leaky Integrator

```
a0 = 1
b1 = 1 - f

out = s += in - f * s;
```

102 Basic Lowpass (all-pole)

A first order lowpass filter, by finite difference approximation (differentials --> differences).

```
a0 = f
b1 = 1 - f

out = s += f * ( in - s );
```

103 Lowpass with inverted control

Same as above, except for different filter parameter is now inverted.
In this case, g equals the location of the pole.

```
a0 = g - 1
b1 = g

out = s = in + g * ( s - in );
```

104 Lowpass with zero at Nyquist

A first order lowpass filter, by via the conformal map of the z-plane (0..infinity --> 0..Nyquist).

```
a0 = f
a1 = f
b1 = 1 - 2 * f

s = temp + ( out = s + ( temp = f * ( in - s ) ) );
```

105 Basic Highpass (DC-blocker)

Input complement to basic lowpass, yields a finite difference highpass filter.

```
a0 = 1 - f
a1 = f - 1
b1 = 1 - f

out = in - ( s += f * ( in - s ) );
```

106 Highpass with forced unity gain at Nyquist

Input complement to filter 104, yields a conformal map highpass filter.

```
a0 = 1 - f
a1 = f - 1
b1 = 1 - 2 * f

out = in + temp - ( s += 2 * ( temp = f * ( in - s ) ) );
```

107 Basic Allpass

This corresponds to a first order allpass filter,
where g is the location of the pole in the range -1..1.

```
a0 = -g
a1 = 1
b1 = g

s = in + g * ( out = s - g * in );
```


[Peak/Notch filter](#) (click this to go back to the index)

Type : peak/notch

References : Posted by tobybear[AT]web[DOT]de

Notes :

```
// Peak/Notch filter
// I don't know anymore where this came from, just found it on
// my hard drive :-)
// Seems to be a peak/notch filter with adjustable slope
// steepness, though slope gets rather wide the lower the
// frequency is.
// "cut" and "steep" range is from 0..1
// Try to feed it with white noise, then the peak output does
// rather well eliminate all other frequencies except the given
// frequency in higher frequency ranges.
```

Code :

```
var f,r:single;
    outp,outpl,outp2:single; // init these with 0!
const p4=1.0e-24; // Pentium 4 denormal problem elimination

function PeakNotch(inp,cut,steep:single;ftype:integer):single;
begin
    r:=steep*0.99609375;
    f:=cos(pi*cut);
    a0:=(1-r)*sqrt(r*(r-4*(f*f)+2)+1);
    b1:=2*f*r;
    b2:=-r*r;
    outp:=a0*inp+b1*outpl+b2*outp2+p4;
    outp2:=outpl;
    outpl:=outp;
    if ftype=0 then
        result:=outp //peak
    else
        result:=inp-outp; //notch
    end;
```

Perfect LP4 filter (click this to go back to the index)

Type : LP

References : Posted by azertopia at free dot fr

Notes :

hacked from the exemple of user script in FL Edison

```
Code :
TLP24DB = class
constructor create;
procedure process(inp,Frq,Res:single;SR:integer);
private
t, t2, x, f, k, p, r, y1, y2, y3, y4, oldx, oldy1, oldy2, oldy3: Single;
public outlp:single;
end;
-----
implementation

constructor TLP24DB.create;
begin
  y1:=0;
  y2:=0;
  y3:=0;
  y4:=0;
  oldx:=0;
  oldy1:=0;
  oldy2:=0;
  oldy3:=0;
end;
procedure TLP24DB.process(inp: Single; Frq: Single; Res: Single; SR: Integer);
begin
  f := (Frq+Frq) / SR;
  p:=f*(1.8-0.8*f);
  k:=p+p-1.0;
  t:=(1.0-p)*1.386249;
  t2:=12.0+t*t;
  r := res*(t2+6.0*t)/(t2-6.0*t);
  x := inp - r*y4;
  y1:=x*p + oldx*p - k*y1;
  y2:=y1*p+oldy1*p - k*y2;
  y3:=y2*p+oldy2*p - k*y3;
  y4:=y3*p+oldy3*p - k*y4;
  y4 := y4 - ((y4*y4*y4)/6.0);
  oldx := x;
  oldy1 := y1+_kd;
  oldy2 := y2+_kd;;
  oldy3 := y3+_kd;;
  outlp := y4;
end;

// the result is in outlp
// 1/ call MyTLP24DB.Process
// 2/then get the result from outlp.
// this filter have a fantastic sound w/a very special res
// _kd is the denormal killer value.
```

Comments

from : neolit123 [[a t]] gmail.com

comment : This is basically a Moog filter.

[Phase equalization](#) (click this to go back to the index)

Type : Allpass

References : Posted by Uli the Grasso

Notes :

The idea is simple: One can equalize the phase response of a system, for example of a loudspeaker, by approximating its phase response by an FIR filter and then turn around the coefficients of the filter. At <http://grassomusic.de/english/phaseeq.htm> you find more info and an Octave script.

Comments

from : piem

comment : hi.

i couldn't find any script there...

cheers, piem

from : grasso789HUHUUHU [[a t]] versanet.de

comment : Now it is there. Exactly, the script is <http://grassomusic.de/download/octave/octaverc>

from : bob [[a t]] yahoot.com

comment : No link exists, or for updated message.

[Pink noise filter](#) (click this to go back to the index)

[References](#) : Posted by Paul Kellett

[Linked file](#) : [pink.txt](#) (this linked file is included below)

[Notes](#) :
(see linked file)

[Comments](#)

[from](#) : Tomy [[a t]] tomy-pa.de

[comment](#) : Hi, first of all thanks a lot for this parameters.

I'm new to digital filtering, and need a 3dB highpass to correct a pink spectrum which is used for measurement back to white for displaying the impulseresponse.

I checked some pages, but all demand a fixed ratio between d0 and d1 for a 6db lowpass. But this ratio is not given on your filters, so I'm not able to transform them into highpasses.

Any hints?

Tomy

[from](#) : Tomy [[a t]] tomy-pa.de

[comment](#) : Hi, first of all thanks a lot for this parameters. I'm new to digital filtering, and need a 3dB highpass to correct a pink spectrum which is used for measurement back to white for displaying the impulseresponse. I checked some pages, but all demand a fixed ratio between d0 and d1 for a 6db lowpass. But this ratio is not given on your filters, so I'm not able to transform them into highpasses. Any hints? Tomy

[from](#) : Christian [[a t]] savioursofsoul.de

[comment](#) : If computing power doesn't matter, than you may want do design the pink noise in the frequency domain and transform it back to timedomain via fft.

Christian

[from](#) : haddadmajed [[a t]] yahoo.fr

[comment](#) : Hi, could you please give me a code matlab to have a pink noise. I tested a code where one did all into frequential mode then made an ifft.

Thank you

[Linked files](#)

[Filter to make pink noise from white](#) (updated March 2000)

This is an approximation to a -10dB/decade filter using a weighted sum of first order filters. It is accurate to within +/-0.05dB above 9.2Hz (44100Hz sampling rate). Unity gain is at Nyquist, but can be adjusted by scaling the numbers at the end of each line.

If 'white' consists of uniform random numbers, such as those generated by the rand() function, 'pink' will have an almost gaussian level distribution.

```
b0 = 0.99886 * b0 + white * 0.0555179;
b1 = 0.99332 * b1 + white * 0.0750759;
b2 = 0.96900 * b2 + white * 0.1538520;
b3 = 0.86650 * b3 + white * 0.3104856;
b4 = 0.55000 * b4 + white * 0.5329522;
b5 = -0.7616 * b5 - white * 0.0168980;
pink = b0 + b1 + b2 + b3 + b4 + b5 + b6 + white * 0.5362;
b6 = white * 0.115926;
```

An 'economy' version with accuracy of +/-0.5dB is also available.

```
b0 = 0.99765 * b0 + white * 0.0990460;
b1 = 0.96300 * b1 + white * 0.2965164;
b2 = 0.57000 * b2 + white * 1.0526913;
pink = b0 + b1 + b2 + white * 0.1848;
```

paul.kellett@maxim.abel.co.uk
<http://www.abel.co.uk/~maxim/>

Plot Filter (Analyze filter characteristics) (click this to go back to the index)

Type : Test

References : Posted by scanner598 at yahoo dot co dot uk

Notes :
As a newbie, and one that has very, very little mathematical background, I wanted to see what all the filters posted here were doing to get a feeling of what was going on here. So with what I picked up from this site, I wrote a little filter test program. Hope it is of any use to you.

```
Code :
//
// plotFilter.cpp
//
// Simple test program to plot filter characteristics of a particular
// filter to stdout. Nice to see how the filter behaves under various
// conditions (cutoff/resonance/samplerate/etc.).
//
// Should work on any platform that supports C++ and should work on C
// as well with a little rework. It just prints text, so no graphical
// stuff is used.
//
// Filter input and filter output should be between -1 and 1 (floating point)
//
// Output is a plotted graph (as text) with a logarithmic scale
// (so half a plotted bar is half of what the human ear can hear).
// If you dont like the vertical output, just print it and turn the paper :-
//

#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>

#define myPI 3.1415926535897932384626433832795

#define FP double
#define DWORD unsigned long
#define CUTOFF 5000
#define SAMPLERATE 48000

// take enough samples to test the 20 herz frequency 2 times
#define TESTSAMPLES (SAMPLERATE/20) * 2

//
// Any filter can be tested, as long as it outputs
// between -1 and 1 (floating point). This filter
// can be replaced with any filter you would like
// to test.
//
class Filter {
    FP sdml; // sample data minus 1
    FP a0; // multiply factor on current sample
    FP b1; // multiply factor on sdml
public:
    Filter (void) {
        sdml = 0;
        // init on no filtering
        a0 = 1;
        b1 = 0;
    }
    void init(FP freq, FP samplerate) {
        FP x;
        x = exp(-2.0 * myPI * freq / samplerate);
        sdml = 0;
        a0 = 1.0 - x;
        b1 = -x;
    }
    FP getSample(FP sample) {
        FP out;
        out = (a0 * sample) - (b1 * sdml);
        sdml = out;
        return out;
    }
};

int
main(void)
{
    DWORD freq;
    DWORD spos;
    double sIn;
    double sOut;
    double tIn;
    double tOut;
    double dB;
    DWORD tmp;

    // define the test filter
    Filter filter;
```

```

printf("          9dB      6dB      3dB      0dB\n");
printf(" freq      dB      |      |      |      | \n");

// test frequencies 20 - 20020 with 100 herz steps
for (freq=20; freq<20020; freq+=100) {

    // (re)initialize the filter
    filter.init(CUTOFF, SAMPLERATE);

    // let the filter do it's thing here
    tIn = tOut = 0;
    for (spos=0; spos<TESTSAMPLES; spos++) {
        sIn = sin((2 * myPI * spos * freq) / SAMPLERATE);
        sOut = filter.getSample(sIn);
        if ((sOut>1) || (sOut<-1)) {
            // If filter is no good, stop the test
            printf("Error! Clipping!\n");
            return(1);
        }
        if (sIn >0) tIn += sIn;
        if (sIn <0) tIn -= sIn;
        if (sOut>0) tOut += sOut;
        if (sOut<0) tOut -= sOut;
    }

    // analyse the results
    dB = 20*log(tIn/tOut);
    printf("%5d %5.1f ", freq, dB);
    tmp = (DWORD)(60.0/pow(2, (dB/3)));
    while (tmp) {
        putchar('#');
        tmp--;
    }
    putchar('\n');
}
return 0;
}

```

Comments

from : niels m.

comment : You need to change the log() to log10() to get the correct answer in dB's.

You can also replace the if(sIn >0) .. -= sOut; by:

```

tIn += sIn*sIn;
tOut += sOut*sOut;

```

this will measure signal power instead of amplitude. If you do this, you also need to change 20*log10() to 10*log10().

Nice and useful tool for exploring filters. Thanks!

Plotting R B-J Equalisers in Excel (click this to go back to the index)

References : Posted by Web Surf

Linked file : [rbj_eq.xls](#)

Notes :

Interactive XL sheet that shows frequency response of the R B-K high pass/low pass, Peaking and Shelf filters

Useful if --

--You want to validate your implementation against mine

--You want to convert given Biquad coefficients into Fo/Q/dBgain by visual curve fitting.

--You want the Coefficients to implement a particular fixed filter

-- Educational aid

Many thanks to R B-J for his personal support !!

Web Surf WebsurffATgmailDOTcom

Code :

see attached file

Comments

from : Paul_Sernine75 [[a t]] hotmail.fr

comment : It also works perfectly with the openoffice2.02 suite :-)

from : jackmattson att gmial

comment : Ok, so I'm about to make my first filter so I really have no idea about coefficients yet but damn this is the coolest .xls I've ever seen! And I see a bunch every day. :?

from : WebsurffATgmailDOTcom

comment : Thanks,

It's just an implementation of the R B-J cookbook formulae for parametric equalisers. RB-J personally assisted me while I was writing this XLS. The real Kudos goes out to him !!

I wrote this as an intermediate tool while I was writing some Guitar effect DSP routines.

One prob it has : If the Q is too sharp, the peak filters have a very sharp tip. It is possible then as you slide the F sliders, that the chosen F is not one of the data points that I am plotting.

The net result is that the peak of a hi-Q filter seems to increase/decrease as you move F. This is a shortcoming of the way I programmed my XLS and is not a problem with the R B-J equations.

PS : I saw the same problem with other similar S/W on the net !!!

PS : Anyone know how to do curve fitting so that we enter in some points through which the frequency response must pass, and it generates best set of F,G,Q ?

Polyphase Filters (click this to go back to the index)

Type : polyphase filters, used for up and down-sampling

References : C++ source code by Dave from Muon Software

Linked file : [BandLimit.cpp](#) (this linked file is included below)

Linked file : [BandLimit.h](#) (this linked file is included below)

Comments

from : saionara_tagushi [[a t]] web.de

comment : can someone give me a hint for a paper where this stuff is from?

from : ABC

comment : From there: <http://www.cmsa.wmin.ac.uk/~artur/Poly.html>

There is also this library, implementing the same filter, but optimised for down/upsampling and ported to SSE and 3DNow!
http://ldesoras.free.fr/prod.html#src_hrir

from : rob.belcham [[a t]] zen.co.uk

comment : There is an error in the 12th order, steep filter coefficients. Having checked the output against that produced by HIR (see previous comment), i have identified the source of the error - the 4th b coefficient 0.06329609551399348, should be 0.6329609551399348.

from : 1nf0 [[a t]] aud1osp1llag3.bot.com

comment : some questions.. is it normal for these halfband filters to cause significant gain loss? sonogram analysis shows a decrease in SNR if I have read the results correctly.

if using these filters for oversampling and I do this:

```
upsample
halfband filter
*process*
halfband filter
decimate (discard samples)
```

then presumably the second halfband filter does the antialiasing at half the new sampling rate?

from : bla

comment : you also need to delete the pointers inside the array

```
CAIIPassFilterCascade::~CAIIPassFilterCascade()
{
for (int i=0;i<numfilters;i++)
{
delete allpassfilter[i];
}

delete[] allpassfilter;
};
```

Linked files

```
CAIIPassFilter::CAIIPassFilter(const double coefficient)
{
a=coefficient;

x0=0.0;
x1=0.0;
x2=0.0;

y0=0.0;
y1=0.0;
y2=0.0;
};
```

```
CAIIPassFilter::~CAIIPassFilter()
{
};
```

```
double CAIIPassFilter::process(const double input)
{
//shuffle inputs
x2=x1;
x1=x0;
```

```

x0=input;

//shuffle outputs
y2=y1;
y1=y0;

//allpass filter 1
const double output=x2+((input-y2)*a);

y0=output;

return output;
};

CallPassFilterCascade::CallPassFilterCascade(const double* coefficient, const int N)
{
    allpassfilter=new CallPassFilter*[N];

    for (int i=0;i<N;i++)
    {
        allpassfilter[i]=new CallPassFilter(coefficient[i]);
    }

    numfilters=N;
};

CallPassFilterCascade::~CallPassFilterCascade()
{
    delete[] allpassfilter;
};

double CallPassFilterCascade::process(const double input)
{
    double output=input;

    int i=0;

    do
    {
        output=allpassfilter[i]->process(output);
        i++;
    } while (i<numfilters);

    return output;
};

CHalfBandFilter::CHalfBandFilter(const int order, const bool steep)
{
    if (steep==true)
    {
        if (order==12) //rejection=104dB, transition band=0.01
        {
            double a_coefficients[6]=
            {0.036681502163648017
            ,0.2746317593794541
            ,0.56109896978791948
            ,0.769741833862266
            ,0.8922608180038789
            ,0.962094548378084
            };

            double b_coefficients[6]=
            {0.13654762463195771
            ,0.42313861743656667
            ,0.6775400499741616
            ,0.839889624849638
            ,0.9315419599631839
            ,0.9878163707328971
            };

            filter_a=new CallPassFilterCascade(a_coefficients,6);
            filter_b=new CallPassFilterCascade(b_coefficients,6);
        }
        else if (order==10) //rejection=86dB, transition band=0.01
        {
            double a_coefficients[5]=
            {0.051457617441190984
            ,0.35978656070567017

```

```

,0.6725475931034693
,0.8590884928249939
,0.9540209867860787
};

double b_coefficients[5]=
{0.18621906251989334
,0.529951372847964
,0.7810257527489514
,0.9141815687605308
,0.985475023014907
};

filter_a=new CallPassFilterCascade(a_coefficients,5);
filter_b=new CallPassFilterCascade(b_coefficients,5);
}
else if (order==8) //rejection=69dB, transition band=0.01
{
double a_coefficients[4]=
{0.07711507983241622
,0.4820706250610472
,0.7968204713315797
,0.9412514277740471
};

double b_coefficients[4]=
{0.2659685265210946
,0.6651041532634957
,0.8841015085506159
,0.9820054141886075
};

filter_a=new CallPassFilterCascade(a_coefficients,4);
filter_b=new CallPassFilterCascade(b_coefficients,4);
}
else if (order==6) //rejection=51dB, transition band=0.01
{
double a_coefficients[3]=
{0.1271414136264853
,0.6528245886369117
,0.9176942834328115
};

double b_coefficients[3]=
{0.40056789819445626
,0.8204163891923343
,0.9763114515836773
};

filter_a=new CallPassFilterCascade(a_coefficients,3);
filter_b=new CallPassFilterCascade(b_coefficients,3);
}
else if (order==4) //rejection=53dB,transition band=0.05
{
double a_coefficients[2]=
{0.12073211751675449
,0.6632020224193995
};

double b_coefficients[2]=
{0.3903621872345006
,0.890786832653497
};

filter_a=new CallPassFilterCascade(a_coefficients,2);
filter_b=new CallPassFilterCascade(b_coefficients,2);
}
else //order=2, rejection=36dB, transition band=0.1
{
double a_coefficients=0.23647102099689224;
double b_coefficients=0.7145421497126001;

filter_a=new CallPassFilterCascade(&a_coefficients,1);
filter_b=new CallPassFilterCascade(&b_coefficients,1);
}
}
else //softer slopes, more attenuation and less stopband ripple
{
if (order==12) //rejection=150dB, transition band=0.05
{
double a_coefficients[6]=
{0.01677466677723562

```

```

,0.13902148819717805
,0.33250111117394731
,0.53766105314488
,0.7214184024215805
,0.8821858402078155
};

double b_coefficients[6]=
{0.06501319274445962
,0.23094129990840923
,0.4364942348420355
,0.06329609551399348
,0.80378086794111226
,0.9599687404800694
};

filter_a=new CallPassFilterCascade(a_coefficients,6);
filter_b=new CallPassFilterCascade(b_coefficients,6);
}
else if (order==10) //rejection=133dB, transition band=0.05
{
double a_coefficients[5]=
{0.02366831419883467
,0.18989476227180174
,0.43157318062118555
,0.6632020224193995
,0.860015542499582
};

double b_coefficients[5]=
{0.09056555904993387
,0.3078575723749043
,0.5516782402507934
,0.7652146863779808
,0.95247728378667541
};

filter_a=new CallPassFilterCascade(a_coefficients,5);
filter_b=new CallPassFilterCascade(b_coefficients,5);
}
else if (order==8) //rejection=106dB, transition band=0.05
{
double a_coefficients[4]=
{0.03583278843106211
,0.2720401433964576
,0.5720571972357003
,0.827124761997324
};

double b_coefficients[4]=
{0.1340901419430669
,0.4243248712718685
,0.7062921421386394
,0.9415030941737551
};

filter_a=new CallPassFilterCascade(a_coefficients,4);
filter_b=new CallPassFilterCascade(b_coefficients,4);
}
else if (order==6) //rejection=80dB, transition band=0.05
{
double a_coefficients[3]=
{0.06029739095712437
,0.4125907203610563
,0.7727156537429234
};

double b_coefficients[3]=
{0.21597144456092948
,0.6043586264658363
,0.9238861386532906
};

filter_a=new CallPassFilterCascade(a_coefficients,3);
filter_b=new CallPassFilterCascade(b_coefficients,3);
}
else if (order==4) //rejection=70dB,transition band=0.1
{
double a_coefficients[2]=
{0.07986642623635751
,0.5453536510711322
};

```

```

double b_coefficients[2]=
{0.28382934487410993
,0.8344118914807379
};

filter_a=new CAllPassFilterCascade(a_coefficients,2);
filter_b=new CAllPassFilterCascade(b_coefficients,2);
}

else //order=2, rejection=36dB, transition band=0.1
{
double a_coefficients=0.23647102099689224;
double b_coefficients=0.7145421497126001;

filter_a=new CAllPassFilterCascade(&a_coefficients,1);
filter_b=new CAllPassFilterCascade(&b_coefficients,1);
}
}

oldout=0.0;
};

CHalfBandFilter::~CHalfBandFilter()
{
delete filter_a;
delete filter_b;
};

double CHalfBandFilter::process(const double input)
{
const double output=(filter_a->process(input)+oldout)*0.5;
oldout=filter_b->process(input);

return output;
}
class CAllPassFilter
{
public:

CAllPassFilter(const double coefficient);
~CAllPassFilter();
double process(double input);

private:
double a;

double x0;
double x1;
double x2;

double y0;
double y1;
double y2;
};

class CAllPassFilterCascade
{
public:
CAllPassFilterCascade(const double* coefficients, int N);
~CAllPassFilterCascade();

double process(double input);

private:
CAllPassFilter** allpassfilter;
int numfilters;
};

class CHalfBandFilter
{
public:
CHalfBandFilter(const int order, const bool steep);
~CHalfBandFilter();

double process(const double input);

private:
CAllPassFilterCascade* filter_a;
CAllPassFilterCascade* filter_b;
};

```



```
double oldout;  
};
```

Polyphase Filters (Delphi) (click this to go back to the index)

Type : polyphase filters, used for up and down-sampling

References : Posted by veryangrymobster[AT]hotmail[DOT]com

Notes :

Pascal conversion of C++ code by Dave from Muon Software. Conversion by Shannon Faulkner.

Code :

```
{
polyphase filters, used for up and down-sampling
original c++ code by Dave from Muon Software found
at MusicDSP.
rewritten in pascal by Shannon Faulkner, 4/7/06.
}

unit uPolyphase;

interface

type
  TAllPass=class
  private
    a,x0,x1,x2,y0,y1,y2:single;
  public
    constructor create(coefficient:single);
    function Process(input:single):single;
  end;

  TAllPassFilterCascade=class
  private
    AllPassFilters:array of TAllPass;
    fOrder:integer;
  public
    constructor create(coefficients:psingle;Order:integer);
    function Process(input:single):single;
  end;

  THalfBandFilter=class
  private
    fOrder:integer;
    OldOut:single;
    aCoeffs,bCoeffs:array of single;
    FilterA,FilterB:TAllPassFilterCascade;
  public
    constructor create(order:integer;Steep:boolean);
    function process(input:single):single;
  end;

implementation
//----- AllPass Filter -----
//----- AllPass Filter -----
//----- AllPass Filter -----
constructor TAllPass.create(coefficient:single);
begin
  a:=coefficient;

  x0:=0;
  x1:=0;
  x2:=0;

  y0:=0;
  y1:=0;
  y2:=0;
end;

function TAllPass.Process(input:single):single;
var output:single;
begin
  //shuffle inputs
  x2:=x1;
  x1:=x0;
  x0:=input;

  //shuffle outputs
  y2:=y1;
  y1:=y0;

  //allpass filter 1
  output:=x2+((input-y2)*a);

  y0:=output;

  result:=output;
end;
//----- AllPass Filter Cascade -----
```

```

//----- AllPass Filter Cascade -----
//----- AllPass Filter Cascade -----
constructor TAllPassFilterCascade.create(coefficients:psingle;Order:integer);
var i:integer;
begin
    fOrder:=Order;
    setlength(AllPassFilters,fOrder);
    for i:=0 to fOrder-1 do
    begin
        AllPassFilters[i]:=TAllPass.create(coefficients^);
        inc(coefficients);
    end;
end;

function TAllPassFilterCascade.Process(input:single):single;
var
    output:single;
    i:integer;
begin
    output:=input;
    for i:=0 to fOrder-1 do
    begin
        output:=allpassfilters[i].Process(output);
    end;
    result:=output;
end;
//----- Halfband Filter -----
//----- Halfband Filter -----
//----- Halfband Filter -----
constructor THalfBandFilter.create(order:integer;Steep:boolean);
begin
    fOrder:=order;
    setlength(aCoeffs,Order div 2);
    setlength(bCoeffs,Order div 2);

    if steep=true then
    begin
        if (order=12) then //rejection=104dB, transition band=0.01
        begin
            aCoeffs[0]:=0.036681502163648017;
            aCoeffs[1]:=0.2746317593794541;
            aCoeffs[2]:=0.56109896978791948;
            aCoeffs[3]:=0.769741833862266;
            aCoeffs[4]:=0.8922608180038789;
            aCoeffs[5]:=0.962094548378084;

            bCoeffs[0]:=0.13654762463195771;
            bCoeffs[1]:=0.42313861743656667;
            bCoeffs[2]:=0.6775400499741616;
            bCoeffs[3]:=0.839889624849638;
            bCoeffs[4]:=0.9315419599631839;
            bCoeffs[5]:=0.9878163707328971;
        end
        else if (order=10) then //rejection=86dB, transition band=0.01
        begin
            aCoeffs[0]:=0.051457617441190984;
            aCoeffs[1]:=0.35978656070567017;
            aCoeffs[2]:=0.6725475931034693;
            aCoeffs[3]:=0.8590884928249939;
            aCoeffs[4]:=0.9540209867860787;

            bCoeffs[0]:=0.18621906251989334;
            bCoeffs[1]:=0.529951372847964;
            bCoeffs[2]:=0.7810257527489514;
            bCoeffs[3]:=0.9141815687605308;
            bCoeffs[4]:=0.985475023014907;
        end
        else if (order=8) then //rejection=69dB, transition band=0.01
        begin
            aCoeffs[0]:=0.07711507983241622;
            aCoeffs[1]:=0.4820706250610472;
            aCoeffs[2]:=0.7968204713315797;
            aCoeffs[3]:=0.9412514277740471;

            bCoeffs[0]:=0.2659685265210946;
            bCoeffs[1]:=0.6651041532634957;
            bCoeffs[2]:=0.8841015085506159;
            bCoeffs[3]:=0.9820054141886075;
        end
        else if (order=6) then //rejection=51dB, transition band=0.01
        begin
            aCoeffs[0]:=0.1271414136264853;
            aCoeffs[1]:=0.6528245886369117;
            aCoeffs[2]:=0.9176942834328115;

            bCoeffs[0]:=0.40056789819445626;
            bCoeffs[1]:=0.8204163891923343;
            bCoeffs[2]:=0.9763114515836773;
        end
        else if (order=4) then //rejection=53dB,transition band=0.05
        begin
            aCoeffs[0]:=0.12073211751675449;
            aCoeffs[1]:=0.6632020224193995;

```

```

    bCoeffs[0]:=0.3903621872345006;
    bCoeffs[1]:=0.890786832653497;
end
else //order=2, rejection=36dB, transition band=0.1
begin
    aCoeffs[0]:=0.23647102099689224;
    bCoeffs[0]:=0.7145421497126001;
end;
end else //softer slopes, more attenuation and less stopband ripple
begin
    if (order=12) then //rejection=104dB, transition band=0.01
    begin
        aCoeffs[0]:=0.01677466677723562;
        aCoeffs[1]:=0.13902148819717805;
        aCoeffs[2]:=0.3325011117394731;
        aCoeffs[3]:=0.53766105314488;
        aCoeffs[4]:=0.7214184024215805;
        aCoeffs[5]:=0.8821858402078155;

        bCoeffs[0]:=0.06501319274445962;
        bCoeffs[1]:=0.23094129990840923;
        bCoeffs[2]:=0.4364942348420355;

        //bug fix - coefficient changed,
        //rob[DOT]belcham[AT]zen[DOT]co[DOT]uk
        //bCoeffs[3]:=0.06329609551399348; //original coefficient
        bCoeffs[3]:=0.6329609551399348; //correct coefficient

        bCoeffs[4]:=0.80378086794111226;
        bCoeffs[5]:=0.9599687404800694;
    end
    else if (order=10) then //rejection=86dB, transition band=0.01
    begin
        aCoeffs[0]:=0.02366831419883467;
        aCoeffs[1]:=0.18989476227180174;
        aCoeffs[2]:=0.43157318062118555;
        aCoeffs[3]:=0.6632020224193995;
        aCoeffs[4]:=0.860015542499582;

        bCoeffs[0]:=0.09056555904993387;
        bCoeffs[1]:=0.3078575723749043;
        bCoeffs[2]:=0.5516782402507934;
        bCoeffs[3]:=0.7652146863779808;
        bCoeffs[4]:=0.95247728378667541;
    end
    else if (order=8) then //rejection=69dB, transition band=0.01
    begin
        aCoeffs[0]:=0.03583278843106211;
        aCoeffs[1]:=0.2720401433964576;
        aCoeffs[2]:=0.5720571972357003;
        aCoeffs[3]:=0.827124761997324;

        bCoeffs[0]:=0.1340901419430669;
        bCoeffs[1]:=0.4243248712718685;
        bCoeffs[2]:=0.7062921421386394;
        bCoeffs[3]:=0.9415030941737551;
    end
    else if (order=6) then //rejection=51dB, transition band=0.01
    begin
        aCoeffs[0]:=0.06029739095712437;
        aCoeffs[1]:=0.4125907203610563;
        aCoeffs[2]:=0.7727156537429234;

        bCoeffs[0]:=0.21597144456092948;
        bCoeffs[1]:=0.6043586264658363;
        bCoeffs[2]:=0.9238861386532906;
    end
    else if (order=4) then //rejection=53dB, transition band=0.05
    begin
        aCoeffs[0]:=0.07986642623635751;
        aCoeffs[1]:=0.5453536510711322;

        bCoeffs[0]:=0.28382934487410993;
        bCoeffs[1]:=0.8344118914807379;
    end
    else //order=2, rejection=36dB, transition band=0.1
    begin
        aCoeffs[0]:=0.23647102099689224;
        bCoeffs[0]:=0.7145421497126001;
    end;
end;

FilterA:=TAllPassFilterCascade.create(@aCoeffs[0],fOrder div 2);
FilterB:=TAllPassFilterCascade.create(@bCoeffs[0],fOrder div 2);

oldout:=0;
end;

function THalfBandFilter.process(input:single):single;
begin
    result:=(FilterA.Process(input)+oldout)*0.5;
    oldout:=FilterB.Process(input);
end;

```

end;

end.

Poor Man's FIWIZ (click this to go back to the index)

Type : Filter Design Utility

References : Posted by mailbj[at]yahoo[dot]com

Notes :
FIWIZ is a neat little filter design utility that uses a genetic programming technique called Differential Evolution. As useful as it is, it looks like it took about a week to write, and is thus very undeserving of the \$70 license fee. So I decided to write my own. There's a freely available optimizer class that uses Differential Evolution and I patched it together with some filter-specific logic.

Use of the utility requires the ability to do simple coding in C, but you need only revise a single function, which basically describes your filter specification. There's a big comment in the main source file that clarifies things a bit more.

Although it's not as easy to use as FIWIZ, it's arguably more powerful because your specifications are limited only by what you can express in C, whereas FIWIZ is completely GUI based.

Another thing: I'm afraid that due to the use of `_kbhit` and `_getch`, the code is a bit Microsofty, but you can take those out and the code will still be basically usable.

```
Code :
// First File: DESolver.cpp

#include <stdio.h>
#include <memory.h>
#include <conio.h>
#include "DESolver.h"

#define Element(a,b,c) a[b*nDim+c]
#define RowVector(a,b) (&a[b*nDim])
#define CopyVector(a,b) memcpy((a),(b),nDim*sizeof(double))

DESolver::DESolver(int dim,int popSize) :
    nDim(dim), nPop(popSize),
    generations(0), strategy(stRandlExp),
    scale(0.7), probability(0.5), bestEnergy(0.0),
    trialSolution(0), bestSolution(0),
    popEnergy(0), population(0)
{
    trialSolution = new double[nDim];
    bestSolution = new double[nDim];
    popEnergy = new double[nPop];
    population = new double[nPop * nDim];

    return;
}

DESolver::~DESolver(void)
{
    if (trialSolution) delete trialSolution;
    if (bestSolution) delete bestSolution;
    if (popEnergy) delete popEnergy;
    if (population) delete population;

    trialSolution = bestSolution = popEnergy = population = 0;
    return;
}

void DESolver::Setup(double *min,double *max,
    int deStrategy,double diffScale,double crossoverProb)
{
    int i;

    strategy = deStrategy;
    scale = diffScale;
    probability = crossoverProb;

    for (i=0; i < nPop; i++)
    {
        for (int j=0; j < nDim; j++)
            Element(population,i,j) = RandomUniform(min[j],max[j]);

        popEnergy[i] = 1.0E20;
    }

    for (i=0; i < nDim; i++)
        bestSolution[i] = 0.0;

    switch (strategy)
    {
    case stBestlExp:
        calcTrialSolution = &DESolver::BestlExp;
        break;

    case stRandlExp:
        calcTrialSolution = &DESolver::RandlExp;
        break;

    case stRandToBestlExp:
        calcTrialSolution = &DESolver::RandToBestlExp;
```

```

    break;

case stBest2Exp:
    calcTrialSolution = &DESolver::Best2Exp;
    break;

case stRand2Exp:
    calcTrialSolution = &DESolver::Rand2Exp;
    break;

case stBest1Bin:
    calcTrialSolution = &DESolver::Best1Bin;
    break;

case stRand1Bin:
    calcTrialSolution = &DESolver::Rand1Bin;
    break;

case stRandToBest1Bin:
    calcTrialSolution = &DESolver::RandToBest1Bin;
    break;

case stBest2Bin:
    calcTrialSolution = &DESolver::Best2Bin;
    break;

case stRand2Bin:
    calcTrialSolution = &DESolver::Rand2Bin;
    break;
}

return;
}

bool DESolver::Solve(int maxGenerations)
{
    int generation;
    int candidate;
    bool bAtSolution;
    int generationsPerLoop = 10;

    bestEnergy = 1.0E20;
    bAtSolution = false;

    for (generation=0;
        (generation < maxGenerations) && !bAtSolution && (0 == _kbhit());
        generation++)
    {
        for (candidate=0; candidate < nPop; candidate++)
        {
            (this->*calcTrialSolution)(candidate);
            trialEnergy = EnergyFunction(trialSolution,bAtSolution);

            if (trialEnergy < popEnergy[candidate])
            {
                // New low for this candidate
                popEnergy[candidate] = trialEnergy;
                CopyVector(RowVector(population,candidate),trialSolution);

                // Check if all-time low
                if (trialEnergy < bestEnergy)
                {
                    bestEnergy = trialEnergy;
                    CopyVector(bestSolution,trialSolution);
                }
            }
        }

        if ((generation % generationsPerLoop) == (generationsPerLoop - 1))
        {
            printf("Gens %u Cost %.15g\n", generation+1, Energy());
        }

        if (0 != _kbhit())
        {
            _getch();
        }

        generations = generation;
        return(bAtSolution);
    }
}

void DESolver::Best1Exp(int candidate)
{
    int r1, r2;
    int n;

    SelectSamples(candidate,&r1,&r2);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; (RandomUniform(0.0,1.0) < probability) && (i < nDim); i++)

```

```

{
    trialSolution[n] = bestSolution[n]
        + scale * (Element(population,r1,n)
        - Element(population,r2,n));
    n = (n + 1) % nDim;
}

return;
}

void DESolver::Rand1Exp(int candidate)
{
    int r1, r2, r3;
    int n;

    SelectSamples(candidate,&r1,&r2,&r3);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; (RandomUniform(0.0,1.0) < probability) && (i < nDim); i++)
    {
        trialSolution[n] = Element(population,r1,n)
            + scale * (Element(population,r2,n)
            - Element(population,r3,n));
        n = (n + 1) % nDim;
    }

    return;
}

void DESolver::RandToBest1Exp(int candidate)
{
    int r1, r2;
    int n;

    SelectSamples(candidate,&r1,&r2);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; (RandomUniform(0.0,1.0) < probability) && (i < nDim); i++)
    {
        trialSolution[n] += scale * (bestSolution[n] - trialSolution[n])
            + scale * (Element(population,r1,n)
            - Element(population,r2,n));
        n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Best2Exp(int candidate)
{
    int r1, r2, r3, r4;
    int n;

    SelectSamples(candidate,&r1,&r2,&r3,&r4);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; (RandomUniform(0.0,1.0) < probability) && (i < nDim); i++)
    {
        trialSolution[n] = bestSolution[n] +
            scale * (Element(population,r1,n)
            + Element(population,r2,n)
            - Element(population,r3,n)
            - Element(population,r4,n));
        n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Rand2Exp(int candidate)
{
    int r1, r2, r3, r4, r5;
    int n;

    SelectSamples(candidate,&r1,&r2,&r3,&r4,&r5);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; (RandomUniform(0.0,1.0) < probability) && (i < nDim); i++)
    {
        trialSolution[n] = Element(population,r1,n)
            + scale * (Element(population,r2,n)
            + Element(population,r3,n)
            - Element(population,r4,n)
            - Element(population,r5,n));
        n = (n + 1) % nDim;
    }

    return;
}

```



```

void DESolver::Best1Bin(int candidate)
{
    int r1, r2;
    int n;

    SelectSamples(candidate, &r1, &r2);
    n = (int)RandomUniform(0.0, (double)nDim);

    CopyVector(trialSolution, RowVector(population, candidate));
    for (int i=0; i < nDim; i++)
    {
        if ((RandomUniform(0.0, 1.0) < probability) || (i == (nDim - 1)))
            trialSolution[n] = bestSolution[n]
                + scale * (Element(population, r1, n)
                    - Element(population, r2, n));
        n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Rand1Bin(int candidate)
{
    int r1, r2, r3;
    int n;

    SelectSamples(candidate, &r1, &r2, &r3);
    n = (int)RandomUniform(0.0, (double)nDim);

    CopyVector(trialSolution, RowVector(population, candidate));
    for (int i=0; i < nDim; i++)
    {
        if ((RandomUniform(0.0, 1.0) < probability) || (i == (nDim - 1)))
            trialSolution[n] = Element(population, r1, n)
                + scale * (Element(population, r2, n)
                    - Element(population, r3, n));
        n = (n + 1) % nDim;
    }

    return;
}

void DESolver::RandToBest1Bin(int candidate)
{
    int r1, r2;
    int n;

    SelectSamples(candidate, &r1, &r2);
    n = (int)RandomUniform(0.0, (double)nDim);

    CopyVector(trialSolution, RowVector(population, candidate));
    for (int i=0; i < nDim; i++)
    {
        if ((RandomUniform(0.0, 1.0) < probability) || (i == (nDim - 1)))
            trialSolution[n] += scale * (bestSolution[n] - trialSolution[n])
                + scale * (Element(population, r1, n)
                    - Element(population, r2, n));
        n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Best2Bin(int candidate)
{
    int r1, r2, r3, r4;
    int n;

    SelectSamples(candidate, &r1, &r2, &r3, &r4);
    n = (int)RandomUniform(0.0, (double)nDim);

    CopyVector(trialSolution, RowVector(population, candidate));
    for (int i=0; i < nDim; i++)
    {
        if ((RandomUniform(0.0, 1.0) < probability) || (i == (nDim - 1)))
            trialSolution[n] = bestSolution[n]
                + scale * (Element(population, r1, n)
                    + Element(population, r2, n)
                    - Element(population, r3, n)
                    - Element(population, r4, n));
        n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Rand2Bin(int candidate)
{
    int r1, r2, r3, r4, r5;
    int n;

    SelectSamples(candidate, &r1, &r2, &r3, &r4, &r5);

```

```

n = (int)RandomUniform(0.0,(double)nDim);

CopyVector(trialSolution,RowVector(population,candidate));
for (int i=0; i < nDim; i++)
{
    if ((RandomUniform(0.0,1.0) < probability) || (i == (nDim - 1)))
        trialSolution[n] = Element(population,r1,n)
            + scale * (Element(population,r2,n)
                + Element(population,r3,n)
                - Element(population,r4,n)
                - Element(population,r5,n));
    n = (n + 1) % nDim;
}

return;
}

void DESolver::SelectSamples(int candidate,int *r1,int *r2,
    int *r3,int *r4,int *r5)
{
    if (r1)
    {
        do
        {
            *r1 = (int)RandomUniform(0.0,(double)nPop);
        }
        while (*r1 == candidate);
    }

    if (r2)
    {
        do
        {
            *r2 = (int)RandomUniform(0.0,(double)nPop);
        }
        while ((*r2 == candidate) || (*r2 == *r1));
    }

    if (r3)
    {
        do
        {
            *r3 = (int)RandomUniform(0.0,(double)nPop);
        }
        while ((*r3 == candidate) || (*r3 == *r2) || (*r3 == *r1));
    }

    if (r4)
    {
        do
        {
            *r4 = (int)RandomUniform(0.0,(double)nPop);
        }
        while ((*r4 == candidate) || (*r4 == *r3) || (*r4 == *r2) || (*r4 == *r1));
    }

    if (r5)
    {
        do
        {
            *r5 = (int)RandomUniform(0.0,(double)nPop);
        }
        while ((*r5 == candidate) || (*r5 == *r4) || (*r5 == *r3)
            || (*r5 == *r2) || (*r5 == *r1));
    }

    return;
}

/*-----Constants for RandomUniform()-----*/
#define SEED 3
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

double DESolver::RandomUniform(double minValue,double maxValue)
{
    long j;
    long k;
    static long idum;
    static long idum2=123456789;
    static long iy=0;

```

```

static long iv[NTAB];
double result;

if (iy == 0)
    idum = SEED;

if (idum <= 0)
{
    if (-idum < 1)
        idum = 1;
    else
        idum = -idum;

    idum2 = idum;

    for (j=NTAB+7; j>=0; j--)
    {
        k = idum / IQ1;
        idum = IA1 * (idum - k*IQ1) - k*IR1;
        if (idum < 0) idum += IM1;
        if (j < NTAB) iv[j] = idum;
    }

    iy = iv[0];
}

k = idum / IQ1;
idum = IA1 * (idum - k*IQ1) - k*IR1;

if (idum < 0)
    idum += IM1;

k = idum2 / IQ2;
idum2 = IA2 * (idum2 - k*IQ2) - k*IR2;

if (idum2 < 0)
    idum2 += IM2;

j = iy / NDIV;
iy = iv[j] - idum2;
iv[j] = idum;

if (iy < 1)
    iy += IMM1;

result = AM * iy;

if (result > RNMX)
    result = RNMX;

result = minValue + result * (maxValue - minValue);
return(result);
}

// END FIRST FILE

// BEGIN SECOND FILE: DESolver.h
// Differential Evolution Solver Class
// Based on algorithms developed by Dr. Rainer Storn & Kenneth Price
// Written By: Lester E. Godwin
//           PushCorp, Inc.
//           Dallas, Texas
//           972-840-0208 x102
//           godwin@pushcorp.com
// Created: 6/8/98
// Last Modified: 6/8/98
// Revision: 1.0

#ifndef _DESOLVER_H
#define _DESOLVER_H

#define stBest1Exp 0
#define stRand1Exp 1
#define stRandToBest1Exp 2
#define stBest2Exp 3
#define stRand2Exp 4
#define stBest1Bin 5
#define stRand1Bin 6
#define stRandToBest1Bin 7
#define stBest2Bin 8
#define stRand2Bin 9

class DESolver;

typedef void (DESolver::*StrategyFunction)(int);

class DESolver
{
public:
    DESolver(int dim,int popSize);
    ~DESolver(void);

    // Setup() must be called before solve to set min, max, strategy etc.

```

```

void Setup(double min[],double max[],int deStrategy,
           double diffScale,double crossoverProb);

// Solve() returns true if EnergyFunction() returns true.
// Otherwise it runs maxGenerations generations and returns false.
virtual bool Solve(int maxGenerations);

// EnergyFunction must be overridden for problem to solve
// testSolution[] is nDim array for a candidate solution
// setting bAtSolution = true indicates solution is found
// and Solve() immediately returns true.
virtual double EnergyFunction(double testSolution[],bool &bAtSolution) = 0;

int Dimension(void) { return(nDim); }
int Population(void) { return(nPop); }

// Call these functions after Solve() to get results.
double Energy(void) { return(bestEnergy); }
double *Solution(void) { return(bestSolution); }

int Generations(void) { return(generations); }

protected:
void SelectSamples(int candidate,int *r1,int *r2=0,int *r3=0,
                  int *r4=0,int *r5=0);
double RandomUniform(double min,double max);

int nDim;
int nPop;
int generations;

int strategy;
StrategyFunction calcTrialSolution;
double scale;
double probability;

double trialEnergy;
double bestEnergy;

double *trialSolution;
double *bestSolution;
double *popEnergy;
double *population;

private:
void Best1Exp(int candidate);
void Rand1Exp(int candidate);
void RandToBest1Exp(int candidate);
void Best2Exp(int candidate);
void Rand2Exp(int candidate);
void Best1Bin(int candidate);
void Rand1Bin(int candidate);
void RandToBest1Bin(int candidate);
void Best2Bin(int candidate);
void Rand2Bin(int candidate);
};

// I added the following stuff 19 March 2007
// Brent Lehman

struct ASpectrum
{
    unsigned mNumValues;
    double* mReals;
    double* mImags;
};

bool ComputeSpectrum(double* evenZeros, unsigned numEvenZeros, double* oddZero,
                    double* evenPoles, unsigned numEvenPoles, double* oddPole,
                    double gain, ASpectrum* spectrum);

class FilterSolver : public DESolver
{
public:
    FilterSolver(int dim, int popSize, int spectrumSize,
                unsigned numZeros, unsigned numPoles, bool minimumPhase) :
        DESolver(dim, popSize)
    {
        mSpectrum.mNumValues = spectrumSize;
        mSpectrum.mReals = new double[spectrumSize];
        mSpectrum.mImags = new double[spectrumSize];
        mNumZeros = numZeros;
        mNumPoles = numPoles;
        mMinimumPhase = minimumPhase;
    }
    virtual ~FilterSolver()
    {
        delete[] mSpectrum.mReals;
        delete[] mSpectrum.mImags;
    }
    virtual double EnergyFunction(double testSolution[], bool& bAtSolution);
    virtual ASpectrum* Spectrum() {return &mSpectrum;}
};

```

```

private:
    unsigned    mNumZeros;
    unsigned    mNumPoles;
    bool        mMinimumPhase;
    ASpectrum   mSpectrum;
};

#endif // _DESOLVER_H

// END SECOND FILE DESolver.h

// BEGIN FINAL FILE: FilterDesign.cpp
/*
 *
 * Filter Design Utility
 * Source
 *
 * Brent Lehman
 * 16 March 2007
 *
 */

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// The idea is that an optimization algorithm passes a bunch of //
// different filter specifications to the function //
// "EnergyFunction" below. That function is supposed to //
// compute an "error" or "cost" value for each specification //
// it receives, which the algorithm uses to decide on other //
// filter specifications to try. Over the course of several //
// thousand different specifications, the algorithm will //
// eventually converge on a single best one. This one has the //
// lowest error value of all possible specifications. Thus, //
// you effectively tell the optimization algorithm what it's //
// looking for through code that you put into EnergyFunction. //
//
// Look for a note in the code like this one to see what part //
// you need to change for your own uses. //
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <conio.h>
#include <math.h>
#include <time.h>
#include "DESolver.h"

#define kIntIsOdd(x) (((x) & 0x00000001) == 1)

double FilterSolver::EnergyFunction(double testSolution[], bool& bAtSolution)
{
    unsigned i;
    double tempReal;
    double tempImag;

    // You probably will want to keep this if statement and its contents
    if (mMinimumPhase)
    {
        // Make sure there are no zeros outside the unit circle
        unsigned lastEvenZero = (mNumZeros & 0xffffffe) - 1;
        for (i = 0; i <= lastEvenZero; i+=2)
        {
            tempReal = testSolution[i];
            tempImag = testSolution[i+1];
            if ((tempReal*tempReal + tempImag*tempImag) > 1.0)
            {
                return 1.0e+300;
            }
        }

        if (kIntIsOdd(mNumZeros))
        {
            tempReal = testSolution[mNumZeros - 1];
            if ((tempReal * tempReal) > 1.0)
            {
                return 1.0e+300;
            }
        }
    }

    // Make sure there are no poles on or outside the unit circle
    // You probably will want to keep this too
    unsigned lastEvenPole = mNumZeros + (mNumPoles & 0xffffffe) - 2;
    for (i = mNumZeros; i <= lastEvenPole; i+=2)
    {

```

```

tempReal = testSolution[i];
tempImag = testSolution[i+1];
if ((tempReal*tempReal + tempImag*tempImag) > 0.999999999)
{
return 1.0e+300;
}
}

// If you keep the for loop above, keep this too
if (kIntIsOdd(mNumPoles))
{
tempReal = testSolution[mNumZeros + mNumPoles - 1];
if ((tempReal * tempReal) > 1.0)
{
return 1.0e+300;
}
}

double* evenZeros = &(testSolution[0]);
double* evenPoles = &(testSolution[mNumZeros]);
double* oddZero = NULL;
double* oddPole = NULL;
double gain = testSolution[mNumZeros + mNumPoles];

if (kIntIsOdd(mNumZeros))
{
oddZero = &(testSolution[mNumZeros - 1]);
}

if (kIntIsOdd(mNumPoles))
{
oddPole = &(testSolution[mNumZeros + mNumPoles - 1]);
}

ComputeSpectrum(evenZeros, mNumZeros & 0xffffffff, oddZero,
evenPoles, mNumPoles & 0xffffffff, oddPole,
gain, &mSpectrum);

unsigned numPoints = mSpectrum.mNumValues;

/////////////////////////////////////////////////////////////////
//
// Use the impulse response, held in the variable //
// "mSpectrum", to compute a score for the solution that //
// has been passed into this function. You probably don't //
// want to touch any of the code above this point, but //
// from here to the end of this function, it's all you! //
// //
/////////////////////////////////////////////////////////////////

#define kLnTwoToThe127 88.02969193111305
#define kRecipLn10 0.4342944819032518

// Compute square sum of errors for magnitude
double magnitudeError = 0.0;
double magnitude = 0.0;
double logMagnitude = 0.0;
tempReal = mSpectrum.mReals[0];
tempImag = mSpectrum.mImags[0];
magnitude = tempReal*tempReal + tempImag*tempImag;
double baseMagnitude = 0.0;
if (0.0 == magnitude)
{
baseMagnitude = -kLnTwoToThe127;
}
else
{
baseMagnitude = log(magnitude) * kRecipLn10;
baseMagnitude *= 0.5;
}

for (i = 0; i < numPoints; i++)
{
tempReal = mSpectrum.mReals[i];
tempImag = mSpectrum.mImags[i];
magnitude = tempReal*tempReal + tempImag*tempImag;
if (0.0 == magnitude)
{
logMagnitude = -kLnTwoToThe127;
}
else
{
logMagnitude = log(magnitude) * kRecipLn10;
logMagnitude *= 0.5; // Half the log because it's mag squared
}

logMagnitude -= baseMagnitude;
magnitudeError += logMagnitude * logMagnitude;
}

// Compute errors for phase
double phaseError = 0.0;
double phase = 0.0;

```

```

double componentError = 0.0;
double degree = 1; (((mNumZeros + 1) & 0xffffffff) - 1);
double angleSpacing = -3.141592653589793 * 0.5 / numPoints * degree;
double targetPhase = 0.0;
double oldPhase = 0.0;
double phaseDifference = 0;
double totalPhaseTraversal = 0.0;
double traversalError = 0.0;
for (i = 0; i < (numPoints - 5); i++)
{
    tempReal = mSpectrum.mReals[i];
    tempImag = mSpectrum.mImags[i];
    oldPhase = phase;
    phase = atan2(tempImag, tempReal);
    phaseDifference = phase - oldPhase;
    if (phaseDifference > 3.141592653589793)
    {
        phaseDifference -= 3.141592653589793;
        phaseDifference -= 3.141592653589793;
    }
    else if (phaseDifference < -3.141592653589793)
    {
        phaseDifference += 3.141592653589793;
        phaseDifference += 3.141592653589793;
    }
    totalPhaseTraversal += phaseDifference;
    componentError = cosh(200.0*(phaseDifference - angleSpacing)) - 0.5;
    phaseError += componentError * componentError;
    targetPhase += angleSpacing;
    if (targetPhase < -3.141592653589793)
    {
        targetPhase += 3.141592653589793;
        targetPhase += 3.141592653589793;
    }
}

traversalError = totalPhaseTraversal - angleSpacing * numPoints;
traversalError *= traversalError;

double baseMagnitudeError = baseMagnitude * baseMagnitude;

// Compute weighted sum of the two subtotals
// Take square root
return sqrt(baseMagnitudeError*1.0 + magnitudeError*100.0 +
            phaseError*400.0 + traversalError*4000000.0);
}

////////////////////////////////////
int main(int argc, char** argv)
{
    srand((unsigned)time(NULL));

    unsigned numZeros;
    unsigned numPoles;
    bool minimumPhase;

    if (argc < 4)
    {
        printf("Usage: FilterDesign.exe <minimumPhase?> <numZeros> <numPoles>\n");
        return 0;
    }
    else
    {
        if (0 == atoi(argv[1]))
        {
            minimumPhase = false;
        }
        else
        {
            minimumPhase = true;
        }

        numZeros = (unsigned)atoi(argv[2]);
        if (0 == numZeros)
        {
            numZeros = 1;
        }

        numPoles = (unsigned)atoi(argv[3]);
    }

    unsigned vectorLength = numZeros + numPoles + 1;
    unsigned populationSize = vectorLength * 10;
    FilterSolver theSolver(vectorLength, populationSize, 200,
                          numZeros, numPoles, minimumPhase);

    double* minimumSolution = new double[vectorLength];
    unsigned i;
    if (minimumPhase)
    {
        for (i = 0; i < numZeros; i++)
        {

```

```

        minimumSolution[i] = -sqrt(0.5);
    }
}
else
{
    for (i = 0; i < numZeros; i++)
    {
        minimumSolution[i] = -10.0;
    }
}

for (; i < (vectorLength - 1); i++)
{
    minimumSolution[i] = -sqrt(0.5);
}

minimumSolution[vectorLength - 1] = 0.0;

double* maximumSolution = new double[vectorLength];
if (minimumPhase)
{
    for (i = 0; i < numZeros; i++)
    {
        maximumSolution[i] = sqrt(0.5);
    }
}
else
{
    for (i = 0; i < numZeros; i++)
    {
        maximumSolution[i] = 10.0;
    }
}

for (i = 0; i < (vectorLength - 1); i++)
{
    maximumSolution[i] = sqrt(0.5);
}

maximumSolution[vectorLength - 1] = 2.0;

theSolver.Setup(minimumSolution, maximumSolution, 0, 0.5, 0.75);
theSolver.Solve(1000000);

double* bestSolution = theSolver.Solution();
printf("\nZeros:\n");
unsigned numEvenZeros = numZeros & 0xffffffe;
for (i = 0; i < numEvenZeros; i+=2)
{
    printf("%.10f +/- %.10fi\n", bestSolution[i], bestSolution[i+1]);
}

if (kIntIsOdd(numZeros))
{
    printf("%.10f\n", bestSolution[numZeros-1]);
}

printf("Poles:\n");
unsigned lastEvenPole = numZeros + (numPoles & 0xffffffe) - 2;
for (i = numZeros; i <= lastEvenPole; i+=2)
{
    printf("%.10f +/- %.10fi\n", bestSolution[i], bestSolution[i+1]);
}

unsigned numRoots = numZeros + numPoles;
if (kIntIsOdd(numPoles))
{
    printf("%.10f\n", bestSolution[numRoots-1]);
}

double gain = bestSolution[numRoots];
printf("Gain: %.10f\n", gain);

_getch();
unsigned j;
ASpectrum* spectrum = theSolver.Spectrum();
double logMagnitude;
printf("Magnitude Response, millibels:\n");
for (i = 0; i < 20; i++)
{
    for (j = 0; j < 10; j++)
    {
        logMagnitude = kRecipLn10 *
            log(spectrum->mReals[i*10 + j] * spectrum->mReals[i*10 + j] +
                spectrum->mImags[i*10 + j] * spectrum->mImags[i*10 + j]);
        if (logMagnitude < -9.999)
        {
            logMagnitude = -9.999;
        }
        printf("%+5.0f ", logMagnitude*1000);
    }
    printf("\n");
}
}

```



```

_getch();
double phase;
printf("Phase Response, milliradians:\n");
for (i = 0; i < 20; i++)
{
    for (j = 0; j < 10; j++)
    {
        phase = atan2(spectrum->mImags[i*10 + j], spectrum->mReals[i*10 + j]);
        printf("%+5.0f ", phase*1000);
    }
    printf("\n");
}

_getch();
printf("Biquad Sections:\n");
unsigned numBiquadSections =
    (numZeros > numPoles) ? ((numZeros + 1) >> 1) : ((numPoles + 1) >> 1);
double x0, x1, x2;
double y0, y1, y2;
if (numZeros >=2)
{
    x0 = (bestSolution[0]*bestSolution[0] + bestSolution[1]*bestSolution[1]) *
        gain;
    x1 = 2.0 * bestSolution[0] * gain;
    x2 = gain;
}
else if (1 == numZeros)
{
    x0 = bestSolution[0] * gain;
    x1 = gain;
    x2 = 0.0;
}
else
{
    x0 = gain;
    x1 = 0.0;
    x2 = 0.0;
}

if (numPoles >= 2)
{
    y0 = (bestSolution[numZeros]*bestSolution[numZeros] +
        bestSolution[numZeros+1]*bestSolution[numZeros+1]);
    y1 = 2.0 * bestSolution[numZeros];
    y2 = 1.0;
}
else if (1 == numPoles)
{
    y0 = bestSolution[numZeros];
    y1 = 1.0;
    y2 = 0.0;
}
else
{
    y0 = 1.0;
    y1 = 0.0;
    y2 = 0.0;
}

x0 /= y0;
x1 /= y0;
x2 /= y0;
y1 /= y0;
y2 /= y0;

printf("y[n] = %.10fx[n]", x0);
if (numZeros > 0)
{
    printf(" + %.10fx[n-1]", x1);
}
if (numZeros > 1)
{
    printf(" + %.10fx[n-2]", x2);
}
printf("\n");

if (numPoles > 0)
{
    printf("          + %.10fy[n-1]", y1);
}
if (numPoles > 1)
{
    printf(" + &.10fy[n-2]", y2);
}
if (numPoles > 0)
{
    printf("\n");
}

int numRemainingZeros = numZeros - 2;
int numRemainingPoles = numPoles - 2;
for (i = 1; i < numBiquadSections; i++)

```

```

{
    if (numRemainingZeros >= 2)
    {
        x0 = (bestSolution[i*2] * bestSolution[i*2] +
            bestSolution[i*2+1] * bestSolution[i*2+1]);
        x1 = -2.0 * bestSolution[i*2];
        x2 = 1.0;
    }
    else if (numRemainingZeros >= 1)
    {
        x0 = bestSolution[i*2];
        x1 = 1.0;
        x2 = 0.0;
    }
    else
    {
        x0 = 1.0;
        x1 = 0.0;
        x2 = 0.0;
    }

    if (numRemainingPoles >= 2)
    {
        y0 = (bestSolution[i*2+numZeros] * bestSolution[i*2+numZeros] +
            bestSolution[i*2+numZeros+1] * bestSolution[i*2+numZeros+1]);
        y1 = -2.0 * bestSolution[i*2+numZeros];
        y2 = 1.0;
    }
    else if (numRemainingPoles >= 1)
    {
        y0 = bestSolution[i*2+numZeros];
        y1 = 1.0;
        y2 = 0.0;
    }
    else
    {
        y0 = 1.0;
        y1 = 0.0;
        y2 = 0.0;
    }

    x0 /= y0;
    x1 /= y0;
    x2 /= y0;
    y1 /= y0;
    y2 /= y0;

    printf("y[n] = %.10fx[n]", x0);
    if (numRemainingZeros > 0)
    {
        printf(" + %.10fx[n-1]", x1);
    }
    if (numRemainingZeros > 1)
    {
        printf(" + %.10fx[n-2]", x2);
    }
    printf("\n");

    if (numRemainingPoles > 0)
    {
        printf("          + %.10fy[n-1]", -y1);
    }
    if (numRemainingPoles > 1)
    {
        printf(" + %.10fy[n-2]", -y2);
    }
    if (numRemainingPoles > 0)
    {
        printf("\n");
    }

    numRemainingZeros -= 2;
    numRemainingPoles -= 2;
}

_getch();
printf("Full Expansion:\n");
double* xpolynomial = new double[numRoots + 1];
memset(xpolynomial, 0, sizeof(double) * (numRoots + 1));
xpolynomial[0] = 1.0;
if (numZeros >= 2)
{
    xpolynomial[0] = bestSolution[0] * bestSolution[0] +
        bestSolution[1] * bestSolution[1];
    xpolynomial[1] = -2.0 * bestSolution[0];
    xpolynomial[2] = 1.0;
}
else if (numZeros == 1)
{
    xpolynomial[0] = bestSolution[0];
    xpolynomial[1] = 1.0;
}
else

```

```

{
    xpolynomial[0] = 1.0;
}

for (i = 2, numRemainingZeros = numZeros; numRemainingZeros >= 2;
     i += 2, numRemainingZeros-=2)
{
    x2 = 1.0;
    x1 = -2.0 * bestSolution[i];
    x0 = bestSolution[i] * bestSolution[i] +
        bestSolution[i+1] * bestSolution[i+1];
    for (j = numRoots; j > 1; j--)
    {
        xpolynomial[j] = xpolynomial[j-2] + xpolynomial[j-1] * x1 +
            xpolynomial[j] * x0;
    }
    xpolynomial[1] = xpolynomial[0] * x1 + xpolynomial[1] * x0;
    xpolynomial[0] *= x0;
}

if (numRemainingZeros > 0)
{
    x1 = 1.0;
    x0 = bestSolution[numZeros-1];
    for (j = numRoots; j > 0; j--)
    {
        xpolynomial[j] = xpolynomial[j-1] + xpolynomial[j] * x0;
    }
    xpolynomial[0] *= x0;
}

double* ypolynomial = new double[numRoots + 1];
memset(ypolynomial, 0, sizeof(double) * (numRoots + 1));
ypolynomial[0] = 1.0;
if (numPoles >= 2)
{
    ypolynomial[0] = bestSolution[numZeros] * bestSolution[numZeros] +
        bestSolution[numZeros+1] * bestSolution[numZeros+1];
    ypolynomial[1] = -2.0 * bestSolution[numZeros];
    ypolynomial[2] = 1.0;
}
else if (numPoles == 1)
{
    ypolynomial[0] = bestSolution[numZeros];
    ypolynomial[1] = 1.0;
}
else
{
    xpolynomial[0] = 1.0;
}

for (i = 2, numRemainingPoles = numPoles; numRemainingPoles >= 2;
     i += 2, numRemainingPoles-=2)
{
    y2 = 1.0;
    y1 = -2.0 * bestSolution[numZeros+i];
    y0 = bestSolution[numZeros+i] * bestSolution[numZeros+i] +
        bestSolution[numZeros+i+1] * bestSolution[numZeros+i+1];
    for (j = numRoots; j > 1; j--)
    {
        ypolynomial[j] = ypolynomial[j-2] + ypolynomial[j-1] * y1 +
            ypolynomial[j] * y0;
    }
    ypolynomial[1] = ypolynomial[0] * y1 + ypolynomial[1] * y0;
    ypolynomial[0] *= y0;
}

if (numRemainingPoles > 0)
{
    y1 = 1.0;
    y0 = bestSolution[numZeros+numPoles-1];
    for (j = numRoots; j > 0; j--)
    {
        ypolynomial[j] = ypolynomial[j-1] + ypolynomial[j] * y0;
    }
    ypolynomial[0] *= y0;
}

y0 = ypolynomial[0];
for (i = 0; i <= numRoots; i++)
{
    xpolynomial[i] /= y0;
    ypolynomial[i] /= y0;
}

printf("y[n] = %.10fx[n]", xpolynomial[0]*gain);
for (i = 1; i <= numZeros; i++)
{
    printf(" + %.10fx[n-%d]", xpolynomial[i]*gain, i);
    if ((i % 3) == 2)
    {
        printf("\n");
    }
}

```

```

}

if ((i % 3) != 0)
{
    printf("\n");
}

if (numPoles > 0)
{
    printf("          ");
}

for (i = 1; i <= numPoles; i++)
{
    printf(" + %.10fy[n-%d]", -ypolynomial[i], i);
    if ((i % 3) == 2)
    {
        printf("\n");
    }
}

if ((i % 3) != 0)
{
    printf("\n");
}

delete[] minimumSolution;
delete[] maximumSolution;
delete[] xpolynomial;
delete[] ypolynomial;
}

bool ComputeSpectrum(double* evenZeros, unsigned numEvenZeros, double* oddZero,
                    double* evenPoles, unsigned numEvenPoles, double* oddPole,
                    double gain, ASpectrum* spectrum)
{
    unsigned i, j;

    // For equally spaced points on the unit circle
    unsigned numPoints = spectrum->mNumValues;
    double spacingAngle = 3.141592653589793 / (numPoints - 1);
    double pointArgument = 0.0;
    double pointReal = 0.0;
    double pointImag = 0.0;
    double rootReal = 0.0;
    double rootImag = 0.0;
    double differenceReal = 0.0;
    double differenceImag = 0.0;
    double responseReal = 1.0;
    double responseImag = 0.0;
    double recipSquareMagnitude = 0.0;
    double recipReal = 0.0;
    double recipImag = 0.0;
    double tempRealReal = 0.0;
    double tempRealImag = 0.0;
    double tempImagReal = 0.0;
    double tempImagImag = 0.0;

    for (i = 0; i < numPoints; i++)
    {
        responseReal = 1.0;
        responseImag = 0.0;

        // The imaginary component is negated because we're using 1/z, not z
        pointReal = cos(pointArgument);
        pointImag = -sin(pointArgument);

        // For each even zero
        for (j = 0; j < numEvenZeros; j+=2)
        {
            rootReal = evenZeros[j];
            rootImag = evenZeros[j + 1];
            // Compute distance from that zero to that point
            differenceReal = pointReal - rootReal;
            differenceImag = pointImag - rootImag;
            // Multiply that distance by the accumulating product
            tempRealReal = responseReal * differenceReal;
            tempRealImag = responseReal * differenceImag;
            tempImagReal = responseImag * differenceReal;
            tempImagImag = responseImag * differenceImag;
            responseReal = tempRealReal - tempImagImag;
            responseImag = tempRealImag + tempImagReal;
            // Do the same with the conjugate root
            differenceImag = pointImag + rootImag;
            tempRealReal = responseReal * differenceReal;
            tempRealImag = responseReal * differenceImag;
            tempImagReal = responseImag * differenceReal;
            tempImagImag = responseImag * differenceImag;
            responseReal = tempRealReal - tempImagImag;
            responseImag = tempRealImag + tempImagReal;
            // The following way is little faster, if any
            // response *= (1/z - r) * (1/z - conj(r))

```

```

//      *= r*conj(r) - (r + conj(r))/z + 1/(z*z)
//      *= real(r)*real(r) + imag(r)*imag(r) - 2*real(r)/z + 1/(z*z)
//      *= ... - 2*real(r)*conj(z) + conj(z)*conj(z)
//      *= ... - 2*real(r)*real(z) + 2i*real(r)*imag(z) +
//      real(z)*real(z) - 2i*real(z)*imag(z) + imag(z)*imag(z)
//      *= real(r)*real(r) + imag(r)*imag(r) - 2*real(r)*real(z) +
//      real(z)*real(z) + imag(z)*imag(z) +
//      2i * imag(z) * (real(r) - real(z))
//      *= (real(r) - real(z))^2 + imag(r)^2 + imag(z)^2 +
//      2i * imag(z) * (real(r) - real(z))
// This ends up being 8 multiplications, 6 additions
}

if (NULL != oddZero)
{
    rootReal = *oddZero;
    // Compute distance from that zero to that point
    differenceReal = pointReal - rootReal;
    differenceImag = pointImag;
    // Multiply that distance by the accumulating product
    tempRealReal = responseReal * differenceReal;
    tempRealImag = responseReal * differenceImag;
    tempImagReal = responseImag * differenceReal;
    tempImagImag = responseImag * differenceImag;
    responseReal = tempRealReal - tempImagImag;
    responseImag = tempRealImag + tempImagReal;
}

// For each pole
for (j = 0; j < numEvenPoles; j+=2)
{
    rootReal = evenPoles[j];
    rootImag = evenPoles[j + 1];
    // Compute distance from that pole to that point
    differenceReal = pointReal - rootReal;
    differenceImag = pointImag - rootImag;
    // Multiply the reciprocal of that distance by the accumulating product
    recipSquareMagnitude = 1.0 / (differenceReal * differenceReal +
        differenceImag * differenceImag);
    recipReal = differenceReal * recipSquareMagnitude;
    recipImag = -differenceImag * recipSquareMagnitude;
    tempRealReal = responseReal * recipReal;
    tempRealImag = responseReal * recipImag;
    tempImagReal = responseImag * recipReal;
    tempImagImag = responseImag * recipImag;
    responseReal = tempRealReal - tempImagImag;
    responseImag = tempRealImag + tempImagReal;
    // Do the same with the conjugate root
    differenceImag = pointImag + rootImag;
    recipSquareMagnitude = 1.0 / (differenceReal * differenceReal +
        differenceImag * differenceImag);
    recipReal = differenceReal * recipSquareMagnitude;
    recipImag = -differenceImag * recipSquareMagnitude;
    tempRealReal = responseReal * recipReal;
    tempRealImag = responseReal * recipImag;
    tempImagReal = responseImag * recipReal;
    tempImagImag = responseImag * recipImag;
    responseReal = tempRealReal - tempImagImag;
    responseImag = tempRealImag + tempImagReal;
}

if (NULL != oddPole)
{
    rootReal = *oddPole;
    // Compute distance from that point to that zero
    differenceReal = pointReal - rootReal;
    differenceImag = pointImag;
    // Multiply the reciprocal of that distance by the accumulating product
    recipSquareMagnitude = 1.0 / (differenceReal * differenceReal +
        differenceImag * differenceImag);
    recipReal = differenceReal * recipSquareMagnitude;
    recipImag = -differenceImag * recipSquareMagnitude;
    tempRealReal = responseReal * recipReal;
    tempRealImag = responseReal * recipImag;
    tempImagReal = responseImag * recipReal;
    tempImagImag = responseImag * recipImag;
    responseReal = tempRealReal - tempImagImag;
    responseImag = tempRealImag + tempImagReal;
}

// Multiply by the gain
responseReal *= gain;
responseImag *= gain;

spectrum->mReals[i] = responseReal;
spectrum->mImags[i] = responseImag;

pointArgument += spacingAngle;
}

return true;
}

```

```

// Half-band lowpass
/*
#define kLnTwoToThe127 88.02969193111305
#define kRecipLn10      0.4342944819032518

// Compute square sum of errors for bottom half band
unsigned numLoBandPoints = numPoints >> 1;
double loBandError = 0.0;
double magnitude = 0.0;
double logMagnitude = 0.0;
for (i = 0; i < numLoBandPoints; i++)
{
    tempReal = mSpectrum.mReals[i];
    tempImag = mSpectrum.mImags[i];
    magnitude = tempReal*tempReal + tempImag*tempImag;
    if (0.0 == magnitude)
    {
        logMagnitude = -kLnTwoToThe127;
    }
    else
    {
        logMagnitude = log(magnitude) * kRecipLn10;
        logMagnitude *= 0.5; // Half the log because it's mag squared
    }

    loBandError += logMagnitude * logMagnitude;
}

// Compute errors for top half of band
double hiBandError = 0.0;
for ( ; i < numPoints; i++)
{
    tempReal = mSpectrum.mReals[i];
    tempImag = mSpectrum.mImags[i];
    magnitude = tempReal*tempReal + tempImag*tempImag;
    hiBandError += magnitude; // Already a squared value
}

// Compute weighted sum of the two subtotals
// Take square root
return sqrt(loBandError + 5000.0 * hiBandError);
*/

```

[Prewarping](#) (click this to go back to the index)

Type : explanation

References : Posted by robert bristow-johnson (better known as "rbj")

Notes :

prewarping is simply recognizing the warping that the BLT introduces. to determine frequency response, we evaluate the digital $H(z)$ at $z = \exp(j\omega T)$ and we evaluate the analog $H_a(s)$ at $s = j\omega W$. the following will confirm the $j\omega$ to unit circle mapping and will show exactly what the mapping is (this is the same stuff in the textbooks):

the BLT says: $s = (2/T) * (z-1)/(z+1)$

substituting: $s = j\omega W = (2/T) * (\exp(j\omega T) - 1) / (\exp(j\omega T) + 1)$

$j\omega W = (2/T) * (\exp(j\omega T/2) - \exp(-j\omega T/2)) / (\exp(j\omega T/2) + \exp(-j\omega T/2))$

$= (2/T) * (j * 2 * \sin(\omega T/2)) / (2 * \cos(\omega T/2))$

$= j * (2/T) * \tan(\omega T/2)$

or

analog $W = (2/T) * \tan(\omega T/2)$

so when the real input frequency is ω , the digital filter will behave with the same amplitude gain and phase shift as the analog filter will have at a hypothetical frequency of W . as ωT approaches π (Nyquist) the digital filter behaves as the analog filter does as $W \rightarrow \infty$. for each degree of freedom that you have in your design equations, you can adjust the analog design frequency to be just right so that when the deterministic BLT warping does its thing, the resultant warped frequency comes out just right. for a simple LPF, you have only one degree of freedom, the cutoff frequency. you can precompensate it so that the true cutoff comes out right but that is it, above the cutoff, you will see that the LPF dives down to $-\infty$ dB faster than an equivalent analog at the same frequencies.

[RBJ Audio-EQ-Cookbook](#) (click this to go back to the index)

Type : EQ filter kookbook

References : Posted by Robert Bristow-Johnson

Linked file : [Audio-EQ-Cookbook.txt](#) (this linked file is included below)

Notes :

Equations for creating different equalization filters.
see linked file

Comments

from : martin.eisenberg [[a t]] udo.edu

comment : Sorry, a slight correction: rewrite the formula as

$$S = s * \log_2(10)/40 * \sin(w0)/w0 * (A^2+1)/\text{abs}(A^2-1)$$

nad make s always positive.

from : martin.eisenberg [[a t]] udo.edu

comment : rbj writes with regard to shelving filters:

> _or_ S, a "shelf slope" parameter (for shelving EQ only). When S = 1,
> the shelf slope is as steep as it can be and remain monotonically
> increasing or decreasing gain with frequency. The shelf slope, in
> dB/octave, remains proportional to S for all other values for a
> fixed f0/Fs and dBgain.

The precise relation for both low and high shelf filters is

$$S = -s * \log_2(10)/40 * \sin(w0)/w0 * (A^2+1)/(A^2-1)$$

where s is the true shelf midpoint slope in dB/oct and w0, A are defined in the Cookbook just below the quoted paragraph. It's your responsibility to keep the overshoots in check by using sensible s values. Also make sure that s has the right sign -- negative for low boost or high cut, positive otherwise.

To find the relation I first differentiated the dB magnitude response of the general transfer function in eq. 1 with regard to log frequency, inserted the low shelf coefficient expressions, and evaluated at w0. Second, I equated this derivative to s and solved for alpha. Third, I equated the result to rbj's expression for alpha and solved for S yielding the above formula. Finally I checked it with the high shelf filter.

Linked files

Cookbook formulae for audio EQ biquad filter coefficients

by Robert Bristow-Johnson <rbj@audioimagination.com>

All filter transfer functions were derived from analog prototypes (that are shown below for each EQ filter type) and had been digitized using the Bilinear Transform. BLT frequency warping has been taken into account for both significant frequency relocation (this is the normal "prewarping" that is necessary when using the BLT) and for bandwidth readjustment (since the bandwidth is compressed when mapped from analog to digital using the BLT).

First, given a biquad transfer function defined as:

$$H(z) = \frac{b_0 + b_1*z^{-1} + b_2*z^{-2}}{a_0 + a_1*z^{-1} + a_2*z^{-2}} \quad (\text{Eq 1})$$

This shows 6 coefficients instead of 5 so, depending on your architecture, you will likely normalize a0 to be 1 and perhaps also b0 to 1 (and collect that into an overall gain coefficient). Then your transfer function would look like:

$$H(z) = \frac{(b_0/a_0) + (b_1/a_0)*z^{-1} + (b_2/a_0)*z^{-2}}{1 + (a_1/a_0)*z^{-1} + (a_2/a_0)*z^{-2}} \quad (\text{Eq 2})$$

or

$$H(z) = (b_0/a_0) * \frac{1 + (b_1/b_0)*z^{-1} + (b_2/b_0)*z^{-2}}{1 + (a_1/a_0)*z^{-1} + (a_2/a_0)*z^{-2}} \quad (\text{Eq 3})$$

The most straight forward implementation would be the "Direct Form 1"
(Eq 2):

$$y[n] = (b_0/a_0)*x[n] + (b_1/a_0)*x[n-1] + (b_2/a_0)*x[n-2] - (a_1/a_0)*y[n-1] - (a_2/a_0)*y[n-2] \quad (\text{Eq 4})$$

This is probably both the best and the easiest method to implement in the 56K and other fixed-point or floating-point architectures with a double wide accumulator.

Begin with these user defined parameters:

Fs (the sampling frequency)

f0 ("wherever it's happenin', man." Center Frequency or Corner Frequency, or shelf midpoint frequency, depending on which filter type. The "significant frequency".)

dBgain (used only for peaking and shelving filters)

Q (the EE kind of definition, except for peakingEQ in which A*Q is the classic EE Q. That adjustment in definition was made so that a boost of N dB followed by a cut of N dB for identical Q and f0/Fs results in a precisely flat unity gain filter or "wire".)

or BW, the bandwidth in octaves (between -3 dB frequencies for BPF and notch or between midpoint (dBgain/2) gain frequencies for peaking EQ)

or S, a "shelf slope" parameter (for shelving EQ only). When S = 1, the shelf slope is as steep as it can be and remain monotonically increasing or decreasing gain with frequency. The shelf slope, in dB/octave, remains proportional to S for all other values for a fixed f0/Fs and dBgain.

Then compute a few intermediate variables:

$$A = \sqrt{10^{(dBgain/20)}} \\ = 10^{(dBgain/40)} \quad (\text{for peaking and shelving EQ filters only})$$

$$w_0 = 2\pi f_0 / F_s$$

$$\cos(w_0) \\ \sin(w_0)$$

$$\alpha = \sin(w_0) / (2*Q) \quad (\text{case: Q}) \\ = \sin(w_0) * \sinh(\ln(2)/2 * BW * w_0 / \sin(w_0)) \quad (\text{case: BW}) \\ = \sin(w_0) / 2 * \sqrt{(A + 1/A) * (1/S - 1) + 2} \quad (\text{case: S})$$

FYI: The relationship between bandwidth and Q is
 $1/Q = 2 * \sinh(\ln(2)/2 * BW * w_0 / \sin(w_0))$ (digital filter w BLT)
 or $1/Q = 2 * \sinh(\ln(2)/2 * BW)$ (analog filter prototype)

The relationship between shelf slope and Q is
 $1/Q = \sqrt{(A + 1/A) * (1/S - 1) + 2}$

$$2 * \sqrt{A} * \alpha = \sin(w_0) * \sqrt{(A^2 + 1) * (1/S - 1) + 2 * A}$$

is a handy intermediate variable for shelving EQ filters.

Finally, compute the coefficients for whichever filter type you want:
 (The analog prototypes, H(s), are shown for each filter type for normalized frequency.)

LPF: $H(s) = 1 / (s^2 + s/Q + 1)$

$$b_0 = (1 - \cos(w_0)) / 2 \\ b_1 = 1 - \cos(w_0) \\ b_2 = (1 - \cos(w_0)) / 2 \\ a_0 = 1 + \alpha \\ a_1 = -2 * \cos(w_0) \\ a_2 = 1 - \alpha$$

HPF: $H(s) = s^2 / (s^2 + s/Q + 1)$

$$b_0 = (1 + \cos(w_0)) / 2$$

```

b1 = -(1 + cos(w0))
b2 = (1 + cos(w0))/2
a0 = 1 + alpha
a1 = -2*cos(w0)
a2 = 1 - alpha

```

BPF: $H(s) = s / (s^2 + s/Q + 1)$ (constant skirt gain, peak gain = Q)

```

b0 = sin(w0)/2 = Q*alpha
b1 = 0
b2 = -sin(w0)/2 = -Q*alpha
a0 = 1 + alpha
a1 = -2*cos(w0)
a2 = 1 - alpha

```

BPF: $H(s) = (s/Q) / (s^2 + s/Q + 1)$ (constant 0 dB peak gain)

```

b0 = alpha
b1 = 0
b2 = -alpha
a0 = 1 + alpha
a1 = -2*cos(w0)
a2 = 1 - alpha

```

notch: $H(s) = (s^2 + 1) / (s^2 + s/Q + 1)$

```

b0 = 1
b1 = -2*cos(w0)
b2 = 1
a0 = 1 + alpha
a1 = -2*cos(w0)
a2 = 1 - alpha

```

APF: $H(s) = (s^2 - s/Q + 1) / (s^2 + s/Q + 1)$

```

b0 = 1 - alpha
b1 = -2*cos(w0)
b2 = 1 + alpha
a0 = 1 + alpha
a1 = -2*cos(w0)
a2 = 1 - alpha

```

peakingEQ: $H(s) = (s^2 + s*(A/Q) + 1) / (s^2 + s/(A*Q) + 1)$

```

b0 = 1 + alpha*A
b1 = -2*cos(w0)
b2 = 1 - alpha*A
a0 = 1 + alpha/A
a1 = -2*cos(w0)
a2 = 1 - alpha/A

```

lowShelf: $H(s) = A * (s^2 + (sqrt(A)/Q)*s + A) / (A*s^2 + (sqrt(A)/Q)*s + 1)$

```

b0 = A*( (A+1) - (A-1)*cos(w0) + 2*sqrt(A)*alpha )
b1 = 2*A*( (A-1) - (A+1)*cos(w0) )
b2 = A*( (A+1) - (A-1)*cos(w0) - 2*sqrt(A)*alpha )
a0 = (A+1) + (A-1)*cos(w0) + 2*sqrt(A)*alpha
a1 = -2*( (A-1) + (A+1)*cos(w0) )
a2 = (A+1) + (A-1)*cos(w0) - 2*sqrt(A)*alpha

```

highShelf: $H(s) = A * (A*s^2 + (sqrt(A)/Q)*s + 1) / (s^2 + (sqrt(A)/Q)*s + A)$

```

b0 = A*( (A+1) + (A-1)*cos(w0) + 2*sqrt(A)*alpha )
b1 = -2*A*( (A-1) + (A+1)*cos(w0) )
b2 = A*( (A+1) + (A-1)*cos(w0) - 2*sqrt(A)*alpha )
a0 = (A+1) - (A-1)*cos(w0) + 2*sqrt(A)*alpha
a1 = 2*( (A-1) - (A+1)*cos(w0) )
a2 = (A+1) - (A-1)*cos(w0) - 2*sqrt(A)*alpha

```

FYI: The bilinear transform (with compensation for frequency warping) substitutes:

$$\text{(normalized) } s \leftarrow \frac{1}{\tan(w_0/2)} * \frac{1 - z^{-1}}{1 + z^{-1}}$$

and makes use of these trig identities:

$$\tan(w_0/2) = \frac{\sin(w_0)}{1 + \cos(w_0)} \quad (\tan(w_0/2))^2 = \frac{1 - \cos(w_0)}{1 + \cos(w_0)}$$

resulting in these substitutions:

$$1 \leftarrow \frac{1 + \cos(w_0)}{1 + \cos(w_0)} * \frac{1 + 2z^{-1} + z^{-2}}{1 + 2z^{-1} + z^{-2}}$$

$$s \leftarrow \frac{1 + \cos(w_0)}{\sin(w_0)} * \frac{1 - z^{-1}}{1 + z^{-1}} \\ = \frac{1 + \cos(w_0)}{\sin(w_0)} * \frac{1 - z^{-2}}{1 + 2z^{-1} + z^{-2}}$$

$$s^2 \leftarrow \frac{1 + \cos(w_0)}{1 - \cos(w_0)} * \frac{1 - 2z^{-1} + z^{-2}}{1 + 2z^{-1} + z^{-2}}$$

The factor:

$$\frac{1 + \cos(w_0)}{1 + 2z^{-1} + z^{-2}}$$

is common to all terms in both numerator and denominator, can be factored out, and thus be left out in the substitutions above resulting in:

$$1 \leftarrow \frac{1 + 2z^{-1} + z^{-2}}{1 + \cos(w_0)}$$

$$s \leftarrow \frac{1 - z^{-2}}{\sin(w_0)}$$

$$s^2 \leftarrow \frac{1 - 2z^{-1} + z^{-2}}{1 - \cos(w_0)}$$

In addition, all terms, numerator and denominator, can be multiplied by a common $(\sin(w_0))^2$ factor, finally resulting in these substitutions:

$$1 \leftarrow (1 + 2z^{-1} + z^{-2}) * (1 - \cos(w_0))$$

$$s \leftarrow (1 - z^{-2}) * \sin(w_0)$$

$$s^2 \leftarrow (1 - 2z^{-1} + z^{-2}) * (1 + \cos(w_0))$$

$$1 + s^2 \leftarrow 2 * (1 - 2\cos(w_0) * z^{-1} + z^{-2})$$

The biquad coefficient formulae above come out after a little simplification.

[RBJ Audio-EQ-Cookbook](#) (click this to go back to the index)

[References](#) : Posted by Robert Bristow-Johnson

[Linked file](#) : [EQ-Coefficients.pdf](#)

[Notes](#) :
see attached file

[Comments](#)
[from](#) : didid [[a t]] skynet.be
[comment](#) : Hi

In your most recent version, you write:

```
--  
alpha = sin(w0)/(2*Q)    (case: Q)  
       = sin(w0)*sinh( ln(2)/2 * BW * w0/sin(w0) )    (case: BW)  
       = sin(w0)/2 * sqrt( (A + 1/A)*(1/S - 1) + 2 )    (case: S)  
--
```

But the 'slope' case doesn't seem to work for me. It results in some kind of bad resonance at higher samplertes.

Now I found this 'beta' in an older version of your paper (I think), describing:

```
--  
beta = sqrt(A)/Q (for shelving EQ filters only)  
      = sqrt(A)*sqrt[ (A + 1/A)*(1/S - 1) + 2 ] (if shelf slope is specified)  
      = sqrt[ (A^2 + 1)/S - (A-1)^2 ]  
--
```

..and here the
 $\sqrt{A} \cdot \sqrt{(A + 1/A) \cdot (1/S - 1) + 2}$
formula works perfectly for me.

I must say I don't understand half of the theory, so it's probably my fault somewhere. But why the change in the newer version?

[from](#) : rbj [[a t]] audioimagination.com
[comment](#) : >But why the change in the newer version?

i wanted to get rid of an extraneous intermediate variable and there was enough similarity between alpha and beta that i changed the lowShelf and highShelf coefficient equations to be in terms of alpha rather than beta.

i believe if you use the new version as shown, in terms of alpha (but remember the coef equations are changed accordingly from the old version), it will come up with the same coefficients given the same boost gain, Q (or S), and shelf frequency (and same Fs). lemme know if you still have trouble.

r b-j

[from](#) : swding [[a t]] yahoo.com
[comment](#) : Hi,

Where can I find the source code?

Thanks.

[from](#) : scoofy (at) inf.elte.hu
[comment](#) : Look for "C++ class implementation of RBJ Filters" or "biquad.c" in the archive.
Peter

[from](#) : scoofy [[a t]] inf.elte.hu
[comment](#) : you have to vary the Q parameter to change the slope.

[from](#) : sundar_ranga [[a t]] yahoo.com
[comment](#) : could u plz tell me how to vary the shelf slope parameter . I want a slope of 6db/octave with second order . what Parameter i have to vary to acheive this???

[from](#) : ben_tj [[a t]] caramoule.com
[comment](#) : hi!,

it's very nice! where can I find the source code please?

[Remez Exchange Algorithm \(Parks/McClellan\)](#) (click this to go back to the index)

Type : Linear Phase FIR Filter

References : Posted by Christian at savioursofsoul dot de

Linked file : <http://www.savioursofsoul.de/Christian/Remez.zip>

Notes :

Here is an object pascal / delphi translation of the Remez Exchange Algorithm by Parks/McClellan

There is at least one small bug in it (compared to the C++ version), which causes the result to be slightly different to the C version.

Code :

<http://www.savioursofsoul.de/Christian/Remez.zip>

[Remez Remez \(Parks/McClellan\)](#) (click this to go back to the index)

Type : FIR Remez (Parks/McClellan)

References : Posted by Christian[AT]savioursofsoul[DOT]de

Notes :

Below you can find a Object Pascal / Delphi Translation of the Remez (Parks/McClellan) FIR Filter Design algorithm. It behaves slightly different from the c++ version, but the results work very well.

Code :

<http://www.savioursofsoul.de/Christian/remez.zip>

Resonant filter (click this to go back to the index)

References : Posted by Paul Kellett

Notes :

This filter consists of two first order low-pass filters in series, with some of the difference between the two filter outputs fed back to give a resonant peak.

You can use more filter stages for a steeper cutoff but the stability criteria get more complicated if the extra stages are within the feedback loop.

Code :

```
//set feedback amount given f and q between 0 and 1
fb = q + q/(1.0 - f);

//for each sample...
buf0 = buf0 + f * (in - buf0 + fb * (buf0 - buf1));
buf1 = buf1 + f * (buf0 - buf1);
out = buf1;
```

Comments

from : mr.just starting

comment : very nice! how could i turn that into a HPF?

from : dsp [[a t]] dsparsons.nospam.co.uk

comment : The cheats way is to use HPF = sample - out;

If you do a plot, you'll find that it isn't as good as designing an HPF from scratch, but it's good enuff for most ears.

This would also mean that you have a quick method for splitting a signal and operating on the (in)discreet parts separately. :) DSP

from : scoofy [[a t]] inf.elte.hu

comment : This filter calculates bandpass and highpass outputs too during calculation, namely bandpass is buf0 - buf1 and highpass is in - buf0. So, we can rewrite the algorithm:

```
// f and fb calculation
f = 2.0*sin(pi*freq/samplerate);
/* you can approximate this with f = 2.0*pi*freq/samplerate with tuning error towards nyquist */
fb = q + q/(1.0 - f);

// loop
hp = in - buf0;
bp = buf0 - buf1;
buf0 = buf0 + f * (hp + fb * bp);
buf1 = buf1 + f * (buf0 - buf1);

out = buf1; // lowpass
out = bp; // bandpass
out = hp; // highpass
```

The slope of the highpass out is not constant, it varies between 6 and 12 dB/Octave with different f and q settings. I'd be interested if anyone derived a proper highpass output from this algorithm.

-- peter schoffhauzer

Resonant IIR lowpass (12dB/oct) (click this to go back to the index)

Type : Resonant IIR lowpass (12dB/oct)

References : Posted by Olli Niemitalo

Notes :

Hard to calculate coefficients, easy to process algorithm

Code :

```
resofreq = pole frequency
amp = magnitude at pole frequency (approx)

double pi = 3.141592654;

/* Parameters. Change these! */
double resofreq = 5000;
double amp = 1.0;

DOUBLEWORD streamofs;
double w = 2.0*pi*resofreq/samplerate; // Pole angle
double q = 1.0-w/(2.0*(amp+0.5/(1.0+w))+w-2.0); // Pole magnitude
double r = q*q;
double c = r+1.0-2.0*cos(w)*q;
double vibrapos = 0;
double vibraspeed = 0;

/* Main loop */
for (streamofs = 0; streamofs < streamsize; streamofs++) {

    /* Accelerate vibra by signal-vibra, multiplied by lowpasscutoff */
    vibraspeed += (fromstream[streamofs] - vibrapos) * c;

    /* Add velocity to vibra's position */
    vibrapos += vibraspeed;

    /* Attenuate/amplify vibra's velocity by resonance */
    vibraspeed *= r;

    /* Check clipping */
    temp = vibrapos;
    if (temp > 32767) {
        temp = 32767;
    } else if (temp < -32768) temp = -32768;

    /* Store new value */
    tostream[streamofs] = temp;
}
}
```

Comments

from : raucous [[a t]] attbi.com

comment : This looks similar to the low-pass filter I used in FilterKing (<http://home.attbi.com/~spaztek4/>) Can you cruft up a high-pass example for me?

Thanks,

__e

from : faster init

comment : thank you! works nicely...

here a simplified init version for faster changes of the filter properties for amps = 1.0

```
void init( double resofreq )
{
    static const double FAC = pi * 2.0 /samplerate;
    double q, w;

    w = FAC * resofreq;
    q = 1.0f - w / ( ( 3.0 / ( 1.0+w ) ) + w - 2.0 );

    _r = q * q;
    _c = r + 1.0f - 2.0f * cos(w) * q;
}
}
```


Resonant low pass filter (click this to go back to the index)

Type : 24dB lowpass

References : Posted by "Zxform"

Linked file : [filters004.txt](#) (this linked file is included below)

Linked files

```
// ----- file filterIIR00.c begin -----
/*
Resonant low pass filter source code.
By baltrax@hotmail.com (Zxform)
*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/*****
FILTER.C - Source code for filter functions

    iir_filter          IIR filter floats sample by sample (real time)
*****/

/* FILTER INFORMATION STRUCTURE FOR FILTER ROUTINES */

typedef struct {
    unsigned int length;          /* size of filter */
    float *history;              /* pointer to history in filter */
    float *coef;                 /* pointer to coefficients of filter */
} FILTER;

#define FILTER_SECTIONS    2    /* 2 filter sections for 24 db/oct filter */

typedef struct {
    double a0, a1, a2;           /* numerator coefficients */
    double b0, b1, b2;           /* denominator coefficients */
} BIQUAD;

BIQUAD ProtoCoef[FILTER_SECTIONS]; /* Filter prototype coefficients,
                                     1 for each filter section
*/

void szxform(
    double *a0, double *a1, double *a2, /* numerator coefficients */
    double *b0, double *b1, double *b2, /* denominator coefficients */
    double fc, /* Filter cutoff frequency */
    double fs, /* sampling rate */
    double *k, /* overall gain factor */
    float *coef); /* pointer to 4 iir coefficients */

/*
* -----
*
* iir_filter - Perform IIR filtering sample by sample on floats
*
* Implements cascaded direct form II second order sections.
* Requires FILTER structure for history and coefficients.
* The length in the filter structure specifies the number of sections.
* The size of the history array is 2*iir->length.
* The size of the coefficient array is 4*iir->length + 1 because
* the first coefficient is the overall scale factor for the filter.
* Returns one output sample for each input sample. Allocates history
* array if not previously allocated.
*
* float iir_filter(float input,FILTER *iir)
*
*     float input          new float input sample
*     FILTER *iir          pointer to FILTER structure
*
* Returns float value giving the current output.
*
* Allocation errors cause an error message and a call to exit.
* -----
*/
float iir_filter(input,iir)
    float input; /* new input sample */
    FILTER *iir; /* pointer to FILTER structure */
```

```

{
    unsigned int i;
    float *hist1_ptr,*hist2_ptr,*coef_ptr;
    float output,new_hist,history1,history2;

/* allocate history array if different size than last call */

    if(!iir->history) {
        iir->history = (float *) calloc(2*iir->length,sizeof(float));
        if(!iir->history) {
            printf("\nUnable to allocate history array in iir_filter\n");
            exit(1);
        }
    }

    coef_ptr = iir->coef;                /* coefficient pointer */

    hist1_ptr = iir->history;            /* first history */
    hist2_ptr = hist1_ptr + 1;          /* next history */

    /* 1st number of coefficients array is overall input scale factor,
    * or filter gain */
    output = input * (*coef_ptr++);

    for (i = 0 ; i < iir->length; i++)
    {
        history1 = *hist1_ptr;          /* history values */
        history2 = *hist2_ptr;

        output = output - history1 * (*coef_ptr++);
        new_hist = output - history2 * (*coef_ptr++);    /* poles */

        output = new_hist + history1 * (*coef_ptr++);
        output = output + history2 * (*coef_ptr++);    /* zeros */

        *hist2_ptr++ = *hist1_ptr;
        *hist1_ptr++ = new_hist;
        hist1_ptr++;
        hist2_ptr++;
    }

    return(output);
}

/*
* -----
*
* main()
*
* Example main function to show how to update filter coefficients.
* We create a 4th order filter (24 db/oct rolloff), consisting
* of two second order sections.
* -----
*/
int main()
{
    FILTER    iir;
    float     *coef;
    double    fs, fc;    /* Sampling frequency, cutoff frequency */
    double    Q;        /* Resonance > 1.0 < 1000 */
    unsigned  nInd;
    double    a0, a1, a2, b0, b1, b2;
    double    k;        /* overall gain factor */

/*
* Setup filter s-domain coefficients
*/

        /* Section 1 */
        ProtoCoef[0].a0 = 1.0;
        ProtoCoef[0].a1 = 0;
        ProtoCoef[0].a2 = 0;
        ProtoCoef[0].b0 = 1.0;
        ProtoCoef[0].b1 = 0.765367;
        ProtoCoef[0].b2 = 1.0;

        /* Section 2 */
        ProtoCoef[1].a0 = 1.0;
        ProtoCoef[1].a1 = 0;
        ProtoCoef[1].a2 = 0;
        ProtoCoef[1].b0 = 1.0;
        ProtoCoef[1].b1 = 1.847759;
        ProtoCoef[1].b2 = 1.0;

```

```

    iir.length = FILTER_SECTIONS;          /* Number of filter sections */

/*
 * Allocate array of z-domain coefficients for each filter section
 * plus filter gain variable
 */
    iir.coef = (float *) calloc(4 * iir.length + 1, sizeof(float));
    if (!iir.coef)
    {
        printf("Unable to allocate coef array, exiting\n");
        exit(1);
    }

    k = 1.0;          /* Set overall filter gain */
    coef = iir.coef + 1; /* Skip k, or gain */

    Q = 1;           /* Resonance */
    fc = 5000;       /* Filter cutoff (Hz) */
    fs = 44100;      /* Sampling frequency (Hz) */

/*
 * Compute z-domain coefficients for each biquad section
 * for new Cutoff Frequency and Resonance
 */
    for (nInd = 0; nInd < iir.length; nInd++)
    {
        a0 = ProtoCoef[nInd].a0;
        a1 = ProtoCoef[nInd].a1;
        a2 = ProtoCoef[nInd].a2;

        b0 = ProtoCoef[nInd].b0;
        b1 = ProtoCoef[nInd].b1 / Q; /* Divide by resonance or Q
*/
        b2 = ProtoCoef[nInd].b2;
        szxform(&a0, &a1, &a2, &b0, &b1, &b2, fc, fs, &k, coef);
        coef += 4; /* Point to next filter
section */
    }

    /* Update overall filter gain in coef array */
    iir.coef[0] = k;

    /* Display filter coefficients */
    for (nInd = 0; nInd < (iir.length * 4 + 1); nInd++)
        printf("C[%d] = %15.10f\n", nInd, iir.coef[nInd]);

/*
 * To process audio samples, call function iir_filter()
 * for each audio sample
 */
    return (0);
}

// ----- file filterIIR00.c end -----
>

Reposting bilinear.c just in case the other one was not the latest version.

// ----- file bilinear.c begin -----
/*
 * -----
 * bilinear.c
 *
 * Perform bilinear transformation on s-domain coefficients
 * of 2nd order biquad section.
 * First design an analog filter and use s-domain coefficients
 * as input to szxform() to convert them to z-domain.
 *
 * Here's the butterworth polinomials for 2nd, 4th and 6th order sections.
 * When we construct a 24 db/oct filter, we take to 2nd order
 * sections and compute the coefficients separately for each section.
 *
 * n          Polinomials
 * -----
 * 2          s^2 + 1.4142s + 1
 * 4          (s^2 + 0.765367s + 1) (s^2 + 1.847759s + 1)
 * 6          (s^2 + 0.5176387s + 1) (s^2 + 1.414214 + 1) (s^2 + 1.931852s +
1)
 *
 * Where n is a filter order.

```

```

*      For n=4, or two second order sections, we have following equasions for
each
*      2nd order stage:
*
*      (1 / (s^2 + (1/Q) * 0.765367s + 1)) * (1 / (s^2 + (1/Q) * 1.847759s +
1))
*
*      Where Q is filter quality factor in the range of
*      1 to 1000. The overall filter Q is a product of all
*      2nd order stages. For example, the 6th order filter
*      (3 stages, or biquads) with individual Q of 2 will
*      have filter Q = 2 * 2 * 2 = 8.
*
*      The nominator part is just 1.
*      The denominator coefficients for stage 1 of filter are:
*      b2 = 1; b1 = 0.765367; b0 = 1;
*      numerator is
*      a2 = 0; a1 = 0; a0 = 1;
*
*      The denominator coefficients for stage 1 of filter are:
*      b2 = 1; b1 = 1.847759; b0 = 1;
*      numerator is
*      a2 = 0; a1 = 0; a0 = 1;
*
*      These coefficients are used directly by the szxform()
*      and bilinear() functions. For all stages the numerator
*      is the same and the only thing that is different between
*      different stages is 1st order coefficient. The rest of
*      coefficients are the same for any stage and equal to 1.
*
*      Any filter could be constructed using this approach.
*
*      References:
*          Van Valkenburg, "Analog Filter Design"
*          Oxford University Press 1982
*          ISBN 0-19-510734-9
*
*          C Language Algorithms for Digital Signal Processing
*          Paul Embree, Bruce Kimble
*          Prentice Hall, 1991
*          ISBN 0-13-133406-9
*
*          Digital Filter Designer's Handbook
*          With C++ Algorithms
*          Britton Rorabaugh
*          McGraw Hill, 1997
*          ISBN 0-07-053806-9
* -----
*/

#include <math.h>

void prewarp(double *a0, double *a1, double *a2, double fc, double fs);
void bilinear(
    double a0, double a1, double a2,    /* numerator coefficients */
    double b0, double b1, double b2,    /* denominator coefficients */
    double *k,                          /* overall gain factor */
    double fs,                          /* sampling rate */
    float *coef);                       /* pointer to 4 iir coefficients */

/*
* -----
*      Pre-warp the coefficients of a numerator or denominator.
*      Note that a0 is assumed to be 1, so there is no wrapping
*      of it.
* -----
*/
void prewarp(
    double *a0, double *a1, double *a2,
    double fc, double fs)
{
    double wp, pi;

    pi = 4.0 * atan(1.0);
    wp = 2.0 * fs * tan(pi * fc / fs);

    *a2 = (*a2) / (wp * wp);
    *a1 = (*a1) / wp;
}

/*

```



```

{
    /* Calculate a1 and a2 and overwrite the original values */
    prewarp(a0, a1, a2, fc, fs);
    prewarp(b0, b1, b2, fc, fs);
    bilinear(*a0, *a1, *a2, *b0, *b1, *b2, k, fs, coef);
}

```

// ----- file bilinear.c end -----

And here is how it all works.

// ----- file filter.txt begin -----
How to construct a kewl low pass resonant filter?

Lets assume we want to create a filter for analog synth.
The filter rolloff is 24 db/oct, which corresponds to 4th
order filter. Filter of first order is equivalent to RC circuit
and has max rolloff of 6 db/oct.

We will use classical Butterworth IIR filter design, as it
exactly corresponds to our requirements.

A common practice is to chain several 2nd order sections,
or biquads, as they commonly called, in order to achive a higher
order filter. Each 2nd order section is a 2nd order filter, which
has 12 db/oct rolloff. So, we need 2 of those sections in series.

To compute those sections, we use standard Butterworth polinomials,
or so called s-domain representation and convert it into z-domain,
or digital domain. The reason we need to do this is because
the filter theory exists for analog filters for a long time
and there exist no theory of working in digital domain directly.
So the common practice is to take standard analog filter design
and use so called bilinear transform to convert the butterworth
equasion coefficients into z-domain.

Once we compute the z-domain coefficients, we can use them in
a very simple transfer function, such as iir_filter() in our
C source code, in order to perform the filtering function.
The filter itself is the simpliest thing in the world.
The most complicated thing is computing the coefficients
for z-domain.

Ok, lets look at butterworth polynomials, arranged as a series
of 2nd order sections:

```

* Note: n is filter order.
*
*      n      Polynomials
*      -----
*      2      s^2 + 1.4142s + 1
*      4      (s^2 + 0.765367s + 1) * (s^2 + 1.847759s + 1)
*      6      (s^2 + 0.5176387s + 1) * (s^2 + 1.414214 + 1) * (s^2 +
1.931852s + 1)
*
* For n=4 we have following equasion for the filter transfer function:
*
*      1              1
* T(s) = ----- * -----
*      s^2 + (1/Q) * 0.765367s + 1   s^2 + (1/Q) * 1.847759s + 1
*

```

The filter consists of two 2nd order seccion since highest s power is 2.
Now we can take the coefficients, or the numbers by which s is multiplied
and plug them into a standard formula to be used by bilinear transform.

Our standard form for each 2nd order seccion is:

$$H(s) = \frac{a2 * s^2 + a1 * s + a0}{b2 * s^2 + b1 * s + b0}$$

Note that butterworth nominator is 1 for all filter sections,
which means s^2 = 0 and s^1 = 0

Lets convert standard butterworth polinomials into this form:

$$\frac{0 + 0 + 1}{1 + ((1/Q) * 0.765367) + 1} * \frac{0 + 0 + 1}{1 + ((1/Q) * 1.847759) + 1}$$

Section 1:

```
a2 = 0; a1 = 0; a0 = 1;
b2 = 1; b1 = 0.5176387; b0 = 1;
```

Section 2:

```
a2 = 0; a1 = 0; a0 = 1;
b2 = 1; b1 = 1.847759; b0 = 1;
```

That Q is filter quality factor or resonance, in the range of 1 to 1000. The overall filter Q is a product of all 2nd order stages. For example, the 6th order filter (3 stages, or biquads) with individual Q of 2 will have filter $Q = 2 * 2 * 2 = 8$.

These a and b coefficients are used directly by the `szxform()` and `bilinear()` functions.

The transfer function for z-domain is:

$$H(z) = \frac{1 + \text{alpha} * z^{(-1)} + \text{alpha}2 * z^{(-2)}}{1 + \text{beta}1 * z^{(-1)} + \text{beta}2 * z^{(-2)}}$$

When you need to change the filter frequency cutoff or resonance, or Q, you call the `szxform()` function with proper a and b coefficients and the new filter cutoff frequency or resonance. You also need to supply the sampling rate and filter gain you want to achieve. For our purposes the gain = 1.

We call `szxform()` function 2 times because we have 2 filter sections. Each call provides different coefficients.

The gain argument to `szxform()` is a pointer to desired filter gain variable.

```
double k = 1.0;      /* overall gain factor */
```

Upon return from each call, the k argument will be set to a value, by which to multiply our actual signal in order for the gain to be one. On second call to `szxform()` we provide k that was changed by the previous section. During actual audio filtering function `iir_filter()` will use this k

Summary:

Our filter is pretty close to ideal in terms of all relevant parameters and filter stability even with extremely large values of resonance. This filter design has been verified under all variations of parameters and it all appears to work as advertized.

Good luck with it.

If you ever make a DirectX wrapper for it, post it to comp.dsp.

```
*
* -----
*References:
*Van Valkenburg, "Analog Filter Design"
*Oxford University Press 1982
*ISBN 0-19-510734-9
*
*C Language Algorithms for Digital Signal Processing
*Paul Embree, Bruce Kimble
*Prentice Hall, 1991
*ISBN 0-13-133406-9
*
*Digital Filter Designer's Handbook
*With C++ Algorithms
*Britton Rorabaugh
*McGraw Hill, 1997
*ISBN 0-07-053806-9
* -----
```

Reverb Filter Generator (click this to go back to the index)

Type : FIR

References : Posted by Stephen McGovern

Notes :

This is a MATLAB function that makes a rough calculation of a room's impulse response. The output can then be convolved with an audio clip to produce good and realistic sounding reverb. I have written a paper discussing the theory used by this algorithm. It is available at <http://stevem.us/rir.html>.

NOTES:

- 1) Large values of N will use large amounts of memory.
- 2) The output is normalized to the largest value of the output.

Code :

```
function [h]=rir(fs, mic, n, r, rm, src);
%RIR Room Impulse Response.
% [h] = RIR(FS, MIC, N, R, RM, SRC) performs a room impulse
% response calculation by means of the mirror image method.
%
% FS = sample rate.
% MIC = row vector giving the x,y,z coordinates of
% the microphone.
% N = The program will account for (2*N+1)3 virtual sources
% R = reflection coefficient for the walls, in general -1<R<1.
% RM = row vector giving the dimensions of the room.
% SRC = row vector giving the x,y,z coordinates of
% the sound source.
%
% EXAMPLE:
%
% >>fs=44100;
% >>mic=[19 18 1.6];
% >>n=12;
% >>r=0.3;
% >>rm=[20 19 21];
% >>src=[5 2 1];
% >>h=rir(fs, mic, n, r, rm, src);
%
% NOTES:
%
% 1) All distances are in meters.
% 2) The output is scaled such that the largest value of the
% absolute value of the output vector is equal to one.
% 3) To implement this filter, you will need to do a fast
% convolution. The program FCONV.m will do this. It can be
% found on the Mathworks File Exchange at
% www.mathworks.com/matlabcentral/fileexchange/. It can also
% be found at www.2pi.us/code/fconv.m
% 4) A paper has been written on this model. It is available at:
% www.2pi.us/rir.html
%
%
%Version 3.4.1
%Copyright © 2003 Stephen G. McGovern
%
%Some of the following comments are references to equations the my paper.

nn=-n:1:n; % Index for the sequence
rms=nn+0.5-0.5*(-1).^nn; % Part of equations 2,3,& 4
srcs=(-1).^nn; % part of equations 2,3,& 4
xi=srcs*src(1)+rms*rm(1)-mic(1); % Equation 2
yj=srcs*src(2)+rms*rm(2)-mic(2); % Equation 3
zk=srcs*src(3)+rms*rm(3)-mic(3); % Equation 4

[i,j,k]=meshgrid(xi,yj,zk); % convert vectors to 3D matrices
d=sqrt(i.^2+j.^2+k.^2); % Equation 5
time=round(fs*d/343)+1; % Similar to Equation 6

[e,f,g]=meshgrid(nn, nn, nn); % convert vectors to 3D matrices
c=r.^(abs(e)+abs(f)+abs(g)); % Equation 9
e=c./d; % Equivalent to Equation 10

h=full(sparse(time(:),1,e(:))); % Equivalent to equation 11
h=h/max(abs(h)); % Scale output
```

Comments

from : thaddy [[a t]] thaddy.com
comment : Can someone put this is C code? I have no problems with the fft/iff parts and the convolution itself, but this gets me stuck:
// from the Fconv.m file in the article
1:1:Ym

from : ibanez82 [[a t]] katamail.com
comment : Fate una versione in C!!!!

[Simple biquad filter from apple.com](#) (click this to go back to the index)

Type : LP

References : Posted by neolit123 at gmail dot com

Notes :

Simple Biquad LP filter from the AU tutorial at apple.com

Code :

```
//cutoff_slider range 20-20000hz
//res_slider range -25/25db
//srate - sample rate

//init
mX1 = 0;
mX2 = 0;
mY1 = 0;
mY2 = 0;
pi = 22/7;

//coefficients
cutoff = cutoff_slider;
res = res_slider;

cutoff = 2 * cutoff_slider / srate;
res = pow(10, 0.05 * -res_slider);
k = 0.5 * res * sin(pi * cutoff);
c1 = 0.5 * (1 - k) / (1 + k);
c2 = (0.5 + c1) * cos(pi * cutoff);
c3 = (0.5 + c1 - c2) * 0.25;

mA0 = 2 * c3;
mA1 = 2 * 2 * c3;
mA2 = 2 * c3;
mB1 = 2 * -c2;
mB2 = 2 * c1;

//loop
output = mA0*input + mA1*mX1 + mA2*mX2 - mB1*mY1 - mB2*mY2;

mX2 = mX1;
mX1 = input;
mY2 = mY1;
mY1 = output;
```

[Spuc's open source filters](#) (click this to go back to the index)

Type : Elliptic, Butterworth, Chebyshev

References : Posted by neolit123 at gmail dot com

Notes :

<http://www.koders.com/info.aspx?c=ProjectInfo&pid=FQLFTV9LA27MF421YKXV224VWH>

Spuc has good C++ versions of some classic filter models.

Download full package from:

<http://spuc.sourceforge.net>

[State variable](#) (click this to go back to the index)

Type : 12db resonant low, high or bandpass

References : Effect Deisgn Part 1, Jon Dattorro, J. Audio Eng. Soc., Vol 45, No. 9, 1997 September

Notes :

Digital approximation of Chamberlin two-pole low pass. Easy to calculate coefficients, easy to process algorithm.

```
Code :
cutoff = cutoff freq in Hz
fs = sampling frequency //(e.g. 44100Hz)
f = 2 sin (pi * cutoff / fs) //[approximately]
q = resonance/bandwidth [0 < q <= 1] most res: q=1, less: q=0
low = lowpass output
high = highpass output
band = bandpass output
notch = notch output
```

```
scale = q
```

```
low=high=band=0;
```

```
//--beginloop
low = low + f * band;
high = scale * input - low - q*band;
band = f * high + band;
notch = high + low;
//--endloop
```

Comments

from : nope

comment : Wow, great. Sounds good, thanks.

from : no.spam [[a t]] plea.se

comment : The variable "high" doesn't have to be initialised, does it? It looks to me like the only variables that need to be kept around between iterations are "low" and "band".

from : nobody [[a t]] nowhere.com

comment : Right. High and notch are calculated from low and band every iteration.

from : kb [[a t]] kebyy.org

comment : One drawback of this is that the cutoff frequency can only go up to SR/4 instead of SR/2 - but you can easily compensate it by using 2x oversampling, eg. simply running this thing twice per sample (apply input interpolation or further output filtering ad lib, but from my experience simple linear interpolation of the input values (in and (in+lastin)/2) works well enough).

from : lala [[a t]] no.go

comment : Anyone know what the difference is between q and scale?

from : jabberdabber [[a t]] hotmail.com

comment : "most res: q=1, less: q=0"

Someone correct me if I'm wrong, but isn't that backwards? q=0 is max res, q=1 is min res.

q and scale are the same value. What the algorithm is doing is scaling the input the higher the resonance is turned up to prevent clipping. One reason why I think 0 equals max resonance and 1 equals no resonance.

So as q approaches zero, the input is attenuated more and more. In other words, as you turn up the resonance, the input is turned down.

from : does [[a t]] not.matter

comment : scale = sqrt(q);

and

```
//value (0;100) - for example
q = sqrt(1.0 - atan(sqrt(value)) * 2.0 / PI);
f = frqHz / sampleRate*4.;
```

```
ufffffff :)
```

Now enjoy!

State Variable Filter (Chamberlin version) (click this to go back to the index)

References : Hal Chamberlin, "Musical Applications of Microprocessors," 2nd Ed, Hayden Book Company 1985. pp 490-492.

```
Code :
//Input/Output
I - input sample
L - lowpass output sample
B - bandpass output sample
H - highpass output sample
N - notch output sample
F1 - Frequency control parameter
Q1 - Q control parameter
D1 - delay associated with bandpass output
D2 - delay associated with low-pass output

// parameters:
Q1 = 1/Q
// where Q1 goes from 2 to 0, ie Q goes from .5 to infinity

// simple frequency tuning with error towards nyquist
// F is the filter's center frequency, and Fs is the sampling rate
F1 = 2*pi*F/Fs

// ideal tuning:
F1 = 2 * sin( pi * F / Fs )

// algorithm
// loop
L = D2 + F1 * D1
H = I - L - Q1*D1
B = F1 * H + D1
N = H + L

// store delays
D1 = B
D2 = L

// outputs
L,H,B,N
```

Comments

from : Christian [[a t]] savioursofsoul.de

comment : Object Pascal Implementation

-denormal fixed
-not optimized

unit SVFUnit;

interface

type

TFrequencyTuningMethod= (ftmSimple, ftmIdeal);

TSVF = class

private

fQ1,fQ : Single;

fF1,fF : Single;

fFS : Single;

fD1,fD2 : Single;

fFTM : TFrequencyTuningMethod;

procedure SetFrequency(v:Single);

procedure SetQ(v:Single);

public

constructor Create;

destructor Destroy; override;

procedure Process(const I : Single; var L,B,N,H: Single);

property Frequency: Single read fF write SetFrequency;

property SampleRate: Single read fFS write fFS;

property Q: Single read fQ write SetQ;

property FrequencyTuningMethod: TFrequencyTuningMethod read fFTM write fFTM;

end;

implementation

uses sysutils;

const kDenorm = 1.0e-24;

constructor TSVF.Create;

begin

inherited;

fQ1:=1;

fF1:=1000;

```
fFS:=44100;
fFTM:=ftmIdeal;
end;
```

```
destructor TSVF.Destroy;
begin
inherited;
end;
```

```
procedure TSVF.SetFrequency(v:Single);
begin
if fFS<=0 then raise exception.create('Sample Rate Error!');
if v<>fF then
begin
fF:=v;
case fFTM of
ftmSimple:
begin
// simple frequency tuning with error towards nyquist
// F is the filter's center frequency, and Fs is the sampling rate
fF1:=2*pi*fF/fFS;
end;
ftmIdeal:
begin
// ideal tuning:
fF1:=2*sin(pi*fF/fFS);
end;
end;
end;
end;
```

```
procedure TSVF.SetQ(v:Single);
begin
if v<>fQ then
begin
if v>=0.5
then fQ:=v
else fQ:=0.5;
fQ1:=1/fQ;
end;
end;
```

```
procedure TSVF.Process(const I : Single; var L,B,N,H: Single);
begin
L:=fD2+fF1*fD1-kDenorm;
H:=I-L-fQ1*fD1;
B:=fF1*H+fD1;
N:=H+L;
// store delays
fD1:=B;
fD2:=kDenorm+L;
end;

end.
```

from : Christian [[a t]] savioursofsoul.de
comment : Ups, there are still denormal bugs in it...
(zu früh gefreut...)

State Variable Filter (Double Sampled, Stable) (click this to go back to the index)

Type : 2 Pole Low, High, Band, Notch and Peaking

References : Posted by Andrew Simper

Notes :

Thanks to Laurent de Soras for the stability limit
and Steffan Diedrichsen for the correct notch output.

Code :

```
input = input buffer;
output = output buffer;
fs = sampling frequency;
fc = cutoff frequency normally something like:
    440.0*pow(2.0, (midi_note - 69.0)/12.0);
res = resonance 0 to 1;
drive = internal distortion 0 to 0.1
freq = 2.0*sin(PI*MIN(0.25, fc/(fs*2))); // the fs*2 is because it's double sampled
damp = MIN(2.0*(1.0 - pow(res, 0.25)), MIN(2.0, 2.0/freq - freq*0.5));
notch = notch output
low = low pass output
high = high pass output
band = band pass output
peak = peaking output = low - high
--
double sampled svf loop:
for (i=0; i<numSamples; i++)
{
    in = input[i];
    notch = in - damp*band;
    low = low + freq*band;
    high = notch - low;
    band = freq*high + band - drive*band*band*band;
    out = 0.5*(notch or low or high or band or peak);
    notch = in - damp*band;
    low = low + freq*band;
    high = notch - low;
    band = freq*high + band - drive*band*band*band;
    out += 0.5*(same out as above);
    output[i] = out;
}
```

Comments

from : jm [[a t]] kampsax.dtu.dk

comment : Oh, just noticed that Eli's SVF stability measurement code has already been made available at <http://www-2.cs.cmu.edu/~eli/tmp/svf/>
However, I think it is up to him to decide whether he wants to include it in the archive or not.

from : jm [[a t]] kampsax.dtu.dk

comment : Interesting that this question pops up right now. Lately I have been wondering about the same thing, not so much about the (possibly limited) frequency range, but about stability problems of the filter that I have had (even when using smoothed control signals). The non-linearity introduced by the "drive*band*band*band" factor does not seem to be covered by the stability measurements.
In particular I would like to know, how the filter graphs in <http://vellocet.com/dsp/svf/svf-stability.html> and <http://www-2.cs.cmu.edu/~eli/tmp/svf/stability.png> were obtained? Would you like to post the code that generated the stability graph to the musicdsp archive? For the double-sampling scheme, wouldn't it make more sense to zero-stuff the input signal (that is interleave all input samples with zeros) instead of doubling the samples?

from : didid [[a t]] skynet.be

comment : Correct me if I'm wrong, but the double-sampling here looks like doubling the input, which is a bad resampling introducing aliasing, followed by an averaging of the 2 outputs, thus filtering that aliasing.
It works, but I think it (the averaging) has the side effect of smoothing up the high freqs in the source material, thus with this filter you can't really fully open it and have the original signal.
At least, it's what seems to happen practically in my tests.

Problem is that this SVF indeed has a crap stability near nyquist, but I can't think of any better way to make it work better, unless you use a better but much more costly upsampling/downsampling.

Anyone confirms?

from : williamk [[a t]] wusik.com

comment : I was having problems with this filter when DRIVE is set to MAX and Rezonance is set to MIN. A quick way to fix it was to make DRIVE*REZO, so when there's no resonance, there's no need for DRIVE anyway. That fixed the problem.

Stilson's Moog filter code (click this to go back to the index)

Type : 4-pole LP, with fruity BP/HP

References : Posted by DFL

Notes :

Mind your p's and Q's...

This code was borrowed from Tim Stilson, and rewritten by me into a pd extern (moog~) available here:
<http://www-ccrma.stanford.edu/~dfll/pd/index.htm>

I ripped out the essential code and pasted it here...

Code :

WARNING: messy code follows ;)

```
// table to fixup Q in order to remain constant for various pole frequencies, from Tim Stilson's code @ CCRMA
(also in CLM distribution)

static float gaintable[199] = { 0.999969, 0.990082, 0.980347, 0.970764, 0.961304, 0.951996, 0.94281, 0.933777,
0.924866, 0.916077, 0.90741, 0.898865, 0.89044
2, 0.882141 , 0.873962, 0.865906, 0.857941, 0.850067, 0.842346, 0.834686, 0.827148, 0.819733, 0.812378,
0.805145, 0.798004, 0.790955, 0.783997, 0.77713, 0.77
0355, 0.763672, 0.75708 , 0.75058, 0.744141, 0.737793, 0.731537, 0.725342, 0.719238, 0.713196, 0.707245,
0.701355, 0.695557, 0.689819, 0.684174, 0.678558, 0.
673035, 0.667572, 0.66217, 0.65686, 0.651581, 0.646393, 0.641235, 0.636169, 0.631134, 0.62619, 0.621277,
0.616425, 0.611633, 0.606903, 0.602234, 0.597626, 0.
593048, 0.588531, 0.584045, 0.579651, 0.575287 , 0.570953, 0.566681, 0.562469, 0.558289, 0.554169, 0.550079,
0.546051, 0.542053, 0.538116, 0.53421, 0.530334,
0.52652, 0.522736, 0.518982, 0.515289, 0.511627, 0.507996 , 0.504425, 0.500885, 0.497375, 0.493896, 0.490448,
0.487061, 0.483704, 0.480377, 0.477081, 0.4738
16, 0.470581, 0.467377, 0.464203, 0.46109, 0.457977, 0.454926, 0.451874, 0.448883, 0.445892, 0.442932,
0.440033, 0.437134, 0.434265, 0.431427, 0.428619, 0.42
5842, 0.423096, 0.42038, 0.417664, 0.415009, 0.412354, 0.409729, 0.407135, 0.404572, 0.402008, 0.399506,
0.397003, 0.394501, 0.392059, 0.389618, 0.387207, 0.
384827, 0.382477, 0.380127, 0.377808, 0.375488, 0.37323, 0.370972, 0.368713, 0.366516, 0.364319, 0.362122,
0.359985, 0.357849, 0.355713, 0.353607, 0.351532,
0.349457, 0.347412, 0.345398, 0.343384, 0.34137, 0.339417, 0.337463, 0.33551, 0.333588, 0.331665, 0.329773,
0.327911, 0.32605, 0.324188, 0.322357, 0.320557,
0.318756, 0.316986, 0.315216, 0.313446, 0.311707, 0.309998, 0.308289, 0.30658, 0.304901, 0.303223, 0.301575,
0.299927, 0.298309, 0.296692, 0.295074, 0.293488
, 0.291931, 0.290375, 0.288818, 0.287262, 0.285736, 0.284241, 0.282715, 0.28125, 0.279755, 0.27829, 0.276825,
0.275391, 0.273956, 0.272552, 0.271118, 0.26974
5, 0.268341, 0.266968, 0.265594, 0.264252, 0.262909, 0.261566, 0.260223, 0.258911, 0.257599, 0.256317,
0.255035, 0.25375 };

static inline float saturate( float input ) { //clamp without branching
#define _limit 0.95
float x1 = fabsf( input + _limit );
float x2 = fabsf( input - _limit );
return 0.5 * (x1 - x2);
}

static inline float crossfade( float amount, float a, float b ) {
return (1-amount)*a + amount*b;
}

//code for setting Q
float ix, ixfrac;
int ixint;
ix = x->p * 99;
ixint = floor( ix );
ixfrac = ix - ixint;
Q = resonance * crossfade( ixfrac, gaintable[ ixint + 99 ], gaintable[ ixint + 100 ] );

//code for setting pole coefficient based on frequency
float fc = 2 * frequency / x->srate;
float x2 = fc*fc;
float x3 = fc*x2;
p = -0.69346 * x3 - 0.59515 * x2 + 3.2937 * fc - 1.0072; //cubic fit by DFL, not 100% accurate but better
than nothing...
}

process loop:
float state[4], output; //should be global scope / preserved between calls
int i,pole;
float temp, input;

for ( i=0; i < numSamples; i++ ) {
input = *(in++);
output = 0.25 * ( input - output ); //negative feedback

for( pole = 0; pole < 4; pole++ ) {
temp = state[pole];
output = saturate( output + p * (output - temp));
state[pole] = output;
output = saturate( output + temp );
}
}
```



```
    lowpass = output;  
    highpass = input - output;  
    bandpass = 3 * x->state[2] - x->lowpass; //got this one from paul kellet  
    *out++ = lowpass;  
  
    output *= Q; //scale the feedback  
}
```

Comments

from : john [[a t]] humanoidsounds.co.uk

comment : What is "x->p" in the code for setting Q?

from : DFL

comment : you should set the frequency first, to get the value of p.
Then use that value to get the normalized Q value.

from : john[AT]humanoidsounds.co.uk

comment : Ah! That p. Thanks.

from : soeren.parton->soerenskleinewelt,de

comment : Hi!

My Output gets stuck at about 1E-7 even when the input is way below. Is that a quantisation problem? Looks as if it's the saturation's fault...

Cheers

Sören

Time domain convolution with $O(n^{\log_2(3)})$ (click this to go back to the index)

References : Wilfried Welti

Notes :

[Quoted from Wilfrieds mail...]

I found last weekend that it is possible to do convolution in time domain (no complex numbers, 100% exact result with int) with $O(n^{\log_2(3)})$ (about $O(n^{1.58})$).

Due to smaller overhead compared to FFT-based convolution, it should be the fastest algorithm for medium sized FIR's. Though, it's slower as FFT-based convolution for large n .

It's pretty easy:

Let's say we have two finite signals of length $2n$, which we want convolve : A and B . Now we split both signals into parts of size n , so we get $A = A_1 + A_2$, and $B = B_1 + B_2$.

Now we can write:

$$(1) A * B = (A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + A_2 * B_1 + A_1 * B_2 + A_2 * B_2$$

where $*$ means convolution.

This we knew already: We can split a convolution into four convolutions of halved size.

Things become interesting when we start shifting blocks in time:

Be z a signal which has the value 1 at $x=1$ and zero elsewhere. Convoluting a signal X with z is equivalent to shifting X by one rightwards. When I define z^n as n -fold convolution of z with itself, like: $z^1 = z$, $z^2 = z * z$, $z^0 = z$ shifted leftwards by 1 = impulse at $x=0$, and so on, I can use it to shift signals:

$X * z^n$ means shifting the signal X by the value n rightwards.

$X * z^{-n}$ means shifting the signal X by the value n leftwards.

Now we look at the following term:

$$(2) (A_1 + A_2 * z^{-n}) * (B_1 + B_2 * z^{-n})$$

This is a convolution of two blocks of size n : We shift A_2 by n leftwards so it completely overlaps A_1 , then we add them. We do the same thing with B_1 and B_2 . Then we convolute the two resulting blocks.

now let's transform this term:

$$(3) (A_1 + A_2 * z^{-n}) * (B_1 + B_2 * z^{-n})$$

$$\begin{aligned} &= A_1 * B_1 + A_1 * B_2 * z^{-n} + A_2 * z^{-n} * B_1 + A_2 * z^{-n} * B_2 * z^{-n} \\ &= A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) * z^{-n} + A_2 * B_2 * z^{-2n} \end{aligned}$$

$$(4) (A_1 + A_2 * z^{-n}) * (B_1 + B_2 * z^{-n}) - A_1 * B_1 - A_2 * B_2 * z^{-2n}$$

$$= (A_1 * B_2 + A_2 * B_1) * z^{-n}$$

Now we convolute both sides of the equation (4) by z^n :

$$(5) (A_1 + A_2 * z^{-n}) * (B_1 + B_2 * z^{-n}) * z^n - A_1 * B_1 * z^n - A_2 * B_2 * z^{-n}$$

$$= (A_1 * B_2 + A_2 * B_1)$$

Now we see that the right part of equation (5) appears within equation (1), so we can replace this appearance by the left part of eq (5).

$$(6) A * B = (A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + A_2 * B_1 + A_1 * B_2 + A_2 * B_2$$

$$\begin{aligned} &= A_1 * B_1 \\ &+ (A_1 + A_2 * z^{-n}) * (B_1 + B_2 * z^{-n}) * z^n - A_1 * B_1 * z^n - A_2 * B_2 * z^{-n} \\ &+ A_2 * B_2 \end{aligned}$$

Voila!

We have constructed the convolution of $A * B$ with only three convolutions of halved size. (Since the convolutions with z^n and z^{-n} are only shifts of blocks with size n , they of course need only n operations for processing :)

This can be used to construct an easy recursive algorithm of Order $O(n^{\log_2(3)})$

Code :

```
void convolution(value* in1, value* in2, value* out, value* buffer, int size)
{
    value* temp1 = buffer;
    value* temp2 = buffer + size/2;
    int i;

    // clear output.
    for (i=0; i<size*2; i++) out[i] = 0;

    // Break condition for recursion: 1x1 convolution is multiplication.
```

```

if (size == 1)
{
    out[0] = in1[0] * in2[0];
    return;
}

// first calculate (A1 + A2 * z^-n)*(B1 + B2 * z^-n)*z^n

signal_add(in1, in1+size/2, temp1, size/2);
signal_add(in2, in2+size/2, temp2, size/2);
convolution(temp1, temp2, out+size/2, buffer+size, size/2);

// then add A1*B1 and subtract A1*B1*z^n

convolution(in1, in2, temp1, buffer+size, size/2);
signal_add_to(out, temp1, size);
signal_sub_from(out+size/2, temp1, size);

// then add A2*B2 and subtract A2*B2*z^-n

convolution(in1+size/2, in2+size/2, temp1, buffer+size, size/2);
signal_add_to(out+size, temp1, size);
signal_sub_from(out+size/2, temp1, size);
}

```

"value" may be a suitable type like int or float.

Parameter "size" is the size of the input signals and must be a power of 2. out and buffer must point to arrays of size 2*n.

Just to be complete, the helper functions:

```

void signal_add(value* in1, value* in2, value* out, int size)
{
    int i;
    for (i=0; i<size; i++) out[i] = in1[i] + in2[i];
}

void signal_sub_from(value* out, value* in, int size)
{
    int i;
    for (i=0; i<size; i++) out[i] -= in[i];
}

void signal_add_to(value* out, value* in, int size)
{
    int i;
    for (i=0; i<size; i++) out[i] += in[i];
}

```

Comments

from : Christian [[a t]] savioursofsoul.de

comment : Here is a delphi translation of the code:

// "value" may be a suitable type like int or float.

// Parameter "size" is the size of the input signals and must be a power of 2.

// out and buffer must point to arrays of size 2*n.

```

procedure signal_add(in1, in2, ou1 :PValue; Size:Integer);
var i      : Integer;
begin
    for i:=0 to Size-1 do
        begin
            ou1^[i] := in1^[i] + in2^[i];
        end;
    end;
end;

```

```

procedure signal_sub_from(in1, ou1 :PValue; Size:Integer);
var i      : Integer;
begin
    for i:=0 to Size-1 do
        begin
            ou1^[i] := ou1^[i] - in1^[i];
        end;
    end;
end;

```

```

procedure signal_add_to(in1, ou1: PValue; Size:Integer);
var i      : Integer;
    po, pi1 : PValue;
begin
    po:=ou1;
    pi1:=in1;
    for i:=0 to Size-1 do
        begin
            ou1^[i] := ou1^[i] + in1^[i];
            Inc(po);
            Inc(pi1);
        end;
    end;
end;

```

```

procedure convolution(in1, in2, ou1, buffer :PValue; Size:Integer);
var tmp1, tmp2 : PValue;
    i      : Integer;
begin
tmp1:=Buffer;
tmp2:=@(Buffer^(Size div 2));

// clear output.
for i:=0 to size*2 do ou1^[i]:=0;

// Break condition for recursion: 1x1 convolution is multiplication.
if Size = 1 then
begin
ou1^[0] := in1^[0] * in2^[0];
exit;
end;

// first calculate (A1 + A2 * z^-n)*(B1 + B2 * z^-n)*z^n
signal_add(in1, @(in1^(Size div 2)), tmp1, Size div 2);
signal_add(in2, @(in1^(Size div 2)), tmp2, Size div 2);
convolution(tmp1, tmp2, @(ou1^(Size div 2)), @(Buffer^[Size]), Size div 2);

// then add A1*B1 and subtract A1*B1*z^n
convolution(in1, in2, tmp1, @(Buffer^[Size]), Size div 2);
signal_add_to(ou1, tmp1, size);
signal_sub_from(@(ou1^(Size div 2)), tmp1, size);

// then add A2*B2 and subtract A2*B2*z^-n
convolution(@(in1^(Size div 2)), @(in2^(Size div 2)), tmp1, @(Buffer^[Size]), Size div 2);
signal_add_to(@(ou1^[Size]), tmp1, size);
signal_sub_from(@(ou1^[Size]), tmp1, size);
end;

```

from : Christian [[a t]] savioursofsoul.de
comment : Sorry, i forgot the definitions:

```

type
Values = Array[0..0] of Single;
PValue = ^Values;

```

from : Christian [[a t]] savioursofsoul.de
comment : I have implemented a Surround-Plugin using this Source-Code.
Basically a FIR-Filter with 512 Taps, bundled with some HRTF's for sourround panning

<http://www.savioursofsoul.de/Christian/ITA-HRTF.EXE>

(Delphi Sourcecode available on request)

from : b0nk [[a t]] free.fr
comment : I think "buffer" is a temporary buffer for the algorithm. Am I right ?

Time domain convolution with $O(n^2 \log_2(3))$ (click this to go back to the index)

References : Posted by Magnus Jonsson

Notes :

[see other code by Wilfried Welti too!]

Code :

```
void mul_brute(float *r, float *a, float *b, int w)
{
    for (int i = 0; i < w+w; i++)
        r[i] = 0;
    for (int i = 0; i < w; i++)
    {
        float *rr = r+i;
        float ai = a[i];
        for (int j = 0; j < w; j++)
            rr[j] += ai*b[j];
    }
}

// tmp must be of length 2*w
void mul_knuth(float *r, float *a, float *b, int w, float *tmp)
{
    if (w < 30)
    {
        mul_brute(r, a, b, w);
    }
    else
    {
        int m = w>>1;

        for (int i = 0; i < m; i++)
        {
            r[i ] = a[m+i]-a[i ];
            r[i+m] = b[i ]-b[m+i];
        }

        mul_knuth(tmp, r , r+m, m, tmp+w);
        mul_knuth(r , a , b , m, tmp+w);
        mul_knuth(r+w, a+m, b+m, m, tmp+w);

        for (int i = 0; i < m; i++)
        {
            float bla = r[m+i]+r[w+i];
            r[m+i] = bla+r[i ]+tmp[i ];
            r[w+i] = bla+r[w+m+i]+tmp[i+m];
        }
    }
}
```

[Type : LPF 24dB/Oct](#) (click this to go back to the index)

[Type](#) : Bessel Lowpass

[References](#) : Posted by Christian[AT]savioursofsoul[DOT]de

Notes :

The filter tends to be unstable for low frequencies in the way, that it seems to flutter, but it never explodes. At least here it doesn't.

Code :

First calculate the prewarped digital frequency:

```
K = tan(Pi * Frequency / Samplerate);
K2 = K*K; // speed improvement
```

Then calc the digital filter coefficients:

```
A0 = (((((105*K + 105)*K + 45)*K + 10)*K + 1);
A1 = -(((420*K + 210)*K2 - 20)*K - 4)*t;
A2 = -((630*K2 - 90)*K2 + 6)*t;
A3 = -(((420*K - 210)*K2 + 20)*K - 4)*t;
A4 = -((((105*K - 105)*K + 45)*K - 10)*K + 1)*t;
```

```
B0 = 105*K2*K2;
B1 = 420*K2*K2;
B2 = 630*K2*K2;
B3 = 420*K2*K2;
B4 = 105*K2*K2;
```

Per sample calculate:

```
Output = B0*Input + State0;
State0 = B1*Input + A1/A0*Output + State1;
State1 = B2*Input + A2/A0*Output + State2;
State2 = B3*Input + A3/A0*Output + State3;
State3 = B4*Input + A4/A0*Output;
```

For high speed substitute A1/A0 with A1' = A1/A0...

Comments

[from](#) : Christian [[a t]] savioursofsoul.de

[comment](#) : Just found out, that I forgot to remove the temporary variable 't'. It was used in my code for the speedup. You can simply ignore and delete it.

[from](#) : Christian [[a t]] savioursofsoul.de

[comment](#) : It turns out, that the filter is only unstable if the coefficient/state precision isn't high enough. Using double instead of single precision already makes it a lot more stable.

Various Biquad filters (click this to go back to the index)

References : JAES, Vol. 31, No. 11, 1983 November

Linked file : [filters003.txt](#) (this linked file is included below)

Notes :

(see linkfile)

Filters included are:

presence

shelvelowpass

2polebp

peaknotch

peaknotch2

Comments

from : neolit123 [[a t]] gmail.com

comment : Managed to port the presence eq properly. And its sounds great!

Altho I did some changes to some of the code.

changed "d /= mag" to "d = mag"

"bw/srate" to "bw"

There results I got are stable within there parameters:

freq: 3100-18500hz

boost: 0-15db

bw: 0.07-0.40

Really good sound from this filter!

from : neolit123 [[a t]] gmail.com

comment : I'm kinda stuck trying to figure out the 'pointer' 'structure pointer' loop in the presence EQ.

Can someone explain:

...

```
*a0 = a2plus1 + alphan*ma2plus1;
```

```
*a1 = 4.0*a;
```

```
*a2 = a2plus1 - alphan*ma2plus1;
```

```
b0 = a2plus1 + alphad*ma2plus1;
```

```
*b2 = a2plus1 - alphad*ma2plus1;
```

```
recipb0 = 1.0/b0;
```

```
*a0 *= recipb0;
```

```
*a1 *= recipb0;
```

```
*a2 *= recipb0;
```

```
*b1 = *a1;
```

```
*b2 *= recipb0;
```

....

```
void setfilter_presence(f,freq,boost,bw)
```

```
filter *f;
```

```
double freq,boost,bw;
```

```
{
  presence(freq/(double)SR,boost,bw/(double)SR,
    &f->cx,&f->cx1,&f->cx2,&f->cy1,&f->cy2);
  f->cy1 = -f->cy1;
  f->cy2 = -f->cy2;
}
```

How can this be translated into something more easy to understand.

Input = ...

Output = ...

Linked files

```
/*
 * Presence and Shelf filters as given in
 * James A. Moorer
 * The manifold joys of conformal mapping:
 * applications to digital filtering in the studio
 * JAES, Vol. 31, No. 11, 1983 November
 */
```

```
#define SPN MINDOUBLE
```

```

double bw2angle(a,bw)
double a,bw;
{
    double T,d,sn,cs,mag,delta,theta,tmp,a2,a4,asnd;

    T = tan(2.0*PI*bw);
    a2 = a*a;
    a4 = a2*a2;
    d = 2.0*a2*T;
    sn = (1.0 + a4)*T;
    cs = (1.0 - a4);
    mag = sqrt(sn*sn + cs*cs);
    d /= mag;
    delta = atan2(sn,cs);
    asnd = asin(d);
    theta = 0.5*(PI - asnd - delta);
    tmp = 0.5*(asnd-delta);
    if ((tmp > 0.0) && (tmp < theta)) theta = tmp;
    return(theta/(2.0*PI));
}

void presence(cf,boost,bw,a0,a1,a2,b1,b2)
double cf,boost,bw,*a0,*a1,*a2,*b1,*b2;
{
    double a,A,F,xfmbw,C,tmp,alphan,alphad,b0,recipb0,asq,F2,a2plus1,ma2plus1;

    a = tan(PI*(cf-0.25));
    asq = a*a;
    A = pow(10.0,boost/20.0);
    if ((boost < 6.0) && (boost > -6.0)) F = sqrt(A);
    else if (A > 1.0) F = A/sqrt(2.0);
    else F = A*sqrt(2.0);
    xfmbw = bw2angle(a,bw);

    C = 1.0/tan(2.0*PI*xfmbw);
    F2 = F*F;
    tmp = A*A - F2;
    if (fabs(tmp) <= SPN) alphad = C;
    else alphad = sqrt(C*C*(F2-1.0)/tmp);
    alphan = A*alphad;

    a2plus1 = 1.0 + asq;
    ma2plus1 = 1.0 - asq;
    *a0 = a2plus1 + alphan*ma2plus1;
    *a1 = 4.0*a;
    *a2 = a2plus1 - alphan*ma2plus1;

    b0 = a2plus1 + alphad*ma2plus1;
    *b2 = a2plus1 - alphad*ma2plus1;

    recipb0 = 1.0/b0;
    *a0 *= recipb0;
    *a1 *= recipb0;
    *a2 *= recipb0;
    *b1 = *a1;
    *b2 *= recipb0;
}

void shelve(cf,boost,a0,a1,a2,b1,b2)
double cf,boost,*a0,*a1,*a2,*b1,*b2;
{
    double a,A,F,tmp,b0,recipb0,asq,F2,gamma2,siggam2,gam2p1;
    double gamman,gammad,ta0,ta1,ta2,tb0,tb1,tb2,aal,abl;

    a = tan(PI*(cf-0.25));
    asq = a*a;
    A = pow(10.0,boost/20.0);
    if ((boost < 6.0) && (boost > -6.0)) F = sqrt(A);
    else if (A > 1.0) F = A/sqrt(2.0);
    else F = A*sqrt(2.0);

    F2 = F*F;
    tmp = A*A - F2;
    if (fabs(tmp) <= SPN) gammad = 1.0;
    else gammad = pow((F2-1.0)/tmp,0.25);
    gamman = sqrt(A)*gammad;

    gamma2 = gamman*gamman;
    gam2p1 = 1.0 + gamma2;
    siggam2 = 2.0*sqrt(2.0)/2.0*gamman;
    ta0 = gam2p1 + siggam2;
    ta1 = -2.0*(1.0 - gamma2);
}

```



```

ta2 = gam2p1 - siggam2;

gamma2 = gammad*gammad;
gam2p1 = 1.0 + gamma2;
siggam2 = 2.0*sqrt(2.0)/2.0*gammad;
tb0 = gam2p1 + siggam2;
tb1 = -2.0*(1.0 - gamma2);
tb2 = gam2p1 - siggam2;

a1 = a*ta1;
*a0 = ta0 + a1 + asq*ta2;
*a1 = 2.0*a*(ta0+ta2)+(1.0+asq)*ta1;
*a2 = asq*ta0 + a1 + ta2;

b1 = a*tb1;
b0 = tb0 + b1 + asq*tb2;
*b1 = 2.0*a*(tb0+tb2)+(1.0+asq)*tb1;
*b2 = asq*tb0 + b1 + tb2;

recipb0 = 1.0/b0;
*a0 *= recipb0;
*a1 *= recipb0;
*a2 *= recipb0;
*b1 *= recipb0;
*b2 *= recipb0;
}

void initfilter(f)
filter *f;
{
    f->x1 = 0.0;
    f->x2 = 0.0;
    f->y1 = 0.0;
    f->y2 = 0.0;
    f->y = 0.0;
}

void setfilter_presence(f,freq,boost,bw)
filter *f;
double freq,boost,bw;
{
    presence(freq/(double)SR,boost,bw/(double)SR,
             &f->cx,&f->cx1,&f->cx2,&f->cy1,&f->cy2);
    f->cy1 = -f->cy1;
    f->cy2 = -f->cy2;
}

void setfilter_shelve(f,freq,boost)
filter *f;
double freq,boost;
{
    shelve(freq/(double)SR,boost,
           &f->cx,&f->cx1,&f->cx2,&f->cy1,&f->cy2);
    f->cy1 = -f->cy1;
    f->cy2 = -f->cy2;
}

void setfilter_shelvelowpass(f,freq,boost)
filter *f;
double freq,boost;
{
    double gain;

    gain = pow(10.0,boost/20.0);
    shelve(freq/(double)SR,boost,
           &f->cx,&f->cx1,&f->cx2,&f->cy1,&f->cy2);
    f->cx /= gain;
    f->cx1 /= gain;
    f->cx2 /= gain;
    f->cy1 = -f->cy1;
    f->cy2 = -f->cy2;
}

/*
 * As in 'An introduction to digital filter theory' by Julius O. Smith
 * and in Moore's book; I use the normalized version in Moore's book.
 */
void setfilter_2polebp(f,freq,R)
filter *f;
double freq,R;
{
    double theta;

```

```

theta = 2.0*PI*freq/(double)SR;
f->cx = 1.0-R;
f->cx1 = 0.0;
f->cx2 = -(1.0-R)*R;
f->cy1 = 2.0*R*cos(theta);
f->cy2 = -R*R;
}

/*
 * As in
 * Stanley A. White
 * Design of a digital biquadratic peaking or notch filter
 * for digital audio equalization
 * JAES, Vol. 34, No. 6, 1986 June
 */
void setfilter_peaknotch(f,freq,M,bw)
filter *f;
double freq,M,bw;
{
    double w0,p,om,ta,d;

    w0 = 2.0*PI*freq;
    if ((1.0/sqrt(2.0) < M) && (M < sqrt(2.0))) {
        fprintf(stderr,"peaknotch filter: 1/sqrt(2) < M < sqrt(2)\n");
        exit(-1);
    }
    if (M <= 1.0/sqrt(2.0)) p = sqrt(1.0-2.0*M*M);
    if (sqrt(2.0) <= M) p = sqrt(M*M-2.0);
    om = 2.0*PI*bw;
    ta = tan(om/((double)SR*2.0));
    d = p+ta;
    f->cx = (p+M*ta)/d;
    f->cx1 = -2.0*p*cos(w0/(double)SR)/d;
    f->cx2 = (p-M*ta)/d;
    f->cy1 = 2.0*p*cos(w0/(double)SR)/d;
    f->cy2 = -(p-ta)/d;
}

/*
 * Some JAES's article on ladder filter.
 * freq (Hz), gdb (dB), bw (Hz)
 */
void setfilter_peaknotch2(f,freq,gdb,bw)
filter *f;
double freq,gdb,bw;
{
    double k,w,bwr,abw,gain;

    k = pow(10.0,gdb/20.0);
    w = 2.0*PI*freq/(double)SR;
    bwr = 2.0*PI*bw/(double)SR;
    abw = (1.0-tan(bwr/2.0))/(1.0+tan(bwr/2.0));
    gain = 0.5*(1.0+k+abw-k*abw);
    f->cx = 1.0*gain;
    f->cx1 = gain*(-2.0*cos(w)*(1.0+abw))/(1.0+k+abw-k*abw);
    f->cx2 = gain*(abw+k*abw+1.0-k)/(abw-k*abw+1.0+k);
    f->cy1 = 2.0*cos(w)/(1.0+tan(bwr/2.0));
    f->cy2 = -abw;
}

double applyfilter(f,x)
filter *f;
double x;
{
    f->x = x;
    f->y = f->cx * f->x + f->cx1 * f->x1 + f->cx2 * f->x2
        + f->cy1 * f->y1 + f->cy2 * f->y2;
    f->x2 = f->x1;
    f->x1 = f->x;
    f->y2 = f->y1;
    f->y1 = f->y;
    return(f->y);
}

```

Windowed Sinc FIR Generator (click this to go back to the index)

Type : LP, HP, BP, BS

References : Posted by Bob Maling

Linked file : [wsfir.h](#) (this linked file is included below)

Notes :

This code generates FIR coefficients for lowpass, highpass, bandpass, and bandstop filters by windowing a sinc function.

The purpose of this code is to show how windowed sinc filter coefficients are generated. Also shown is how highpass, bandpass, and bandstop filters can be made from lowpass filters.

Included windows are Blackman, Hanning, and Hamming. Other windows can be added by following the structure outlined in the opening comments of the header file.

Comments

from : Christian [[a t]] savioursofsoul.de

comment : // Object Pascal Port...

unit SincFIR;

(* Windowed Sinc FIR Generator
Bob Maling (BobM.DSP@gmail.com)
Contributed to musicdsp.org Source Code Archive
Last Updated: April 12, 2005
Translated to Object Pascal by Christian-W. Budde

Usage:

Lowpass:wsfirLP(H, WindowType, CutOff)
Highpass:wsfirHP(H, WindowType, CutOff)
Bandpass:wsfirBP(H, WindowType, LowCutOff, HighCutOff)
Bandstop:wsfirBS(H, WindowType, LowCutOff, HighCutOff)

where:

H (TDoubleArray): empty filter coefficient table (SetLength(H,DesiredLength!))
WindowType (TWindowType): wtBlackman, wtHanning, wtHamming
CutOff (double): cutoff ($0 < \text{CutOff} < 0.5$, $\text{CutOff} = f/\text{fs}$)
--> for fs=48kHz and cutoff f=12kHz, $\text{CutOff} = 12\text{k}/48\text{k} \Rightarrow 0.25$

LowCutOff (double):low cutoff ($0 < \text{CutOff} < 0.5$, $\text{CutOff} = f/\text{fs}$)
HighCutOff (double):high cutoff ($0 < \text{CutOff} < 0.5$, $\text{CutOff} = f/\text{fs}$)

Windows included here are Blackman, Blackman-Harris, Gaussian, Hanning and Hamming.*)

interface

uses Math;

type TDoubleArray = array of Double;
TWindowType = (wtBlackman, wtHanning, wtHamming, wtBlackmanHarris, wtGaussian); // Window type constants

// Function prototypes

```
procedure wsfirLP(var H : TDoubleArray; const WindowType : TWindowType; const CutOff : Double);
procedure wsfirHP(var H : TDoubleArray; const WindowType : TWindowType; const CutOff : Double);
procedure wsfirBS(var H : TDoubleArray; const WindowType : TWindowType; const LowCutOff, HighCutOff : Double);
procedure wsfirBP(var H : TDoubleArray; const WindowType : TWindowType; const LowCutOff, HighCutOff : Double);
procedure genSinc(var Sinc : TDoubleArray; const CutOff : Double);
procedure wGaussian(var W : TDoubleArray);
procedure wBlackmanHarris(var W : TDoubleArray);
procedure wBlackman(var W : TDoubleArray);
procedure wHanning(var W : TDoubleArray);
procedure wHamming(var W : TDoubleArray);
```

implementation

```
// Generate lowpass filter
// This is done by generating a sinc function and then windowing it
procedure wsfirLP(var H : TDoubleArray; const WindowType : TWindowType; const CutOff : Double);
begin
  genSinc(H, CutOff); // 1. Generate Sinc function
  case WindowType of // 2. Generate Window function -> lowpass filter!
    wtBlackman: wBlackman(H);
    wtHanning: wHanning(H);
    wtHamming: wHamming(H);
    wtGaussian: wGaussian(H);
    wtBlackmanHarris: wBlackmanHarris(H);
  end;
end;
```

```
// Generate highpass filter
```

```

// This is done by generating a lowpass filter and then spectrally inverting it
procedure wsfirHP(var H : TDoubleArray; const WindowType : TWindowType; const CutOff : Double);
var i : Integer;
begin
wsfirLP(H, WindowType, CutOff); // 1. Generate lowpass filter

// 2. Spectrally invert (negate all samples and add 1 to center sample) lowpass filter
// = delta[n-((N-1)/2)] - h[n], in the time domain
for i:=0 to Length(H)-1
do H[i]:=H[i]*-1.0;
H[(Length(H)-1) div 2]:=H[(Length(H)-1) div 2]+1.0;
end;

// Generate bandstop filter
// This is done by generating a lowpass and highpass filter and adding them
procedure wsfirBS(var H : TDoubleArray; const WindowType : TWindowType; const LowCutOff, HighCutOff : Double);
var i : Integer;
    H2 : TDoubleArray;
begin
SetLength(H2,Length(H));

// 1. Generate lowpass filter at first (low) cutoff frequency
wsfirLP(H, WindowType, LowCutOff);

// 2. Generate highpass filter at second (high) cutoff frequency
wsfirHP(H2, WindowType, HighCutOff);

// 3. Add the 2 filters together
for i:=0 to Length(H)-1
do H[i]:=H[i]+H2[i];

SetLength(H2,0);
end;

// Generate bandpass filter
// This is done by generating a bandstop filter and spectrally inverting it
procedure wsfirBP(var H : TDoubleArray; const WindowType : TWindowType; const LowCutOff, HighCutOff : Double);
var i : Integer;
begin
wsfirBS(H, WindowType, LowCutOff, HighCutOff); // 1. Generate bandstop filter

// 2. Spectrally invert (negate all samples and add 1 to center sample) lowpass filter
// = delta[n-((N-1)/2)] - h[n], in the time domain
for i:=0 to Length(H)-1
do H[i]:=H[i]*-1.0;
H[(Length(H)-1) div 2]:=H[(Length(H)-1) div 2]+1.0;
end;

// Generate sinc function - used for making lowpass filter
procedure genSinc(var Sinc : TDoubleArray; const Cutoff : Double);
var i,j : Integer;
    n,k : Double;
begin
j:=Length(Sinc)-1;
k:=1/j;
// Generate sinc delayed by (N-1)/2
for i:=0 to j do
if (i=j div 2)
then Sinc[i]:=2.0*Cutoff
else
begin
n:=i-j/2.0;
Sinc[i]:=sin(2.0*PI*Cutoff*n)/(PI*n);
end;
end;

// Generate window function (Gaussian)
procedure wGaussian(var W : TDoubleArray);
var i,j : Integer;
    k : Double;
begin
j:=Length(W)-1;
k:=1/j;
for i:=0 to j
do W[i]:=W[i]*(exp(-5.0/(sqr(j))*(2*i-j)*(2*i-j)));
end;

// Generate window function (Blackman-Harris)
procedure wBlackmanHarris(var W : TDoubleArray);
var i,j : Integer;
    k : Double;
begin
j:=Length(W)-1;
k:=1/j;
for i:=0 to j

```

```
do W[i]:=W[i]*(0.35875-0.48829*cos(2*PI*(i+0.5)*k)+0.14128*cos(4*PI*(i+0.5)*k)-0.01168*cos(6*PI*(i+0.5)*k));
end;
```

```
// Generate window function (Blackman)
```

```
procedure wBlackman(var W : TDoubleArray);
var i,j : Integer;
    k : Double;
begin
j:=Length(W)-1;
k:=1/j;
for i:=0 to j
do W[i]:=W[i]*(0.42-(0.5*cos(2*PI*i*k))+(0.08*cos(4*PI*i*k)));
end;
```

```
// Generate window function (Hanning)
```

```
procedure wHanning(var W : TDoubleArray);
var i,j : Integer;
    k : Double;
begin
j:=Length(W)-1;
k:=1/j;
for i:=0 to j
do W[i]:=W[i]*(0.5*(1.0-cos(2*PI*i*k)));
end;
```

```
// Generate window function (Hamming)
```

```
procedure wHamming(var W : TDoubleArray);
var i,j : Integer;
    k : Double;
begin
j:=Length(W)-1;
k:=1/j;
for i:=0 to j
do W[i]:=W[i]*(0.54-(0.46*cos(2*PI*i*k)));
end;
```

```
end.
```

from : scoofy [[a t]] inf.elte.hu

comment : The Hanning window is often incorrectly referred to as 'Hanning', since it was named after a guy called Julius von Hann. So it's more appropriate to call it 'Hann' window.

from : rousicoello [[a t]] gmail.com

comment : What do I have to do to apply this windowed sinc filter to a signal "Y" for example... what is the code for this?
Imagine signal "Y" is a stereo song which means that I have Y1 from channel 1 and Y2 from channel 2.. help me please as soon as possible because at least i want to know how to apply the filter to any signal...

from : Dave in sinc land

comment : I've seen a BASIC version of the same genSinc code.
Shouldn't a 'sin(2.0*Cutoff)' be used when the divide by zero check is 0?

```
...
if (i=j div 2)
then Sinc[i]:=2.0*Cutoff
else
begin
n:=i-j/2.0;
Sinc[i]:=sin(2.0*PI*Cutoff*n)/(PI*n);
end;
```

from : Dave in sinc land

comment : Scrap that, I've just FFT'd the response, and it appears to be correct as it was. Hey it just looked wrong o.k. ;)

Linked files

```
/*
Windowed Sinc FIR Generator
Bob Maling (BobM.DSP@gmail.com)
Contributed to musicdsp.org Source Code Archive
Last Updated: April 12, 2005
```

Usage:

```
Lowpass: wsfirLP(h, N, WINDOW, fc)
Highpass: wsfirHP(h, N, WINDOW, fc)
Bandpass: wsfirBP(h, N, WINDOW, fc1, fc2)
Bandstop: wsfirBS(h, N, WINDOW, fc1, fc2)
```

where:

```
h (double[N]): filter coefficients will be written to this array
N (int): number of taps
WINDOW (int): Window (W_BLACKMAN, W_HANNING, or W_HAMMING)
fc (double): cutoff (0 < fc < 0.5, fc = f/fs)
--> for fs=48kHz and cutoff f=12kHz, fc = 12k/48k => 0.25
```

```
fc1 (double): low cutoff ( $0 < f_c < 0.5$ ,  $f_c = f/f_s$ )
fc2 (double): high cutoff ( $0 < f_c < 0.5$ ,  $f_c = f/f_s$ )
```

Windows included here are Blackman, Hanning, and Hamming. Other windows can be added by doing the following:

1. "Window type constants" section: Define a global constant for the new window.
2. `wsfirLP()` function: Add the new window as a case in the `switch()` statement.
3. Create the function for the window

For windows with a design parameter, such as Kaiser, some modification will be needed for each function in order to pass the parameter.

```
*/
#ifdef WSFIR_H
#define WSFIR_H

#include <math.h>

// Function prototypes
void wsfirLP(double h[], const int &N, const int &WINDOW, const double &fc);
void wsfirHP(double h[], const int &N, const int &WINDOW, const double &fc);
void wsfirBS(double h[], const int &N, const int &WINDOW, const double &fc1, const double &fc2);
void wsfirBP(double h[], const int &N, const int &WINDOW, const double &fc1, const double &fc2);
void genSinc(double sinc[], const int &N, const double &fc);
void wBlackman(double w[], const int &N);
void wHanning(double w[], const int &N);
void wHamming(double w[], const int &N);

// Window type constants
const int W_BLACKMAN = 1;
const int W_HANNING = 2;
const int W_HAMMING = 3;

// Generate lowpass filter
//
// This is done by generating a sinc function and then windowing it
void wsfirLP(double h[], // h[] will be written with the filter coefficients
             const int &N, // size of the filter (number of taps)
             const int &WINDOW, // window function (W_BLACKMAN, W_HANNING, etc.)
             const double &fc) // cutoff frequency
{
    int i;
    double *w = new double[N]; // window function
    double *sinc = new double[N]; // sinc function

    // 1. Generate Sinc function
    genSinc(sinc, N, fc);

    // 2. Generate Window function
    switch (WINDOW) {
        case W_BLACKMAN: // W_BLACKMAN
            wBlackman(w, N);
            break;
        case W_HANNING: // W_HANNING
            wHanning(w, N);
            break;
        case W_HAMMING: // W_HAMMING
            wHamming(w, N);
            break;
        default:
            break;
    }

    // 3. Make lowpass filter
    for (i = 0; i < N; i++) {
        h[i] = sinc[i] * w[i];
    }

    // Delete dynamic storage
    delete []w;
    delete []sinc;

    return;
}

// Generate highpass filter
//
// This is done by generating a lowpass filter and then spectrally inverting it
void wsfirHP(double h[], // h[] will be written with the filter coefficients
             const int &N, // size of the filter
             const int &WINDOW, // window function (W_BLACKMAN, W_HANNING, etc.)
             const double &fc) // cutoff frequency
```

```

{
    int i;

    // 1. Generate lowpass filter
    wsfirLP(h, N, WINDOW, fc);

    // 2. Spectrally invert (negate all samples and add 1 to center sample) lowpass filter
    // = delta[n-((N-1)/2)] - h[n], in the time domain
    for (i = 0; i < N; i++) {
        h[i] *= -1.0; // = 0 - h[i]
    }
    h[(N-1)/2] += 1.0; // = 1 - h[(N-1)/2]

    return;
}

// Generate bandstop filter
//
// This is done by generating a lowpass and highpass filter and adding them
void wsfirBS(double h[], // h[] will be written with the filter taps
             const int &N, // size of the filter
             const int &WINDOW, // window function (W_BLACKMAN, W_HANNING, etc.)
             const double &fc1, // low cutoff frequency
             const double &fc2) // high cutoff frequency
{
    int i;
    double *h1 = new double[N];
    double *h2 = new double[N];

    // 1. Generate lowpass filter at first (low) cutoff frequency
    wsfirLP(h1, N, WINDOW, fc1);

    // 2. Generate highpass filter at second (high) cutoff frequency
    wsfirHP(h2, N, WINDOW, fc2);

    // 3. Add the 2 filters together
    for (i = 0; i < N; i++) {
        h[i] = h1[i] + h2[i];
    }

    // Delete dynamic memory
    delete []h1;
    delete []h2;

    return;
}

// Generate bandpass filter
//
// This is done by generating a bandstop filter and spectrally inverting it
void wsfirBP(double h[], // h[] will be written with the filter taps
             const int &N, // size of the filter
             const int &WINDOW, // window function (W_BLACKMAN, W_HANNING, etc.)
             const double &fc1, // low cutoff frequency
             const double &fc2) // high cutoff frequency
{
    int i;

    // 1. Generate bandstop filter
    wsfirBS(h, N, WINDOW, fc1, fc2);

    // 2. Spectrally invert bandstop (negate all, and add 1 to center sample)
    for (i = 0; i < N; i++) {
        h[i] *= -1.0; // = 0 - h[i]
    }
    h[(N-1)/2] += 1.0; // = 1 - h[(N-1)/2]

    return;
}

// Generate sinc function - used for making lowpass filter
void genSinc(double sinc[], // sinc[] will be written with the sinc function
            const int &N, // size (number of taps)
            const double &fc) // cutoff frequency
{
    int i;
    const double M = N-1;
    double n;

    // Constants
    const double PI = 3.14159265358979323846;

    // Generate sinc delayed by (N-1)/2

```

```

for (i = 0; i < N; i++) {
    if (i == M/2.0) {
        sinc[i] = 2.0 * fc;
    }
    else {
        n = (double)i - M/2.0;
        sinc[i] = sin(2.0*PI*fc*n) / (PI*n);
    }
}

return;
}

// Generate window function (Blackman)
void wBlackman(double w[], // w[] will be written with the Blackman window
               const int &N) // size of the window
{
    int i;
    const double M = N-1;
    const double PI = 3.14159265358979323846;

    for (i = 0; i < N; i++) {
        w[i] = 0.42 - (0.5 * cos(2.0*PI*(double)i/M)) + (0.08*cos(4.0*PI*(double)i/M));
    }

    return;
}

// Generate window function (Hanning)
void wHanning(double w[], // w[] will be written with the Hanning window
              const int &N) // size of the window
{
    int i;
    const double M = N-1;
    const double PI = 3.14159265358979323846;

    for (i = 0; i < N; i++) {
        w[i] = 0.5 * (1.0 - cos(2.0*PI*(double)i/M));
    }

    return;
}

// Generate window function (Hamming)
void wHamming(double w[], // w[] will be written with the Hamming window
              const int &N) // size of the window
{
    int i;
    const double M = N-1;
    const double PI = 3.14159265358979323846;

    for (i = 0; i < N; i++) {
        w[i] = 0.54 - (0.46*cos(2.0*PI*(double)i/M));
    }

    return;
}

#endif

```


Zoelzer biquad filters (click this to go back to the index)

Type : biquad IIR

References : Udo Zoelzer: Digital Audio Signal Processing (John Wiley & Sons, ISBN 0 471 97226 6), Chris Townsend

Notes :

Here's the formulas for the Low Pass, Peaking, and Low Shelf, which should cover the basics. I tried to convert the formulas so they are little more consistent. Also, the Zoelzer low pass/shelf formulas didn't have adjustable Q, so I added that for consistency with Roberts formulas as well. I think someone may want to check that I did it right.

----- Chris Townsend

I mistranscribed the low shelf cut formulas.

Hopefully this is correct. Thanks to James McCartney for noticing.

----- Chris Townsend

Code :

```
omega = 2*PI*frequency/sample_rate
```

```
K=tan(omega/2)
```

```
Q=Quality Factor
```

```
V=gain
```

```
LPF:  b0 = K^2
      b1 = 2*K^2
      b2 = K^2
      a0 = 1 + K/Q + K^2
      a1 = 2*(K^2 - 1)
      a2 = 1 - K/Q + K^2
```

peakingEQ:

```
boost:
```

```
b0 = 1 + V*K/Q + K^2
```

```
b1 = 2*(K^2 - 1)
```

```
b2 = 1 - V*K/Q + K^2
```

```
a0 = 1 + K/Q + K^2
```

```
a1 = 2*(K^2 - 1)
```

```
a2 = 1 - K/Q + K^2
```

```
cut:
```

```
b0 = 1 + K/Q + K^2
```

```
b1 = 2*(K^2 - 1)
```

```
b2 = 1 - K/Q + K^2
```

```
a0 = 1 + V*K/Q + K^2
```

```
a1 = 2*(K^2 - 1)
```

```
a2 = 1 - V*K/Q + K^2
```

lowShelf:

```
boost:
```

```
b0 = 1 + sqrt(2*V)*K + V*K^2
```

```
b1 = 2*(V*K^2 - 1)
```

```
b2 = 1 - sqrt(2*V)*K + V*K^2
```

```
a0 = 1 + K/Q + K^2
```

```
a1 = 2*(K^2 - 1)
```

```
a2 = 1 - K/Q + K^2
```

```
cut:
```

```
b0 = 1 + K/Q + K^2
```

```
b1 = 2*(K^2 - 1)
```

```
b2 = 1 - K/Q + K^2
```

```
a0 = 1 + sqrt(2*V)*K + V*K^2
```

```
a1 = 2*(V*K^2 - 1)
```

```
a2 = 1 - sqrt(2*V)*K + V*K^2
```

Comments

from : signalzerodb [[a t]] yahoo.com

comment : I get a different result for the low-shelf boost with parametric control.

Zolzer builds his lp shelf from a pair of poles and a pair of zeros at:

poles = $Q(-1 + j)$

zeros = $\sqrt{V}Q(-1 + j)$

Where (in the book) $Q=1/\sqrt{2}$

So,

$$s^2 + 2\sqrt{V}Qs + 2VQ^2$$

$$H(s) = \frac{\dots}{s^2 + 2Qs + 2Q^2}$$

If you analyse this in terms of:

$H(s) = \text{LPF}(s) + 1$, it sort of falls apart, as we've gained a zero in the LPF. (as does zolzers)

Then, if we bilinear transform that, we get:

$$a0 = 1 + 2\sqrt{V}Q^2K + 2VQ^2K^2$$

$$a1 = 2(2\sqrt{V}Q^2K^2 - 1)$$

$$a2 = 1 - 2\sqrt{V}Q^2K + 2VQ^2K^2$$

$$b0 = 1 + 2Q^2K + 2Q^2K^2$$

$$b1 = 2(2Q^2K^2 - 1)$$

$$b2 = 1 - 2Q^2K + 2Q^2K^2$$

For:

$$H(z) = a_0z^2 + a_1z + a_2 / b_0z^2 + b_1z + b_2$$

Which, i /think/ is right...

Dave.

from : signalzerodb [[a t]] yahoo.com

comment : Very sorry, I interpreted Zolzer's s-plane poles as z-plane poles. Too much digital stuff.

After getting back to grips with s-plane maths :) and much graphing to test that it's right, I still get slightly different results.

$$b_0 = 1 + \sqrt{V} * K / Q + V * K^2$$

$$b_1 = 2 * (V * K^2 - 1)$$

$$b_2 = 1 - \sqrt{V} * K / Q + V * K^2$$

$$a_0 = 1 + K / Q + K^2$$

$$a_1 = 2 * (K^2 - 1)$$

$$a_2 = 1 - K / Q + K^2$$

The way the filter works is to have two poles on a unit circle around the origin in the s-plane, and two zeros that start at the poles at $V_0=1$, and move outwards. The above co-efficients represent that. Chris's original results put the poles in the right place, but put the zeros at the location where the poles would be if they were butterworth, and move out from there - yielding some rather strange results...

But I've graphed that extensively, and it works fine now :)

Dave.

from : asynth(at)io(dot)com

comment : Once you divide through by a_0 , the Zoelzer LPF gives coefficient values that are _identical_ to the RBJ LPF.

This one is a cheaper formulation because there is only one transcendental function call (tan) instead of two (sin, cos) for RBJ.

-- james mccartney

from : asynth(at)io(dot)com

comment : Actually, sin and cos are pretty cheap when done via taylor series, so I take that last bit back.

-- james mccartney

from : unkargherth [[a t]] terra.es

comment : Anybody know the formulation for Band Pass, High Pass and High shelf ?

from : Christian [[a t]] savioursofsoul.de

comment : Have a look at tobybear's Filter Explorer: http://www.tobybear.de/p_filterexp.html

Usually you can derivate a highpass from a lowpass and vice versa.

from : unkargherth [[a t]] terra.es

comment : Thanks Christian, lots of things solved now !!

Unfortunately, Bandpass continues missing. I don't know if it's really possible to obtain a Bandpass filter out of this (my filters math knowlegde isn't so deep), but i asked for it because would be nice to have the complete set of Zoeltzer filters

I suppose thta YOU can derive one from another as you stated, but this is not my case. Anyway, lots of thanks for your help

from : rbj [[a t]] audioimagination.com

comment : >Actually, sin and cos are pretty cheap when done via taylor series, so I take that last bit back

also, James, the sin() and cos() are less of a problem for implementing in a fixed-point context. tan() is a bitch.

r b-j

from : Christian [[a t]] savioursofsoul.de

comment : Here's also a highshelf filters for completeness

$$K := \tan(\omega_0 * 0.5);$$

$$t2 :=;$$

$$t3 := K * fQ; t6 := (V);$$

$$t5 := \sqrt{2 * V} * K;$$

$$t1 := 1/;$$

$$b_0 = (K * K + \sqrt{2 * V} * K + V);$$

$$b_1 = 2 * (K * K - V);$$

$$b_2 = (K * K - \sqrt{2 * V} * K + V);$$

$$a_0 = (K * K + K * fQ + 1)$$

$$a_1 = -2 * (K * K - 1);$$

$$a_2 = - (K * K - K * fQ + 1);$$

from : scoofy [[a t]] inf.elte.hu

comment : Yes, there is a Taylor serie for tan(x), but near pi/2, it converges very slowly, so high frequencies is a problem again.

Let's suppose you approximate sin(x) with $x - x^3/6 + x^5/120$, and cos(x) with $1 - x^2/2 + x^4/24$.

So tan(x) would be

$$x - x^3/6 + x^5/120$$

$$1 - x^2/2 + x^4/24$$

How do you do a polynomial division for that?

from : wolven.ivan [[a t]] free.fr

comment : I think it is possible to get a Taylor serie of the tan() function ? And it is possible to do a polynomial division of the sin & cos series to get rid of the division, you get the same thing...

from : scoofy [[a t]] inf.elte.hu

comment : Highpass version:

HPF:

$$b0 = 1 - K^2$$

$$b1 = -2*K^2$$

$$b2 = 1 - K^2$$

$$a0 = 1 + K/Q + K^2$$

$$a1 = 2*(K^2 - 1)$$

$$a2 = 1 - K/Q + K^2$$

Bandpass version:

BPF1 (peak gain = Q):

$$b0 = K$$

$$b1 = 0$$

$$b2 = -K$$

$$a0 = 1 + K/Q + K^2$$

$$a1 = 2*(K^2 - 1)$$

$$a2 = 1 - K/Q + K^2$$

BPF2 (peak gain = zero):

$$b0 = K/Q$$

$$b1 = 0$$

$$b2 = -K/Q$$

$$a0 = 1 + K/Q + K^2$$

$$a1 = 2*(K^2 - 1)$$

$$a2 = 1 - K/Q + K^2$$

Couldn't figure out the notch coeffs yet...

-- peter schoffhauzer

from : scoofy [[a t]] inf.elte.hu

comment : Got the notch too finally ;)

Notch

$$b0 = 1 + K^2$$

$$b1 = 2*(K^2 - 1)$$

$$b2 = 1 + K^2$$

$$a0 = 1 + K/Q + K^2$$

$$a1 = 2*(K^2 - 1)$$

$$a2 = 1 - K/Q + K^2$$

The HPF seems to have an error in the previous post. The correct HPF version:

HPF:

$$b0 = 1 + K/Q$$

$$b1 = -2$$

$$b2 = 1 - K/Q$$

$$a0 = 1 + K/Q + K^2$$

$$a1 = 2*(K^2 - 1)$$

$$a2 = 1 - K/Q + K^2$$

Hopefully it works now. Anyone confirms?

The set is complete now. Happy coding :)

-- peter schoffhauzer

from : scoofy [[a t]] inf.elte.hu

comment : For sake of completeness ;)

Allpass:

$$b0 = 1 - K/Q + K^2$$

$$b1 = 2*(K^2 - 1)$$

$$b2 = 1$$

$$a0 = 1 + K/Q + K^2$$

$$a1 = 2*(K^2 - 1)$$

$$a2 = 1 - K/Q + K^2$$

-- peter schoffhauzer

from : Christian [[a t]] savioursofsoul.de
comment : What was wrong with the first version:

HPF:
 $b_0 = 1 - K^2$
 $b_1 = -2$ (!!)
 $b_2 = 1 - K^2$
 $a_0 = 1 + K/Q + K^2$
 $a_1 = 2*(K^2 - 1)$
 $a_2 = 1 - K/Q + K^2$

you only have to delete K^2 . In the other version the cutoff frequency depends on the Q!

from : Christian [[a t]] savioursofsoul.de
comment : Also the Allpass should be symmetrical:

$b_0 = 1 - K/Q + K^2$
 $b_1 = 2*(K^2 - 1)$
 $b_2 = 1 + K/Q + K^2$ (!!)
 $a_0 = 1 + K/Q + K^2$
 $a_1 = 2*(K^2 - 1)$
 $a_2 = 1 - K/Q + K^2$

If you divide by a_0 (to reduce a coefficient) b_2 will get 1 of course.

from : scoofy [[a t]] inf.elte.hu
comment : Ah, thanks for the allpass correction! I used TobyBear Filter Explorer, where I see only 5 coeffs instead of six, that was the source of confusion.

However, the highpass is still not perfect. In my 2nd version, the cutoff is not dependent of Q, because the cutoff is determined by the pole positions, which are set by a_1 and a_2 . Instead, as the zero positions change according to Q, the cutoff slope varies. So it has an interesting behaviour, for low Qs it has a 6dB/Oct slope, for infinite resonance, the slope becomes 12dB/Oct.

However, with your suggested HPF version, I got only a strange highshelf-like filter. So here is my 3rd version, which I hope works fine:

HPF:
 $b_0 = 1$
 $b_1 = -2$
 $b_2 = 1$
 $a_0 = 1 + K/Q + K^2$
 $a_1 = 2*(K^2 - 1)$
 $a_2 = 1 - K/Q + K^2$

Quite simple isn't it? ;)

Cheers
Peter

from : scoofy [[a t]] inf.elte.hu
comment : james mccartney:

Tan can also be approximated using Taylor series (approx sin and cos with Taylor, then $\tan(x)=\sin(x)/\cos(x)$) well, there's a heavy division that you can't get rid of... well, that's not true in all cases. The advantage of $\tan()$ is that you can use that $\tan(x) \sim x$ when x is small. So you can get coefficients without any transcendental functions for low and middle frequencies.

Peter

2 Wave shaping things (click this to go back to the index)

References : Posted by Frederic Petrot

Notes :

Makes nice saturations effects that can be easily computed using cordic

First using a atan function:

y1 using k=16

max is the max value you can reach (32767 would be a good guess)

Harmonics scale down linealy and not that fast

Second using the hyperbolic tangent function:

y2 using k=2

Harmonics scale down linealy very fast

Code :

```
y1 = (max>>1) * atan(k * x/max)
```

```
y2 = max * th(x/max)
```

Comments

from : ibalthor [[a t]] sbcglobal.net

comment : Why are you calling decompiled script code?BALTHOR

[Alien Wah](#) (click this to go back to the index)

[References](#) : Nasca Octavian Paul (paulnasca[AT]email.ro)

[Linked file](#) : [alienwah.c](#) (this linked file is included below)

[Notes](#) :

"I found this algoritm by "playing around" with complex numbers. Please email me your opinions about it.

Paul."

[Comments](#)

[from](#) : ignatz [[a t]] webmail.co.za

[comment](#) : need help porting this alienwah to C, i'm running linux d;>

[from](#) : antiprosynthesis [[a t]] hotmail.com

[comment](#) : Where to download the complex.h you included?

[from](#) : michelangelo79 [[a t]] libero.it

[comment](#) : I need help in a linux porting too...

I think the problem lyes on complex.h

I'm just at the very beginning of DSP and I'd like to learn.

I choosed alienwah.c because it seemed short and simple... but for now wont compile under linux...

Bye

[Linked files](#)

```
/*
 Alien-Wah by Nasca Octavian Paul from Tg. Mures, Romania
 e-mail: <paulnasca@email.ro> or <paulnasca@yahoo.com>.
 */

/*
 The algorithm was found by me by mistake(I was looking for something else);
 I called this effect "Alien Wah" because sounds a bit like wahwah, but more strange.
 The ideaa of this effect is very simple: It is a feedback delay who uses complex numbers.
 If x[] represents the input and y[] is the output, so a simple feedback delay looks like this:
 y[n]=y[n-delay]*fb+x[n]*(1-fb)

 'fb' is a real number between 0 and 1.
 If you change the fb with a complex number who has the MODULUS smaller than 1, it will look like
 this.

 fb=R*(cos(alpha)+i*sin(alpha)); i^2=-1; R<1;
 y[n]=y[n-delay]*R*(cos(alpha)+i*sin(alpha))+x[n]*(1-R);

 alpha is the phase of the number and is controlled by the LFO(Low Frequency Oscillator).
 If the 'delay' parameter is low, the effect sounds more like wah-wah,
 but if it is big, the effect will sound very interesting.
 The input x[n] has the real part of the samples from the wavefile and the imaginary part is zero.
 The output of this effect is the real part of y[n].

 Here it is a simple and unoptimised implementation of the effect. All parameters should be
 changed at compile time.
 It was tested only with Borland C++ 3.1.

 Please send me your opinions about this effect.
 Hope you like it (especially if you are play to guitar).
 Paul.
 */

/*
 Alien Wah Parameters

 freq          - "Alien Wah" LFO frequency
 startphase    - "Alien Wah" LFO startphase (radians), needed for stereo
 fb            - "Alien Wah" FeedBack (0.0 - low feedback, 1.0 = 100% high feedback)
 delay         - delay in samples at 44100 KHz (recomanded from 5 to 50...)
 */

#include <complex.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <math.h>

/*
 .raw files are raw files (without header), signed 16 bit,mono
 */
```

```

#define infile "a.raw" //input file
#define outfile "b.raw" //input file
#define samplerate 44100

#define bufsize 1024
int buf1[bufsize]; //input buffer
int buf2[bufsize]; //output buffer

#define lfoskipsamples 25 // How many samples are processed before compute the lfo value again

struct params
{
    float freq,startphase,fb;
    int delay;
} awparams;
//alien wah internal parameters

struct alienwahinternals
{
    complex *delaybuf;
    float lfoskip;
    long int t;
    complex c;
    int k;
} awint;

//effect initialisation
void init(float freq,float startphase,float fb,int delay){
    awparams.freq=freq;
    awparams.startphase=startphase;
    awparams.fb=fb/4+0.74;
    awparams.delay=(int)(delay/44100.0*samplerate);
    if (delay<1) delay=1;
    awint.delaybuf=new complex[awparams.delay];
    int i;
    for (i=0;i<delay;i++) awint.delaybuf[i]=complex(0,0);
    awint.lfoskip=freq*2*3.141592653589/samplerate;
    awint.t=0;
}

//process buffer
void process()
{
    int i;
    float lfo,out;
    complex outc;
    for(i=0;i<bufsize;i++)
    {
        if (awint.t++%lfoskipsamples==0)
        {
            lfo=(1+cos(awint.t*awint.lfoskip+awparams.startphase));
            awint.c=complex(cos(lfo)*awparams.fb,sin(lfo)*awparams.fb);
        };
        outc=awint.c*awint.delaybuf[awint.k]+(1-awparams.fb)*buf1[i];
        awint.delaybuf[awint.k]=outc;
        if ((++awint.k)>=awparams.delay)
            awint.k=0;
        out=real(outc)*3; //take real part of outc
        if (out<-32768) out=-32768;
        else if (out>32767) out=32767; //Prevents clipping
        buf2[i]=out;
    };
}

int main()
{
    char f1,f2;
    int readed;
    long int filereaded=0;
    printf("\n");
    f1=open(infile,O_RDONLY|O_BINARY);
    remove(outfile);
    f2=open(outfile,O_BINARY|O_CREAT,S_IWRITE);
    long int i;

    init(0.6,0,0.5,20); //effects parameters

    do
    {
        readed=read(f1,buf1,bufsize*2);

        process();
    }
}

```

```
    write(f2,buf2,readed);
    printf("%ld bytes \r",filereaded);
    filereaded+=readed;
}while (readed==bufsize*2);

delete(awint.delaybuf);
close(f1);
close(f2);
printf("\n\n");

return(0);
}
```


Band Limited PWM Generator (click this to go back to the index)

Type : PWM generator

References : Posted by paul_sernine75 AT hotmail DOT fr

Notes :

This is a commented and deobfuscated version of my 1st April fish. It is based on a tutorial code by Thierry Rochebois. I just translated and added comments.

Regards,

Paul Sernine.

Code :

```
// SelfPmpwm.cpp

// Antialised PWM oscillator

// Based on a tutorial code by Thierry Rochebois (98).
// Itself inspired by US patent 4249447 by Norio Tomisawa (81).
// Comments added/translated by P.Sernine (06).

// This program generates a 44100Hz-raw-PCM-mono-wavefile.
// It is based on Tomisawa's self-phase-modulated sinewave generators.
// Rochebois uses a common phase accumulator to feed two half-Tomisawa-
// oscillators. Each half-Tomisawa-oscillator generates a bandlimited
// sawtooth (band limitation depending on the feedback coeff B).
// These half oscillators are phase offseted according to the desired
// pulse width. They are finally combined to obtain the PW signal.
// Note: the anti-"hunting" filter is a critical feature of a good
// implementation of Tomisawa's method.
#include <math.h>
#include <stdio.h>
const float pi=3.14159265359f;
int main()
{
    float freq,dphi; ///< frequency (Hz) and phase increment(rad/sample)
    float dphif=0;   ///< filtered (anti click) phase increment
    float phi=-pi;   ///< phase
    float Y0=0,Y1=0; ///< feedback memories
    float PW=pi;     ///< pulse width ]0,2pi[
    float B=2.3f;    ///< feedback coef
    FILE *f=fopen("SelfPmpwm.pcm","wb");
    // séquence ('a'=mi=E)
    // you can edit this if you prefer another melody.
    static char seq[]="aiakahiafahadfaiakahiahafahadf"; ///< sequence
    int note=sizeof(seq)-2; ///< note number in the sequence
    int octave=0;         ///< octave number
    float env,envf=0;    ///< envelopped and filtered envelopped
    for(int ns=0;ns<8*(sizeof(seq)-1)*44100/6;ns++)
    {
//waveform control -----
        //Frequency
        //freq=27.5f*powf(2.0f,8*ns/(8*30*44100.0f/6)); //sweep
        freq=27.5f*powf(2.0f,octave+(seq[ns]-'a'-5)/12.0f);
        //freq*=(1.0f+0.01f*sinf(ns*0.0015f)); //vibrato
        dphi=freq*(pi/22050.0f); // phase increment
        dphif+=0.1f*(dphi-dphif);
        //notes and envelope trigger
        if((ns%(44100/6))==0)
        {
            note++;
            if(note>=(sizeof(seq)-1))// sequence loop
            {
                note=0;
                octave++;
            }
            env=1; //env set
            //PW=pi*(0.4+0.5f*(rand()%1000)/1000.0f); //random PW
        }
        env*=0.9998f; // exp envelope
        envf+=0.1f*(env-envf); // de-clicked envelope
        B=1.0f; // feedback coefficient
        //try this for a nice bass sound:
        //B*=envf*envf; // feedback controlled by envelope
        B*=2.3f*(1-0.0001f*freq); // feedback limitation
        if(B<0)
            B=0;
//waveform generation -----
        //Common phase
        phi+=dphif; // phase increment
        if(phi>=pi)
            phi-=2*pi; // phase wrapping

        // "phase" half Tomisawa generator 0
        // B*Y0 -> self phase modulation
        float out0=cosf(phi+B*Y0); // half-output 0
        Y0=0.5f*(out0+Y0); // anti "hunting" filter
    }
}
```

```
// "phase+PW" half Tomisawa generator 1
// B*Y1 -> self phase modulation
// PW -> phase offset
float out1=cosf(phi+B*Y1+PW); // half-output 1
Y1=0.5f*(out1+Y1); // anti "hunting" filter

// combination, envelope and output
short s=short(15000.0f*(out0-out1)*envf);
fwrite(&s,2,1,f); // file output
}
fclose(f);
return 0;
}
```

Comments

from : ---

comment : Did anyone try this?

How is the antialiasing compared to applying phaserror between two oscs in zerocross, one aliasing the other not (but pitcherror).

Best Regards,
Arif Ove Karlsen.

Bit quantization/reduction effect (click this to go back to the index)

Type : Bit-level noise-generating effect

References : Posted by Jon Watte

Notes :

This function, run on each sample, will emulate half the effect of running your signal through a Speak-N-Spell or similar low-bit-depth circuitry.

The other half would come from downsampling with no aliasing control, i e replicating every N-th sample N times in the output signal.

Code :

```
short keep_bits_from_16( short input, int keepBits ) {  
    return (input & (-1 << (16-keepBits)));  
}
```

Comments

from : illdoc [[a t]] gmail.com

comment : I add some code to prevent offset.

Code :

```
short keep_bits_from_16( short input, int keepBits ) {  
    short prevent_offset = static_cast<unsigned short>(-1) >> keepBits+1;  
    input &= (-1 << (16-keepBits));  
    return input + prevent_offset  
}
```

[Class for waveguide/delay effects](#) (click this to go back to the index)

Type : IIR filter

References : Posted by arguru[AT]smartelectronix.com

Notes :

Flexible-time, non-sample quantized delay , can be used for stuff like waveguide synthesis or time-based (chorus/flanger) fx.

MAX_WG_DELAY is a constant determining MAX buffer size (in samples)

Code :

```
class cwaveguide
{
public:
    cwaveguide(){clear();}
    virtual ~cwaveguide(){};

    void clear()
    {
        counter=0;
        for(int s=0;s<MAX_WG_DELAY;s++)
            buffer[s]=0;
    }

    inline float feed(float const in,float const feedback,double const delay)
    {
        // calculate delay offset
        double back=(double)counter-delay;

        // clip lookback buffer-bound
        if(back<0.0)
            back=MAX_WG_DELAY+back;

        // compute interpolation left-floor
        int const index0=floor_int(back);

        // compute interpolation right-floor
        int index_1=index0-1;
        int index1=index0+1;
        int index2=index0+2;

        // clip interp. buffer-bound
        if(index_1<0)index_1=MAX_WG_DELAY-1;
        if(index1>=MAX_WG_DELAY)index1=0;
        if(index2>=MAX_WG_DELAY)index2=0;

        // get neighbour samples
        float const y_1= buffer [index_1];
        float const y0 = buffer [index0];
        float const y1 = buffer [index1];
        float const y2 = buffer [index2];

        // compute interpolation x
        float const x=(float)back-(float)index0;

        // calculate
        float const c0 = y0;
        float const c1 = 0.5f*(y1-y_1);
        float const c2 = y_1 - 2.5f*y0 + 2.0f*y1 - 0.5f*y2;
        float const c3 = 0.5f*(y2-y_1) + 1.5f*(y0-y1);

        float const output=((c3*x+c2)*x+c1)*x+c0;

        // add to delay buffer
        buffer[counter]=in+output*feedback;

        // increment delay counter
        counter++;

        // clip delay counter
        if(counter>=MAX_WG_DELAY)
            counter=0;

        // return output
        return output;
    }

    float buffer[MAX_WG_DELAY];
    int counter;
};
```

Compressor (click this to go back to the index)

Type : Hardknee compressor with RMS look-ahead envelope calculation and adjustable attack/decay

References : Posted by flashinc[AT]mail[DOT]ru

Notes :

RMS is a true way to estimate _musical_ signal energy, our ears behaves in a same way.

to making all it work,
try this values (as is, routine accepts percents and milliseconds) for first time:

```
threshold = 50%
slope = 50%
RMS window width = 1 ms
lookahead = 3 ms
attack time = 0.1 ms
release time = 300 ms
```

This code can be significantly improved in speed by changing RMS calculation loop to 'running summ' (keeping the summ in 'window' - adding next newest sample and subtracting oldest on each step)

Code :

```
void compress
(
    float* wav_in,      // signal
    int    n,          // N samples
    double threshold,  // threshold (percents)
    double slope,      // slope angle (percents)
    int    sr,         // sample rate (smp/sec)
    double tla,        // lookahead (ms)
    double twnd,       // window time (ms)
    double tatt,       // attack time (ms)
    double trel        // release time (ms)
)
{
    typedef float  stereodata[2];
    stereodata*   wav = (stereodata*) wav_in; // our stereo signal
    threshold *= 0.01; // threshold to unity (0..1)
    slope *= 0.01; // slope to unity
    tla *= 1e-3; // lookahead time to seconds
    twnd *= 1e-3; // window time to seconds
    tatt *= 1e-3; // attack time to seconds
    trel *= 1e-3; // release time to seconds

    // attack and release "per sample decay"
    double att = (tatt == 0.0) ? (0.0) : exp (-1.0 / (sr * tatt));
    double rel = (trel == 0.0) ? (0.0) : exp (-1.0 / (sr * trel));

    // envelope
    double env = 0.0;

    // sample offset to lookahead wnd start
    int    lhsmp = (int) (sr * tla);

    // samples count in lookahead window
    int    nrms = (int) (sr * twnd);

    // for each sample...
    for (int i = 0; i < n; ++i)
    {
        // now compute RMS
        double summ = 0;

        // for each sample in window
        for (int j = 0; j < nrms; ++j)
        {
            int    lki = i + j + lhsmp;
            double smp;

            // if we in bounds of signal?
            // if so, convert to mono
            if (lki < n)
                smp = 0.5 * wav[lki][0] + 0.5 * wav[lki][1];
            else
                smp = 0.0; // if we out of bounds we just get zero in smp

            summ += smp * smp; // square em..
        }

        double rms = sqrt (summ / nrms); // root-mean-square

        // dynamic selection: attack or release?
        double theta = rms > env ? att : rel;
    }
}
```

```

// smoothing with capacitor, envelope extraction...
// here be aware of pIV denormal numbers glitch
env = (1.0 - theta) * rms + theta * env;

// the very easy hard knee 1:N compressor
double gain = 1.0;
if (env > threshold)
    gain = gain - (env - threshold) * slope;

// result - two hard kneed compressed channels...
float leftchannel = wav[i][0] * gain;
float rightchannel = wav[i][1] * gain;
}
}

```

Comments

from : music-dsp [[a t]] umminger.com

comment : My comments:

A rectangular window is not physical. It would make more physical sense, and be a lot cheaper, to use a 1-pole low pass filter to do the RMS averaging. A 1-pole filter means you can lose the bounds checks in the RMS calculation.

It does not make sense to convert to mono before squaring, you should square each channel separately and then add them together to get the total signal power.

You might also consider whether you even need any filtering other than the attack/release filter. You could modify the attack/release rates appropriately, place the sqrt after the attack/release, and lose the rms averager entirely.

I don't think your compressor actually approaches a linear slope in the decibel domain. You need a gain law more like

```
double gain = exp(max(0.0,log(env)-log(thresh))*slope);
```

Sincerely,
Frederick Umminger

from : xeeton[AT]gmail[DOT]com

comment : To sum up (and maybe augment) the RMS calculation method, this question and answer may be of use...

music-dsp@shoko.calarts.edu writes:

I am looking at gain processing algorithms. I haven't found much in the way of reference material on this, any pointers? In the level detection code, if one is doing peak detection, how many samples does one generally average over (if at all)? What kind of window size for RMS level detection? Is the RMS level detection generally the same algo. as peak, but with a bigger window?

The peak detector can be easily implemented as a one-pole low pass, you just have to modify it, so that it tracks the peaks and gently falls down afterwards. RMS detection is done squaring the input signal, averaging with a lowpass and taking the root afterwards.

Hope this helps.

Kind regards

Steffan Diedrichsen

DSP developer

emagic GmbH

I found the thread by searching old [music-dsp] forum posts. Hope it helps.

from : graue [[a t]] oceanbase.org

comment : How would you use a 1-pole lowpass filter to do RMS averaging? How do you pick a coefficient to use?

from : scoofy [[a t]] inf.elte.hu

comment : Use $x = \exp(-1/d)$, where d is the time constant in samples. A 1 pole IIR filter has an infinite impulse response, so instead of window width, this coeff determines the time when the impulse response reaches 36.8% of the original value.

Coeffs:

$a0 = 1.0 - x$;

$b1 = -x$;

Loop:

$out = a0 * in - b1 * tmp$;

$tmp = out$;

-- peter schoffhauzer

from : txutao [[a t]] 163.com

comment : I am looking at gain processing algorithms²

There are too such sentences :

```
double att = (tatt == 0.0) ? (0.0) : exp (-1.0 / (sr * tatt));  
double rel = (trei == 0.0) ? (0.0) : exp (-1.0 / (sr * trei));
```

can you tell me something about the $\exp(-1.0 / (sr * tatt))$?

New day ~~
thanks

Decimator (click this to go back to the index)

Type : Bit-reducer and sample&hold unit

References : Posted by tobyear[AT]web[DOT]de

Notes :

This is a simple bit and sample rate reduction code, maybe some of you can use it. The parameters are bits (1..32) and rate (0..1, 1 is the original samplerate).

Call the function like this:

```
y=decimate(x);
```

A VST plugin implementing this algorithm (with full Delphi source code included) can be downloaded from here:

<http://tobybear.phreque.com/decimator.zip>

Comments/suggestions/improvements are welcome, send them to: tobybear@web.de

Code :

```
// bits: 1..32
// rate: 0..1 (1 is original samplerate)

***** Pascal source *****
var m:longint;
    y,cnt,rate:single;

// call this at least once before calling
// decimate() the first time
procedure setparams(bits:integer;shrate:single);
begin
  m:=1 shl (bits-1);
  cnt:=1;
  rate:=shrate;
end;

function decimate(i:single):single;
begin
  cnt:=cnt+rate;
  if (cnt>1) then
  begin
    cnt:=cnt-1;
    y:=round(i*m)/m;
  end;
  result:=y;
end;

***** C source *****
int bits=16;
float rate=0.5;

long int m=1<<(bits-1);
float y=0,cnt=0;

float decimate(float i)
{
  cnt+=rate;
  if (cnt>=1)
  {
    cnt-=1;
    y=(long int)(i*m)/(float)m;
  }
  return y;
}
```

Comments

from : kaleja [[a t]] estarcion.com
comment : Nothing wrong with that, but you can also do fractional-bit-depth decimations, allowing smooth degradation from high bit depth to low and back:

```
// something like this -- this is
// completely off the top of my head
// precalculate the quantization level
float bits; // effective bit depth
float quantum = powf( 2.0f, bits );
```

```
// per sample
y = floorf( x * quantum ) / quantum;
```

from : dr.kef [[a t]] spray.se
comment : it looks to me like the c-line


```
long int m=1<<(bits-1);
```

doesn't give the correct number of quantisation levels if the number of levels is defined as 2^{bits} . if bits=2 for instance, the above code line returns a bit pattern of 10 (3) and not 11 (2^2) like one would expect.

please, do correct me if im wrong.

/heatrof

from : resofactor [[a t]] hotmail.com

comment : just getting into coding, i've mainly been working with synthedit...but would really like to move on into the bigger arena? any pointers for a DSP newbie-totally not hip on structured programming...yet! :O)

Delay time calculation for reverberation (click this to go back to the index)

References : Posted by Andy Mucho

Notes :

This is from some notes I had scribbled down from a while back on automatically calculating diffuse delays. Given an initial delay line gain and time, calculate the times and feedback gain for numlines delay lines..

Code :

```
int numlines = 8;
float t1 = 50.0; // d0 time
float g1 = 0.75; // d0 gain
float rev = -3*t1 / log10 (g1);

for (int n = 0; n < numlines; ++n)
{
    float dt = t1 / pow (2, (float (n) / numlines));
    float g = pow (10, -((3*dt) / rev));
    printf ("d%d t=%.3f g=%.3f\n", n, dt, g);
}
```

The above with t1=50.0 and g1=0.75 yields:

```
d0 t=50.000 g=0.750
d1 t=45.850 g=0.768
d2 t=42.045 g=0.785
d3 t=38.555 g=0.801
d4 t=35.355 g=0.816
d5 t=32.421 g=0.830
d6 t=29.730 g=0.843
d7 t=27.263 g=0.855
```

To go more diffuse, chuck in dual feedback paths with a one cycle delay effectively creating a phase-shifter in the feedback path, then things get more exciting.. Though what the optimum phase shifts would be I couldn't tell you right now..

Comments

from : bob [[a t]] yahoob.com

comment : Hello, when you say 'dual feedback paths with one cycle delay' do you mean dual as in stereo? And one cycle means one sample? My experiments have massive energy build up of energy unless the feedback was something like * 0.01, but there was still ringing in the sound. Can you hint a little more about what you mean?

Thanks

DIRAC - Free C/C++ Library for Time and Pitch Manipulation of Audio Based on Time-Frequency Transforms (click this to go back to the index)

Type : Time Stretch / Pitch Shift

References : Posted by Stephan M. Bernsee

Notes :

This is an availability notification for a free object library, no source code.

Code :

Past research has shown time domain [pitch] synchronized overlap-add ([P]SOLA) algorithms for independent time and pitch manipulation of audio ("time stretching" and "pitch shifting") to be the method of choice for single-pitched sounds such as voice and musically monophonic instrument recordings due to the prominent periodicity at the fundamental period. On the other hand, frequency domain methods have recently evolved around the concept of the phase vocoder that have proven to be vastly superior for multi-pitched sounds and entire musical pieces.

"Dirac" is a free cross-platform C/C++ object library that exploits the good localization of time-frequency transforms in both domains to build an algorithm for time and pitch manipulation that uses an arbitrary time-frequency tiling depending on the underlying signal. Additionally, the time and frequency localization parameter of the basis can be user-defined, making the algorithm smoothly scalable to provide either the phase coherence properties of a time domain process or the good frequency resolution of the phase vocoder.

The basic "Dirac" library comes as a free download off the DSPdimension web site and is currently available for Microsoft Visual C6+, CodeWarrior 8.x on Windows and MacOS, and for Xcode 2.x on MacOS X. Optional "Studio" and "Pro" versions with increased feature set are available commercially from the author.

More information and download at <http://www.dspdimension.com>

Comments

from : tahome [[a t]] postino.ch
comment : The quality of this is just amazing!!! I'm using Stefan's TimeFactory on a daily basis but this is even better imho. I hope it will be upgraded to use dirac soon!

from : ce [[a t]] ce.com
comment : Awesome, but I need a Windows CE-library using integer math.

from : WER [[a t]] 163.COM
comment : JINF NMCXFH BNHHJK

from : info [[a t]] into3d.com
comment : Anyone has a VST framework for this?

from : who [[a t]] cares.com
comment : I hate false publicity... There's no free code.

from : tahome[AT]postino[DOT]ch
comment : Why not?

[dynamic convolution](#) (click this to go back to the index)

Type : a naive implementation in C++

References : Posted by Risto Holopainen

Notes :
This class illustrates the use of dynamic convolution with a set of IR:s consisting of exponentially damped sinusoids with glissando. There's lots of things to improve for efficiency.

```
Code :
#include <cmath>

class dynaconv
{
public:
// sr=sample rate, cf=resonance frequency,
// dp=frq sweep or nonlinearity amount
dynaconv(const int sr, float cf, float dp);
double operator()(double);

private:
// steps: number of amplitude regions, L: length of impulse response
enum {steps=258, dv=steps-2, L=200};
double x[L];
double h[steps][L];
int S[L];
double conv(double *x, int d);
};

dynaconv::dynaconv(const int sr, float cfr, float dp)
{
for(int i=0; i<L; i++)
x[i] = S[i] = 0;

double sc = 6.0/L;
double frq = twopi*cfr/sr;

// IR's initialised here.
// h[0] holds the IR for samples with lowest amplitude.
for(int k=0; k<steps; k++)
{
double sum = 0;
double theta=0;
double w;
for(int i=0; i<L; i++)
{
// IR of exp. decaying sinusoid with glissando
h[k][i] = sin(theta)*exp(-sc*i);
w = (double)i/L;
theta += frq*(1 + dp*w*(k - 0.4*steps)/steps);
sum += fabs(h[k][i]);
}

double norm = 1.0/sum;
for(int i=0; i<L; i++)
h[k][i] *= norm;
}
}

double dynaconv::operator()(double in)
{
double A = fabs(in);
double a, b, w, y;
int sel = int(dv*A);

for(int j=L-1; j>0; j--)
{
x[j] = x[j-1];
S[j] = S[j-1];
}
x[0] = in;
S[0] = sel;

if(sel == 0)
y = conv(x, 0);

else if(sel > 0)
{
a = conv(x, 0);
b = conv(x, 1);
w = dv*A - sel;
y = w*a + (1-w)*b;
}

return y;
}

double dynaconv::conv(double *x, int d)
```

```

{
double y=0;
for(int i=0; i<L; i++)
y += x[i] * h[ S[i]+d ][i];

return y;
}

```

Comments

from : Christian [[a t]] savioursofsoul.de

comment : You can speed things up by:

a) rewriting the "double dynaconv::conv(double *x, int d)" function using Assembler, SSE and 3DNow routines.

b) instead of this

```

"else if(sel > 0)
{
a = conv(x, 0);
b = conv(x, 1);
w = dv*A - sel;
y = w*a + (1-w)*b;
}"

```

you can create a temp IR by fading the two impulse responses before convolution. Then you'll only need ONE CPU-expensive-convolution.

c) this one only works with the upper half wave!

d) only nonlinear components can be modeled. For time-variant modeling (compressor/limiter) you'll need more than this.

e) the algo is protected by a patent. But it's easy to find more efficient ways, which aren't protected by the patent.

With my implementation i can fold up to 4000 Samples (IR) in realtime on my machine.

from : correction [[a t]] point.d

comment : Correction to d:

d) only time invariant nonlinear components can be modeled; and then adequate memory must be used. Compressors/Limiters can be modelled, but the memory requirements will be somewhat frightening. Time-variant systems, such as flangers, phasors, and sub-harmonic generators (i.e. anything with an internal clock) will need more than this.

[Early echo's with image-mirror technique](#) (click this to go back to the index)

References : Donald Schulz

Linked file : [early_echo.c](#) (this linked file is included below)

Linked file : [early_echo_eng.c](#) (this linked file is included below)

Notes :

(see linked files)

Donald Schulz's code for computing early echoes using the image-mirror method. There's an english and a german version.

Linked files

/*

```
From: "Donald Schulz" <d.schulz@gmx.de>
To: <music-dsp@shoko.calarts.edu>
Subject: [music-dsp] Image-mirror method source code
Date: Sun, 11 Jun 2000 15:01:51 +0200
```

A while ago I wrote a program to calculate early echo responses. As there seems to be some interest in this, I now post it into the public domain.

Have phun,

Donald.

*/

```
/*
 *
 * Early Echo Computation using image-mirror method
 *
 * Position of listener, 2 sound-sources, room-size may be set.
 * Four early echo responses are calculated (from left sound source and
 * right sound source to left and right ear). Angle with which the sound
 * meets the ears is taken into account.
 * The early echo response from left sound source to left ear is printed
 * to screen for demonstration, the first table contains the delay times
 * and the second one the weights.
 *
 * Program is released into the public domain.
 *
 * Sorry for german comments :-(
 * Some frequently used german words:
 * hoerpos : listening position
 * breite : width
 * laenge : length
 * hoehe : height
 * daempfung : damping
 * links : left
 * rechts : right
 * Ohr : ear
 * Winkel : angle
 * Wichtung : weight
 * Normierung : normalization
 *
 * If someone does some improvements on this, I (Donald, d.schulz@gmx.de)
 * would be happy to get the improved code.
 *
 *****/
```

```
#include <math.h>
#include <stdio.h>
```

```
#define early_length 0x4000 /* Laenge der Puffer fuer early-echo */
#define early_length_1 0x3fff
```

```
#define early_tap_num 20 /* Anzahl an early-echo taps */
```

```
#define breite 15.0 /* 15 m breiter Raum (x)*/
#define laenge 20.0 /* 20 m lang (y) */
#define hoehe 10.0 /* 10 m hoch (z)*/
#define daempfung 0.999 /* Daempfungsfaktor bei Reflexion */
```

```
#define hoerposx 7.91 /* hier sitzt der Hoerer (linkes Ohr) */
#define hoerposy 5.0
#define hoerposz 2.0
```

```
#define leftposx 5.1 /* hier steht die linke Schallquelle */
```

```

#define leftposy 16.3
#define leftposz 2.5

#define rightposx 5.9          /* hier steht die rechte Schallquelle */
#define rightposy 6.3
#define rightposz 1.5

#define i_length 32           /* Laenge des Eingangs-Zwischenpuffers */
#define i_length_1 31
#define o_length 32          /* Laenge des Ausgangs-Zwischenpuffers */
#define o_length_1 31

float *early_l2l; /* linker Kanal nach linkem Ohr */
float *early_l2r; /* linker Kanal nach rechtem Ohr */
float *early_r2l; /* rechter Kanal nach linkem Ohr */
float *early_r2r; /* rechter Kanal nach rechtem Ohr */

int early_pos=0;

int e_delays_l2l[early_tap_num]; /* Delays der early-echos */
float e_values_l2l[early_tap_num]; /* Gewichtungen der delays */
int e_delays_l2r[early_tap_num];
float e_values_l2r[early_tap_num];
int e_delays_r2l[early_tap_num];
float e_values_r2l[early_tap_num];
int e_delays_r2r[early_tap_num];
float e_values_r2r[early_tap_num];

/* Early-echo Berechnung mittels Spiegelquellenverfahren

Raummodell:
H - Hoererposition
L - Spiegelschallquellen des linken Kanales
U - Koordinatenursprung
Raum sei 11 meter breit und 5 meter lang (1 Zeichen = 1 meter)
Linker Kanal stehe bei x=2 y=4 z=?
Hoerer stehe bei x=5 y=1 z=?

|-----|
|       |       |       |       |       |
|       |       |       |       |       |
|_L_    |_L_    |_L_    |_L_    |_L_
L|L    |L|L    |L|L    |L|L    |L|
|       |       |       |       |       |
|       |       |       |       |       |
|       |       |       |       |       |
|       |       |       |       |       |
|_L_    |_L_    |_L_    |_L_
*/

main()
{
  int i,j,select;
  float dist_max;
  float x,y,z,xref,yref,zref;
  float x_pos,y_pos,z_pos;
  float distance,winkel;
  float wichtung;
  float normierungr,normierungl;

  early_l2l=(float *)malloc(early_length*sizeof(float));
  early_l2r=(float *)malloc(early_length*sizeof(float));
  early_r2l=(float *)malloc(early_length*sizeof(float));
  early_r2r=(float *)malloc(early_length*sizeof(float));

  /* Erst mal Echos loeschen: */
  for (i=0;i<early_length;i++)
    early_l2l[i]=early_l2r[i]=early_r2l[i]=early_r2r[i]=0.0;

  dist_max=300.0*early_length/44100.0; /* 300 m/s Schallgeschwindigkeit */

  /* Echo vom LINKEN Kanal auf linkes/rechtes Ohr berechnen */

  for (x=-ceil(dist_max/(2*laenge));x<=ceil(dist_max/(2*laenge));x++)

```

```

for (y=-ceil(dist_max/(2*breite));y<=ceil(dist_max/(2*breite));y++)
for (z=-ceil(dist_max/(2*hoehe));z<=ceil(dist_max/(2*hoehe));z++)
{
  xref=2*x*breite;
  yref=2*y*laenge;
  zref=2*z*hoehe;
  for (select=0;select<8;select++) /* vollstaendige Permutation */
  {
    if (select&1) x_pos=xref+leftposx;
    else x_pos=xref-leftposx;
    if (select&2) y_pos=yref+leftposy;
    else y_pos=yref-leftposy;
    if (select&4) z_pos=zref+leftposz;
    else z_pos=zref-leftposz;
    /* Jetzt steht die absolute Position der Quelle in ?_pos */

    /* Relative Position zum linken Ohr des Hoerers bestimmen: */
    x_pos-=hoerposx;
    y_pos-=hoerposy;
    z_pos-=hoerposz;

    distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
    /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHz-> *106 */
    if ((distance*147)<early_length)
    {
      /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
      winkel=atan(y_pos/x_pos);
      if (y_pos>0) /* Klang kommt aus vorderem Bereich: */
      { /* 0=links=>Verstaerkung=1 Pi=rechts=>Verstaerkung=0.22 (?) */
        winkel+=3.1415926/2; wichtung = 1 - winkel/4;
      }
      else /* Klang kommt von hinten: */
      {
        winkel-=3.1415926/2; wichtung= 1 + winkel/4;
      }
      /* Early-echo gemaess Winkel und Entfernung gewichten: */
      early_l2l[(int) (distance*147.)]+=wichtung/(pow(distance,3.1));
    }

    /* Relative Position zum rechten Ohr des Hoerers bestimmen: */
    x_pos-=0.18; /* Kopf ist 18 cm breit */

    distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
    /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHz-> *106 */
    if ((distance*147)<early_length)
    {
      /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
      winkel=atan(y_pos/x_pos);
      if (y_pos>0) /* Klang kommt aus vorderem Bereich: */
      { /* 0=links=>Verstaerkung=1 Pi=rechts=>Verstaerkung=0.22 (?) */
        winkel-=3.1415926/2; wichtung = 1 + winkel/4;
      }
      else /* Klang kommt von hinten: */
      {
        winkel+=3.1415926/2; wichtung= 1 - winkel/4;
      }
      /* Early-echo gemaess Winkel und Entfernung gewichten: */
      early_l2r[(int) (distance*147.)]+=wichtung/(pow(distance,3.1));
    }
  }
}

/* Echo vom RECHTEN Kanal auf linkes/rechtes Ohr berechnen */

for (x=-ceil(dist_max/(2*laenge));x<=ceil(dist_max/(2*laenge));x++)
for (y=-ceil(dist_max/(2*breite));y<=ceil(dist_max/(2*breite));y++)
for (z=-ceil(dist_max/(2*hoehe));z<=ceil(dist_max/(2*hoehe));z++)
{
  xref=2*x*breite;
  yref=2*y*laenge;
  zref=2*z*hoehe;
  for (select=0;select<8;select++) /* vollstaendige Permutation */
  {
    if (select&1) x_pos=xref+rightposx;
    else x_pos=xref-rightposx;
    if (select&2) y_pos=yref+rightposy;
    else y_pos=yref-rightposy;
    if (select&4) z_pos=zref+rightposz;
    else z_pos=zref-rightposz;
    /* Jetzt steht die absolute Position der Quelle in ?_pos */

    /* Relative Position zum linken Ohr des Hoerers bestimmen: */

```



```

x_pos-=hoerposx;
y_pos-=hoerposy;
z_pos-=hoerposz;

distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
/* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHz-> *106 */
if ((distance*147.)<early_length)
{
/* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
winkel=atan(y_pos/x_pos);
if (y_pos>0) /* Klang kommt aus vorderem Bereich: */
{ /* 0=links=>Verstaerkung=1 Pi=rechts=>Verstaerkung=0.22 (?) */
winkel+=3.1415926/2; wichtung = 1 - winkel/4;
}
else /* Klang kommt von hinten: */
{
winkel-=3.1415926/2; wichtung= 1 + winkel/4;
}
/* Early-echo gemass Winkel und Entfernung gewichten: */
early_r2l[(int) (distance*147.)]+=wichtung/(pow(distance,3.1));
}

/* Und jetzt Early-Echo zweiter Kanal auf LINKES Ohr berechnen */
x_pos-=0.18; /* Kopfbreite addieren */

distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
/* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHz-> *106 */
if ((distance*147.)<early_length)
{
/* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
winkel=atan(y_pos/x_pos);
if (y_pos>0) /* Klang kommt aus vorderem Bereich: */
{ /* 0=links=>Verstaerkung=1 Pi=rechts=>Verstaerkung=0.22 (?) */
winkel-=3.1415926/2; wichtung = 1 + winkel/4;
}
else /* Klang kommt von hinten: */
{
winkel+=3.1415926/2; wichtung= 1 - winkel/4;
}
/* Early-echo gemass Winkel und Entfernung gewichten: */
early_r2r[(int) (distance*147.)]+=wichtung/(pow(distance,3.1));
}
}

/* Und jetzt aus berechnetem Echo die ersten early_tap_num Werte holen */
/* Erst mal e's zuruecksetzen: */
for (i=0;i<early_tap_num;i++)
{
e_values_l2l[i]=e_values_l2r[i]=0.0;
e_delays_l2l[i]=e_delays_l2r[i]=0; /* Unangenehme Speicherzugriffe vermeiden */
e_values_r2l[i]=e_values_r2r[i]=0.0;
e_delays_r2l[i]=e_delays_r2r[i]=0;
}

/* und jetzt e_delays und e_values extrahieren: */
j=0;
normierungl=normierungr=0.0;
for(i=0;i<early_length;i++)
{
if (early_l2l[i]!=0)
{
e_delays_l2l[j]=i;
e_values_l2l[j]=early_l2l[i];
normierungl+=early_l2l[i];
j++;
}
if (j==early_tap_num) break;
}
j=0;
for(i=0;i<early_length;i++)
{
if (early_l2r[i]!=0)
{
e_delays_l2r[j]=i;
e_values_l2r[j]=early_l2r[i];
normierungr+=early_l2r[i];
j++;
}
if (j==early_tap_num) break;
}

```

```

}
j=0;
for(i=0;i<early_length;i++)
{
    if (early_r2l[i]!=0)
    {
        e_delays_r2l[j]=i;
        e_values_r2l[j]=early_r2l[i];
        normierungl+=early_r2l[i];
        j++;
    }
    if (j==early_tap_num) break;
}
j=0;
for(i=0;i<early_length;i++)
{
    if (early_r2r[i]!=0)
    {
        e_delays_r2r[j]=i;
        e_values_r2r[j]=early_r2r[i];
        normierungr+=early_r2r[i];
        j++;
    }
    if (j==early_tap_num) break;
}

/* groessere von beiden Normierungen verwenden: */
if (normierungr>normierungl) normierungr=normierungl;

for (j=0;j<early_tap_num;j++)
{
    e_values_l2l[j]/=normierungr;
    e_values_l2r[j]/=normierungr;
    e_values_r2l[j]/=normierungr;
    e_values_r2r[j]/=normierungr;
}
/* Ausgeben nur der l2l-Werte fuer schnelles Reverb */
printf("int e_delays[%d]={",early_tap_num);
for (j=0;j<(early_tap_num-1);j++)
    printf("%d, ",e_delays_l2l[j]);
printf("%d};\n\n",e_delays_l2l[j]);

printf("float e_values[%d]={",early_tap_num);
for (j=0;j<(early_tap_num-1);j++)
    printf("%0.4f, ",e_values_l2l[j]);
printf("%0.4f};\n\n",e_values_l2l[j]);
}

/*
From: "Donald Schulz" <d.schulz@gmx.de>
To: <music-dsp@shoko.calarts.edu>
Subject: [music-dsp] Image-mirror method source code
Date: Sun, 11 Jun 2000 15:01:51 +0200

A while ago I wrote a program to calculate early echo responses.
As there seems to be some interest in this, I now post it into the
public domain.

Have phun,

Donald.
*/

/*

i have taken the liberty of renaming some of donald's variables to make
his code easier to use for english speaking, non-german speakers. i did
a simple search and replace on all the german words mentioned in the
note below. all of the comments are still in german, but the english
variable names make the code easier to follow, at least for me. i don't
think that i messed anything up by doing this, but if something's not
working you might want to try the original file, just in case.

douglas

*/

/*****
*
* Early Echo Computation using image-mirror method

```

```

*
* Position of listener, 2 sound-sources, room-size may be set.
* Four early echo responses are calculated (from left sound source and
* right sound source to left and right ear). Angle with which the sound
* meets the ears is taken into account.
* The early echo response from left sound source to left ear is printed
* to screen for demonstration, the first table contains the delay times
* and the second one the weights.
*
* Program is released into the public domain.
*
* Sorry for german comments :-(
* Some frequently used german words:
* hoerpos : listening position
* breite : width
* laenge : length
* hoehe : height
* daempfung : damping
* links : left
* rechts : right
* Ohr : ear
* Winkel : angle
* Wichtung : weight
* Normierung : normalization
*
*
* If someone does some improvements on this, I (Donald, d.schulz@gmx.de)
* would be happy to get the improved code.
*
*****/

#include <math.h>
#include <stdio.h>

#define early_length 0x4000 /* Length der Puffer fuer early-echo */
#define early_length_1 0x3fff

#define early_tap_num 20 /* Anzahl an early-echo taps */

#define width 15.0 /* 15 m widthr Raum (x)*/
#define length 20.0 /* 20 m lang (y) */
#define height 10.0 /* 10 m hoch (z)*/
#define damping 0.999 /* Dampingsfaktor bei Reflexion */

#define listening_positionx 7.91 /* hier sitzt der Hoerer (linkes ear) */
#define listening_positiony 5.0
#define listening_positionz 2.0

#define leftposx 5.1 /* hier steht die linke Schallquelle */
#define leftposy 16.3
#define leftposz 2.5

#define rightposx 5.9 /* hier steht die rechte Schallquelle */
#define rightposy 6.3
#define rightposz 1.5

#define i_length 32 /* Length des Eingangs-Zwischenpuffers */
#define i_length_1 31
#define o_length 32 /* Length des Ausgangs-Zwischenpuffers */
#define o_length_1 31

float *early_l2l; /* linker Kanal nach linkem ear */
float *early_l2r; /* linker Kanal nach rechtem ear */
float *early_r2l; /* rechter Kanal nach linkem ear */
float *early_r2r; /* rechter Kanal nach rechtem ear */

int early_pos=0;

int e_delays_l2l[early_tap_num]; /* Delays der early-echos */
float e_values_l2l[early_tap_num]; /* Geweichten der delays */
int e_delays_l2r[early_tap_num];
float e_values_l2r[early_tap_num];
int e_delays_r2l[early_tap_num];
float e_values_r2l[early_tap_num];
int e_delays_r2r[early_tap_num];
float e_values_r2r[early_tap_num];

/* Early-echo Berechnung mittels Spiegelquellenverfahren

```

Raummodell:

H - Hoererposition

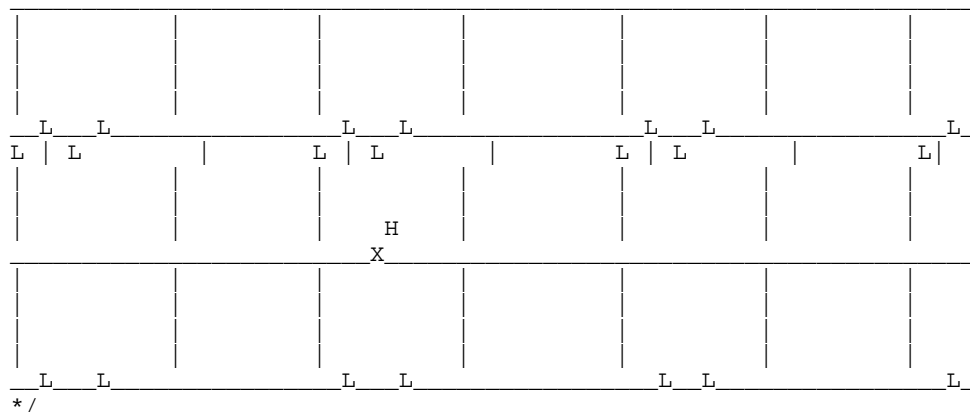
L - Spiegelschallquellen des linken Kanales

U - Koordinatenursprung

Raum sei 11 meter breit und 5 meter lang (1 Zeichen = 1 meter)

Linker Kanal stehe bei x=2 y=4 z=?

Hoerer stehe bei x=5 y=1 z=?



main()

```
{
  int i,j,select;
  float dist_max;
  float x,y,z,xref,yref,zref;
  float x_pos,y_pos,z_pos;
  float distance,angle;
  float weight;
  float normalizationr,normalizationl;

  early_l2l=(float *)malloc(early_length*sizeof(float));
  early_l2r=(float *)malloc(early_length*sizeof(float));
  early_r2l=(float *)malloc(early_length*sizeof(float));
  early_r2r=(float *)malloc(early_length*sizeof(float));

  /* Erst mal Echos loeschen: */
  for (i=0;i<early_length;i++)
  early_l2l[i]=early_l2r[i]=early_r2l[i]=early_r2r[i]=0.0;

  dist_max=300.0*early_length/44100.0; /* 300 m/s Schallgeschwindigkeit */

  /* Echo vom LINKEN Kanal auf linkes/rechtes ear berechnen */

  for (x=-ceil(dist_max/(2*length));x<=ceil(dist_max/(2*length));x++)
  for (y=-ceil(dist_max/(2*width));y<=ceil(dist_max/(2*width));y++)
  for (z=-ceil(dist_max/(2*height));z<=ceil(dist_max/(2*height));z++)
  {
    xref=2*x*width;
    yref=2*y*length;
    zref=2*z*height;
    for (select=0;select<8;select++) /* vollstaendige Permutation */
    {
      if (select&1) x_pos=xref+leftposx;
      else x_pos=xref-leftposx;
      if (select&2) y_pos=yref+leftposy;
      else y_pos=yref-leftposy;
      if (select&4) z_pos=zref+leftposz;
      else z_pos=zref-leftposz;
      /* Jetzt steht die absolute Position der Quelle in ?_pos */

      /* Relative Position zum linken ear des Hoerers bestimmen: */
      x_pos-=listening_positionx;
      y_pos-=listening_positiony;
      z_pos-=listening_positionz;

      distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
      /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHz-> *106 */
      if ((distance*147)<early_length)
      {
        /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
        angle=atan(y_pos/x_pos);
        if (y_pos>0) /* Klang kommt aus vorderem Bereich: */
        { /* 0=left=>Verstaerkung=1 Pi=right=>Verstaerkung=0.22 (?) */
          angle+=3.1415926/2; weight = 1 - angle/4;
        }
      }
    }
  }
}
```

```

else          /* Klang kommt von hinten: */
{
    angle-=3.1415926/2;  weight= 1 + angle/4;
}
/* Early-echo gemaess angle und Entfernung gewichten: */
early_l2l[(int) (distance*147.)]+=weight/(pow(distance,3.1));
}

/* Relative Position zum rechten ear des Hoerers bestimmen: */
x_pos-=0.18; /* Kopf ist 18 cm breit */

distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
/* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHz-> *106 */
if ((distance*147)<early_length)
{
    /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
    angle=atan(y_pos/x_pos);
    if (y_pos>0) /* Klang kommt aus vorderem Bereich: */
    { /* 0=left=>Verstaerkung=1 Pi=right=>Verstaerkung=0.22 (?) */
        angle-=3.1415926/2;  weight = 1 + angle/4;
    }
    else          /* Klang kommt von hinten: */
    {
        angle+=3.1415926/2;  weight= 1 - angle/4;
    }
    /* Early-echo gemaess angle und Entfernung gewichten: */
    early_l2r[(int) (distance*147.)]+=weight/(pow(distance,3.1));
}
}
}

/* Echo vom RECHTEN Kanal auf linkes/rechtes ear berechnen */

for (x=-ceil(dist_max/(2*length));x<=ceil(dist_max/(2*length));x++)
for (y=-ceil(dist_max/(2*width));y<=ceil(dist_max/(2*width));y++)
for (z=-ceil(dist_max/(2*height));z<=ceil(dist_max/(2*height));z++)
{
    xref=2*x*width;
    yref=2*y*length;
    zref=2*z*height;
    for (select=0;select<8;select++) /* vollstaendige Permutation */
    {
        if (select&1) x_pos=xref+rightposx;
        else x_pos=xref-rightposx;
        if (select&2) y_pos=yref+rightposy;
        else y_pos=yref-rightposy;
        if (select&4) z_pos=zref+rightposz;
        else z_pos=zref-rightposz;
        /* Jetzt steht die absolute Position der Quelle in ?_pos */

        /* Relative Position zum linken ear des Hoerers bestimmen: */
        x_pos-=listening_positionx;
        y_pos-=listening_positiony;
        z_pos-=listening_positionz;

        distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
        /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHz-> *106 */
        if ((distance*147.)<early_length)
        {
            /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
            angle=atan(y_pos/x_pos);
            if (y_pos>0) /* Klang kommt aus vorderem Bereich: */
            { /* 0=left=>Verstaerkung=1 Pi=right=>Verstaerkung=0.22 (?) */
                angle+=3.1415926/2;  weight = 1 - angle/4;
            }
            else          /* Klang kommt von hinten: */
            {
                angle-=3.1415926/2;  weight= 1 + angle/4;
            }
            /* Early-echo gemaess angle und Entfernung gewichten: */
            early_r2l[(int) (distance*147.)]+=weight/(pow(distance,3.1));
        }
    }
}

/* Und jetzt Early-Echo zweiter Kanal auf LINKES ear berechnen */
x_pos-=0.18; /* Kopfwidth addieren */

distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
/* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHz-> *106 */
if ((distance*147)<early_length)
{
    /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */

```

```

    angle=atan(y_pos/x_pos);
    if (y_pos>0) /* Klang kommt aus vorderem Bereich: */
    { /* 0=left=>Verstaerkung=1 Pi=right=>Verstaerkung=0.22 (?) */
      angle-=3.1415926/2; weight = 1 + angle/4;
    }
    else /* Klang kommt von hinten: */
    {
      angle+=3.1415926/2; weight= 1 - angle/4;
    }
    /* Early-echo gemaeass angle und Entfernung gewichten: */
    early_r2r[(int) (distance*147.)]+=weight/(pow(distance,3.1));
  }
}

/* Und jetzt aus berechnetem Echo die ersten early_tap_num Werte holen */
/* Erst mal e's zuruecksetzen: */
for (i=0;i<early_tap_num;i++)
{
  e_values_l2l[i]=e_values_l2r[i]=0.0;
  e_delays_l2l[i]=e_delays_l2r[i]=0; /* Unangenehme Speicherzugriffe vermeiden */
  e_values_r2l[i]=e_values_r2r[i]=0.0;
  e_delays_r2l[i]=e_delays_r2r[i]=0;
}

/* und jetzt e_delays und e_values extrahieren: */
j=0;
normalizationl=normalizationr=0.0;
for(i=0;i<early_length;i++)
{
  if (early_l2l[i]!=0)
  {
    e_delays_l2l[j]=i;
    e_values_l2l[j]=early_l2l[i];
    normalizationl+=early_l2l[i];
    j++;
  }
  if (j==early_tap_num) break;
}
j=0;
for(i=0;i<early_length;i++)
{
  if (early_l2r[i]!=0)
  {
    e_delays_l2r[j]=i;
    e_values_l2r[j]=early_l2r[i];
    normalizationr+=early_l2r[i];
    j++;
  }
  if (j==early_tap_num) break;
}
j=0;
for(i=0;i<early_length;i++)
{
  if (early_r2l[i]!=0)
  {
    e_delays_r2l[j]=i;
    e_values_r2l[j]=early_r2l[i];
    normalizationl+=early_r2l[i];
    j++;
  }
  if (j==early_tap_num) break;
}
j=0;
for(i=0;i<early_length;i++)
{
  if (early_r2r[i]!=0)
  {
    e_delays_r2r[j]=i;
    e_values_r2r[j]=early_r2r[i];
    normalizationr+=early_r2r[i];
    j++;
  }
  if (j==early_tap_num) break;
}

/* groessere von beiden normalizationen verwenden: */
if (normalizationr>normalizationl) normalizationr=normalizationl;

for (j=0;j<early_tap_num;j++)
{
  e_values_l2l[j]/=normalizationr;

```

```
    e_values_l2r[j]/=normalizationr;
    e_values_r2l[j]/=normalizationr;
    e_values_r2r[j]/=normalizationr;
}
/* Ausgeben nur der l2l-Werte fuer schnelles Reverb */
printf("int e_delays[%d]={",early_tap_num);
for (j=0;j<(early_tap_num-1);j++)
    printf("%d, ",e_delays_l2l[j]);
printf("%d};\n\n",e_delays_l2l[j]);

printf("float e_values[%d]={",early_tap_num);
for (j=0;j<(early_tap_num-1);j++)
    printf("%0.4f, ",e_values_l2l[j]);
printf("%0.4f};\n\n",e_values_l2l[j]);
}
```

ECE320 project: Reverberation w/ parameter control from PC (click this to go back to the index)

References : Posted by Brahim Hamadicharef (project by Hua Zheng and Shobhit Jain)

Linked file : [rev.txt](#) (this linked file is included below)

Notes :

rev.asm
ECE320 project: Reverberation w/ parameter control from PC
Hua Zheng and Shobhit Jain
12/02/98 ~ 12/11/98
(se linked file)

Linked files

```
; rev.asm
;
; ECE320 project: Reverberation w/ parameter control from PC
;
; Hua Zheng and Shobhit Jain
; 12/02/98 ~ 12/11/98
;
; bugs fixed
; wrong co coefficients
; using current out point to calculate new in point
; r6 changed in set_dl (now changed to r4)
; initialize er delaylines to be 0 causes no output -- program memory
; periodic pops: getting garbage because external memory is configured to 16k

;=====
; Initialization
;=====

SAMPLERATE equ 48
    nolist
    include 'core302.asm'
    list

;-----
; Variables and storage setup
;-----

RESET equ 255 ; serial data reset character
n_para equ 29 ; number of parameters expected from serial port
delayline_number equ 12
delay_pointers equ 24
er_number equ 3
co_number equ 6
ap_number equ 3

    org x:$0
ser_data ds n_para ; location of serial data chunk
delays    ; default rev parameters: delay length
    dc 967    ; er1 $3c7
    dc 1867   ; er2 $74b
    dc 3359   ; er3 $d1f
    dc 2399   ; co1 $95f
    dc 2687   ; co2 $a7f
    dc 2927   ; co3 $b6f
    dc 3271   ; co4 $cc7
    dc 3457   ; co5 $d81
    dc 3761   ; co6 $eb1
    dc 293    ; ap1 $125
    dc 83     ; ap2 $53
    dc 29     ; ap3 $1d
coeffs    ; default rev parameters: coefficients
    dc 0.78   ; er1
    dc 0.635  ; er2
    dc 0.267  ; er3
; dc 0
; dc 0
; dc 0
    dc 0.652149 ; co1 $7774de
    dc 0.301209
    dc 0.615737 ; co2 $53799e
    dc 0.334735
    dc 0.586396 ; co3 $4ed078
    dc 0.362044
    dc 0.546884 ; co4 $4b0f06
    dc 0.399249
    dc 0.525135 ; co5 $4337a0
    dc 0.419943
    dc 0.493653 ; co6 $3f3006
```



```

dc 0.450179
dc 0.2 ; brightness
dc 0.4 ; mix
comb ds 6 ; one sample delay in comb filters
in ds 1 ; input sample
lpf ds 1 ; one sample delay in LPF
dl_p ds delay_pointers ; delayline in/out pointers

dl_er_1 equ $1000 ; max er delayline length 4096/48=85.3ms
dl_co_1 equ $1000 ; max co delayline length 85.3ms
dl_ap_1 equ $200 ; max ap delayline length 512/48=10.67ms

org p:$1000
dl_er1 dsm dl_er_1 ; er1 delayline
dl_er2 dsm dl_er_1 ; er2 delayline
dl_er3 dsm dl_er_1 ; er3 delayline
org y:$8000
dl_co1 dsm dl_co_1 ; co1 delayline
dl_co2 dsm dl_co_1 ; co2 delayline
dl_co3 dsm dl_co_1 ; co3 delayline
dl_co4 dsm dl_co_1 ; co4 delayline
dl_co5 dsm dl_co_1 ; co5 delayline
dl_co6 dsm dl_co_1 ; co6 delayline
org y:$F000
dl_ap1 dsm dl_ap_1 ; ap1 delayline
dl_ap2 dsm dl_ap_1 ; ap2 delayline
dl_ap3 dsm dl_ap_1 ; ap3 delayline

;-----
; Memory usage
;-----
; P:$0000 -- $0200 core file
; P:$0200 -- $0300 program
;
; X:$0000 -- $1BFF data 7168=149.3ms serial data, parameters, pointers
; Y:$0000 -- $1BFF data 7168=149.3ms not used
; P:$1000 -- $4FFF data 16384=341.3ms er(85*3=255ms)
; Y:$8000 -- $FFFF data 32768=682.67ms co(80*6=480ms) and ap(10*3=30ms)
;
; X,Y:$1C00 -- $1BFF reserved for system
;

;=====
; Main program
;=====

org p:$200
main

;-----
; Initialization
;-----

move #0,x0
; move #dl_er1,r0
; move #dl_er_1,y0
; do #er_number,clear_dl_er_loop
; rep y0
; movem x0,p:(r0)+
; nop
;clear_dl_er_loop

move #dl_co1,r0
move #dl_co_1,y0
do #co_number,clear_dl_co_loop
rep y0
move x0,y:(r0)+
nop
clear_dl_co_loop

move #dl_ap1,r0
move #dl_ap_1,y0
do #ap_number,clear_dl_ap_loop
rep y0
move x0,y:(r0)+
nop
clear_dl_ap_loop

move #comb,r0
rep #co_number
move x0,x:(r0)+ ; init comb filter states
move #lpf,r0
move x0,x:(r0) ; init LPF state

```

```

move #ser_data,r6 ; incoming data buffer pointer
move #(n_para-1),m6

jsr set_dl ; set all delayline pointers

; initialize SCI
movep #$0302,x:M_SCR ; R/T enable, INT disable
movep #$006a,x:M_SCCR ; SCP=0, CD=106 (9636 bps)
movep #7,x:M_PCRE

;-----
; Main loop
; Register usage
; r0: delayline pointer pointer
; r1: coefficients pointer
; r2: comb filter internal state pointer
; r4,r5: used in delayline subroutine
; r6: incoming buffer pointer
; a: output of each segment
;-----

main_loop:

move #dl_p,r0
move #coeffs,r1

waitdata r3,buflen,1
move x:(r3)+,a
move a,x:<in ; save input sample for mix

;-----
; Early reflection
;-----
; temp = in;
; for (int i=0; i<earlyRefNum; i++)
; {
; in = delayLineEarlyRef[c][i]->tick(in);
; temp += earlyRefCoeff[i] * in;
; }
; return temp;

move a,b ; b=temp=in
move #(dl_er_l-1),n6
do #er_number,er_loop
jsr use_dl_er
move a,y0 ; y0=a=in (delayline out)
move x:(r1)+,x0 ; x0=coeff
mac x0,y0,b ; b=temp
er_loop
asr #2,b,a
move b,a

;-----
; Comb filter
;-----
; float temp1 = 0., temp2;
; for (int i=0; i<combNum; i++)
; {
; temp2 = delayLineComb[c][i]->tick
; (in + combCoeff[i] * combLastOut[c][i]);
; combLastOut[c][i] = temp2+combDamp[i]*combLastOut[c][i];
; temp1 += temp2;
; }
; return temp1 / float(combNum);

move #comb,r2
move a,y1
clr b
move #(dl_co_l-1),n6
do #co_number,co_loop
move y1,a ; a=in
move x:(r1)+,x0 ; x0=coeff
move x:(r2),y0 ; y0=lastout
mac x0,y0,a x:(r1)+,x0 ; x0=damp
jsr use_dl
move a,x1 ; a=x1=temp2
mac x0,y0,a ; a=lastout
move a,x:(r2)+
add x1,b ; b=temp1
co_loop
; asr #2,b,a
move b,a

```

```

;-----
; All-pass filter
;-----
; float temp1, temp2;
; for (int i=0; i<allPassNum; i++)
; {
;   temp1 = delayLineAllPass[c][i]->lastOut();
;   temp2 = allPassCoeff[i] * (in - temp1);
;   delayLineAllPass[c][i]->tick(in + temp2);
;   in = temp1 + temp2;
; }
; return in;

move #1,n0
move #0.7,x1
move #(dl_ap_1-1),n6
do #ap_number,ap_loop
  move y:(r0+n0),x0 ; x0=temp1
  sub x0,a ; a=in-temp1
  move a,y0
  mpy x1,y0,b ; b=temp2
  add b,a ; a=in+temp2
  jsr use_dl
  add x0,b ; b=temp1+temp2
  move b,a ; a=in
ap_loop

;-----
; Brightness
;-----
; lastout = lastout + BW * (in - lastout);
move x:<lpf,b
sub b,a x:(r1)+,x0 ; a=in-lastout, x0=bri
move a,y0
mpy x0,y0,a
add b,a
move a,x:<lpf

;-----
; Mix
;-----
; out = (1-mix)*in + mix*out = in + mix * (out - in);
move x:<in,y0 ; y0=in
sub y0,a x:(r1)+,x0 ; a=out-in, x0=mix
move a,y1 ; y1=out-in
mpy x0,y1,b y0,a ; b=mix*(out-in), a=in
add b,a

;-----
; Spit out
;-----
move a,y:(r3)+
move a,y:(r3)+
move (r3)+

;-----
; Get parameters and reformat them
;-----
jclr #2,x:M_SSR,main_loop ; do nothing if no new data arrived

movep x:M_SRXL,a ; get next 8-bit word from SCI

cmp #RESET,a
jeq reformat_data ; if it's RESET, then reformat data

move a,x:(r6)+ ; save one incoming data for later reformatting

jmp main_loop

reformat_data:
; order of parameters:
; er1 delay, er1 coeff, er2 ..., er3 ...
; col delay, coeff_c, coeff_d, co2 ... , ... , co6
; ap1 delay, ap2, ap3
; brightness
; mix

move #ser_data,r0
move #delays,r1
move #coeffs,r2

do #3,format_er_loop

```

```

    move x:(r0)+,a ; er delay
    asr #20,a,a
    ; max delay 4096=2^12, max value 256=2^8, scale 256/4096=2^-4
    move a0,x:(r1)+
    move x:(r0)+,a ; er coeff
    asr #9,a,a
    move a0,x:(r2)+
format_er_loop

    move #>$000001,x0
    do #6,format_co_loop
        move x:(r0)+,a ; co delay
        asr #20,a,a ; max delay 4096=2^12
        move a0,a1
;try this: asl #4,a,a
        or x0,a
        move a1,x:(r1)+
        move x:(r0)+,a ; co coeff
        asr #9,a,a
        move a0,x:(r2)+
        move x:(r0)+,a ; co damping
        asr #9,a,a
        move a0,x:(r2)+
format_co_loop

    do #3,format_ap_loop
        move x:(r0)+,a ; ap delay
        asr #23,a,a ; max delay 528=2^9
        move a0,a1
        or x0,a
        move a1,x:(r1)+
format_ap_loop

    jsr set_dl

    move x:(r0)+,a ; brightness
    asr #9,a,a
    move a0,x:(r2)+

    move x:(r0)+,a ; mix
    asr #9,a,a
    move a0,x:(r2)+

    jmp main_loop

;=====
; Set all delayline length subroutine
; IN: nothing
; OUT: out pointer UNCHANGED
; in pointer = out + length e.g. (#(dl_p+3))=(#(dl_p+4))+x:(r4)
; r4=r4+1: next delay length
;=====
set_dl:
    move #(dl_p+1),r5 ; first out pointer
    move #dl_er1,r4
    move r4,x:(r5)+ ; initial out point=delayline starting address
    move (r5)+
    move #dl_er2,r4
    move r4,x:(r5)+
    move (r5)+
    move #dl_er3,r4
    move r4,x:(r5)+
    move (r5)+
    move #dl_co1,r4
    move r4,x:(r5)+
    move (r5)+
    move #dl_co2,r4
    move r4,x:(r5)+
    move (r5)+
    move #dl_co3,r4
    move r4,x:(r5)+
    move (r5)+
    move #dl_co4,r4
    move r4,x:(r5)+
    move (r5)+
    move #dl_co5,r4
    move r4,x:(r5)+
    move (r5)+
    move #dl_co6,r4
    move r4,x:(r5)+
    move (r5)+
    move #dl_ap1,r4
    move r4,x:(r5)+

```

```

move (r5)+
move #dl_ap2,r4
move r4,x:(r5)+
move (r5)+
move #dl_ap3,r4
move r4,x:(r5)+
move (r5)+

move #delays,r4 ; delayline length
move #(dl_p+1),r5 ; first out pointer
move #2,n5
do #delayline_number,set_dl_loop
  move x:(r4)+,x0 ; x0=length
  move x:(r5)-,a ; a=out pointer
  add x0,a
  move a,x:(r5)+ ; in=out+length
  move (r5)+n5 ; next out pointer
set_dl_loop
rts

;=====
; Access delayline subroutine
; IN: in and out pointers in r4,r5
; modulo (delayline length-1) in n6
; input sample in a
; OUT: in and out pointers modulo incremented
; output sample in a
;=====
; inputs[inPoint++] = sample;
; inPoint &= lengthm1;
; lastOutput = inputs[outPoint++];
; outPoint &= lengthm1;
; return lastOutput;
use_dl:
  move n6,m4
  move n6,m5
  move x:(r0)+,r4 ; in point
  move x:(r0)-,r5 ; out point
  move a,y:(r4)+ ; queue in
  move y:(r5)+,a ; queue out
  move r4,x:(r0)+ ; modulo incremented in point
  move r5,x:(r0)+ ; modulo incremented out point
  rts

use_dl_er: ; using P memory
  move n6,m4
  move n6,m5
  move x:(r0)+,r4 ; in point
  move x:(r0)-,r5 ; out point
  movem a,p:(r4)+ ; queue in
  movem p:(r5)+,a ; queue out
  move r4,x:(r0)+ ; modulusly incremented in point
  move r5,x:(r0)+ ; modulusly incremented out point
  rts

```

[fold back distortion](#) (click this to go back to the index)

Type : distortion

References : Posted by hellfire[AT]upb[DOT]de

Notes :

a simple fold-back distortion filter.

if the signal exceeds the given threshold-level, it mirrors at the positive/negative threshold-border as long as the signal lies in the legal range (-threshold..+threshold).

there is no range limit, so inputs doesn't need to be in -1..+1 scale.

threshold should be >0

depending on use (low thresholds) it makes sense to rescale the input to full amplitude

performs approximately the following code
(just without the loop)

```
while (in>threshold || in<-threshold)
{
  // mirror at positive threshold
  if (in>threshold) in= threshold - (in-threshold);
  // mirror at negative threshold
  if (in<-threshold) in= -threshold + (-threshold-in);
}
```

Code :

```
float foldback(float in, float threshold)
{
  if (in>threshold || in<-threshold)
  {
    in= fabs(fabs(fmod(in - threshold, threshold*4)) - threshold*2) - threshold;
  }
  return in;
}
```

[Guitar feedback](#) (click this to go back to the index)

[References](#) : Posted by Sean Costello

Notes :

It is fairly simple to simulate guitar feedback with a simple Karplus-Strong algorithm (this was described in a CMJ article in the early 90's):

Code :

```
Run the output of the Karplus-Strong delay lines into a nonlinear shaping function for distortion (i.e. 6 parallel delay lines for 6 strings, going into 1 nonlinear shaping function that simulates an overdriven amplifier, fuzzbox, etc.);
```

```
Run part of the output into a delay line, to simulate the distance from the amplifier to the "strings";
```

```
The delay line feeds back into the Karplus-Strong delay lines. By controlling the amount of the output fed into the delay line, and the length of the delay line, you can control the intensity and pitch of the feedback note.
```

Comments

from : ignatz [[a t]] webmail.co.za

comment : any C snippet code ???

thx

from : lavirosa21 [[a t]] aol.com

comment : Are you Sean Costello the Blues Musician/Guitarist?

Lenora

lavirosa21@aol.com

from : hagenkaiser [[a t]] gmx.de

comment : what is a Karplus-Strong-Delay??

from : brainslayer [[a t]] braincontrol.org

comment : a physical modelling algorithm

[Lo-Fi Crusher](#) (click this to go back to the index)

Type : Quantizer / Decimator with smooth control

References : Posted by David Lowenfels

Notes :

Yet another bitcrusher algorithm. But this one has smooth parameter control.

Normfreq goes from 0 to 1.0; (freq/samplerate)

Input is assumed to be between 0 and 1.

Output gain is greater than unity when bits < 1.0;

Code :

```
function output = crusher( input, normfreq, bits );
    step = 1/2^(bits);
    phasor = 0;
    last = 0;

    for i = 1:length(input)
        phasor = phasor + normfreq;
        if (phasor >= 1.0)
            phasor = phasor - 1.0;
            last = step * floor( input(i)/step + 0.5 ); %quantize
        end
        output(i) = last; %sample and hold
    end
end
```

Comments

from : kk791231 [[a t]] hotmail.com

comment : what's the "bits" here? I tried to run the code, it seems it's a dead loop here, can not figure out why

from : dfl

comment : bits goes from 1 to 16

Most simple and smooth feedback delay (click this to go back to the index)

Type : Feedback delay

References : Posted by antiprosynthesis[AT]hotmail[DOT]com

Notes :

fDlyTime = delay time parameter (0-1)

i = input index

j = delay index

Code :

```
if( i >= SampleRate )
    i = 0;

j = i - (fDlyTime * SampleRate);

if( j < 0 )
    j += SampleRate;

Output = DlyBuffer[ i++ ] = Input + (DlyBuffer[ j ] * fFeedback);
```

Comments

from : antiprosynthesis [[a t]] hotmail.com

comment : This algo didn't seem to work on testing again, just change:

Output = DlyBuffer[i++] = Input + (DlyBuffer[j] * fFeedback);

to

Output = DlyBuffer[i] = Input + (DlyBuffer[j] * fFeedback);

i++;

and it will work fine.

from : antiprosynthesis [[a t]] hotmail.com

comment : Here's a more clear source. both BufferSize and MaxDlyTime are amounts of samples. BufferSize should best be 2*MaxDlyTime to have proper sound.

-
if(i >= BufferSize)
 i = 0;

j = i - (fDlyTime * MaxDlyTime);

if(j < 0)
 j += BufferSize;

Output = DlyBuffer[i] = Input + (DlyBuffer[j] * fFeedback);

i++;

from : amalinaharif [[a t]] yahoo.com

comment : hi,can anyone help me with the c code for flanging effect using C6711 DSK board??

Most simple static delay (click this to go back to the index)

Type : Static delay

References : Posted by antiprosynthesis[AT]hotmail[DOT]com

Notes :
This is the most simple static delay (just delays the input sound an amount of samples). Very useful for newbies also probably very easy to change in a feedback delay (for comb filters for example).

Note: fDlyTime is the delay time parameter (0 to 1)

i = input index
j = output index

```
Code :
if( i >= SampleRate )
    i = 0;

DlyBuffer[ i ] = Input;

j = i - (fDlyTime * SampleRate);

i++;

if( j < 0 )
    j = SampleRate + j;

Output = DlyBuffer[ j ];
```

Comments

from : antiprosynthesis [[a t]] hotmail.com
comment : Another note: The delay time will be 0 if fDlyTime is 0 or 1.

from : anonymous [[a t]] fake.org
comment : I think you should be careful with mixing floats and integers that way (error-prone, slow float-to-int conversions, etc).

This should also work (haven't checked, not best way of doing it):

... (initializing) ..

```
float numSecondsDelay = 0.3f;
int numSamplesDelay_ = (int)(numSecondsDelay * sampleRate); // maybe want to round to an integer instead of truncating..
```

```
float *buffer_ = new float[2*numSamplesDelay];
```

```
for (int i = 0; i < numSamplesDelay_; ++i)
{
    buffer_[i] = 0.f;
}
```

```
int readPtr_ = 0;
int writePtr_ = numSamplesDelay_;
```

... (processing) ...

```
for (i = each sample)
{
    buffer_[writePtr_] = input[i];
    output[i] = buffer_[readPtr_];

    ++writePtr_;
    if (writePtr_ >= 2*numSamplesDelay_)
        writePtr_ = 0;

    ++readPtr_;
    if (readPtr_ >= 2*numSamplesDelay_)
        readPtr_ = 0;
}
```

from : xeeton[at]gmail[dot]com
comment : I may be missing something, but I think it gets a little simpler than the previous two examples. The difference in result is that actual delay will be 1 if d is 0 or 1.

i = input/output index
d = delay (in samples)

Code:

```
out = buffer[i];
buffer[i] = in;
i++;
if(i >= delay) i = 0;
```

from : xeeton[at]gmail[dot]com
comment : or even in three lines...

```
out = buffer[i];  
buffer[i++] = in;  
if(i >= delay) i = 0;
```

from : gravitate [[a t]] dj-gravitate.co.uk

comment : The only problem with this implementation, is that it is not really an audio effect! all this will do is to delay the input signal by a given number of samples! ...why would you ever want to do that? ...this would only ever work if you had a DSP and speakers both connected to the audio source and run them at the same time, so the speakers would be playing the original source and the DSP containing the delayed source connected to another set of speakers! this is not really an audio effect!

...Here is a pseudo code example of a delay effect that will mix both the original sound with the delayed sound:

Pseudo Code implementation for simple delay:

- This implementation will put the current audio signal to the left channel and the delayed audio signal to the right channel.
this is suitable for any stereo codec!

```
delay_function {  
  
    left_channel // for stereo left  
    right_channel // for stereo right  
    mono        // mono representation of stereo input  
    delay_time  // amount of time to delay input  
    counter = 0 // counter  
  
    //setup an array that is the same length as the maximum delay time:  
    delay_array[max delay time] // array containing delayed data  
  
    // convert stereo to mono:  
    (left_channel + right_channel) / 2  
  
    // initialise time to delay signal - maybe input from user  
    delay_time = x  
  
    if (delay_time = 0){  
        left_out = mono  
        right_out = mono  
    }  
    else {  
        // put current input data to left channel:  
        left_out = mono  
        // put oldest delayed input data to right channel:  
        right_out = delay_array[index]  
  
        // overwrite with newest input:  
        delay_array[index] = mono;  
  
        // is index at end of delay buffer? if not increment, else set to zero  
        if (index < delay_time) index++  
        else index = 0  
    }  
}
```

from : nobody [[a t]] nowhere.com

comment : I've need a delay before, many a time.

- 1) To compensate for a delay in another effect.
- 2) To manually build a high-pass delay.
- 3) In other ways for interesting effects in a modular host.
- 4) Building block for physical modeling or filtering.

As long as you have multiple ins and/or outs, you want something like this.

from : kibibu [[a t]] gmail.com

comment : With (-1 < feedback < 1):

```
out = buffer[i];  
buffer[i++] = (out * feedback) + in;  
if(i >= delay) i = 0;
```

Parallel combs delay calculation (click this to go back to the index)

References : Posted by Juhana Sadeharju (kouhia[AT]nic[DOT]funet[DOT]fi)

Notes :

This formula can be found from a patent related to parallel combs structure. The formula places the first echoes coming out of parallel combs to uniformly distributed sequence. If T_1, \dots, T_n are the delay lines in increasing order, the formula can be derived by setting $T_{(k-1)}/T_k = \text{Constant}$ and $T_n/(2*T_1) = \text{Constant}$, where $2*T_1$ is the echo coming just after the echo T_n .

I figured this out myself as it is not told in the patent. The formula is not the best which one can come up. I use a search method to find echo sequences which are uniform enough for long enough time. The formula is uniform for a short time only.

The formula doesn't work good for series allpass and FDN structures, for which a similar formula can be derived with the same idea. The search method works for these structures as well.

[Phaser code](#) (click this to go back to the index)

[References](#) : Posted by Ross Bencina

[Linked file](#) : [phaser.cpp](#) (this linked file is included below)

[Notes](#) :
(see linked file)

[Comments](#)

[from](#) : dj_ikke [[a t]] hotmail.com

[comment](#) : What range should be the float parameter of the Update function? From -1 to 1, 0 to 1 or -32768 to 32767?

[from](#) : rossb [[a t]] audiomulch.com

[comment](#) : It doesn't matter what the range of the parameter to the Update function is. Usually in a floating point signal chain you would use -1 to 1, but anything else will work just as well.

[from](#) : askywhale [[a t]] free.fr

[comment](#) : Please what is the usual range of frequencies ?

[from](#) : Christian [[a t]] savioursofsoul.de

[comment](#) : Delphi / Object Pascal Translation:

unit Phaser;

// Still not efficient, but avoiding denormalisation.

interface

type

TAllPass=class(TObject)

private

fDelay : Single;

fA1, fZM1 : Single;

fSampleRate : Single;

procedure SetDelay(v:Single);

public

constructor Create;

destructor Destroy; override;

function Process(const x:single):single;

property SampleRate : Single read fSampleRate write fSampleRate;

property Delay: Single read fDelay write SetDelay;

end;

TPhaser=class(TObject)

private

fZM1 : Single;

fDepth : Single;

fLFOInc : Single;

fLFOPhase : Single;

fFeedBack : Single;

fRate : Single;

fMinimum: Single;

fMaximum: Single;

fMin: Single;

fMax: Single;

fSampleRate : Single;

fAllpassDelay: array[0..5] of TAllPass;

procedure SetSampleRate(v:Single);

procedure SetMinimum(v:Single);

procedure SetMaximum(v:Single);

procedure SetRate(v:Single);

procedure Calculate;

public

constructor Create;

destructor Destroy; override;

function Process(const x:single):single;

property SampleRate : Single read fSampleRate write SetSampleRate;

property Depth: Single read fDepth write fDepth; //0..1

property Feedback: Single read fFeedBack write fFeedBack; // 0..<1

property Minimum: Single read fMin write SetMinimum;

property Maximum: Single read fMax write SetMaximum;

property Rate: Single read fRate write SetRate;

end;

implementation

uses DDSPUtils;

const kDenorm=1E-25;

constructor TAllpass.Create;

begin

```

inherited;
fA1:=0;
fZM1:=0;
end;

destructor TAllpass.Destroy;
begin
inherited;
end;

function TAllpass.Process(const x:single):single;
begin
Result:=x*-fA1+fZM1;
fZM1:=Result*fA1+x;
end;

procedure TAllpass.SetDelay(v:Single);
begin
fDelay:=v;
fA1:=(1-v)/(1+v);
end;

constructor TPhaser.Create;
var i : Integer;
begin
inherited;
fSampleRate:=44100;
fFeedBack:=0.7;
fLFOPhase:=0;
fDepth:=1;
fZM1:=0;
Minimum:=440;
Maximum:=1600;
Rate:=5;
for i:=0 to Length(fAllpassDelay)-1
do fAllpassDelay[i]:=TAllpass.Create;
end;

destructor TPhaser.Destroy;
var i : Integer;
begin
for i:=0 to Length(fAllpassDelay)-1
do fAllpassDelay[i].Free;
inherited;
end;

procedure TPhaser.SetRate(v:Single);
begin
fLFOInc:=2*Pi*(v/SampleRate);
end;

procedure TPhaser.Calculate;
begin
fMin:= fMinimum / (fSampleRate/2);
fMax:= fMaximum / (fSampleRate/2);
end;

procedure TPhaser.SetMinimum(v:Single);
begin
fMinimum:=v;
Calculate;
end;

procedure TPhaser.SetMaximum(v:Single);
begin
fMaximum:=v;
Calculate;
end;

function TPhaser.Process(const x:single):single;
var d: Single;
i: Integer;
begin
//calculate and update phaser sweep lfo...
d := fMin + (fMax-fMin) * ((sin( fLFOPhase )+1)/2);
fLFOPhase := fLFOPhase + fLFOInc;
if fLFOPhase>=Pi*2
then fLFOPhase:=fLFOPhase-Pi*2;

//update filter coeffs
for i:=0 to 5 do fAllpassDelay[i].Delay:=d;

//calculate output
Result:= fAllpassDelay[0].Process(
fAllpassDelay[1].Process(

```

```

    fAllpassDelay[2].Process(
    fAllpassDelay[3].Process(
    fAllpassDelay[4].Process(
    fAllpassDelay[5].Process(kDenorm + x + fZM1 * fFeedBack ))));
fZM1:=tanh2a(Result);

```

```

Result:=tanh2a(1.4*(x + Result * fDepth));
end;

```

```

procedure TPhaser.SetSampleRate(v:Single);

```

```

begin
    fSampleRate:=v;
end;

```

```

end.

```

from : Christian [[a t]] savioursofsoul.de

comment : Ups, forgot to remove my special, magic ingredients "tanh2a(1.4*(". It's just to make the sound even warmer.

The frequency range i used for Minimum and Maximum is 0..22000. But I believe there is still an error in that formula. The input range doesn't matter (if you remove my special ingredient), because it is a linear system.

from : thaddy [[a t]] thaddy.com

comment : I thought I already posted this but here's my interpretation for Delphi and KOL. The reason I repost this, is that it is rather efficient and has no denormal problems.

```

unit Phaser;

```

```

{

```

```

    Unit: Phaser

```

```

    purpose: Phaser is a six stage phase shifter, intended to reproduce the
             sound of a traditional analogue phaser effect.

```

```

    Author: Thaddy de Koning, based on a musicdsp.pdf C++ Phaser by
            Ross Bencina.http://www.musicdsp.org/musicdsp.pdf

```

```

    Copyright: This version (c) 2003, Thaddy de Koning
              Copyrighted Freeware

```

```

    Remarks: his implementation uses six first order all-pass filters in
             series, with delay time modulated by a sinusoidal.
             This implementation was created to be clear, not efficient.
             Obvious modifications include using a table lookup for the lfo,
             not updating the filter delay times every sample, and not
             tuning all of the filters to the same delay time.

```

```

        It sounds sensationally good!

```

```

}

```

```

interface

```

```

uses Kol, AudioUtils, SimpleAllpass;

```

```

type

```

```

    PPhaser = ^TPhaser;

```

```

    TPhaser = object(Tobj)

```

```

    private

```

```

        FSamplerate: single;

```

```

        FFeedback: single;

```

```

        FlfoPhase: single;

```

```

        FDepth: single;

```

```

        FOldOutput: single;

```

```

        FMinDelta: single;

```

```

        FMaxDelta: single;

```

```

        FLfoStep: single;

```

```

        FAllpDelays: array[0..5] of PAllpassdelay;

```

```

        FLowFrequency: single;

```

```

        FHighFrequency: single;

```

```

        procedure SetRate(TheRate: single); // cps

```

```

        procedure SetFeedback(TheFeedback: single); // 0 -> <1.

```

```

        procedure SetDepth(TheDepth: single);

```

```

        procedure SetHighFrequency(const Value: single);

```

```

        procedure SetLowFrequency(const Value: single); // 0 -> 1.

```

```

        procedure SetRange(LowFreq, HighFreq: single); // Hz

```

```

    public

```

```

        destructor Destroy; virtual;

```

```

        function Process(inSamp: single): single;

```

```

        property Rate: single write setrate;//In Cycles per second

```

```

        property Depth: single read Fdepth write setdepth;//0.. 1

```

```

        property Feedback: single read FFeedback write setfeedback; //0..< 1

```

```

        property Samplerate: single read Fsamplerate write Fsamplerate;

```

```

        property LowFrequency: single read FLowFrequency write SetLowFrequency;

```

```

        property HighFrequency: single read FHighFrequency write SetHighFrequency;

```

```

    end;

```

```

function NewPhaser: PPhaser;

```

implementation

```
{ TPhaser }
function NewPhaser: PPhaser;
var
  i: integer;
begin
  New(Result, Create);
  with Result^ do
  begin
    FsampleRate := 44100;
    FFeedback := 0.7;
    FlfoPhase := 0;
    Fdepth := 1;
    FOldOutput := 0;
    setrange(440,1720);
    setrate(0.5);
    for i := 0 to 5 do
      FAllpDelays[i] := NewAllpassDelay;
    end;
  end;
end;

destructor TPhaser.Destroy;
var
  i: integer;
begin
  for i := 5 downto 0 do FAllpDelays[i].Free;
  inherited;
end;

procedure TPhaser.SetDepth(TheDepth: single); // 0 -> 1.
begin
  Fdepth := TheDepth;
end;

procedure TPhaser.SetFeedback(TheFeedback: single); // 0..1;
begin
  FFeedback := TheFeedback;
end;

procedure TPhaser.SetRange(LowFreq, HighFreq: single);
begin
  FMinDelta := LowFreq / (FsampleRate / 2);
  FMaxDelta := HighFreq / (FsampleRate / 2);
end;

procedure TPhaser.SetRate(TheRate: single);
begin
  FLfoStep := 2 * _PI * (TheRate / FsampleRate);
end;

const
  _1:single=1;
  _2:single=2;
function TPhaser.Process(inSamp: single): single;
var
  Delaytime, Output: single;
  i: integer;
begin
  //calculate and Process phaser sweep lfo...
  Delaytime := FMinDelta + (FMaxDelta - FMinDelta) * ((sin(FlfoPhase) + 1) / 2);
  FlfoPhase := FlfoPhase + FLfoStep;
  if (FlfoPhase >= _PI * 2) then
    FlfoPhase := FlfoPhase - _PI * 2;
  //Process filter coeffs
  for i := 0 to 5 do
    FAllpDelays[i].setdelay(Delaytime);
  //calculate output
  Output := FAllpDelays[0].Process(FAllpDelays[1].Process
    (FAllpDelays[2].Process(FAllpDelays[3].Process(FAllpDelays[4].Process
      (FAllpDelays[5].Process(inSamp + FOldOutput * FFeedback))))));
  FOldOutput := Output;
  Result := kDenorm + inSamp + Output * Fdepth;
end;

procedure TPhaser.SetHighFrequency(const Value: single);
begin
  FHighFrequency := Value;
  setrange(FlowFrequency, FHighFrequency);
end;

procedure TPhaser.SetLowFrequency(const Value: single);
```



```
begin
  FLowFrequency := Value;
  setrange(FlowFrequency, FHighFrequency);
end;
```

end.

```

from : thaddy [ [ a t ] ] thaddy.com
comment : And here the allpass:
unit SimpleAllpass;
{
  Unit: SimpleAllpass
  purpose: Simple allpass delay for creating reverbs and phasing/flanging
  Author:
  Copyright:
  Remarks:
}
interface
uses kol, audioutils;
```

```
type
PAllpassDelay = ^TAllpassDelay;
TAllpassdelay = object(Tobj)
protected
  Fa1,
  Fzm1: single;
public
  procedure SetDelay(delay: single); //sample delay time
  function Process(inSamp: single): single;
end;
```

```
function NewAllpassDelay: PAllpassDelay;
```

implementation

```
function NewAllpassDelay: PAllpassDelay;
```

```
begin
  New(Result, Create);
  with Result^ do
  begin
    Fa1 := 0;
    Fzm1 := 0;
  end;
end;
```

```
function TallpassDelay.Process(Insamp: single): single;
```

```
begin
  Result := kDenorm+inSamp * -Fa1 + Fzm1;
  Fzm1 := Result * Fa1 + inSamp + kDenorm;
end;
```

```
procedure TAllpassDelay.setdelay(delay: single); // In sample time
```

```
begin
  Fa1 := (1 - delay) / (1 + delay);
end;
```

end.

```
from : Christian [ [ a t ] ] savioursofsoul.de
```

comment : You'll get a good performance boost by combining the 6 allpasses to one and rewriting that one to FPU code. Heavy speed increase AND you can make the number of allpasses variable as well.

This would look similar to this:

```
function TMasterAllpass.Process(const x:single):single;
```

```
var a : array[0..1] of Single;
```

```
  b : Single;
```

```
  i : Integer;
```

```
begin
  a[0]:=x*fA1+fY[0];
```

```
  b:=a[0]*fA1;
```

```
  fY[0]:=b-x;
```

```
i:=0;
```

```
while i<fStages do
```

```
  begin
```

```
    a[1]:=b-fY[i+1];
```

```
    b:=a[1]*fA1;
```

```
    fY[i+1]:=a[0]-b;
```

```
    a[0]:=b-fY[i+2];
```

```
    b:=a[0]*fA1;
```

```
    fY[i+2]:=a[1]-b;
```

```
    Inc(i,2);
```

```
end;

a[1]:=b-fY[5];
b:=a[1]*fA1;
fY[5]:=a[0]-b;
Result:=a[1];
end;
```

Now all you have to do is crawling into the FPU registers...

```
from : thaddy [[ a t ]] thaddy.com
comment : Point taken ;)
Maybe we should combine all the stuff ;)
Btw:
It's lots of fun working from each others code, don't you think?
```

Linked files

```
/*
Date: Mon, 24 Aug 1998 07:02:40 -0700
Reply-To: music-dsp
Originator: music-dsp@shoko.calarts.edu
Sender: music-dsp
Precedence: bulk
From: "Ross Bencina" <rbencina@hotmail.com>
To: Multiple recipients of list <music-dsp>
Subject: Re: Phaser revisited [code included]
X-Comment: Music Hackers Unite! http://shoko.calarts.edu/~glmrboy/musicdsp/music-dsp.html
Status: RO
```

Hi again,

Thanks to Chris Townsend and Marc Lindahl for their helpful contributions. I now have a working phaser and it sounds great! It seems my main error was using a 'sub-sampled' all-pass reverberator instead of a single sample all-pass filter [what was I thinking? :)].

I have included a working prototype (C++) below for anyone who is interested. My only remaining doubt is whether the conversion from frequency to delay time [$_dmin = fMin / (SR/2.f);$] makes any sense what-so-ever.

Ross B.

```
*/
/*
```

```
class Phaser
implemented by: Ross Bencina <rossb@kagi.com>
date: 24/8/98
```

Phaser is a six stage phase shifter, intended to reproduce the sound of a traditional analogue phaser effect. This implementation uses six first order all-pass filters in series, with delay time modulated by a sinusoidal.

This implementation was created to be clear, not efficient. Obvious modifications include using a table lookup for the lfo, not updating the filter delay times every sample, and not tuning all of the filters to the same delay time.

Thanks to:
The nice folks on the music-dsp mailing list, including...
Chris Townsend and Marc Lindahl

...and Scott Lehman's Phase Shifting page at harmony central:
http://www.harmony-central.com/Effects/Articles/Phase_Shifting/

```
*/
```

```
#define SR (44100.f) //sample rate
#define F_PI (3.14159f)
```

```
class Phaser{
public:
Phaser() //initialise to some usefull defaults...
: _fb( .7f )
, _lfoPhase( 0.f )
, _depth( 1.f )
, _zml( 0.f )
{
Range( 440.f, 1600.f );
Rate( .5f );
```

```

}

void Range( float fMin, float fMax ){ // Hz
    _dmin = fMin / (SR/2.f);
    _dmax = fMax / (SR/2.f);
}

void Rate( float rate ){ // cps
    _lfoInc = 2.f * F_PI * (rate / SR);
}

void Feedback( float fb ){ // 0 -> <1.
    _fb = fb;
}

void Depth( float depth ){ // 0 -> 1.
    _depth = depth;
}

float Update( float inSamp ){
    //calculate and update phaser sweep lfo...
    float d = _dmin + (_dmax-_dmin) * ((sin( _lfoPhase ) +
1.f)/2.f);
    _lfoPhase += _lfoInc;
    if( _lfoPhase >= F_PI * 2.f )
        _lfoPhase -= F_PI * 2.f;

    //update filter coeffs
    for( int i=0; i<6; i++ )
        _alps[i].Delay( d );

    //calculate output
    float y = _alps[0].Update(
        _alps[1].Update(
            _alps[2].Update(
                _alps[3].Update(
                    _alps[4].Update(
                        _alps[5].Update( inSamp + _zml * _fb ))))));
    _zml = y;

    return inSamp + y * _depth;
}

private:
class AllpassDelay{
public:
    AllpassDelay()
        : _a1( 0.f )
        , _zml( 0.f )
        {}

    void Delay( float delay ){ //sample delay time
        _a1 = (1.f - delay) / (1.f + delay);
    }

    float Update( float inSamp ){
        float y = inSamp * -_a1 + _zml;
        _zml = y * _a1 + inSamp;

        return y;
    }
private:
    float _a1, _zml;
};

AllpassDelay _alps[6];

float _dmin, _dmax; //range
float _fb; //feedback
float _lfoPhase;
float _lfoInc;
float _depth;

float _zml;
};

```

Polynomial Waveshaper (click this to go back to the index)

Type : (discrete harmonics)

References : Posted by Christian[AT]savioursofsoul[DOT]de

Notes :
The following code will describe how to excite discrete harmonics and only these harmonics. A simple polynomial waveshaper for processing the data is included as well. However the code don't claim to be optimized. Using a horner scheme with precalculated coefficients should be your choice here.

Also remember to oversample the data (optimal in the order of the harmonics) to have them alias free.

Code :
We assume the input is a sinewave (works for any input signal, but this makes everything more clear). Then we have $x = \sin(a)$

the first harmonic is plain simple (using trigonometric identities):

$$\cos(2*a) = \cos^2(a) - \sin^2(a) = 1 - 2 \sin^2(a)$$

using the general trigonometric identities:

$$\sin(x + y) = \sin(x)*\cos(y) + \sin(y)*\cos(x)$$

$$\cos(x + y) = \cos(x)*\cos(y) - \sin(y)*\sin(x)$$

together with some math, you can easily calculate: $\sin(3x)$, $\cos(4x)$, $\sin(5x)$, and so on...

Here's how the resulting waveshaper may look like:

```
// o = output, i = input
o = fPhase[1]* i * fGains[0]+
  fPhase[1]*( 2*i*i - 1 ) * fGains[1]+
  fPhase[2]*( 4*i*i*i - 3*i ) * fGains[2]+
  fPhase[3]*( 8*i*i*i*i + 1 ) * fGains[3]-
  fPhase[4]*( 16*i*i*i*i*i - 20*i*i*i*i + 5 * i ) * fGains[4]+
  fPhase[5]*( 32*i*i*i*i*i*i - 48*i*i*i*i*i + 18 * i*i - 1 ) * fGains[5]-
  fPhase[6]*( 64*i*i*i*i*i*i*i - 112*i*i*i*i*i*i + 56 * i*i*i - 7 * i ) * fGains[6]+
  fPhase[7]*(128*i*i*i*i*i*i*i*i - 256*i*i*i*i*i*i*i + 160 * i*i*i*i - 32 * i*i + 1 ) * fGains[7];
```

fPhase[...] is the sign array and fGains[...] is the gain factor array.

P.S.: I don't want to see a single comment about the fact that the code above is unoptimized. I know that!

Comments

from : Christian [[a t]] savioursofsoul.de

comment : Do'oh. You're right. Once more I got fooled by the way of my measurement. That explains a lot of things...

Thanks for the clarification!

from : Christian [[a t]] savioursofsoul.de

comment : Btw. the coefficients follow the chebyshev polynomials. Just in case you wonder about the logic behind. Maybe we can call it chebyshev waveshaper from now on...

from : Christian [[a t]] savioursofsoul.de

comment : Here's the more math like version:

```
// o = output, i = input
o = fPhase[1]* i * fGains[0]+
  fPhase[1]*( 2*i^2 - 1 ) * fGains[1]+
  fPhase[2]*( 4*i^3 - 3*i ) * fGains[2]+
  fPhase[3]*( 8*i^4 - 8*i^2 + 1 ) * fGains[3]-
  fPhase[4]*( 16*i^5 - 20*i^3 + 5*i ) * fGains[4]+
  fPhase[5]*( 32*i^6 - 48*i^4 + 18 * i^2 - 1 ) * fGains[5]-
  fPhase[6]*( 64*i^7 - 112*i^5 + 56 * i^3 - 7 * i ) * fGains[6]+
  fPhase[7]*(128*i^8 - 256*i^6 + 160 * i^4 - 32 * i^2 + 1 ) * fGains[7];
```

from : fuzzpilz [[a t]] gmx.net

comment : Actually, this *doesn't* work in the way described on any input, only on single sinusoids of amplitude 1. It's nonlinear - $(a+b)^n$ is not the same thing as a^n+b^n , nor are $(a*b)^n$ and $a*(b^n)$. Even just a sum of two sinusoids or a single sinusoid of a different amplitude breaks the chosen-harmonics-only thing.

from : physicstrick [[a t]] optonline.net

comment : I played with this idea for a while yesterday to no avail before discovering this post tonight. I thought I could excite any harmonic I wanted using select Chebyshev Polynomials. But you are totally right - it doesn't work that way. Any complex waveform that can be broken down into a Fourier series is a linear sum of terms. Squaring or cubing the waveform, and therefor this sum, leads to multiple cross terms which introduce additional frequencies. It does only work with normalized single sinusoids . . .which is too bad.

Right now, the only way I can see to do this sort of thing where you excite select harmonics at will is to run an FFT and then work from there in the frequency domain.

But my question is, if we are looking to simulate tube saturation, is the Chebyshev method good enough. What, after all, do tubes do? Does a tube amp actually add discrete harmonics or is it introducing all of those cross term frequencies as well?

Reverberation Algorithms in Matlab (click this to go back to the index)

References : Posted by Gautham J. Mysore (gauthamjm [AT] yahoo [DOT] com)

Linked file : [MATLABReverb.zip](#)

Notes :

These M-files implement a few reverberation algorithms (based on Schroeder's and Moorer's algorithms). Each of the M-files include a short description.

There are 5 M-files that implement reverberation. They are:

- schroeder1.m
- schroeder2.m
- schroeder3.m
- moorer.m
- stereoverb.m

The remaining 8 M-files implement filters, delay lines etc. Most of these are used in the above M-files. They can also be used as building blocks for other reverberation algorithms.

Comments

from : brewjinm [[a t]] aol.com

comment : StereoVerb is the name of an old car stereo "enhancer" from way back. I was just trying to find it's roots.

from : evanus_y [[a t]] yahoo.com

comment : There is another allpass filter transfer function.

$$H(z) = \frac{-g + Z^{(-m)}}{1 - gZ^{(-m)}}$$

g is the attenuation

m is the number of delay (in sampel)

This allpass will give exponential decay impulse response, compare to your allpass that give half sinc decay impulse response.

Reverberation techniques (click this to go back to the index)

References : Posted by Sean Costello

Notes :

* Parallel comb filters, followed by series allpass filters. This was the original design by Schroeder, and was extended by Moorer. Has a VERY metallic sound for sharp transients.

* Several allpass filters in serie (also proposed by Schroeder). Also suffers from metallic sound.

* 2nd-order comb and allpass filters (described by Moorer). Not supposed to give much of an advantage over first order sections.

* Nested allpass filters, where an allpass filter will replace the delay line in another allpass filter. Pioneered by Gardner. Haven't heard the results.

* Strange allpass amp delay line based structure in Jon Dattorro article (JAES). Four allpass filters are used as an input to a cool "figure-8" feedback loop, where four allpass reverberators are used in series with a few delay lines. Outputs derived from various taps in structure. Supposedly based on a Lexicon reverb design. Modulating delay lines are used in some of the allpass structures to "spread out" the eigentones.

* Feedback Delay Networks. Pioneered by Puckette/Stautner, with Jot conducting extensive recent research. Sound VERY good, based on initial experiments. Modulating delay lines and feedback matrixes used to spread out eigentones.

* Waveguide-based reverbs, where the reverb structure is based upon the junction of many waveguides. Julius Smith developed these. Recently, these have been shown to be essentially equivalent to the feedback delay network reverbs. Also sound very nice. Modulating delay lines and scattering values used to spread out eigentones.

* Convolution-based reverbs, where the sound to be reverbed is convolved with the impulse response of a room, or with exponentially-decaying white noise. Supposedly the best sound, but very computationally expensive, and not very flexible.

* FIR-based reverbs. Essentially the same as convolution. Probably not used, but shorter FIR filters are probably used in combination with many of the above techniques, to provide early reflections.

Comments

from : kristian_bauza [[a t]] yahoo.it

comment : Hello I am doing de algorithm of moore in Matlab simulink and I compare direntes reverb processor to improve something but i dont find anything to improve i ask if you have some idea to make something better in C++ a library DLL and making a plugin....
thanks

Simple Compressor class (C++) (click this to go back to the index)

Type : stereo, feed-forward, peak compressor

References : Posted by Citizen Chunk

Linked file : <http://www.chunkware.com/opensource/SimpleComp.zip>

Notes :

Everyone seems to want to make their own compressor plugin these days, but very few know where to start. After replying to so many questions on the KVR Dev Forum, I figured I might as well just post some ready-to-use C++ source code.

This is a C++ implementation of a simple, stereo, peak compressor. It uses a feed-forward topology, detecting the sidechain level pre-gain reduction. The sidechain detects the rectified peak level, with stereo linking to preserve imaging. The attack/release uses the EnvelopeDetector class (posted in the Analysis section).

Notes:

- Make sure to call `initRuntime()` before processing starts (i.e. call it in `resume()`).
- The process function takes a stereo input.
- VST params must be mapped to a practical range when setting compressor parameters. (i.e. don't try `setAttack(0.f)`.)

(see linked files)

Comments

from : tl_163 [[a t]] hotmail.com

comment : This code works perfectly, and I have tried a number of sound and each worked correctly. The conversion is linear in logarithm domain.

The code has been written in such a professional style, can not believe it is FREE!!

Keep it up. Two super huge thumbs up.

Ting

from : scuzzphut [[a t]] gmail.com

comment : source file seems to be down :-(
anyone have a local copy ???

from : citizenchunk[at]chunkware[dot]com

comment : there are some updates: mainly, there is now a SimpleGate class, which implements a simple gate. other than that, minor changes to the code -- but all public functions should work the same.

*** NEW FILE NAME: <http://www.chunkware.com/opensource/SimpleCompGate.zip>

from : jinjing20012001 [[a t]] yahoo.com

comment : hello everybody!

I am a Chinese student, I want to download the code from <http://www.chunkware.com/opensource/SimpleCompGate.zip> but because the server can not do his job well, I have no chance to download the code. I really hope kind people who have been have the code can transfer a copy to me! I will be appreciated very much.

my email is listed as following:

jinjing20012001@yahoo.com

I am looking for your reply as soon as possible.

from : citizenchunk[at]chunkware.com

comment : sorry about the link. the new source is at
<http://www.chunkware.com/downloads/simpleSource.zip>

from : ben [[a t]] benvesco.com

comment : MIRROR: <http://www.benvesco.com/dump/simpleSource.zip>

This code has been missing for some time at the posted locations. I found someone with a copy and am now mirroring these files for download on my server. I believe the software license allows this mirroring. The code is unchanged and belongs to the original owner.

[smsPitchScale Source Code](#) (click this to go back to the index)

Type : Pitch Scaling (often incorrectly referred to as "Pitch Shifting") using the Fourier transform

References : Posted by sms[AT]dspdimension.com

Linked file : <http://www.dspdimension.com>

Code :

See above web site

Comments

from : chinagreatman [[a t]] 163.net

comment : Hi:

Not Bad.

[Soft saturation](#) (click this to go back to the index)

Type : waveshaper

References : Posted by Bram de Jong

Notes :

This only works for positive values of x. a should be in the range 0..1

Code :

```
x < a:
  f(x) = x
x > a:
  f(x) = a + (x-a)/(1+((x-a)/(1-a))^2)
x > 1:
  f(x) = (a+1)/2
```

Comments

from : graue [[a t]] oceanbase.org

comment : This is a most excellent waveshaper.

I have implemented it as an effect for the music tracker Psyche, and so far I am very pleased with the results. Thanks for sharing this knowledge, Bram!

from : brobinson [[a t]] toptensoftware.com

comment : I'm wondering about the >1 condition here. If a is 0.8, values <1 approach 0.85 but values >1 are clamped to 0.9 (there's a gap)

If you substitute x=1 to the equation for x>a you get: a+((1-a)/4) not (a+1)/2

Have I missed something or is there a reason for this?

(Go easy I'm new to all of this)

from : kibibu [[a t]] gmail.com

comment : Substituting x=1 into equation 2 (taking many steps)

```
f(x) = a + (x-a)/(1+((x-a)/(1-a))^2)
      = a + (1-a)/(1+((1-a)/(1-a))^2)
      = a + (1-a)/(1+ 1^2)
      = a + (1-a)/2
      = 2a/2 + (1-a)/2
      = (2a + 1 - a) / 2
      = (a+1) / 2
```

from : musicdsp[at] Nospam dsparsons[dot]co[dot]uk

comment : You can normalise the output:

```
f(x)'=f(x)*(1/((a+1)/2))
```

This gives a nice variable shaper with smooth curve upto clipping at 0dBFS

[Stereo Enhancer](#) (click this to go back to the index)

[References](#) : Posted by kurmisk[at]inbox[DOT]lv

[Notes](#) :

Stereo Enhance

[Code](#) :

```
// WideCoeff 0.0 .... 1.5

#define StereoEnhance(SampL,SampR,MonoSign, \
    DeltaLeft,WideCoeff ) \
    MonoSign = (SampL + SampR)/2.0; \
    DeltaLeft = SampL - MonoSign; \
    DeltaLeft = DeltaLeft * WideCoeff; \
    SampL=SampL + DeltaLeft; \
    SampR=SampR - DeltaLeft;
```

[Comments](#)

[from](#) : tahome [[a t]] postino.ch

[comment](#) : This code is nonsense, all it does is create an out-of-balance stereo field...

--th

[from](#) : xeeton[AT]gmail[DOT]com

[comment](#) : #define StereoEnhance(SampL,SampR,MonoSign,
DeltaLeft,DeltaRight,WideCoeff)
MonoSign = (SampL + SampR)/2.0;

```
DeltaLeft = SampL - MonoSign;
DeltaLeft *= WideCoeff;
DeltaRight = SampR - MonoSign;
DeltaRight *= WideCoeff;
```

```
SampL += DeltaLeft;
SampR += DeltaRight;
```

I think this is more along the lines of what you were trying to accomplish. I doubt this is the correct way of implementing this type of thing however.

[from](#) : mark_hamburg[AT]baymoon[DOT]com

[comment](#) : I believe both pieces of code do the same thing. Since MonoSign is set equal to the average of the two signals, in the second case
DeltaRight = -DeltaLeft.

[from](#) : thaddy[[[a t]]] thaddy.com

[comment](#) :

Here's an implementation of the classic stereo enhancer in Delphi BASM

Values below 0.1 have a narrowing effect

Values above 0.1 widens

parameters:

Buffer = eax

Amount = edx

Samples = ecx

const

Spread:single = 6.5536;

procedure Sound3d32f(Buffer:PSingle;Amount:Single;Samples:integer);

asm

fld Amount

fmul spread

mov ecx,edx // move samples to ecx

shr ecx,1 // divide by two, stereo = 2 samples

@Start:

fld [eax].dword // left sample

fld [eax+4].dword // right sample, whole calculation runs on the stack

fld st(0) // copy

fadd st(0),st(2)

fmul half // average =st(0), right sample = st(1), left = st(2), amount=st(3)

fld st(0) // copy average

fsubr st(0),st(3) // left diffence

fmul st(0),st(4) // amount

fadd st(0),st(1) // add average

fadd st(0),st(3) // add original

fmul half // divide by two

fstp [eax].dword // and store

fld st(0)

fsubr st(0),st(2) // right difference

fmul st(0),st(4) // amount

faddp // add average

faddp // add original

```

fmul  half      // divide by 2
fstp  [eax+4].dword; // and store
fxch                      // Dangling average?? remove it later, tdk
ffree st(1)
add   eax, 8      // advance to next stereo pair
loop  @Start
ffree st(0);     // Cleanup amount
end;

```

from : thaddy [[a t]] thaddy.com
comment : Note 'half' is defined as const half:single = 0.5;
 This is an omission in the above posting

from : gtekprog [[a t]] hotmail.com
comment : This original code makes indeed no sense.

```

>#define StereoEnhanca(SampL,SampR,MonoSign, \
>DeltaLeft,WideCoeff ) \
>MonoSign = (SampL + SampR)/2.0; \
>DeltaLeft = SampL - MonoSign; \
>DeltaLeft = DeltaLeft * WideCoeff; \
>SampL=SampL + DeltaLeft; \
>SampR=SampR - DeltaLeft;
Deltaleft hold no stereoinformation.
explained: Deltaleft=L-(L+R) = R!!!
So, in this example your stereo image would slide to the right more as you put widecoeff higher.

```

A better implementation is the following code.

```

#define StereoEnhanca(SampL,SampR,MonoSign, \
stereo,WideCoeff ) \
MonoSign = (SampL + SampR)/2.0; \
stereo = SampL - SampR; \
stereo = DeltaLeft * WideCoeff; \
SampL=SampR + stereo; // R+Stereo = L
SampR=SampL - stereo; // L-Stereo = R

```

This way of stereoenhancement will lead to exaggerated reverberation effects (snaredrums).
 This is not the best way to do widening, but it is the easiest.

Gtekprog.

Evert Verduin

from : gtekprog [[a t]] hotmail.com
comment : oops,

```

stereo = SampL - SampR;
needs ofcourse to be
stereo = SampL - SampR;

```

and

```

stereo = DeltaLeft * WideCoeff; \
needs to be
stereo = stereo * WideCoeff; \

```

Again the correct code:

```

#define StereoEnhanca(SampL,SampR,MonoSign, \
stereo,WideCoeff ) \
MonoSign = (SampL + SampR)/2.0; \
stereo = SampL - SampR; \
stereo = stereo * WideCoeff; \
SampL=SampR + stereo; // R+Stereo = L
SampR=SampL - stereo; // L-Stereo = R

```

This will do.

Evert

Stereo Field Rotation Via Transformation Matrix (click this to go back to the index)

Type : Stereo Field Rotation

References : Posted by Michael Gruhn

Notes :

This work is hereby placed in the public domain for all purposes, including use in commercial applications.

'angle' is the angle by which you want to rotate your stereo field.

Code :

```
// Calculate transformation matrix's coefficients
cos_coef = cos(angle);
sin_coef = sin(angle);

// Do this per sample
out_left  = in_left * cos_coef - in_right * sin_coef;
out_right = in_left * sin_coef + in_right * cos_coef;
```

Comments

from : Foo

comment : If the source would be dead center a 180° rotation would mean the source would be behind you, but since in stereo there is no front or behind (just left and right), behind gets indicated by phase reversal (I know it doesn't reflect the position, but you can't because there is only left and right).

Also the rotation is clockwise, so a positive angles shift the source to the right, which means for your example if you'd rotate from 0° to -90° you'd indeed get the signal one the left channel and the right blank. For a mono signal (both channels identical that is) and a rotation range of -45° to 45° is the same as panning (with a 0dB pan law).

But I'll admit I was totally wrong and this entry in the musicdsp is the most faultiest that there ever was and isn't going to be useful at all, to no one. Anyway if this is not stereo field rotation, how would YOU call it? I'd happily forward the new terminology to the siteadmin, so the entries' description can be changed as soon as possible to whatever you think it is.

I'm just glad that I'm not the only one that is using wrong terminology, e.g. the Waves S1-Imager's "Rotation" does the same as the above posted code, as does Nick Whitehurst's c_superstereo and others ...

So tell me what it is called and I'll see if I can get the name changed, so everyone can be happy. Though I doubt I can get Waves nor any audio engineers to also adapt the new, correct terminology, that you will proved, for this kind of effect.

BTW if you want to discuss this further please mail to: 1337foo42bar69@trashmail.net because there is no need to waste more comment space about this (I now think or at least hope that it only is a ...) terminology discussion, because there is nothing wrong with the code itself I posted, or is there?

from : Bar

comment : Yet another childish thought. If one can treat signals as if they were space locations, then surely translations will work just as well as rotations. So to move a sound source to another location, one just add constants to the signals?

from : Bar

comment : Let's try another experiment. You're in the midpoint of the line joining the two speakers and is the center of rotation. Your signal happens to have all zeros for the left channel. The formula simplifies to:

```
out_left = -in_right * sin
out_right = in_right * cos
```

As you rotate from 0 to 90, sin goes from 0 to 1, cos goes from 1 to 0. So the formula predicts that the left channel goes from silence to a phase inverted right, and the right channel goes from full sound to silence. Whereas physically the sound should move from my right to directly in front of me. Please explain.

from : Bar

comment : Then I'm not sure what you mean by rotation. In my mind, I see two sound sources at arbitrary locations and I'm at the center of rotation. So the effect of a rotation would depend on the angle subtended by the three points to begin with, which doesn't even show up in the formula. Also please explains what does it mean by the two channels being orthogonal dimensions, which is what the formula is based on. (I assume you understand the mathematical basis of how the formula is derived.)

No, a phase inversion on both channels don't sound 180 deg rotated. It sounds exactly the same as before.

from : Foo

comment : So you want mathematical prove? Even though I consider this childish, because it'd take you <5 minutes to put this in Matlab or any other DSP prototyping bench and hear the rotation effect for yourself. Anyway ...

For 180° the output should be totally inverted. So:

```
cos(180) = -1
sin(180) = 0
out_left = -in_left
out_right = -in_right
```

at 90° this means for a mono signal that the left channel will be a phase inverse of the right channel, so ... go directly to result, do not calculate:

```
out_left = -in_right
out_right = in_left
```

at 45° is just like hard panning to the right (with a 3dB volume attenuation), so for a mono signal the expected results would be one channel silence and the other would have the signal, so we calculate:

```
cos(45) = sqrt(2)/2
sin(45) = sqrt(2)/2
```

for mono signal we assume: mono = in_left = in_right ... so it follows:

out_left = mono * sqrt(2)/2 - mono * sqrt(2)/2 = 0
out_right = mono * sqrt(2)/2 + mono * sqrt(2)/2 = mono * sqrt(2) == mono * 3dB

and one more, 360° means same output as input, calculate for yourself.

Valid enough?

from : Bar

comment : Sorry this makes no sense at all. The rotation formula is predicated on the assumption that (x,y) are coordinates of two orthogonal dimensions. Now you can choose to visualize stereo signals anyway you like, including being on a Cartesian plan, or as polar coordinates, what have you... But this visualization has no relationship to the physical location of the sound. The left and right channels are NOT orthogonal dimensions physically. What the formula does is just some weird panning. As the previous comment pointed out, just plug in some easy angles like 90, 180 ... and see if you can make any valid interpretations out of them. You can't.

from : Foo

comment : It IS the exact formula as rotation for a point in a 2D space (around its origin). Now this is applied to the stereo field. Imagine it as a left-right plot, so the values of the left and right channel get plotted (just like a goniometer: [http://en.wikipedia.org/wiki/Goniometer_\(audio\)](http://en.wikipedia.org/wiki/Goniometer_(audio))). So now you can see the stereo image (mono = straight line, stereo = circle, etc...). Now when you rotate THIS plot and then use the values of the rotated plot for the new left and right sample values, you rotated the stereo image. So just get a goniometer and look at how the signal changes when you run it through the algorithm, it will be pretty obvious.

Hope this helps.

from : jgiulini [[a t]] hotmail.com

comment : This looks like the rotation formula for a point in space. Can you explain how does it work for a sound signal? Let's say that angle is 90 degrees, then you formula gives

out_left = -in_right

out_right = in_left

How would this be a 90 deg rotation of the sound?

Stereo Width Control (Obtained Via Transformation Matrix) (click this to go back to the index)

Type : Stereo Widener

References : Posted by Michael Gruhn

Notes :

(I was quite surprised that this wasn't already in the archive, so here it is.)

This work is hereby placed in the public domain for all purposes, including use in commercial applications.

'width' is the stretch factor of the stereo field:

width < 1: decrease in stereo width

width = 1: no change

width > 1: increase in stereo width

width = 0: mono

Code :

```
// calculate scale coefficient
coef_S = width*0.5;

// then do this per sample
m = (in_left + in_right)*0.5;
s = (in_right - in_left )*coef_S;

out_left  = m - s;
out_right = m + s;
```

Comments

from : scanner598 [[a t]] yahoo.co.uk

comment : Nice peace of code. I would add the following code at the end of the source to compensate for the loss/gain of amplitude:

```
out_left /= 0.5 + coef_S;
out_right /= 0.5 + coef_S;
```

from : -

comment : Scanner, no I wouldn't add that. First off it is unnecessary calculation you can rescale the MS matrix to your liking already! Plus your method will cause a boost by 6dBs when you set the width to 0 = mono. So mono signals get boosted by 6dB which I'm sure isn't what you intended.

Note: My original code is correct that is, when you'd look at an audio signal on a goniometer it would scale the audio signal at the S-axis and leaving everything else unaffected.

But as I don't want people that add the additional calculation that scanner requested (sorry not trying to mock you), an volume adjusted version.

[code]

```
// calc coefs
tmp = 1/max(1 + width,2);
coef_M = 1 * tmp;
coef_S = width * tmp;

// then do this per sample
m = (in_left + in_right)*coef_M;
s = (in_right - in_left)*coef_S;
```

```
out_left = m - s;
out_right = m + s;
```

[/code]

from : scanner598 [[a t]] yahoo.co.uk

comment : Hi Michael,

Thanks for the correction, I have build your solution in PureData and it is better than my suggestion was. B.t.w. there was already a posting on stereo enhancement on this site, you can find it under the effects section.

from : -

comment : Scanner, no problem and yes I've seen the "Stereo Enhancer" entry, though (even though it seems to try to achieve the same as this here) it is (as far as I can see) broken.

Time compression-expansion using standard phase vocoder (click this to go back to the index)

Type : vocoder phase time stretching

References : Posted by Cournape

Linked file : [vocoder.m](#) (this linked file is included below)

Notes :

Standard phase vocoder. For improved techniques (faster), see paper of Laroche : "Improved phase vocoder time-scale modification of audio"

Laroche, J.; Dolson, M.

Speech and Audio Processing, IEEE Transactions on , Volume: 7 Issue: 3 , May 1999

Page(s): 323 -332

Comments

from : dsp[at]rymix.net

comment : Anyone know what language this is in? It really would be nice to understand the syntax.

from : null [[a t]] null.null

comment : It's matlab code see mathworks.

/Daniel

from : dsp [[a t]] rymix.net

comment : thanks =)

from : ericlee280.at.hotmail.com

comment : The code seems to contain an undefined variable lss_frame, can someone explain what this is?

from : yyc [[a t]] cad.el.yuntech.edu.tw

comment : The code seems to contain an undefined variable lss_frame, can someone explain what this is?

from : cournape[at]enst[dot]fr

comment : There is indeed an error in the script. I will post the correction to the administrator.

For now, here is the correction. You have to replace

lss_frame by Ls (which is properly defined before the main loop in the script which is online). When I checked the code, there can be also some out of range error for the output vector : a change in the max variable definition seems to solve the problem (at least for overlapp below 0.75).

replace max = (nb_frame-2)*La+Nfft

by max = (nb_frame)*La+Nfft.

from : yyc [[a t]] cad.el.yuntech.edu.tw

comment : is it really follow this paper "Improved phase vocoder time-scale modification of audio"??

because i get the another source code for original phase vocoder on net, and the performance is better than the code....

it is let me confuse....

have anyone can slove my question?

and thanks.....

ps:the sorce code from"http://www.ee.columbia.edu/~dpwe/resources/matlab/pvoc/" .

Linked files

```
function [S] = vocoder(E,Nfft,over,alpha)
```

```
% Phase vocoder time-scaling :
```

```
% - E      : input vector
```

```
% - Nfft   : size per frame
```

```
% - over   : overlapp ( between 0 et 1 )
```

```
% - alpha  : time-expansion/compression factor ( alpha < 1 : compression; alpha > 1 : expansion ).
```

```
%
```

```
% 30.11.02. Cournapeau David.
```

```
%
```

```
% Time expansion using standart phase vocoder technique. Based on Dolson and Laroche's paper :
```

```
% "NEW PHASE-VOCODER TECHNIQUES FOR PITCH-SHIFTING, HARMONIZING AND
```

```
% OTHER EXOTIC EFFECTS", in
```

```
%=====
```

```
% The synthesis signal's length isn't exactly alpha*input's length. %
```

```
%=====
```

```
% Verifiy the overlapp
```



```

if( over < 0 | over >= 1 )
    error('error : overlapp must be between 0 and 1');
end;

if ( alpha <= 0)
    error('alpha must be strictly positive');
end;

E    = E(:);
N    = length(E);
Nfft = 2^(nextpow2(Nfft));

% Computing vocoder's parameters :
% - La      : number of samples to "advance" for each anamysis frame : analysis hop size.
% - nb_frames : number of frames to compute
% - Ls      : number of samples ot "advance" for each synthesis frame : synthesis hop size.
% - S       : S is the result vector
% - h       : hanning window

La      = floor((1-over) * Nfft);
nb_frames = floor((N-Nfft) / La);
max      = (nb_frames-2)*La + Nfft;
ls       = floor(alpha * La);

S       = zeros(floor(max*alpha),1);
h       = hanning(Nfft);

% Init process :

X       = h.*E(1:Nfft);
tX      = fft(X,Nfft);
Phis1   = angle(tX)
Phial   = Phis1;

for loop=2:nb_frames-1

    %=====
    % ( classic analysis part of a phase vocoder )

    % Take a frame, and windowing it

    X = h.*E((loop-1) * La + 1:(loop-1)*La + Nfft);

    % XI is the amplitude spectrum, and Phia2 the phase spectrum.

    tX    = fft(X, Nfft);
    Xi    = abs(tX);
    Phia2 = angle(tX);

    %=====
    % the part which actually does the time scaling

    % One compute the actual pulsations, and shift them. The tricky part is here...

    omega = mod( (Phia2-Phial)-2*pi*([0:Nfft-1].')/Nfft * La + pi, 2*pi) - pi;
    omega = 2 * pi * ([0:Nfft-1].') / Nfft + omega / La;
    Phis2 = Phis1 + lss_frame*omega;

    % The new phases values :

    Phis1 = Phis2;
    Phial = Phia2;

    %=====
    % Synthetise the frame, thanks to the computed phase and amplitude spectrum :
    % ( classic synthetisis part of a phase vocoder )

    tfs = Xi.*exp(j*Phis2);
    Xr  = real(ifft(tfs)).*h;

    % overlapp-add the synthetised frame Xr

    S((loop-1)*lss_frame+1:(loop-1)*lss_frame+Nfft) ...
        = S((loop-1)*lss_frame+1:(loop-1)*lss_frame+Nfft) + Xr;

end;

```

[transistor differential amplifier simulation](#) (click this to go back to the index)

Type : Waveshaper

References : Posted by Christian[at]savioursofsoul[dot]de

Notes :
Writing an exam about electronic components, i learned several equations about simulating that stuff. One simplified equation was the $\tanh(x)$ formula for the differential amplifier. It is not exact, but since the amplifiers are driven with only small amplitudes the behaviour is most often even advanced linear.

The fact, that the amp is differential, means, that the 2n order is eliminated, so the sound is also similar to a tube.

For a very fast use, this code is in pure assembly language (not optimized with SSE-Code yet) and performs in VST-Plugins very fast.

The code was written in delphi and if you want to translate the assembly code, you should know, the the parameters passing is done via registers. So pinp=EAX pout=EDX sf=ECX.

```
Code :
procedure Transistor(pinp,pout : PSingle; sf:Integer; Faktor: Single);
asm
    fld Faktor
@Start:
    fld [eax].single
    fmul st(0),st(1)

    fldl2e
    fmul
    fld st(0)
    frndint
    fsub st(1),st
    fxch st(1)
    f2xm1
    fld1
    fadd
    fscale    { result := z * 2**i }
    fstp st(1)

    fld st(0)
    fmulp

    fld st(0)
    fld1
    faddp
    fld1
    fsubp st(2),st(0)
    fdivp

    fstp [edx].single

    add eax,4
    add edx,4
    loop @Start
    fstp st(0)
end;
```

Variable-hardness clipping function (click this to go back to the index)

References : Posted by Laurent de Soras <laurent[AT]ohmforce[DOT]com>

Linked file : [laurent.gif](#)

Notes :

$k \geq 1$ is the "clipping hardness". 1 gives a smooth clipping, and a high value gives hardclipping.

Don't set k too high, because the formula use the `pow()` function, which use `exp()` and would overflow easily. 100 seems to be a reasonable value for "hardclipping"

Code :

```
f (x) = sign (x) * pow (atan (pow (abs (x), k)), (1 / k));
```

Comments

from : antiprosynthesis [[a t]] hotmail.com

comment : Use this function instead of atan and see performance increase drastically :)

```
inline double fastatan( double x )
{
    return (x / (1.0 + 0.28 * (x * x)));
}
```

from : spam [[a t]] musicdsp.org

comment : The greater k becomes the lesser is the change in the form of $f(x, k)$. I recommend using

$f_2(x, k_2) = \text{sign}(x) * \text{pow}(\text{atan}(\text{pow}(\text{abs}(x), 1 / k_2)), k_2)$, k_2 in $[0.01, 1]$

where k_2 is the "clipping softness" ($k_2 = 0.01$ means "hardclipping", $k_2 = 1$ means "softclipping"). This gives better control over the clipping effect.

from : notinformed [[a t]] nomail.org

comment : Don't know if i understood ok , but, how can i clip at diferent levels than -1.0/1.0 using this func? tried several ways but none seems to work

from : xeeton[AT]gmail[DOT]com

comment : The most straightforward way to adjust the level (x) at which the signal is clipped would be to multiply the signal by $1/x$ before the clipping function then multiply it again by x afterwards.

from : cschueler[at]gmx[dot]de

comment :

Atan is a nice softclipping function, but you can do without `pow()`.

x : input value

a : clipping factor (0 = none, infinity = hard)

ainv : $1/a$

```
y = ainv * atan( x * a );
```

from : scoofy [[a t]] inf.elte.hu

comment : Even better, you can normalize the output using:

```
shape = 1..infinity
```

precalc:

```
inv_atan_shape=1.0/atan(shape);
```

process:

```
output = inv_atan_shape * atan(input*shape);
```

This gives a very soft transition from no distortion to hard clipping.

[WaveShaper](#) (click this to go back to the index)

Type : waveshaper

References : Posted by Bram de Jong

Notes :

where x (in $[-1..1]$) will be distorted and a is a distortion parameter that goes from 1 to infinity
The equation is valid for positive and negativ values.

If a is 1, it results in a slight distortion and with bigger a 's the signal get's more funky.

A good thing about the shaper is that feeding it with bigger-than-one values, doesn't create strange fx. The maximum this function will reach is 1.2 for $a=1$.

Code :

```
f(x,a) = x*(abs(x) + a)/(x^2 + (a-1)*abs(x) + 1)
```

[Waveshaper](#) (click this to go back to the index)

Type : waveshaper

References : Posted by Jon Watte

Notes :

A favourite of mine is using a `sin()` function instead. This will have the "unfortunate" side effect of removing odd harmonics if you take it to the extreme: a triangle wave gets mapped to a pure sine wave. This will work with `a` going from .1 or so to `a= 5` and bigger! The mathematical limits for `a = 0` actually turns it into a linear function at that point, but unfortunately FPUs aren't that good with calculus :-). Once `a` goes above 1, you start getting clipping in addition to the "soft" wave shaping. It starts getting into more of an effect and less of a mastering tool, though :-)

Seeing as this is just various forms of wave shaping, you could do it all with a look-up table, too. In my version, that would get rid of the somewhat-expensive `sin()` function.

Code :

```
(input: a == "overdrive amount")
```

```
z = M_PI * a;  
s = 1/sin(z)  
b = 1/a
```

```
if (x > b)  
    f(x) = 1  
else  
    f(x) = sin(z*x)*s
```

Comments

from : Christian [[a t]] savioursofsoul.de

comment : This one doesn't work for me. What have i done wrong? On positive inputs, the output get messed up (usually one constant instead of a sine-like function)

from : nobody [[a t]] nowhere.com

comment : >>This one doesn't work for me.

I haven't tried it yet, but it's always possible your compiler decided the constants are ints instead of floats. Try "1.0f" everywhere you see "1". May not be it, but you never know.

Anyone have any luck with this one?

[Waveshaper](#) (click this to go back to the index)

[References](#) : Posted by Partice Tarrabia and Bram de Jong

[Notes](#) :

amount should be in [-1..1] [Plot it and stand back in astonishment! ;)]

[Code](#) :

```
x = input in [-1..1]
y = output
k = 2*amount/(1-amount);
```

```
f(x) = (1+k)*x/(1+k*abs(x))
```

[Comments](#)

[from](#) : kaleja [[a t]] estarcion.com

[comment](#) : I haven't compared this to the other waveshapers, but its behavior with input outside the [-1..1] range is interesting. With a relatively moderate shaping amounts which don't distort in-range signals severely, it damps extremely out-of-range signals fairly hard, e.g. $x = 100$, $k = 0.1$ yields $y = 5.26$; as x goes to infinity, y approaches 5.5. This might come in handy to control nonlinear processes which would otherwise be prone to computational blowup.

Waveshaper (simple description) (click this to go back to the index)

Type : Polynomial; Distortion

References : Posted by Jon Watte

Notes :

> The other question; what's a 'waveshaper' algorithm. Is it simply another
> word for distortion?

A typical "waveshaper" is some function which takes an input sample value X and transforms it to an output sample X' . A typical implementation would be a look-up table of some number of points, and some level of interpolation between those points (say, cubic). When people talk about a wave shaper, this is most often what they mean. Note that a wave shaper, as opposed to a filter, does not have any state. The mapping from $X \rightarrow X'$ is stateless.

Some wave shapers are implemented as polynomials, or using other math functions. Hard clipping is a wave shaper implemented using the `min()` and `max()` functions (or the three-argument `clamp()` function, which is the same thing). A very mellow and musical-sounding distortion is implemented using a third-degree polynomial; something like $X' = (3/2)X - (1/2)X^3$. The nice thing with polynomial wave shapers is that you know that the maximum they will expand bandwidth is their order. Thus, you need to oversample 3x to make sure that a third-degree polynomial is aliasing free. With a lookup table based wave shaper, you don't know this (unless you treat an N-point table as an N-point polynomial :-)

Code :

```
float waveshape_distort( float in ) {  
    return 1.5f * in - 0.5f * in *in * in;  
}
```

Comments

from : dspgoes [[a t]] hotmail.com

comment : Has anyone tried implementing this distortion effect? All I get on the output is static... thanks

from : Christian [[a t]] savioursofsoul.de

comment : Yes! It's one of the most simple waveshaper and you know the amount of oversampling! Works very nice (and fast).

from : kibibu [[a t]] gmail.com

comment : If you are getting static, make sure your input is scaled to between -1 and +1.

[Waveshaper :: Gloubi-boulga](#) (click this to go back to the index)

[References](#) : Laurent de Soras on IRC

[Notes](#) :

Multiply input by gain before processing

[Code](#) :

```
const double x = input * 0.686306;
const double a = 1 + exp (sqrt (fabs (x)) * -0.75);
output = (exp (x) - exp (-x * a)) / (exp (x) + exp (-x));
```

[Comments](#)

[from](#) : theo[DOT]burt[[[a t]] [[a t]] [[a t]] T]virgin[dot]net

[comment](#) : Just tried this out, sound is incredible, but is horribly expensive... Can anyone think of any realistic ways to optimize/approximate this?

[from](#) : mailthing [[a t]] trilete.net

[comment](#) : you can use a taylor series approximation for the exp , save time by realizing that $\exp(-x) = 1/\exp(x)$, use newton's method to calculate the sqrt with less precision... and if you use SIMD instructions, you can calculate several values in parallel. dunno what the savings would be like, but it would surely be faster.

[from](#) : Christian [[a t]] savioursofsoul.de

[comment](#) : Maybe something like this:

```
function GloubiBoulga(x:Single):Single;
var a,b:Single;
begin
x:=x*0.686306;
a:=1+exp(sqrt(f_abs(x))*-0.75);
b:=exp(x);
Result:=(b-exp(-x*a))*b/(b*b+1);
end;
```

still expensive, but...

[from](#) : Christian [[a t]] savioursofsoul.de

[comment](#) : A Taylor series doesn't work very well, because the approximation effects the result very early due to
a) numerical critical additions & subtractions of approximations
b) approximating approximated "a" makes the result evne more worse.

The above version has already been improved, by removing 2 of 5 exp() functions.

You can also try to express the $\exp(x)+\exp(-x)$ as $\cosh(x)$ with its approximation. So:

```
b:=exp(x);
Result:=(b-exp(-x*a))*b/(b*b+1);
```

would be:

```
Result:=(exp(x)-exp(-x*a))*20160/40320+x*x*(20160+ x*x*(1680+x*x*(56+x*x)));
```

but this is again more worse. Anyone else?

[from](#) : ShadowHugger [[a t]] dev.null

[comment](#) : Use table lookup with interpolation.

[from](#) : decil [[a t]] gtlab.net

[comment](#) : IMHO, you can use
 $x-0.15*x^2-0.15*x^3$
instead of this scary formula.

I try to explain my position with this small graph:

<http://liteprint.com/download/replacment.png>

This is only first step, if you want to get more correct result you can use interpolation method called method of minimal squares (this is translation from russian, maybe in england it has another name)

[from](#) : musicdsp [[a t]] dsparsons.co.uk

[comment](#) : That's much better decil - thx for that!

DSP

[from](#) : decil [[a t]] gtlab.net

[comment](#) : You are welcome :)

Now I've working under plugin with waveshapping processing like this. I've put a link to it here, when I've done it.

[from](#) : decil [[a t]] gtlab.net

[comment](#) : You can check my version:
<http://liteprint.com/download/SweetVST.zip>

Please, send comments and suggestions to my email.

Dmitry.

from : jayman_21 [[a t]] hotmail.com

comment : Which formula exactly did you use decil, for your plugin? How do you get different harmonics from this algo. thanx

jay

from : mail [[a t]] trilete.net

comment : wow, blast from the past seeing this turn up on kvraudio.

christian - i'd have thought that an advantage of using a taylor series approximation would be that it limits the order of the polynomial (and the resulting bandwidth) somewhat. it's been ages since i tested, but i thought i got some reasonable sounding results using the taylor series approximation. maybe not.

decil - isn't that a completely unrelated polynomial (similar to the common and cheap $x - a x^3$?). i'd think you'd have to do something about the dc from the x^2 term, too (or do a $\text{sign}(x) * x^2$). anyway, your plugin sounds to be popular so i look forward to checking it out later at home.

VST SDK GUI Switch without (click this to go back to the index)

References : Posted by quintosardo[AT]yahoo[DOT]it

Notes :

In VST GUI an on-value is represented by 1.0 and off by 0.0.

Code :

Say you have two signals you want to switch between when the user changes a switch.
You could do:

```
if(fSwitch == 0.f) //fSwitch is either 0.0 or 1.0
    output = input1
else
    output = input2
```

However, you can avoid the branch by doing:

```
output = input1 * (1.f - fSwitch) + input2 * fSwitch
```

Which would be like a quick mix. You could make the change clickless by adding a simple one-pole filter:

```
smooth = filter(fSwitch)
output = input1 * (1.f - smooth) + input2 * smooth
```

Comments

from : citizenchunk[AT]gmail[DOT]com

comment : Not trying to be incredulous, but ... Is this really worth it? Assuming that you pre-calc the (1-fSwitch), you still have 2 multiplies and 1 add, instead of just an assignment. Are branches really bad enough to justify spending those cycles?

Also, does it matter where in the signal flow the branch is? For instance, if it were at the output, the branch wouldn't be such a problem. But at the input, with many calculations downstream, would it matter more?

Also, what if your branches are much more complicated--i.e. multiple lines per case?

from : quintosardo [[a t]] yahoo.it

comment : I use it when I have to compute the (1-fSwitch) signal anyway.

Example: apply a LFO to amplitude and not to frequency. I compute LFO anyway, then I apply (1-fSwitch) to frequency and (fSwitch) to amplitude.

Yes, branches are really bad!:-)

This is because you "break" your cache waiting for a decision

Even if the branch is at the end of your routine, you are leaving a branch to successive code (i.e. to host)

Anyway, this is not ever worth to use, just consider single cases...

from : kaleja [[a t]] estarcion.com

comment : Kids, kids, you're both wrong!

chunk: Two multiplies and an add are really cheap on modern hardware - P2/P3 take about 2 clocks for fmul and 1 clock for fadd.

quintosardo: Modern hardware also has good branch prediction, so if the switch is, e.g., a VST parameter that only changes once per process() block, it will branch the same way on the order of 100 times in a row. Correctly predicted branches are basically free; mispredicted branches blow the instruction pipeline, which is a penalty of about 20 cycles or so. If you spread the cost of a single missed prediction over 100 samples, it's cheap enough to not worry about. So yes, use this "predicate transform" to optimize away branches which are unpredictable, but don't worry about branches which are predictable.

16-Point Fast Integer Sinc Interpolator. (click this to go back to the index)

References : Posted by mumart[at]gmail[dot]com

Notes :
This is designed for fast upsampling with good quality using only a 32-bit accumulator. Sound quality is very good. Conceptually it resamples the input signal 32768x and performs nearest-neighbour to get the requested sample rate. As a result downsampling will result in aliasing.

The provided Sinc table is Blackman-Harris windowed with a slight lowpass. The table entries are 16-bit and are 16x linear-oversampled. It should be pretty easy to figure out how to make your own table for it.

Code provided is in Java. Converting to C/MMX etc. should be pretty trivial.

Remember the interpolator requires a number of samples before and after the sample to be interpolated, so you can't resample the whole of a passed input buffer in one go.

Have fun,
Martin

Code :

```
public class SincResampler {
    private final int FP_SHIFT = 15;
    private final int FP_ONE = 1 << FP_SHIFT;
    private final int FP_MASK = FP_ONE - 1;

    private final int POINT_SHIFT = 4; // 16 points
    private final int OVER_SHIFT = 4; // 16x oversampling
    private final short[] table = {
        0, -7, 27, -71, 142, -227, 299, 32439, 299, -227, 142, -71, 27, -7, 0, 0,
        0, 0, -5, 36, -142, 450, -1439, 32224, 2302, -974, 455, -190, 64, -15, 2, 0,
        0, 6, -33, 128, -391, 1042, -2894, 31584, 4540, -1765, 786, -318, 105, -25, 3, 0,
        0, 10, -55, 204, -597, 1533, -4056, 30535, 6977, -2573, 1121, -449, 148, -36, 5, 0,
        -1, 13, -71, 261, -757, 1916, -4922, 29105, 9568, -3366, 1448, -578, 191, -47, 7, 0,
        -1, 15, -81, 300, -870, 2185, -5498, 27328, 12263, -4109, 1749, -698, 232, -58, 9, 0,
        -1, 15, -86, 322, -936, 2343, -5800, 25249, 15006, -4765, 2011, -802, 269, -68, 10, 0,
        -1, 15, -87, 328, -957, 2394, -5849, 22920, 17738, -5298, 2215, -885, 299, -77, 12, 0,
        0, 14, -83, 319, -938, 2347, -5671, 20396, 20396, -5671, 2347, -938, 319, -83, 14, 0,
        0, 12, -77, 299, -885, 2215, -5298, 17738, 22920, -5849, 2394, -957, 328, -87, 15, -1,
        0, 10, -68, 269, -802, 2011, -4765, 15006, 25249, -5800, 2343, -936, 322, -86, 15, -1,
        0, 9, -58, 232, -698, 1749, -4109, 12263, 27328, -5498, 2185, -870, 300, -81, 15, -1,
        0, 7, -47, 191, -578, 1448, -3366, 9568, 29105, -4922, 1916, -757, 261, -71, 13, -1,
        0, 5, -36, 148, -449, 1121, -2573, 6977, 30535, -4056, 1533, -597, 204, -55, 10, 0,
        0, 3, -25, 105, -318, 786, -1765, 4540, 31584, -2894, 1042, -391, 128, -33, 6, 0,
        0, 2, -15, 64, -190, 455, -974, 2302, 32224, -1439, 450, -142, 36, -5, 0, 0,
        0, 0, -7, 27, -71, 142, -227, 299, 32439, 299, -227, 142, -71, 27, -7, 0
    };

    /*
    private final int POINT_SHIFT = 1; // 2 points
    private final int OVER_SHIFT = 0; // 1x oversampling
    private final short[] table = {
        32767, 0,
        0, 32767
    };
    */
}
```

```

private final int POINTS = 1 << POINT_SHIFT;

private final int INTERP_SHIFT = FP_SHIFT - OVER_SHIFT;

private final int INTERP_BITMASK = ( 1 << INTERP_SHIFT ) - 1;

/*
input - array of input samples
inputPos - sample position ( must be at least POINTS/2 + 1, ie. 7 )
inputFrac - fractional sample position ( 0 <= inputFrac < FP_ONE )
step - number of input samples per output sample * FP_ONE
lAmp - left output amplitude ( 1.0 = FP_ONE )
lBuf - left output buffer
rAmp - right output amplitude ( 1.0 = FP_ONE )
rBuf - right output buffer
pos - offset into output buffers
count - number of output samples to produce
*/

public void resample( short[] input, int inputPos, int inputFrac, int step,
    int lAmp, int[] lBuf, int rAmp, int[] rBuf, int pos, int count ) {
for( int p = 0; p < count; p++ ) {

    int tabidx1 = ( inputFrac >> INTERP_SHIFT ) << POINT_SHIFT;

    int tabidx2 = tabidx1 + POINTS;

    int bufidx = inputPos - POINTS/2 + 1;

    int a1 = 0, a2 = 0;

    for( int t = 0; t < POINTS; t++ ) {

        a1 += table[ tabidx1++ ] * input[ bufidx ] >> 15;

        a2 += table[ tabidx2++ ] * input[ bufidx ] >> 15;

        bufidx++;

    }

    int out = a1 + ( ( a2 - a1 ) * ( inputFrac & INTERP_BITMASK ) >> INTERP_SHIFT );

    lBuf[ pos ] += out * lAmp >> FP_SHIFT;

    rBuf[ pos ] += out * rAmp >> FP_SHIFT;

    pos++;

    inputFrac += step;

    inputPos += inputFrac >> FP_SHIFT;

    inputFrac &= FP_MASK;

}

}

}

```

16-to-8-bit first-order dither (click this to go back to the index)

Type : First order error feedforward dithering code

References : Posted by Jon Watte

Notes :

This is about as simple a dithering algorithm as you can implement, but it's likely to sound better than just truncating to N bits.

Note that you might not want to carry forward the full difference for infinity. It's probably likely that the worst performance hit comes from the saturation conditionals, which can be avoided with appropriate instructions on many DSPs and integer SIMD type instructions, or CMOV.

Last, if sound quality is paramount (such as when going from > 16 bits to 16 bits) you probably want to use a higher-order dither function found elsewhere on this site.

Code :

```
// This code will down-convert and dither a 16-bit signed short
// mono signal into an 8-bit unsigned char signal, using a first
// order forward-feeding error term dither.

#define uchar unsigned char

void dither_one_channel_16_to_8( short * input, uchar * output, int count, int * memory )
{
    int m = *memory;
    while( count-- > 0 ) {
        int i = *input++;
        i += m;
        int j = i + 32768 - 128;
        uchar o;
        if( j < 0 ) {
            o = 0;
        }
        else if( j > 65535 ) {
            o = 255;
        }
        else {
            o = (uchar)((j>>8)&0xff);
        }
        m = ((j-32768+128)-i);
        *output++ = o;
    }
    *memory = m;
}
```

3rd order Spline interpolation (click this to go back to the index)

References : Posted by Dave from Muon Software, originally from Josh Scholar

Notes :
(from Joshua Scholar about Spline interpolation in general...)
According to sampling theory, a perfect interpolation could be found by replacing each sample with a sinc function centered on that sample, ringing at your target nyquist frequency, and at each target point you just sum all of contributions from the sinc functions of every single point in source. The sinc function has ringing that dies away very slowly, so each target sample will have to have contributions from a large neighborhood of source samples. Luckily, by definition the sinc function is bandwidth limited, so once we have a source that is prefiltered for our target nyquist frequency and reasonably oversampled relative to our nyquist frequency, ordinary interpolation techniques are quite fruitful even though they would be pretty useless if we hadn't oversampled.

We want an interpolation routine that at very least has the following characteristics:

1. Obviously it's continuous. But since finite differencing a signal (I don't really know about true differentiation) is equivalent to a low frequency attenuator that drops only about 6 dB per octave, continuity at the higher derivatives is important too.
2. It has to be stiff enough to find peaks when our oversampling missed them. This is where what I said about the combination the sinc function's limited bandwidth and oversampling making interpolation possible comes into play.

I've read some papers on splines, but most stuff on splines relates to graphics and uses a control point descriptions that is completely irrelevant to our sort of interpolation. In reading this stuff I quickly came to the conclusion that splines:

1. Are just piecewise functions made of polynomials designed to have some higher order continuity at the transition points.
2. Splines are highly arbitrary, because you can choose arbitrary derivatives (to any order) at each transition. Of course the more you specify the higher order the polynomials will be.
3. I already know enough about polynomials to construct any sort of spline. A polynomial through 'n' points with a derivative specified at 'm[1]' points and second derivatives specified at 'm[2]' points etc. will be a polynomial of the order $n-1+m[1]+m[2]...$

A way to construct third order splines (that admittedly doesn't help you construct higher order splines), is to linearly interpolate between two parabolas. At each point (they are called knots) you have a parabola going through that point, the previous and the next point. Between each point you linearly interpolate between the polynomials for each point. This may help you imagine splines.

As a starting point I used a polynomial through 5 points for each knot and used MuPad (a free Mathematica like program) to derive a polynomial going through two points (knots) where at each point it has the same first two derivatives as a 4th order polynomial through the surrounding 5 points. My intuition was that basing it on polynomials through 3 points wouldn't be enough of a neighborhood to get good continuity. When I tested it, I found that not only did basing it on 5 point polynomials do much better than basing it on 3 point ones, but that 7 point ones did nearly as badly as 3 point ones. 5 points seems to be a sweet spot.

However, I could have set the derivatives to a nearly arbitrary values - basing the values on those of polynomials through the surrounding points was just a guess.

I've read that the math of sampling theory has different interpretation to the sinc function one where you could upsample by making a polynomial through every point at the same order as the number of points and this would give you the same answer as sinc function interpolation (but this only converges perfectly when there are an infinite number of points). Your head is probably spinning right now - the only point of mentioning that is to point out that perfect interpolation is exactly as stiff as a polynomial through the target points of the same order as the number of target points.

Code :

```
//interpolates between L0 and H0 taking the previous (L1) and next (H1)
points into account
inline float ThirdInter(const float x,const float L1,const float L0,const
float H0,const float H1)
{
    return
    L0 +
    .5f*
    x*(H0-L1 +
    x*(H0 + L0*(-2) + L1 +
    x*( (H0 - L0)*9 + (L1 - H1)*3 +
    x*((L0 - H0)*15 + (H1 - L1)*5 +
    x*((H0 - L0)*6 + (L1 - H1)*2 ))));
}
```

Comments

from : a [[a t]] a.com
comment : What is x ?

from : mike_rawes [[a t]] ku.oc.oohay
comment : The samples being interpolated represent the wave amplitude at a particular instant of time, T - an impulse train. So each sample is the amplitude at T=0,1,2,3 etc.

The purpose of interpolation is to determine the amplitude, a, for an arbitrary t, where t is any real number:

```
p1  p0  a  n0  n1
:   :   :   :
0-----1---t---2-----3-----> T
:
:
:
<-x->
```

$x = t - T(p0)$

-
myk

from : mike_rawes [[a t]] ku.oc.oohay
comment : Dang! My nice diagram had its spacing stolen, and it now makes no sense!

p1, p0, n0, n1 are supposed to line up with 0,1,2,3 respectively. a is supposed to line up with the t. And finally, <-x-> spans between 1 and t.

-
myk

from : fcanessa [[a t]] terra.cl
comment : 1.- What is 5f ?

2.- How I can test this procedure?.

Thank you

from : joshscholar_REMOVE_THIS [[a t]] yahoo.com
comment : This is years later. but just in case anyone has the same problem as fcanessa... In C or C++ you can append an 'f' to a number to make it single precision, so .5f is the same as .5

5-point spline interpolation (click this to go back to the index)

Type : interpolation

References : Joshua Scholar, posted by David Waugh

Code :

```
//nMask = sizeofwavetable-1 where sizeofwavetable is a power of two.
double interpolate(double* wavetable, int nMask, double location)
{
    /* 5-point spline*/

    int nearest_sample = (int) location;
    double x = location - (double) nearest_sample;

    double p0=wavetable[(nearest_sample-2)&nMask];
    double p1=wavetable[(nearest_sample-1)&nMask];
    double p2=wavetable[nearest_sample];
    double p3=wavetable[(nearest_sample+1)&nMask];
    double p4=wavetable[(nearest_sample+2)&nMask];
    double p5=wavetable[(nearest_sample+3)&nMask];

    return p2 + 0.04166666666*x*((p3-p1)*16.0+(p0-p4)*2.0
+ x *((p3+p1)*16.0-p0-p2*30.0- p4
+ x *(p3*66.0-p2*70.0-p4*33.0+p1*39.0+ p5*7.0- p0*9.0
+ x *( p2*126.0-p3*124.0+p4*61.0-p1*64.0- p5*12.0+p0*13.0
+ x *((p3-p2)*50.0+(p1-p4)*25.0+(p5-p0)*5.0)))));
};
```

Comments

from : joshscholarREMOVETHIS [[a t]] yahoo.com

comment : The code works much better if you oversample before interpolating. If you oversample enough (maybe 4 to 6 times oversampling) then the results are audiophile quality.

Allocating aligned memory (click this to go back to the index)

Type : memory allocation

References : Posted by Benno Senoner

Notes :

we waste up to align_size + sizeof(int) bytes when we alloc a memory area.
We store the aligned_ptr - unaligned_ptr delta in an int located before the aligned area.
This is needed for the free() routine since we need to free all the memory not only the aligned area.
You have to use aligned_free() to free the memory allocated with aligned_malloc() !

Code :

```
/* align_size has to be a power of two !! */
void *aligned_malloc(size_t size, size_t align_size) {

    char *ptr,*ptr2,*aligned_ptr;
    int align_mask = align_size - 1;

    ptr=(char *)malloc(size + align_size + sizeof(int));
    if(ptr==NULL) return(NULL);

    ptr2 = ptr + sizeof(int);
    aligned_ptr = ptr2 + (align_size - ((size_t)ptr2 & align_mask));

    ptr2 = aligned_ptr - sizeof(int);
    *((int *)ptr2)=(int)(aligned_ptr - ptr);

    return(aligned_ptr);
}

void aligned_free(void *ptr) {

    int *ptr2=(int *)ptr - 1;
    ptr -= *ptr2;
    free(ptr);
}
```

Antialiased Lines (click this to go back to the index)

Type : A slow, ugly, and unoptimized but short method to perform antialiased lines in a framebuffer

References : Posted by arguru[AT]smartelectronix[DOT]com

Notes :

Simple code to perform antialiased lines in a 32-bit RGBA (1 byte/component) framebuffer.

pframebuffer <- unsigned char* to framebuffer bytes (important: Y flipped line order! [like in the way Win32 CreateDIBSection works...])

client_height=framebuffer height in lines

client_width=framebuffer width in pixels (not in bytes)

This doesnt perform any clip checl so it fails if coordinates are set out of bounds.

sorry for the english

Code :

```
//  
// By Arguru  
//  
void PutTransPixel(int const x,int const y,UCHAR const r,UCHAR const g,UCHAR const b,UCHAR const a)  
{  
    unsigned char* ppix=pframebuffer+(x+(client_height-(y+1))*client_width)*4;  
    ppix[0]=((a*b)+(255-a)*ppix[0])/256;  
    ppix[1]=((a*g)+(255-a)*ppix[1])/256;  
    ppix[2]=((a*r)+(255-a)*ppix[2])/256;  
}  
  
void LineAntialiased(int const x1,int const y1,int const x2,int const y2,UCHAR const r,UCHAR const g,UCHAR  
const b)  
{  
    // some useful constants first  
    double const dw=x2-x1;  
    double const dh=y2-y1;  
    double const slx=dh/dw;  
    double const sly=dw/dh;  
  
    // determine wichever raster scanning behaviour to use  
    if(fabs(slx)<1.0)  
    {  
        // x scan  
        int tx1=x1;  
        int tx2=x2;  
        double raster=y1;  
  
        if(x1>x2)  
        {  
            tx1=x2;  
            tx2=x1;  
            raster=y2;  
        }  
  
        for(int x=tx1;x<=tx2;x++)  
        {  
            int const ri=int(raster);  
  
            double const in_y0=1.0-(raster-ri);  
            double const in_y1=1.0-(ri+1-raster);  
  
            PutTransPixel(x,ri+0,r,g,b,in_y0*255.0);  
            PutTransPixel(x,ri+1,r,g,b,in_y1*255.0);  
  
            raster+=slx;  
        }  
    }  
    else  
    {  
        // y scan  
        int ty1=y1;  
        int ty2=y2;  
        double raster=x1;  
  
        if(y1>y2)  
        {  
            ty1=y2;  
            ty2=y1;  
            raster=x2;  
        }  
  
        for(int y=ty1;y<=ty2;y++)  
        {  
            int const ri=int(raster);  
  
            double const in_x0=1.0-(raster-ri);  
            double const in_x1=1.0-(ri+1-raster);
```

```
PutTransPixel(ri+0,y,r,g,b,in_x0*255.0);
PutTransPixel(ri+1,y,r,g,b,in_x1*255.0);

raster+=sly;
}
}
}
```

Comments

from : Gog

comment : Sorry, but what does this have to do with music DSP ??

from : pav_101 [[a t]] hotmail.com

comment : well, for drawing envelopes, waveforms, etc on screen in your DSP app....

from : Gog

comment : But... there are TONS of graphic toolkits to do just that. No reason to "roll your own". One f.i. is GDI+ (works darn well to be honest), or if you want non-M\$ (and better!) go with AGG at (<http://www.antigrain.com>). And there are even open-source cross-platform toolkits (if you want to do Unix and Mac without coding).

Graphics and GUIs is a very time-consuming task to do from scratch, therefore I think using libraries such as the above is the way to go, liberating energy to do the DSP stuff... ;-)

from : Rich

comment : I don't want a toolkit, I want antialiased line drawing and nothing more. Everything else is fine.

from : Justin

comment : Anyone know how to get the pointer to the framebuffer? Perhaps there is a different answer for different platforms?

from : daniel.schaack [at] basementarts.de

comment : you can also draw everything in a 2x (vertically and horizontally) higher resolution and then reduce the size again by always taking the average of 4 pixels. that works well.

from : aliloko [[a t]] gmail.com

comment : I think it can be useful to those designing graphical synths.

But the Wu line algorithm is considerably more fast and works only with integers.

http://en.wikipedia.org/wiki/Xiaolin_Wu's_line_algorithm

Automatic PDC system (click this to go back to the index)

Type : the type that actually works, completely

References : Posted by Tebello Thejane

Linked file : [pdc.pdf](#)

Notes :

No, people, implementing PDC is actually not as difficult as you might think it is.

This paper presents a solution to the problem of latency inherent in audio effects processors, and the two appendices give examples of the method being applied on Cubase SX (with an example which its native half-baked PDC fails to solve properly) as well as a convoluted example in FL Studio (taking advantage of the flexible routing capabilities introduced in version 6 of the software). All that's necessary to understand it is a grasp of basic algebra and an intermediate understanding of how music production software works (no need to understand the Laplace transform, linear processes, sigma and integral notation... YAY!).

Please do send me any feedback (kudos, errata, flames, job offers, questions, comments) you might have - my email address is included in the paper - or simply use musicdsp.org's own commenting system.

Tebello Thejane.

Code :

(I have sent the PDF to Bram as he suggested)

Comments

from : zyxoas [[a t]] gmail.com

comment : The revised version may be found here:
http://www.vormdicht.nl/misc/PDC_paper-rev.pdf

Naturally, you need to remove the brackets from the address.

from : zyxoas [[a t]] gmail.com

comment : Oops! RBJ's first name is Robert, not Richard! Man, that's a bad one...

from : zyxoas [[a t]] gmail.com

comment : Okay, I've sent a fixed version to Bram. It should be uploaded shortly. Bigger diagrams, too, so there's less aliasing in Adobe Acrobat Reader. Hopefully no more embarrassing bad errors (like misspelling my own name, or something...).

[Base-2 exp](#) (click this to go back to the index)

[References](#) : Posted by Laurent de Soras

Notes :

Linear approx. between 2 integer values of val. Uses 32-bit integers. Not very efficient but fastest than exp()

This code was designed for x86 (little endian), but could be adapted for big endian processors.

Laurent thinks you just have to change the `(*1 + (int *) &ret)` expressions and replace it by `*(int *) &ret`. However, He didn't test it.

Code :

```
inline double fast_exp2 (const double val)
{
    int    e;
    double ret;

    if (val >= 0)
    {
        e = int (val);
        ret = val - (e - 1);
        ((*1 + (int *) &ret) &= ~(2047 << 20)) += (e + 1023) << 20;
    }
    else
    {
        e = int (val + 1023);
        ret = val - (e - 1024);
        ((*1 + (int *) &ret) &= ~(2047 << 20)) += e << 20;
    }
    return (ret);
}
```

Comments

[from](#) : subatomic [[a t]] vrsource.org

[comment](#) :

Here is the code to detect little endian processor:

```
union
{
    short val;
    char  ch[sizeof( short )];
} un;
un.val = 256; // 0x10;

if (un.ch[1] == 1)
{
    // then we're little
}
```

I've tested the fast_exp2() on both little and big endian (intel, AMD, and motorola) processors, and the comment is correct.

Here is the completed function that works on all endian systems:

```
inline double fast_exp2( const double val )
{
    // is the machine little endian?
    union
    {
        short val;
        char  ch[sizeof( short )];
    } un;
    un.val = 256; // 0x10;
    // if un.ch[1] == 1 then we're little

    // return 2 to the power of val (exp base2)
    int e;
    double ret;

    if (val >= 0)
    {
        e = int (val);
        ret = val - (e - 1);

        if (un.ch[1] == 1)
            ((*1 + (int *) &ret) &= ~(2047 << 20)) += (e + 1023) << 20;
        else
            ((*((int *) &ret) &= ~(2047 << 20)) += (e + 1023) << 20;
    }
    else
    {
        e = int (val + 1023);
        ret = val - (e - 1024);

        if (un.ch[1] == 1)
            ((*1 + (int *) &ret) &= ~(2047 << 20)) += e << 20;
        else
            ((*((int *) &ret) &= ~(2047 << 20)) += e << 20;
    }
}
```

```
}  
return ret;  
}
```

Bit-Reversed Counting (click this to go back to the index)

References : Posted by mailbjl[at]yahoo[DOT]com

Notes :

Bit-reversed ordering comes up frequently in FFT implementations. Here is a non-branching algorithm (given in C) that increments the variable "s" bit-reversedly from 0 to N-1, where N is a power of 2.

Code :

```
int r = 0;           // counter
int s = 0;           // bit-reversal of r/2
int N = 256;         // N can be any power of 2
int N2 = N << 1;    // N<<1 == N*2

do {
    printf("%u ", s);
    r += 2;
    s ^= N - (N / (r&-r));
}
while (r < N2);
```

Comments

from : wahida_r[at]yahoo.com

comment : This will give the bit reversal of N number of elements (where N is a power of 2). If we want reversal of a particular number out of N, is there any optimised way other than doing bit wise operations

from : mailbjl[at]yahoo.com

comment : There's a better way that doesn't require counting, branching, or division. It's probably the fastest way of doing bit reversal without a special instruction. I got this from Jörg's FXT book:

unsigned r; // value to be bit-reversed

// Assume r is 32 bits

```
r = ((r & 0x55555555) << 1) | ((r & 0xaaaaaaaa) >> 1);
r = ((r & 0x33333333) << 2) | ((r & 0xcccccccc) >> 2);
r = ((r & 0x0f0f0f0f) << 4) | ((r & 0xf0f0f0f0) >> 4);
r = ((r & 0x00ff00ff) << 8) | ((r & 0xff00ff00) >> 8);
r = ((r & 0x0000ffff) << 16) | ((r & 0xffff0000) >> 16);
```

Block/Loop Benchmarking (click this to go back to the index)

Type : Benchmarking Tool

References : Posted by arguru[AT]smartelectronix[DOT]com

Notes :

Requires CPU with RDTSC support

Code :

```
// Block-Process Benchmarking Code using rdtsc
// useful for measure DSP block stuff
// (based on Intel papers)
// 64-bit precision
// VeryUglyCode(tm) by Arguru

// globals
UINT time,time_low,time_high;

// call this just before enter your loop or whatever
void bpb_start()
{
    // read time stamp to EAX
    __asm rdtsc;
    __asm mov time_low,eax;
    __asm mov time_high,edx;
}

// call the following function just after your loop
// returns average cycles wasted per sample
UINT bpb_finish(UINT const num_samples)
{
    __asm rdtsc
    __asm sub eax,time_low;
    __asm sub edx,time_high;
    __asm div num_samples;
    __asm mov time,eax;
    return time;
}
```

Comments

from : pete [[a t]] bannister25.plus.com

comment : If running windows on a mutliprocessor system, apparently it is worth calling:

```
SetThreadAffinityMask(GetCurrentThread(), 1);
```

to reduce artefacts.

(see http://msdn.microsoft.com/visualc/vctoolkit2003/default.aspx?pull=/library/en-us/dv_vstechart/html/optimization.asp)

from : guillaume.mureu [[a t]] free.fr

comment : __asm sub eax,time_low;
__asm sub edx,time_high;

should be

```
__asm sub eax,time_low
__asm SBB edx,time_high // subtract with borrow
```


Branchless Clipping (click this to go back to the index)

Type : Clipping at 0dB, with none of the usual 'if..then..'

References : Posted by musicdsp[AT]dsparsons[DOT]co[DOT]uk

Notes :

I was working on something that I wanted to ensure that the signal never went above 0dB, and a branchless solution occurred to me.

It works by playing with the structure of a single type, shifting the sign bit down to make a new multiplicand.

calling MaxZerodB(mydBSample) will ensure that it will never stray over 0dB.

By playing with signs or adding/removing offsets, this offers a complete branchless limiting solution, no matter whether dB or not (after all, they're all just numbers...).

eg:

```
Limit to <=0 : sample:=MaxZerodB(sample);
Limit to <=3 : sample:=MaxZerodB(sample-3)+3;
Limit to <=-4 : sample:=MaxZerodB(sample+4)-4;
```

```
Limit to >=0 : sample:=-MaxZerodB(-sample);
Limit to >=2 : sample:=-MaxZerodB(-sample+2)+2;
Limit to >=-1.5: sample:=-MaxZerodB(-sample-1.5)-1.5;
```

Whether it actually saves any CPU cycles remains to be seen, but it was an interesting diversion for half an hour :)

[Translating from pascal to other languages shouldn't be too hard, and for doubles, you'll need to fiddle it abit :)]

Code :

```
function MaxZerodB(dBin:single):single;
var tmp:longint;
begin
    //given that leftmost bit of a longint indicates the negative,
    // if we shift that down to bit0, and multiply dBin by that
    // it will return dBin, or zero :)
    tmp:=(longint((@dBin)^) and $80000000) shr 31;
    result:=dBin*tmp;
end;
```

Comments

from : hotpop.com [[a t]] blargg

comment : Since most processors include a sign-preserving right shift, you can right shift by 31 to end up with either -1 (all bits set) or 0, then mask the original value with it:

```
out = (in >> 31) & in;
```

from : mumart [[a t]] gmail.com

comment : I prefer this method, using a sign-preserving shift, as it can clip a signal to arbitrary bounds:

```
over = upper_limit - samp
mask = over >> 31
over = over & mask
samp = samp + over
over = samp - lower_limit
mask = over >> 31
over = over & mask
samp = samp - over
```

Is it faster? Maybe on modern machines with 20-plus-stage pipelines and if the signal is clipped often, as the branches are not predictable.

from : musicdsp[AT]dsparsons[DOT]co[DOT]uk

comment : hmm.. Did some looking into the sign preserving thing. My laptop has an P3 which didn't preserve as mentioned, and my work PC (P4HT) didn't either. Maybe its an AMD or motorola thing :)

unless it's how delphi interprets the shr.. what does a C++ compiler generate for '>>' ?

from : mumart [[a t]] gmail.com

comment : C and C++ have sign-preserving shifts. If the value is negative, a right shift will add ones onto the left hand side (thus -2 becomes -1 etc).

Java also has a non-sign-preserving right shift operator (>>>).

I tried googling for information on how Delphi handles shifts, but nothing turned up. Looks like you might need to use in-line assembly :/

from : bero [[a t]] 0ok.de

comment : Here my SAR function for Delphi+FreePascal

```
FUNCTION SAR(Value,Shift:INTEGER):INTEGER; {$IFDEF CPU386}ASSEMBLER; REGISTER;{$ELSE}{$IFDEF
FPC}INLINE;{$ELSE}REGISTER;{$ENDIF}{$ENDIF}
{$IFDEF CPU386}
ASM
MOV ECX,EDX
SAR EAX,CL
```

```

END;
{$ELSE}
BEGIN
RESULT:=(Value SHR Shift) OR (($FFFFFFFF+(1-((Value AND (1 SHL 31)) SHR 31) AND ORD(Shift<->0))) SHL (32-Shift));
END;
{$ENDIF}

```

from : bero [[a t]] 0ok.de

comment : Ny branchless clipping functions (the first is faster than the second)

```

FUNCTION Clip(Value,Min,Max:SINGLE):SINGLE; ASSEMBLER; STDCALL;
CONST Constant0Dot5:SINGLE=0.5;

```

```

ASM
FLD DWORD PTR Value
FLD DWORD PTR Min
FLD DWORD PTR Max

```

```

FLD ST(2)
FSUB ST(0),ST(2)
FABS
FADD ST(0),ST(2)
FADD ST(0),ST(1)

```

```

FLD ST(3)
FSUB ST(0),ST(2)
FABS
FSUBP ST(1),ST(0)
FMUL DWORD PTR Constant0Dot5

```

```

FFREE ST(4)
FFREE ST(3)
FFREE ST(2)
FFREE ST(1)
END;

```

```

FUNCTION ClipDSP(Value:SINGLE):SINGLE; {$IFDEF CPU386} ASSEMBLER; REGISTER;

```

```

ASM
MOV EAX,DWORD PTR Value
AND EAX,$80000000

```

```

AND DWORD PTR Value,$7FFFFFFF

```

```

FLD DWORD PTR Value
FLD1
FSUBP ST(1),ST(0)
FSTP DWORD PTR Value

```

```

MOV EDX,DWORD PTR Value
AND EDX,$80000000
SHR EDX,31
NEG EDX
AND DWORD PTR Value,EDX

```

```

FLD DWORD PTR Value
FLD1
FADDP ST(1),ST(0)
FSTP DWORD PTR Value

```

```

OR DWORD PTR Value,EAX

```

```

FLD DWORD PTR Value

```

```

END;
{$ELSE}
VAR ValueCasted:LONGWORD ABSOLUTE Value;
Sign:LONGWORD;
BEGIN

```

```

Sign:=ValueCasted AND $80000000;
ValueCasted:=ValueCasted AND $7FFFFFFF;
Value:=Value-1;
ValueCasted:=ValueCasted AND (-LONGWORD((ValueCasted AND $80000000) SHR 31));
Value:=Value+1;
ValueCasted:=ValueCasted OR Sign;
RESULT:=Value;
END;
{$ENDIF}

```

Calculate notes (java) (click this to go back to the index)

Type : Java class for calculating notes with different in params

References : Posted by larsby[AT]elak[DOT]org

Linked file : [Frequency.java](#) (this linked file is included below)

Notes :

Converts between string notes and frequencies and back. I vaguely remember writing bits of it, and I got it off the net somewhere so dont ask me

- Larsby

Linked files

```
public class Frequency extends Number
{
    private static final double PITCH_OF_A4 = 57D;
    private static final double FACTOR = 12D / Math.log(2D);
    private static final String NOTE_SYMBOL[] = {
        "C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A",
        "A#", "B"
    };
    public static float frequencyOfA4 = 440F;
    private float frequency;

    public static final double getPitch(float f)
    {
        return getPitch(f);
    }

    public static final double getPitch(double d)
    {
        return 57D + FACTOR * Math.log(d / (double)frequencyOfA4);
    }

    public static final float getFrequency(double d)
    {
        return (float)(Math.exp((d - 57D) / FACTOR) * (double)frequencyOfA4);
    }

    public static final String makeNoteSymbol(double d)
    {
        int i = (int)(d + 120.5D);
        StringBuffer stringBuffer = new StringBuffer(NOTE_SYMBOL[i % 12]);
        stringBuffer.append(Integer.toString(i / 12 - 10));
        return new String(stringBuffer);
    }

    public static float valueOf(String s)
        throws IllegalArgumentException
    {
        try
        {
            return (new Float(s)).floatValue();
        }
        catch(NumberFormatException _ex) { }
        try
        {
            return getFrequency(parseNoteSymbol(s));
        }
        catch(IllegalArgumentException _ex)
        {
            throw new IllegalArgumentException("Neither a floating point number nor a valid note
symbol.");
        }
    }

    public static final int parseNoteSymbol(String s)
        throws IllegalArgumentException
    {
        s = s.trim().toUpperCase();
        for(int i = NOTE_SYMBOL.length - 1; i >= 0; i--)
        {
            if(!s.startsWith(NOTE_SYMBOL[i]))
                continue;
            try
            {
                return i + 12 * Integer.parseInt(s.substring(NOTE_SYMBOL[i].length()).trim());
            }
            catch(NumberFormatException _ex) { }
            break;
        }
    }
}
```

```

    }

    throw new IllegalArgumentException("not valid note symbol.");
}

public static void transformPitch(TextComponent textcomponent, boolean flag)
{
    boolean flag1 = false;
    String s = textcomponent.getText();
    if(flag)
    {
        try
        {
            textcomponent.setText(Integer.toString((int)(getFrequency(parseNoteSymbol(s)) +
0.5F)));
            return;
        }
        catch(IllegalArgumentException _ex)
        {
            flag1 = true;
        }
        return;
    }
    try
    {
        textcomponent.setText(makeNoteSymbol(getPitch((new Float(s)).floatValue())));
        return;
    }
    catch(NumberFormatException _ex)
    {
        flag1 = true;
    }
}

public Frequency(float f)
{
    frequency = 1.0F;
    frequency = f;
}

public Frequency(String s)
    throws IllegalArgumentException
{
    frequency = 1.0F;
    frequency = valueOf(s);
}

public byte byteValue()
{
    return (byte)(int)(frequency + 0.5F);
}

public short shortValue()
{
    return (short)(int)(frequency + 0.5F);
}

public long longValue()
{
    return (long)(frequency + 0.5F);
}

public int intValue()
{
    return (int)(frequency + 0.5F);
}

public float floatValue()
{
    return frequency;
}

public double doubleValue()
{
    return (double)frequency;
}

public String toString()
{
    return Integer.toString(intValue());
}

public String toNoteSymbol()

```

```
{
    return makeNoteSymbol(getPitch(frequency));
}

public static void main(String[] args)
{
    System.out.println(Frequency.parseNoteSymbol("C2"));
    System.out.println(Frequency.getFrequency(24));
}
}
```

Center separation in a stereo mixdown (click this to go back to the index)

References : Posted by Thiburce BELAVENTURE

Notes :

One year ago, i found a little trick to isolate or remove the center in a stereo mixdown.

My method use the time-frequency representation (FFT). I use a min fuction between left and right channels (for each bin) to create the pseudo center. I apply a phase correction, and i substract this signal to the left and right signals.

Then, we can remix them after treatments (or without) to produce a stereo signal in output.

This algorithm (I called it "TBIisolator") is not perfect, but the result is very nice, better than the phase technic (L substract R...). I know that it is not mathematically correct, but as an estimation of the center, the exact match is very hard to obtain. So, it is not so bad (just listen the result and see).

My implementation use a 4096 FFT size, with overlap-add method (factor 2). With a lower FFT size, the sound will be more dirty, and with a 16384 FFT size, the center will have too much high frequency (I don't explore why this thing appears).

I just post the TBIisolator code (see FFTReal in this site for implement the FFT engine).

pIns and pOuts buffers use the representation of the FFTReal class (0 to N/2-1: real parts, N/2 to N-1: imaginary parts).

Have fun with the TBIisolator algorithm ! I hope you enjoy it and if you enhance it, contact me (it's my baby...).

P.S.: the following function is not optimized.

```
Code :
/* ===== */
/* nFFTSIZE must be a power of 2 */
/* ===== */
/* Usage examples: */
/* - suppress the center: fAmpL = 1.f, fAmpC = 0.f, fAmpR = 1.f */
/* - keep only the center: fAmpL = 0.f, fAmpC = 1.f, fAmpR = 0.f */
/* ===== */

void processtBIisolator(float *pIns[2], float *pOuts[2], long nFFTSIZE, float fAmpL, float fAmpC, float fAmpR)
{
    float fModL, fModR;
    float fRealL, fRealC, fRealR;
    float fImagL, fImagC, fImagR;
    double u;

    for ( long i = 0, j = nFFTSIZE / 2; i < nFFTSIZE / 2; i++ )
    {
        fModL = pIns[0][i] * pIns[0][i] + pIns[0][j] * pIns[0][j];
        fModR = pIns[1][i] * pIns[1][i] + pIns[1][j] * pIns[1][j];

        // min on complex numbers
        if ( fModL > fModR )
        {
            fRealC = fRealR;
            fImagC = fImagR;
        }
        else
        {
            fRealC = fRealL;
            fImagC = fImagL;
        }

        // phase correction...
        u = fabs(atan2(pIns[0][j], pIns[0][i]) - atan2(pIns[1][j], pIns[1][i])) / 3.141592653589;

        if ( u >= 1 ) u -= 1.;

        u = pow(1 - u*u*u, 24);

        fRealC *= (float) u;
        fImagC *= (float) u;

        // center extraction...
        fRealL = pIns[0][i] - fRealC;
        fImagL = pIns[0][j] - fImagC;

        fRealR = pIns[1][i] - fRealC;
        fImagR = pIns[1][j] - fImagC;

        // You can do some treatments here...

        pOuts[0][i] = fRealL * fAmpL + fRealC * fAmpC;
        pOuts[0][j] = fImagL * fAmpL + fImagC * fAmpC;

        pOuts[1][i] = fRealR * fAmpR + fRealC * fAmpC;
        pOuts[1][j] = fImagR * fAmpR + fImagC * fAmpC;
    }
}
```

Comments

from : fmanhec [[a t]] thiburce.com

comment : I am sorry, my source code is not totally correct.

1 - the for is:

```
for ( long i = 0, j = nFFTSize / 2; i < nFFTSize / 2; i++, j++ )
```

2 - the correct min is:

```
if ( fModL > fModR )
{
    fRealC = pIns[1][i];
    fImagC = pIns[1][j];
}
else
{
    fRealC = pIns[0][i];
    fImagC = pIns[0][j];
}
```

3 - in the phase correction:

```
if ( u >= 1 ) u -= 1.;
```

must be replaced by:

```
if ( u >= 1 ) u = 2 - u;
```

Thiburce 'TB' BELAVENTURE

Center separation in a stereo mixdown (click this to go back to the index)

References : Posted by Thiburce BELAVENTURE

Notes :

One year ago, i found a little trick to isolate or remove the center in a stereo mixdown.

My method use the time-frequency representation (FFT). I use a min fuction between left and right channels (for each bin) to create the pseudo center. I apply a phase correction, and i substract this signal to the left and right signals.

Then, we can remix them after treatments (or without) to produce a stereo signal in output.

This algorithm (I called it "TBIisolator") is not perfect, but the result is very nice, better than the phase technic (L substract R...). I know that it is not mathematically correct, but as an estimation of the center, the exact match is very hard to obtain. So, it is not so bad (just listen the result and see).

My implementation use a 4096 FFT size, with overlap-add method (factor 2). With a lower FFT size, the sound will be more dirty, and with a 16384 FFT size, the center will have too much high frequency (I don't explore why this thing appears).

I just post the TBIisolator code (see FFTReal in this site for implement the FFT engine).

pIns and pOuts buffers use the representation of the FFTReal class (0 to N/2-1: real parts, N/2 to N-1: imaginary parts).

Have fun with the TBIisolator algorithm ! I hope you enjoy it and if you enhance it, contact me (it's my baby...).

P.S.: the following function is not optimized.

```
Code :
/* ===== */
/* nFFTSIZE must be a power of 2 */
/* ===== */
/* Usage examples:
/* - suppress the center: fAmpL = 1.f, fAmpC = 0.f, fAmpR = 1.f */
/* - keep only the center: fAmpL = 0.f, fAmpC = 1.f, fAmpR = 0.f */
/* ===== */

void processtBIisolator(float *pIns[2], float *pOuts[2], long nFFTSIZE, float fAmpL, float fAmpC, float fAmpR)
{
    float fModL, fModR;
    float fRealL, fRealC, fRealR;
    float fImagL, fImagC, fImagR;
    double u;

    for ( long i = 0, j = nFFTSIZE / 2; i < nFFTSIZE / 2; i++ )
    {
        fModL = pIns[0][i] * pIns[0][i] + pIns[0][j] * pIns[0][j];
        fModR = pIns[1][i] * pIns[1][i] + pIns[1][j] * pIns[1][j];

        // min on complex numbers
        if ( fModL > fModR )
        {
            fRealC = fRealR;
            fImagC = fImagR;
        }
        else
        {
            fRealC = fRealL;
            fImagC = fImagL;
        }

        // phase correction...
        u = fabs(atan2(pIns[0][j], pIns[0][i]) - atan2(pIns[1][j], pIns[1][i])) / 3.141592653589;

        if ( u >= 1 ) u -= 1.;

        u = pow(1 - u*u*u, 24);

        fRealC *= (float) u;
        fImagC *= (float) u;

        // center extraction...
        fRealL = pIns[0][i] - fRealC;
        fImagL = pIns[0][j] - fImagC;

        fRealR = pIns[1][i] - fRealC;
        fImagR = pIns[1][j] - fImagC;

        // You can do some treatments here...

        pOuts[0][i] = fRealL * fAmpL + fRealC * fAmpC;
        pOuts[0][j] = fImagL * fAmpL + fImagC * fAmpC;

        pOuts[1][i] = fRealR * fAmpR + fRealC * fAmpC;
        pOuts[1][j] = fImagR * fAmpR + fImagC * fAmpC;
    }
}
```


Cheap pseudo-sinusoidal lfo (click this to go back to the index)

References : Posted by fumminger[AT]umminger[DOT]com

Notes :

Although the code is written in standard C++, this algorithm is really better suited for dsps where one can take advantage of multiply-accumulate instructions and where the required phase accumulator can be easily implemented by masking a counter.

It provides a pretty cheap roughly sinusoidal waveform that is good enough for an lfo.

Code :

```
// x should be between -1.0 and 1.0
inline
double pseudo_sine(double x)
{
    // Compute 2*(x^2-1.0)^2-1.0
    x *= x;
    x -= 1.0;
    x *= x;
    // The following lines modify the range to lie between -1.0 and 1.0.
    // If a range of between 0.0 and 1.0 is acceptable or preferable
    // (as in a modulated delay line) then you can save some cycles.
    x *= 2.0;
    x -= 1.0;
}
```

Comments

from : bekkah [[a t]] web.de

comment : You forgot a

return x;

from : fumminger [[a t]] umminger.com

comment : Doh! You're right.

-Frederick

Clipping without branching (click this to go back to the index)

Type : Min, max and clip

References : Posted by Laurent de Soras <laurent[AT]ohmforce[DOT]com>

Notes :
It may reduce accuracy for small numbers. I.e. if you clip to [-1; 1], fractional part of the result will be quantized to 23 bits (or more, depending on the bit depth of the temporary results). Thus, 1e-20 will be rounded to 0. The other (positive) side effect is the denormal number elimination.

```
Code :
float max (float x, float a)
{
    x -= a;
    x += fabs (x);
    x *= 0.5;
    x += a;
    return (x);
}

float min (float x, float b)
{
    x = b - x;
    x += fabs (x);
    x *= 0.5;
    x = b - x;
    return (x);
}

float clip (float x, float a, float b)
{
    x1 = fabs (x-a);
    x2 = fabs (x-b);
    x = x1 + (a+b);
    x -= x2;
    x *= 0.5;
    return (x);
}
```

Comments

from : kleps [[a t]] refx.net

comment : AFAlK, the fabs() is using if()...

from : andy[AT]vellocet.com

comment : fabs/fabsf do not use if and are quicker than:

if (x<0) x = -x;

Do the speed tests yourself if you don't believe me!

from : kaleja [[a t]] estarcion.com

comment : Depends on CPU and optimization options, but yes, Visual C++/x86/full optimization uses intrinsic fabs, which is very cool.

from : lennart.denninger[AT]guerrilla-games.com

comment : And ofcourse you could always use one of those nifty bit-tricks for fabs :)

(Handy when you don't want to link with the math-library, like when coding a softsynth for a 4Kb-executable demo :))

from : andy [[a t]] a2hd.com

comment : according to my benchmarks (using the cpu clock cycle counter), fabs and the 'nifty bit tricks' have identicle performance characteristics, EXCEPT that with the nifty bit trick, sometimes it has a -horrible- penalty, which depends on the context..., maybe it does not optimize consistently? I use libmath fabs now. (i'm using gcc-3.3/linux on a P3)

from : kaleja [[a t]] estarcion.com

comment : Precision can be a major problem with these. In particular, if you have an algorithm that blows up with negative input, don't guard via clip(in, 0.0, 1.0) - it will occasionally go negative.

Constant-time exponent of 2 detector (click this to go back to the index)

References : Posted by Brent Lehman (mailbij[AT]yahoo.com)

Notes :

In your common FFT program, you want to make sure that the frame you're working with has a size that is a power of 2. This tells you in just a few operations. Granted, you won't be using this algorithm inside a loop, so the savings aren't that great, but every little hack helps ;)

Code :

```
// Quit if size isn't a power of 2
if ((-size ^ size) & size) return;

// If size is an unsigned int, the above might not compile.
// You'd want to use this instead:
if (((~size + 1) ^ size) & size) return;
```

Comments

from : functorx [[a t]] yahoo.com

comment : I think I prefer:

```
if (!(size & (size - 1))) return;
```

I'm not positive this is fewer instructions than the above, but I think it's easier to see why it works (n and n-1 will share bits unless n is a power of two), and it doesn't require two's-complement.

- Tom 7

Conversion and normalization of 16-bit sample to a floating point number (click this to go back to the index)

References : Posted by George Yohng

```
Code :  
float out;  
signed short in;  
  
// This code does the same as  
// out = ((float)in)*(1.0f/32768.0f);  
//  
// Depending on the architecture and conversion settings,  
// it might be more optimal, though it is always  
// advisable to check its speed against genuine  
// algorithms.  
  
((unsigned &)out)=0x43818000^in;  
out-=259.0f;
```

Comments

from : yuri_xl [[a t]] tom.com
comment : Hi George Yohng

I tried it... but it's create the heavy noise!!

from : George Yohng
comment : Correction:
((unsigned &)out)=0x43818000^((unsigned short)in);
out-=259.0f;

(needs to have a cast to 'unsigned short')

Conversions on a PowerPC (click this to go back to the index)

Type : motorola ASM conversions

References : Posted by James McCartney

Code :

```
double ftod(float x) { return (double)x;
00000000: 4E800020 blr
// blr == return from subroutine, i.e. this function is a noop

float dtof(double x) { return (float)x;
00000000: FC200818 frsp      fp1,fp1
00000004: 4E800020 blr

int ftoi(float x) { return (int)x;
00000000: FC00081E fctiwz   fp0,fp1
00000004: D801FFF0 stfd     fp0,-16(SP)
00000008: 8061FFF4 lwz      r3,-12(SP)
0000000C: 4E800020 blr

int dtoi(double x) { return (int)x;
00000000: FC00081E fctiwz   fp0,fp1
00000004: D801FFF0 stfd     fp0,-16(SP)
00000008: 8061FFF4 lwz      r3,-12(SP)
0000000C: 4E800020 blr

double itod(int x) { return (double)x;
00000000: C8220000 lfd      fp1,@1558(RTOC)
00000004: 6C608000 xoris    r0,r3,$8000
00000008: 9001FFF4 stw      r0,-12(SP)
0000000C: 3C004330 lis     r0,17200
00000010: 9001FFF0 stw      r0,-16(SP)
00000014: C801FFF0 lfd      fp0,-16(SP)
00000018: FC200828 fsub     fp1,fp0,fp1
0000001C: 4E800020 blr

float itof(int x) { return (float)x;
00000000: C8220000 lfd      fp1,@1558(RTOC)
00000004: 6C608000 xoris    r0,r3,$8000
00000008: 9001FFF4 stw      r0,-12(SP)
0000000C: 3C004330 lis     r0,17200
00000010: 9001FFF0 stw      r0,-16(SP)
00000014: C801FFF0 lfd      fp0,-16(SP)
00000018: EC200828 fsubs   fp1,fp0,fp1
0000001C: 4E800020 blr
```

[Copy-protection schemes](#) (click this to go back to the index)

References : Posted by Moyer, Andy

Notes :

This post of Andy sums up everything there is to know about copy-protection schemes:

"Build a great product and release improvements regularly so that people will be willing to spend the money on it, thus causing anything that is cracked to be outdated quickly. Build a strong relationship with your customers, because if they've already paid for one of your products, and were satisfied, chances are, they will be more likely to buy another one of your products. Make your copy protection good enough so that somebody can't just do a search in Google and enter in a published serial number, but don't make registered users jump through flaming hoops to be able to use the product. Also use various approaches to copy protection within a release, and vary those approaches over multiple releases so that a hacker that cracked your app's version 1.0 can't just run a recorded macro in a text editor to crack your version 2.0 software [this being simplified]."

Comments

from : ultrano [[a t]] mail.bg

comment : Won't it be good to make several versions of 1.0 with the functions places being scrambled, and unused static data being changed? And if the product uses plugins, to make each plugin detect if the code has been changed?

[Cubic interpolation](#) (click this to go back to the index)

Type : interpolation

References : Posted by Olli Niemitalo

Linked file : [other001.gif](#)

Notes :

(see linkfile)

finpos is the fractional, inpos the integer part.

Code :

```
xm1 = x [inpos - 1];
x0  = x [inpos + 0];
x1  = x [inpos + 1];
x2  = x [inpos + 2];
a = (3 * (x0-x1) - xm1 + x2) / 2;
b = 2*x1 + xm1 - (5*x0 + x2) / 2;
c = (x1 - xm1) / 2;
y [outpos] = ((a * finpos) + b) * finpos + c) * finpos + x0;
```

Cure for malicious samples (click this to go back to the index)

Type : Filters Denormals, NaNs, Infinities

References : Posted by urs[AT]u-he[DOT]com

Notes :

A lot of incidents can happen during processing samples. A nasty one is denormalization, which makes cpus consume insanely many cycles for easiest instructions.

But even worse, if you have NaNs or Infinities inside recursive structures, maybe due to division by zero, all subsequent samples that are multiplied with these values will get "infected" and become NaN or Infinity. Your sound makes BLIPPP and that was it, silence from the speakers.

Thus I've written a small function that sets all of these cases to 0.0f.

You'll notice that I treat a buffer of floats as unsigned integers. And I avoid branches by using comparison results as 0 or 1.

When compiled with GCC, this function should not create any "hidden" branches, but you should verify the assembly code anyway. Sometimes some parenthesis do the trick...

;) Urs

```
Code :
#ifndef UInt32
#define UInt32 unsigned int
#endif

void erase_All_NaN_Infinities_And_Denormals( float* inSamples, int& inNumberOfSamples )
{
    UInt32* inArrayOfFloats = (UInt32*) inSamples;

    for ( int i = 0; i < inNumberOfSamples; i++ )
    {
        UInt32 sample = *inArrayOfFloats;
        UInt32 exponent = sample & 0x7F800000;

        // exponent < 0x7F800000 is 0 if NaN or Infinity, otherwise 1
        // exponent > 0 is 0 if denormalized, otherwise 1

        int aNaN = exponent < 0x7F800000;
        int aDen = exponent > 0;

        *inArrayOfFloats++ = sample * ( aNaN & aDen );
    }
}
```

Comments

from : dont-email-me
comment : #include <inttypes.h>
and use std::uint32_t
or typedef (not #define)

```
int const & inNumberOfSamples
```

from : dont-email-me
comment : #include <inttypes.h>
and use std::uint32_t
or typedef (not #define)

```
int const & inNumberOfSamples
```

from : DevilishHabib
comment : Isn't it bad to declare variables within for loop?

If someone has VC++ standard (no optimizer included, thanks Bill :-), the cycles gained by removing denormals, will be eaten by declaring 4 variables per loop cycle, so watch out !

from : texmex [[a t]] iki.fi
comment : DevilishHabib, that's rubbish. It doesn't matter where the declaration is as long as the code works. Declaring outside the loop is the same thing (you can verify this).

Urs, nice code but you don't get rid of branches just like that. Comparison is comparison no matter what. Your code is equal to "int aNaN = exponent < 0x7F800000 ? 1 : 0;" which is equal to "int aNaN = 0; if (exponent < 0x7F800000) aNaN = 1;" If we are talking about x86 asm here, there is no instruction that would do the conditional assignment needed. MMX/SSE has it, though.

Denormal DOUBLE variables, macro (click this to go back to the index)

References : Posted by Jon Watte

Notes :

Use this macro if you want to find denormal numbers and you're using doubles...

Code :

```
#if PLATFORM_IS_BIG_ENDIAN
#define INDEX 0
#else
#define INDEX 1
#endif
inline bool is_denormal( double const & d ) {
    assert( sizeof( d ) == 2*sizeof( int ) );
    int l = ((int *)&d)[INDEX];
    return (l&0x7fe00000) != 0;
}
```

Comments

from : dont-email-me

comment : put the #if inside the function itself

Denormal numbers (click this to go back to the index)

References : Compiled by Merlijn Blaauw

Linked file : [other001.txt](#) (this linked file is included below)

Notes :

this text describes some ways to avoid denormalisation. Denormalisation happens when FPU's go mad processing very small numbers

Comments

from : andy [[a t]] a2hd.com

comment : See also the entry about 'branchless min, max and clip' by Laurent Soras in this section,

Using the following function,

```
float clip (float x, float a, float b)
{
    x1 = fabs (x-a);
    x2 = fabs (x-b);
    x = x1 + (a+b);
    x -= x2;
    x *= 0.5;
    return (x);
}
```

If you apply clipping from -1.0 to 1.0 will have a side effect of squashing denormal numbers to zero due to loss of precision on the order of $\sim 1.0.e-20$. The upside is that it is branchless, but possibly more expensive than adding noise and certainly more so than adding a DC offset.

Linked files

Denormal numbers

Here's a small recap on all proposed solutions to prevent the FPU from denormalizing:

When you feed the FPU really small values (what's the exact 'threshold' value?) the CPU will go into denormal mode to maintain precision; as such precision isn't required for audio applications and denormal operations are MUCH slower than normal operations, we want to avoid them.

All methods have been proposed by people other than me, and things I say here may be inaccurate or completely wrong :). 'Algorithms' have not been tested and can be implemented more efficiently no doubt. If I made some horrible mistakes, or left some stuff out, please let me/the list know.

**** Checking denormal processing solution:**

To detect if a denormal number has occurred, just trace your code and look up on the STAT FPU register ... if 0x0002 flag is set then a denormal operation occurred (this flag stays fixed until next FINIT)

**** Checking denormal macro solution:**

NOTES:

This is the least computationally efficient method of the lot, but has the advantage of being inaudible.

Please note that in every feedback loop, you should also check for denormals (rendering it useless on algorithms with loads of filters, feedback loops, etc).

CODE:

```
#define IS_DENORMAL(f) (((*(unsigned int *)&f)&0x7f800000)==0)
```

```
// inner-loop:
```

```
is1 = *(++in1); is1 = IS_DENORMAL(is1) ? 0.f : is1;
is2 = *(++in2); is2 = IS_DENORMAL(is2) ? 0.f : is2;
```

**** Adding noise solution:**

NOTES:

Less overhead than the first solution, but still 2 mem accesses. Because a number of the values of denormalBuf will be denormals themselves, there will always be *some* denormal overhead. However, a small percentage denormals probably isn't a problem.

Use this eq. to calculate the appropriate value of id (presuming rand() generates evenly distributed values):
id = 1/percentageOfDenormalsAllowed * denormalThreshold
(I do not know the exact value of the denormalThreshold, the value at which the FPU starts to denormal).

Possible additions to this algorithm include, noiseshaping the noise buffer, which would allow a smaller value of percentageOfDenormalsAllowed without becoming audible - however, in some algorithms, with filters and such, I think this might cause the noise to be removed, thus rendering it useless. Checking for denormals on noise generation might have a similar effect I suspect.

CODE:

```
// on construction:
float **denormalBuf = new float[getBlockSize()];

float id = 1.0E-20;

for (int i = 0; i < getBlockSize(); i++)
{
    denormalBuf[i] = (float)rand()/32768.f * id;
}

// inner-loop:

float noise = *(++noiseBuf);
is1 = *(++in1) + noise;
is2 = *(++in2) + noise;
..

** Flipping number solution:
```

NOTES:

In my opinion the way-to-go method for most applications; very little overhead (no compare/if/etc or memory access needed), there isn't a percentage of the values that will denormal and it will be inaudible in most cases.
The exact value of id will probably differ from algorithm to algorithm, but the proposed value of 1.0E-30 seemed too small to me.

CODE:

```
// on construction:
float id = 1.0E-25;

// inner-loop:
is1 = *(++in1) + id;
is2 = *(++in2) + id;
id = -id;
..

** Adding offset solution:
```

NOTES:

This is the most efficient method of the lot and is also inaudible. However, some filters will probably remove the added DC offset, thus rendering it useless.

CODE:

```
// inner-loop:

is1 = *(++in1) + 1.0E-25;
is2 = *(++in2) + 1.0E-25;

** Fix-up solution
```

You can also walk through your filter and clamp any numbers which are close enough to being denormal at the end of each block, as long as your blocks are not too large. For instance, if you implement EQ using a bi-quad, you can check the delay slots at the end of each process() call, and if any slot has a magnitude of 10^{-15} or smaller, you just clamp it to 0. This will ensure that your filter doesn't run for long times with denormal numbers; ideally (depending on the coefficients) it won't reach 10^{-35} from the 10^{-15} initial state within the time of one block of samples.

That solution uses the least cycles, and also has the nice property of generating absolute-0 output values for long stretches of absolute-0 input values; the others don't.

[Denormal numbers, the meta-text](#) (click this to go back to the index)

References : Laurent de Soras

Linked file : [denormal.pdf](#)

Notes :
This very interesting paper, written by Laurent de Soras (www.ohmforce.com) has everything you ever wanted to know about denormal numbers! And it obviously describes how you can get rid of them too!

(see linked file)

Denormalization preventer (click this to go back to the index)

References : Posted by gol

Notes :

A fast tiny numbers sweeper using integer math. Only for 32bit floats. Den_Thres is your 32bit (normalized) float threshold, something small enough but big enough to prevent future denormalizations.

EAX=input buffer

EDX=length

(adapt to your compiler)

Code :

```
MOV    ECX, EDX
LEA   EDI, [EAX+4*ECX]
NEG   ECX
MOV   EDX, Den_Thres
SHL   EDX, 1
XOR   ESI, ESI
```

@Loop:

```
MOV   EAX, [EDI+4*ECX]
LEA   EBX, [EAX*2]
CMP   EBX, EDX
CMOVB EAX, ESI
MOV   [EDI+4*ECX], EAX

INC   ECX
JNZ   @Loop
```

Denormalization preventer (click this to go back to the index)

References : Posted by didid[AT]skynet[DOT]be

Notes :
Because I still see people adding noise or offset to their signal to avoid slow denormalization, here's a piece of code to zero out (near) tiny numbers instead.

Why zeroing out is better? Because a fully silent signal is better than a little offset, or noise. A host or effect can detect silent signals and choose not to process them in a safe way.

Plus, adding an offset or noise reduces huge packets of denormalization, but still leaves some behind.

Also, truncating is what the DAZ (Denormals Are Zero) SSE flag does.

This code uses integer comparison, and a CMOV, so you need a Pentium Pro or higher.

There's no need for an SSE version, as if you have SSE code you're probably already using the DAZ flag instead (but I advise plugins not to mess with the SSE flags, as the host is likely to have DAZ switched on already). This is for FPU code. Should work much faster than crap FPU comparison.

Den_Thres is your threshold, it cannot be denormalized (would be pointless). The function is Delphi, if you want to adapt, just make sure EAX is the buffer and EDX is length (Delphi register calling convention - it's not the same in C++).

```
Code :
const Den_Thres:Single=1/$1000000;

procedure PrevFPUDen_Buffer(Buffer:Pointer;Length:Integer);
asm
    PUSH    ESI
    PUSH    EDI
    PUSH    EBX

    MOV     ECX,EDX
    LEA    EDI,[EAX+4*ECX]
    NEG    ECX
    MOV    EDX,Den_Thres
    SHL    EDX,1
    XOR    ESI,ESI

    @Loop:
    MOV    EAX,[EDI+4*ECX]
    LEA    EBX,[EAX*2]
    CMP    EBX,EDX
    CMOVB  EAX,ESI
    MOV    [EDI+4*ECX],EAX

    INC    ECX
    JNZ    @Loop

    POP    EBX
    POP    EDI
    POP    ESI
End;
```

Comments

from : scoofy [[a t]] inf.elte.hu

comment : You can zero out denormals by adding and subtracting a small number.

```
void kill_denormal_by_quantization(float &val)
{
    static const float anti_denormal = 1e-18;
    val += anti_denormal;
    val -= anti_denormal;
}
```

Reference: Laurent de Soras' great article on denormal numbers:
ldesoras.free.fr/doc/articles/denormal-en.pdf

I tend to add DC because it is faster than quantization. A slight DC offset (0.000000000000000001) won't hurt. That's -360 decibels...

from : gol

comment : >>You can zero out denormals by adding and subtracting a small number

But with drawbacks as explained in his paper.

As for the speed, not sure which is the faster. Especially since the FPU speed is too manufacturer-dependant (read: it's crap in pentiums), and mine is using integer.

>>A slight DC offset (0.000000000000000001) won't hurt

As I wrote, it really does.. hurt the sequencer, that can't detect pure silence and optimize things accordingly. A host can detect near-silence, but it can't know which offset value YOU chose, so may use a lower threshold.

from : gol

comment : Btw, I happen to see I had already posted this code, probably years ago, doh!

Anyway this version gives more explanation.

And here's more:

The `LEA EBX,[EAX*2]` is to get rid of the sign bit.

And the integer comparison of float values can be done providing both are the same sign (I'm not quite sure it works on denormals, but we don't care, since they're the ones we want to zero out, so our threshold won't be denormalized).

[Dither code](#) (click this to go back to the index)

Type : Dither with noise-shaping

References : Posted by Paul Kellett

Notes :
This is a simple implementation of highpass triangular-PDF dither (a good general-purpose dither) with optional 2nd-order noise shaping (which lowers the noise floor by 11dB below 0.1 Fs).
The code assumes input data is in the range +1 to -1 and doesn't check for overloads!

To save time when generating dither for multiple channels you can re-use lower bits of a previous random number instead of calling rand() again. e.g.
r3=(r1 & 0x7F)<<8;

```
Code :
int  r1, r2;           //rectangular-PDF random numbers
float s1, s2;         //error feedback buffers
float s = 0.5f;       //set to 0.0f for no noise shaping
float w = pow(2.0,bits-1); //word length (usually bits=16)
float wi= 1.0f/w;
float d = wi / RAND_MAX; //dither amplitude (2 lsb)
float o = wi * 0.5f;   //remove dc offset
float in, tmp;
int  out;

//for each sample...

r2=r1;                //can make HP-TRI dither by
r1=rand();            //subtracting previous rand()

in += s * (s1 + s1 - s2); //error feedback
tmp = in + o + d * (float)(r1 - r2); //dc offset and dither

out = (int)(w * tmp); //truncate downwards
if(tmp<0.0f) out--;  //this is faster than floor()

s2 = s1;
s1 = in - wi * (float)out; //error
```


[Dithering](#) (click this to go back to the index)

References : Paul Kellett

Linked file : [nsdither.txt](#) (this linked file is included below)

Notes :
(see linked file)

Linked files

Noise shaped dither (March 2000)

This is a simple implementation of highpass triangular-PDF dither with 2nd-order noise shaping, for use when truncating floating point audio data to fixed point.

The noise shaping lowers the noise floor by 11dB below 5kHz (@ 44100Hz sample rate) compared to triangular-PDF dither. The code below assumes input data is in the range +1 to -1 and doesn't check for overloads!

To save time when generating dither for multiple channels you can do things like this: `r3=(r1 & 0x7F)<<8;` instead of calling `rand()` again.

```
int  r1, r2;           //rectangular-PDF random numbers
float s1, s2;         //error feedback buffers
float s = 0.5f;       //set to 0.0f for no noise shaping
float w = pow(2.0,bits-1); //word length (usually bits=16)
float wi= 1.0f/w;
float d = wi / RAND_MAX; //dither amplitude (2 lsb)
float o = wi * 0.5f;   //remove dc offset
float in, tmp;
int  out;

//for each sample...

r2=r1;                //can make HP-TRI dither by
r1=rand();            //subtracting previous rand()

in += s * (s1 + s1 - s2); //error feedback
tmp = in + o + d * (float)(r1 - r2); //dc offset and dither

out = (int)(w * tmp);  //truncate downwards
if(tmp<0.0f) out--;    //this is faster than floor()

s2 = s1;
s1 = in - wi * (float)out; //error
```

--
paul.kellett@maxim.abel.co.uk
<http://www.maxim.abel.co.uk>

Double to Int (click this to go back to the index)

Type : pointer cast (round to zero, or 'truncate')

References : Posted by many people, implementation by Andy M00cho

Notes :
-Platform independant, literally. You have IEEE FP numbers, this will work, as long as your not expecting a signed integer back larger than 16bits :)
-Will only work correctly for FP numbers within the range of [-32768.0,32767.0]
-The FPU must be in Double-Precision mode

```
Code :
typedef double lreal;
typedef float real;
typedef unsigned long uint32;
typedef long int32;

//2^36 * 1.5, (52-_shiftamt=36) uses limited precision to floor
//16.16 fixed point representation

const lreal _double2fixmagic = 68719476736.0*1.5;
const int32 _shiftamt = 16;

#if BigEndian_
#define iexp_ 0
#define iman_ 1
#else
#define iexp_ 1
#define iman_ 0
#endif //BigEndian_

// Real2Int
inline int32 Real2Int(lreal val)
{
    val= val + _double2fixmagic;
    return ((int32*)&val)[iman_] >> _shiftamt;
}

// Real2Int
inline int32 Real2Int(real val)
{
    return Real2Int ((lreal)val);
}
```

For the x86 assembler freaks here's the assembler equivalent:

```
__double2fixmagic    dd 00000000h,04238000h

fld    AFloatingPoint Number
fadd   QWORD PTR __double2fixmagic
fstp   TEMP
movsx  eax,TEMP+2
```

Comments

from : martin.eisenberg [[a t]] udo.edu
comment : www.stereopsis.com/FPU.html credits one Sree Kotay for this code.

from : rasmus [[a t]] 3kings.dk
comment : On PC this may be faster/easier:

```
int ftoi(float x)
{
    int res;
    __asm
    {
        fld x
        fistp res
    }
    return res;
}

int dtoid(double x)
{
    return ftoi((float)x);
}
```

[Envelope Follower](#) (click this to go back to the index)

[References](#) : Posted by ers

[Code](#) :

```
#define V_ENVELOPE_FOLLOWER_NUM_POINTS 2000
class vEnvelopeFollower :
{
    public:
    vEnvelopeFollower();
    virtual ~vEnvelopeFollower();
    inline void Calculate(float *b)
    {
        envelopeVal -= *buff;
        if (*b < 0)
            envelopeVal += *buff = -*b;
        else
            envelopeVal += *buff = *b;
        if (buff++ == bufferEnd)
            buff = buffer;
    }
    void SetBufferSize(float value);
    void GetControlValue(){return envelopeVal / (float)bufferSize;}

private:
    float buffer[V_ENVELOPE_FOLLOWER_NUM_POINTS];
    float *bufferEnd, *buff, envelopeVal;
    int bufferSize;
    float val;
};

vEnvelopeFollower::vEnvelopeFollower()
{
    bufferEnd = buffer + V_ENVELOPE_FOLLOWER_NUM_POINTS-1;
    buff = buffer;
    val = 0;
    float *b = buffer;
    do
    {
        *b++ = 0;
    }while (b <= bufferEnd);
    bufferSize = V_ENVELOPE_FOLLOWER_NUM_POINTS;
    envelopeVal= 0;
}

vEnvelopeFollower::~vEnvelopeFollower()
{
}

void vEnvelopeFollower::SetBufferSize(float value)
{
    bufferEnd = buffer + (bufferSize = 100 + (int)(value * ((float)V_ENVELOPE_FOLLOWER_NUM_POINTS-102)));
    buff = buffer;
    float val = envelopeVal / bufferSize;
    do
    {
        *buff++ = val;
    }while (buff <= bufferEnd);
    buff = buffer;
}

```

[Comments](#)

[from](#) : steve [[a t]] beeka.org

[comment](#) : Nice contribution, but I have a couple of questions...

Looks like there is a typo on GetControlValue... should return a float. Also, I am not clear on the reason for it taking a pointer to a float.

Is there any noticeable speed improvement with the "if (*b < 0)" code, as opposed to using fabs? I would hope that a decent compiler library would inline this (but haven't cracked open the disassembler to find out).

Exponential curve for (click this to go back to the index)

Type : Exponential curve

References : Posted by neolit123 [at] gmail [dot] com

Notes :

When you design a frequency control for your filters you may need an exponential curve to give the lower frequencies more resolution.

F=20-20000hz

x=0-100%

Case (control middle point):

x=50%

F=1135hz

Ploted diagram with 5 points:

<http://img151.imageshack.us/img151/9938/expplotur3.jpg>

Code :

```
//tweak - 14.15005 to change middle point and F(max)
F = 19+floor(pow(4,x/14.15005))+x*20;
```

Comments

from : neolit123 [[a t]] gmail.com

comment : Yet another one! :)

This is one should be the most linear one out of the 3. The 50% appears to be exactly the same as Voxengo span midpoint.

```
//x - 0-100%
```

```
//y - 20-20k
```

```
y = floor(exp((16+x*1.20103)*log(1.059))*8.17742);
```

```
//x=0, y=20
```

```
//x=50, y=639
```

```
//x=100, y=20000
```

from : neolit123 [[a t]] gmail.com

comment : Here is another function:

This one is much more expensive but should sound more linear.

```
//t - offset
```

```
//x - 0-100%
```

```
//y - 20-20000hz
```

```
t = 64.925;
```

```
y = floor(exp(x*log(1.059))*t - t/1.45);
```

Comparison between the two:

[IMG]<http://img528.imageshack.us/img528/641/plot1nu1.jpg>[/IMG]

from : neolit123 [[a t]] gmail.com

comment : same function with the more friendly exp(x)

```
y = 19+floor(exp(x/10.2071))+x*20;
```

middle point (x=50) is still at 1135hz

[Exponential parameter mapping](#) (click this to go back to the index)

[References](#) : Posted by Russell Borogove

[Notes](#) :

Use this if you want to do an exponential map of a parameter (mParam) to a range (mMin - mMax).

Output is in mData...

[Code](#) :

```
float logmax = log10f( mMax );  
float logmin = log10f( mMin );  
float logdata = (mParam * (logmax-logmin)) + logmin;
```

```
mData = powf( 10.0f, logdata );  
if (mData < mMin)  
{  
    mData = mMin;  
}  
if (mData > mMax)  
{  
    mData = mMax;  
}
```

[Comments](#)

[from](#) : rerdavies [[a t]] msn.com

[comment](#) : No point in using heavy functions when lighter-weight functions work just as well. Use ln instead of log10f, and exp instead of pow(10,x). Log-linear is the same, no matter which base you're using, and base e is way more efficient than base 10.

[from](#) : kaleja [[a t]] estarcion.com

[comment](#) : Thanks for the tip. A set of VST param wrapper classes which offers linear float, exponential float, integer selection, and text selection controls, using this technique for the exponential response, can be found in the VST source code archive -- finally.

[from](#) : act_ion [[a t]] yahoo.com

[comment](#) : Just made my day!
pretty useful :) cheers Aktion

[from](#) : agillesp [[a t]] gmail

[comment](#) : You can trade an (expensive) ln for a (cheaper) divide here because of the logarithmic identity:

$\ln(x) - \ln(y) == \ln(x/y)$

[fast abs/neg/sign for 32bit floats](#) (click this to go back to the index)

Type : floating point functions

References : Posted by tobybear[AT]web[DOT]de

Notes :

Haven't seen this elsewhere, probably because it is too obvious? Anyway, these functions are intended for 32-bit floating point numbers only and should work a bit faster than the regular ones.

fastabs() gives you the absolute value of a float
fastneg() gives you the negative number (faster than multiplying with -1)
fastsgn() gives back +1 for 0 or positive numbers, -1 for negative numbers

Comments are welcome (tobybear[AT]web[DOT]de)

Cheers

Toby (www.tobybear.de)

Code :

```
// C/C++ code:
float fastabs(float f)
{int i=((*(int*)&f)&0x7fffffff);return (*(float*)&i);}

float fastneg(float f)
{int i=((*(int*)&f)^0x80000000);return (*(float*)&i);}

int fastsgn(float f)
{return 1+(((*(int*)&f)>>31)<<1);}

//Delphi/Pascal code:
function fastabs(f:single):single;
begin i:=longint((@f)^) and $7FFFFFFF;result:=single((@i)^) end;

function fastneg(f:single):single;
begin i:=longint((@f)^) xor $80000000;result:=single((@i)^) end;

function fastsgn(f:single):longint;
begin result:=1+((longint((@f)^) shr 31) shl 1) end;
```

Comments

from : tobybear [[a t]] web.de

comment : Matthias (bekkah[AT]web[DOT]de) wrote me a mail with the following further improvements for the C++ parts of the code:

```
// C++ code:
inline float fastabs(const float f)
{int i=((*(int*)&f)&0x7fffffff);return (*(float*)&i);}

inline float fastneg(const float f)
{int i=((*(int*)&f)^0x80000000);return (*(float*)&i);}

inline int fastsgn(const float f)
{return 1+(((*(int*)&f)>>31)<<1);}

Thanks!
```

from : picoder [[a t]] mail.ru

comment : Too bad these 'tricks' need two additional FWAITS to work in a raw FPU code. Maybe standard fabs and fneg are better? Although, that fastsgn() could be useful since there's no FPU equivalent for it.

Cheers,
Aleksey.

from : picoder [[a t]] mail.ru

comment : I meant 'fchs' in place of 'fneg'.

from : david [[a t]] brannvall.net

comment : I don't know if this is any faster, but atleast you can avoid some typecasting.

```
function fastabs(f: Single): Single; var i: Integer absolute f;
begin i := i and $7fffffff; Result := f; end;
```

from : chris [[a t]] m-audio.com

comment : Note that a reasonable compiler should be able to perform these optimizations for you. I seem to recall that GCC in particular has the capability to replace calls to [f]abs() with instructions optimized for the platform.

from : kaleja [[a t]] estarcion.com

comment : On MS compilers for x86, just do:
#pragma intrinsic(fabs)

...and then use fabs() for doubles, fabsf() for floats. The compiler will generate the FABS instruction, which is generally 1 cycle on modern x86 FPUs. (Internally, the FPU just masks the bit.)

Fast binary log approximations (click this to go back to the index)

Type : C code

References : Posted by musicdsp.org[AT]mindcontrol.org

Notes :
This code uses IEEE 32-bit floating point representation knowledge to quickly compute approximations to the log2 of a value. Both functions return under-estimates of the actual value, although the second flavour is less of an under-estimate than the first (and might be sufficient for using in, say, a dBV/FS level meter).

Running the test program, here's the output:

```
0.1: -4 -3.400000
1: 0 0.000000
2: 1 1.000000
5: 2 2.250000
100: 6 6.562500
```

Code :

```
// Fast logarithm (2-based) approximation
// by Jon Watte

#include <assert.h>

int floorOfLn2( float f ) {
    assert( f > 0. );
    assert( sizeof(f) == sizeof(int) );
    assert( sizeof(f) == 4 );
    return (((*(int *)&f)&0x7f800000)>>23)-0x7f;
}

float approxLn2( float f ) {
    assert( f > 0. );
    assert( sizeof(f) == sizeof(int) );
    assert( sizeof(f) == 4 );
    int i = (*(int *)&f);
    return (((i&0x7f800000)>>23)-0x7f)+(i&0x007fffff)/(float)0x800000;
}

// Here's a test program:

#include <stdio.h>

// insert code from above here

int
main()
{
    printf( "0.1: %d %f\n", floorOfLn2( 0.1 ), approxLn2( 0.1 ) );
    printf( "1: %d %f\n", floorOfLn2( 1. ), approxLn2( 1. ) );
    printf( "2: %d %f\n", floorOfLn2( 2. ), approxLn2( 2. ) );
    printf( "5: %d %f\n", floorOfLn2( 5. ), approxLn2( 5. ) );
    printf( "100: %d %f\n", floorOfLn2( 100. ), approxLn2( 100. ) );
    return 0;
}
```

Comments

from : tobybear [[a t]] web.de

comment : Here is some code to do this in Delphi/Pascal:

```
function approxLn2(f:single):single;
begin
    result:=(((longint((@f)^) and $7f800000) shr 23)-$7f)+(longint((@f)^) and $007fffff)/$800000;
end;
```

```
function floorOfLn2(f:single):longint;
begin
    result:=(((longint((@f)^) and $7f800000) shr 23)-$7f);
end;
```

Cheers,

Tobybear
www.tobybear.de

Fast cube root, square root, and reciprocal for x86/SSE CPUs. (click this to go back to the index)

References : Posted by kaleja[AT]estarcion[DOT]com

Notes :

All of these methods use SSE instructions or bit twiddling tricks to get a rough approximation to cube root, square root, or reciprocal, which is then refined with one or more Newton-Raphson approximation steps.

Each is named to indicate its approximate level of accuracy and a comment describes its performance relative to the conventional versions.

```
Code :
// These functions approximate reciprocal, square root, and
// cube root to varying degrees of precision substantially
// faster than the straightforward methods 1/x, sqrtf(x),
// and powf( x, 1.0f/3.0f ). All require SSE-enabled CPU & OS.
// Unlike the powf() solution, the cube roots also correctly
// handle negative inputs.

#define REALLY_INLINE __forceinline

// ~34 clocks on Pentium M vs. ~360 for powf
REALLY_INLINE float cuberoot_sse_8bits( float x )
{
    float z;
    static const float three = 3.0f;
    _asm
    {
        mov  eax, x    // x as bits
        movss xmm2, x  // x2: x
        movss xmm1, three // x1: 3
        // Magic floating point cube root done with integer math.
        // The exponent is divided by three in such a way that
        // remainder bits get shoved into the top of the normalized
        // mantissa.
        mov  ecx, eax // copy of x
        and  eax, 0x7FFFFFFF // exponent & mantissa of x in biased-127
        sub  eax, 0x3F800000 // exponent & mantissa of x in 2's comp
        sar  eax, 10 //
        imul eax, 341 // 341/1024 ~= .333
        add  eax, 0x3F800000 // back to biased-127
        and  eax, 0x7FFFFFFF // remask
        and  ecx, 0x80000000 // original sign and mantissa
        or   eax, ecx // masked new exponent, old sign and mantissa
        mov  z, eax //

        // use SSE to refine the first approximation
        movss xmm0, z // x0: z
        movss xmm3, xmm0 // x3: z
        mulss xmm3, xmm0 // x3: z*z
        movss xmm4, xmm3 // x4: z*z
        mulss xmm3, xmm1 // x3: 3*z*z
        rcpss xmm3, xmm3 // x3: ~ 1/(3*z*z)
        mulss xmm4, xmm0 // x4: z*z*z
        subss xmm4, xmm2 // x4: z*z*z-x
        mulss xmm4, xmm3 // x4: (z*z*z-x) / (3*z*z)
        subss xmm0, xmm4 // x0: z' accurate to within about 0.3%
        movss z, xmm0
    }

    return z;
}

// ~60 clocks on Pentium M vs. ~360 for powf
REALLY_INLINE float cuberoot_sse_16bits( float x )
{
    float z;
    static const float three = 3.0f;
    _asm
    {
        mov  eax, x    // x as bits
        movss xmm2, x  // x2: x
        movss xmm1, three // x1: 3
        // Magic floating point cube root done with integer math.
        // The exponent is divided by three in such a way that
        // remainder bits get shoved into the top of the normalized
        // mantissa.
        mov  ecx, eax // copy of x
        and  eax, 0x7FFFFFFF // exponent & mantissa of x in biased-127
        sub  eax, 0x3F800000 // exponent & mantissa of x in 2's comp
        sar  eax, 10 //
        imul eax, 341 // 341/1024 ~= .333
        add  eax, 0x3F800000 // back to biased-127
        and  eax, 0x7FFFFFFF // remask
        and  ecx, 0x80000000 // original sign and mantissa
        or   eax, ecx // masked new exponent, old sign and mantissa
        mov  z, eax //

        // use SSE to refine the first approximation
    }
}
```

```

movss xmm0, z    ;// x0: z
movss xmm3, xmm0 ;// x3: z
mulss xmm3, xmm0 ;// x3: z*z
movss xmm4, xmm3 ;// x4: z*z
mulss  xmm3, xmm1 ;// x3: 3*z*z
rcpss xmm3, xmm3 ;// x3: ~ 1/(3*z*z)
mulss  xmm4, xmm0 ;// x4: z*z*z
subss  xmm4, xmm2 ;// x4: z*z*z-x
mulss  xmm4, xmm3 ;// x4: (z*z*z-x) / (3*z*z)
subss  xmm0, xmm4 ;// x0: z' accurate to within about 0.3%

movss xmm3, xmm0 ;// x3: z
mulss xmm3, xmm0 ;// x3: z*z
movss xmm4, xmm3 ;// x4: z*z
mulss  xmm3, xmm1 ;// x3: 3*z*z
rcpss xmm3, xmm3 ;// x3: ~ 1/(3*z*z)
mulss  xmm4, xmm0 ;// x4: z*z*z
subss  xmm4, xmm2 ;// x4: z*z*z-x
mulss  xmm4, xmm3 ;// x4: (z*z*z-x) / (3*z*z)
subss  xmm0, xmm4 ;// x0: z'' accurate to within about 0.001%

movss z, xmm0
}

return z;
}

// ~77 clocks on Pentium M vs. ~360 for powf
REALLY_INLINE float cuberoot_sse_22bits( float x )
{
float z;
static const float three = 3.0f;
_asm
{
mov  eax, x    // x as bits
movss xmm2, x  // x2: x
movss xmm1, three // x1: 3
// Magic floating point cube root done with integer math.
// The exponent is divided by three in such a way that
// remainder bits get shoved into the top of the normalized
// mantissa.
mov  ecx, eax // copy of x
and  eax, 0x7FFFFFFF // exponent & mantissa of x in biased-127
sub  eax, 0x3F800000 // exponent & mantissa of x in 2's comp
sar  eax, 10 //
imul eax, 341 // 341/1024 ~= .333
add  eax, 0x3F800000 // back to biased-127
and  eax, 0x7FFFFFFF // remask
and  ecx, 0x80000000 // original sign and mantissa
or   eax, ecx // masked new exponent, old sign and mantissa
mov  z, eax //

// use SSE to refine the first approximation
movss xmm0, z // x0: z
movss xmm3, xmm0 // x3: z
mulss xmm3, xmm0 // x3: z*z
movss xmm4, xmm3 // x4: z*z
mulss  xmm3, xmm1 // x3: 3*z*z
rcpss xmm3, xmm3 // x3: ~ 1/(3*z*z)
mulss  xmm4, xmm0 // x4: z*z*z
subss  xmm4, xmm2 // x4: z*z*z-x
mulss  xmm4, xmm3 // x4: (z*z*z-x) / (3*z*z)
subss  xmm0, xmm4 // x0: z' accurate to within about 0.3%

movss xmm3, xmm0 // x3: z
mulss xmm3, xmm0 // x3: z*z
movss xmm4, xmm3 // x4: z*z
mulss  xmm3, xmm1 // x3: 3*z*z
rcpss xmm3, xmm3 // x3: ~ 1/(3*z*z)
mulss  xmm4, xmm0 // x4: z*z*z
subss  xmm4, xmm2 // x4: z*z*z-x
mulss  xmm4, xmm3 // x4: (z*z*z-x) / (3*z*z)
subss  xmm0, xmm4 // x0: z'' accurate to within about 0.001%

movss xmm3, xmm0 // x3: z
mulss xmm3, xmm0 // x3: z*z
movss xmm4, xmm3 // x4: z*z
mulss  xmm3, xmm1 // x3: 3*z*z
rcpss xmm3, xmm3 // x3: ~ 1/(3*z*z)
mulss  xmm4, xmm0 // x4: z*z*z
subss  xmm4, xmm2 // x4: z*z*z-x
mulss  xmm4, xmm3 // x4: (z*z*z-x) / (3*z*z)
subss  xmm0, xmm4 // x0: z''' accurate to within about 0.000012%

movss z, xmm0
}

return z;
}

// ~6 clocks on Pentium M vs. ~24 for single precision sqrtf
REALLY_INLINE float squareroot_sse_11bits( float x )
{

```

```

float z;
_asm
{
    rsqrtss xmm0, x
    rcpss xmm0, xmm0
    movss z, xmm0 // z ~= sqrt(x) to 0.038%
}
return z;
}

// ~19 clocks on Pentium M vs. ~24 for single precision sqrtf
REALLY_INLINE float squareroot_sse_22bits( float x )
{
    static float half = 0.5f;
    float z;
    _asm
    {
        movss xmm1, x // x1: x
        rsqrtss xmm2, xmm1 // x2: ~1/sqrt(x) = 1/z
        rcpss xmm0, xmm2 // x0: z == ~sqrt(x) to 0.05%

        movss xmm4, xmm0 // x4: z
        movss xmm3, half
        mulss xmm4, xmm4 // x4: z*z
        mulss xmm2, xmm3 // x2: 1 / 2z
        subss xmm4, xmm1 // x4: z*z-x
        mulss xmm4, xmm2 // x4: (z*z-x)/2z
        subss xmm0, xmm4 // x0: z' to 0.000015%

        movss z, xmm0
    }
    return z;
}

// ~12 clocks on Pentium M vs. ~16 for single precision divide
REALLY_INLINE float reciprocal_sse_22bits( float x )
{
    float z;
    _asm
    {
        rcpss xmm0, x // x0: z ~= 1/x
        movss xmm2, x // x2: x
        movss xmm1, xmm0 // x1: z ~= 1/x
        addss xmm0, xmm0 // x0: 2z
        mulss xmm1, xmm1 // x1: z^2
        mulss xmm1, xmm2 // x1: xz^2
        subss xmm0, xmm1 // x0: z' ~= 1/x to 0.000012%

        movss z, xmm0
    }
    return z;
}

```

Comments

from : Christian [[a t]] savioursofsoul.de

comment : Good job!

from : martini [[a t]] slightlyintoxicated.com

comment : Dude, what's the freakin' point of using SSE for scalar functions?

from : kaleja [[a t]] estarcion.com

comment : The freakin' point is that it's faster than x87, and sometimes you can't do 4 at once. Also, the function signatures (float func(float)) are well established for the scalar versions, but different people have different standards for the vector versions (float* vs __m128& or whatever).

Converting from scalar to vector is trivial for the square root and reciprocal examples, and I'm not going to take the time to investigate a 2-way MMX solution to the first approximation of cube root, but you can feel free.

from : nobody [[a t]] nowhere.com

comment : These are terrific. Wish I had seen them sooner.

from : <Mail>

comment : order cheap soma ,cheap soma
cheap soma

[fast exp\(\) approximations](#) (click this to go back to the index)

Type : Taylor series approximation

References : Posted by scoofy[AT]inf[DOT]elte[DOT]hu

Notes :

I needed a fast exp() approximation in the -3.14..3.14 range, so I made some approximations based on the tanh() code posted in the archive by Fuzzpilz. Should be pretty straightforward, but someone may find this useful.

The increasing numbers in the name of the function means increasing precision. Maximum error in the -1..1 range:

fastexp3: 0.05 (1.8%)
fastexp4: 0.01 (0.36%)
fastexp5: 0.0016152 (0.59%)
fastexp6: 0.0002263 (0.0083%)
fastexp7: 0.0000279 (0.001%)
fastexp8: 0.0000031 (0.00011%)
fastexp9: 0.0000003 (0.000011%)

Maximum error in the -3.14..3.14 range:

fastexp3: 8.8742 (38.4%)
fastexp4: 4.8237 (20.8%)
fastexp5: 2.28 (9.8%)
fastexp6: 0.9488 (4.1%)
fastexp7: 0.3516 (1.5%)
fastexp8: 0.1172 (0.5%)
fastexp9: 0.0355 (0.15%)

These were done using the Taylor series, for example I got fastexp4 by using:

```
exp(x) = 1 + x + x^2/2 + x^3/6 + x^4/24 + ...  
= (24 + 24x + x^2*12 + x^3*4 + x^4) / 24  
(using Horner-scheme:  
= (24 + x * (24 + x * (12 + x * (4 + x)))) * 0.041666666f
```

Code :

```
inline float fastexp3(float x) {  
    return (6+x*(6+x*(3+x)))*0.16666666f;  
}  
  
inline float fastexp4(float x) {  
    return (24+x*(24+x*(12+x*(4+x))))*0.041666666f;  
}  
  
inline float fastexp5(float x) {  
    return (120+x*(120+x*(60+x*(20+x*(5+x)))))*0.0083333333f;  
}  
  
inline float fastexp6(float x) {  
    return 720+x*(720+x*(360+x*(120+x*(30+x*(6+x)))))*0.0013888888f;  
}  
  
inline float fastexp7(float x) {  
    return (5040+x*(5040+x*(2520+x*(840+x*(210+x*(42+x*(7+x))))))*0.00019841269f;  
}  
  
inline float fastexp8(float x) {  
    return (40320+x*(40320+x*(20160+x*(6720+x*(1680+x*(336+x*(56+x*(8+x))))))*2.4801587301e-5;  
}  
  
inline float fastexp9(float x) {  
    return (362880+x*(362880+x*(181440+x*(60480+x*(15120+x*(3024+x*(504+x*(72+x*(9+x))))))*2.75573192e-6;  
}
```

Comments

from : scoofy [[a t]] inf.elte.hu

comment : These series converge fast only near zero. But there is an identity:

$\exp(x) = \exp(a) * \exp(x-a)$

So, if you want a relatively fast polynomial approximation for exp(x) for 0 to ~7.5, you can use:

```
// max error in the 0 .. 7.5 range: ~0.45%  
inline float fastexp(float const &x)  
{  
    if (x<2.5)  
        return 2.7182818f * fastexp5(x-1.f);  
    else if (x<5)  
        return 33.115452f * fastexp5(x-3.5f);  
    else  
        return 403.42879f * fastexp5(x-6.f);  
}
```

where 2.7182.. = exp(1), 33.1154.. = exp(3.5) and 403.428.. = exp(6). I chose these values because fastexp5 has a maximum error of 0.45% between -

1 - 1.5 (using fastexp6, the maximum error is 0.09%).

Using the identity

```
pow(a,x) = exp(x * log(a))
```

you can use any base, for example to get 2^x:

```
// max error in the 0-10.58 range: ~0.45%
inline float fastpow2(float const &x)
{
    float const log_two = 0.6931472f;
    return fastexp(x * log_two);
}
```

These functions are about 3x faster than exp().

-- Peter Schoffhauer

[Fast exp2 approximation](#) (click this to go back to the index)

[References](#) : Posted by Laurent de Soras <laurent[AT]ohmforce[DOT]com>

Notes :

Partial approximation of exp2 in fixed-point arithmetic. It is exactly :

$[0 ; 1[\rightarrow [0.5 ; 1[$

$f : x \mapsto 2^{x-1}$

To get the full exp2 function, you have to separate the integer and fractionnal part of the number. The integer part may be processed by bitshifting.

Process the fractionnal part with the function, and multiply the two results.

Maximum error is only 0.3 % which is pretty good for two mul ! You get also the continuity of the first derivate.

-- Laurent

Code :

```
// val is a 16-bit fixed-point value in 0x0 - 0xFFFF ([0 ; 1[)
// Returns a 32-bit fixed-point value in 0x80000000 - 0xFFFFFFFF ([0.5 ; 1[)
unsigned int fast_partial_exp2 (int val)
{
    unsigned int    result;

    __asm
    {
        mov eax, val
        shl eax, 15          ; eax = input [31/31 bits]
        or  eax, 080000000h  ; eax = input + 1 [32/31 bits]
        mul eax
        mov eax, edx        ; eax = (input + 1) ^ 2 [32/30 bits]
        mov edx, 2863311531 ; 2/3 [32/32 bits], rounded to +oo
        mul edx             ; eax = 2/3 (input + 1) ^ 2 [32/30 bits]
        add edx, 1431655766 ; + 4/3 [32/30 bits] + 1
        mov result, edx
    }

    return (result);
}
```

[Fast_log2](#) (click this to go back to the index)

[References](#) : Posted by Laurent de Soras

```
Code :
inline float fast_log2 (float val)
{
    assert (val > 0);

    int * const exp_ptr = reinterpret_cast <int *> (&val);
    int x = *exp_ptr;
    const int log_2 = ((x >> 23) & 255) - 128;
    x &= ~(255 << 23);
    x += 127 << 23;
    *exp_ptr = x;

    return (val + log_2);
}
```

Comments

[from](#) : tobybear [[a t]] web.de
[comment](#) : And here is some native Delphi/Pascal code that does the same thing:

```
function fast_log2(val:single):single;
var log2,x:longint;
begin
x:=longint((@val)^);
log2:=(x shr 23) and 255-128;
x:=x and (not(255 shl 23));
x:=x+127 shl 23;
result:=single((@x)^)+log2;
end;
```

Cheers

Toby

www.tobybear.de

[from](#) : henry [[a t]] cordylus.de
[comment](#) : instead of using this pointer casting expressions one can also use a enum like this:

```
enum FloatInt
{
float f;
```

[from](#) : henry [[a t]] cordylus.de
[comment](#) : instead of using this pointer casting expressions one can also use a enum like this:

```
enum FloatInt
{
float f;
int i;
} p;
```

and then access the data with:

```
p.f = x;
p.i >>= 23;
```

Greetings, Henry

[from](#) : henry [[a t]] cordylus.de
[comment](#) : Sorry :

didnt mean enum, ment UNION !!!

[from](#) : Laurent de Soras
[comment](#) :

More precision can be obtained by adding the following line just before the return() :

```
val = map_lin_2_exp (val, 1.0f / 2);
```

Below is the function (everything is constant, so most operations should be done at compile time) :

```
inline float map_lin_2_exp (float val, float k)
{
    const float a = (k - 1) / (k + 1);
    const float b = (4 - 2*k) / (k + 1); // 1 - 3*a
    const float c = 2*a;
    val = (a * val + b) * val + c;
```

```
    return (val);  
}
```

You can do the mapping you want for the range [1;2] -> [1;2] to approximate the function $\log(x)/\log(2)$.

from : Laurent de Soras

comment : Sorry I meant $\log(x)/\log(2) + 1$

[fast power and root estimates for 32bit floats](#) (click this to go back to the index)

Type : floating point functions

References : Posted by tobybear[AT]web[DOT]de

Notes :

Original code by Stefan Stenzel (also in this archive, see "pow(x,4) approximation") - extended for more flexibility.

fastpow(f,n) gives a rather *rough* estimate of a float number f to the power of an integer number n ($y=f^n$). It is fast but result can be quite a bit off, since we directly mess with the floating point exponent.-> use it only for getting rough estimates of the values and where precision is not that important.

fastroot(f,n) gives the n-th root of f. Same thing concerning precision applies here.

Cheers

Toby (www.tobybear.de)

Code :

//C/C++ source code:

```
float fastpower(float f,int n)
{
    long *lp,l;
    lp=(long*)&f;
    l=*lp;l-=0x3F800001;l<<=(n-1);l+=0x3F800001;
    *lp=l;
    return f;
}
```

```
float fastroot(float f,int n)
{
    long *lp,l;
    lp=(long*)&f;
    l=*lp;l-=0x3F800001;l>=(n-1);l+=0x3F800001;
    *lp=l;
    return f;
}
```

//Delphi/Pascal source code:

```
function fastpower(i:single;n:integer):single;
var l:longint;
begin
    l:=longint((@i)^);
    l:=l-$3F800000;l:=l shl (n-1);l:=l+$3F800000;
    result:=single((@l)^);
end;
```

```
function fastroot(i:single;n:integer):single;
var l:longint;
begin
    l:=longint((@i)^);
    l:=l-$3F800000;l:=l shr (n-1);l:=l+$3F800000;
    result:=single((@l)^);
end;
```

Fast rounding functions in pascal (click this to go back to the index)

Type : round/ceil/floor/trunc

References : Posted by bouba@hotmail.com

Notes :

Original documentation with cpp samples:
http://ldeoras.free.fr/prod.html#doc_rounding

Code :

Pascal translation of the functions (accurates ones) :

```
function ffloor(f:double):integer;
var
  i:integer;
  t : double;
begin
  t := -0.5 ;
  asm
    fld  f
    fadd st,st(0)
    fadd t
    fistp i
    sar  i, 1
end;
result:= i
end;
```

```
function fceil(f:double):integer;
var
  i:integer;
  t : double;
begin
  t := -0.5 ;
  asm
    fld  f
    fadd st,st(0)
    fsubr t
    fistp i
    sar  i, 1
end;
result:= -i
end;
```

```
function ftrunc(f:double):integer;
var
  i:integer;
  t : double;
begin
  t := -0.5 ;
  asm
    fld  f
    fadd st,st(0)
    fabs
    fadd t
    fistp i
    sar  i, 1
end;
if f<0 then i := -i;
result:= i
end;
```

```
function fround(f:double):integer;
var
  i:integer;
  t : double;
begin
  t := 0.5 ;
  asm
    fld  f
    fadd st,st(0)
    fadd t
    fistp i
    sar  i, 1
end;
result:= i
end;
```

Comments

from : didid [[a t]] noskynetspam.be

comment : the fround doesn't make much sense in Pascal, as in Pascal (well, Delphi & I'm pretty sure FreePascal too), the default rounding is already a fast rounding. The default being FPU rounding to nearest mode, the compiler doesn't change it back & forth. & since it's inlined (well, compiler magic), it's very fast.

Fast sign for 32 bit floats (click this to go back to the index)

References : Posted by Peter Schoffhauzer

Notes :

Fast functions which give the sign of a 32 bit floating point number by checking the sign bit. There are two versions, one which gives the value as a float, and the other gives an int.

The `_nozero` versions are faster, but they give incorrect 1 or -1 for zero (depending on the sign bit set in the number). The int version should be faster than the Tobybear one in the archive, since this one doesn't have an addition, just bit operations.

Code :

```
/*-----
   fast sign, returns float
-----*/

// returns 1.0f for positive floats, -1.0f for negative floats
// for zero it does not work (may gives 1.0f or -1.0f), but it's faster
inline float fast_sign_nozero(float f) {
    float r = 1.0f;
    (int&r) |= ((int&)f & 0x80000000); // mask sign bit in f, set it in r if necessary
    return r;
}

// returns 1.0f for positive floats, -1.0f for negative floats, 0.0f for zero
inline float fast_sign(float f) {
    if (((int&)f & 0x7FFFFFFF)==0) return 0.f; // test exponent & mantissa bits: is input zero?
    else {
        float r = 1.0f;
        (int&r) |= ((int&)f & 0x80000000); // mask sign bit in f, set it in r if necessary
        return r;
    }
}

/*-----
   fast sign, returns int
-----*/

// returns 1 for positive floats, -1 for negative floats
// for 0.0f input it does not work (may give 1 or -1), but it's faster
inline int fast_sign_int_nozero(float f) {
    return (signed((int&)f & 0x80000000) >> 31) | 1;
}

// returns 1 for positive floats, -1 for negative floats, 0 for 0.0f
inline int fast_sign_int(float f) {
    if (((int&)f & 0x7FFFFFFF)==0) return 0; // test exponent & mantissa bits: is input zero?
    return (signed((int&)f & 0x80000000) >> 31) | 1;
}
}
```

Comments

from : scoofy [[a t]] inf.elte.hu

comment : Now consider when you want to multiply a number by the sign of another:

```
if (a>0.f) b = b;
else b = -b;
```

This involves 1) a comparison, 2) a branch, 3) an inversion (multiply or bit flip) in one branch. Another method for calculating the same:

```
b *= fast_sign_nozero_(a);
```

This involves 1) a bitwise and, 2) a bitwise or 3) and a multiply. Using only bit operations, the branch and/or multiply can be totally eliminated:

```
// equivalent to dest *= sgn(source)
inline void mul_sign(float &dest, float &source) {
    (int&dest) &= 0x7FFFFFFF; // clear sign bit
    (int&dest) |= ((int&dest) ^ (int&source) & 0x80000000); // set sign bit if necessary
}
}
```

This function has only three bitwise operations, which should be very fast. Usage:

```
mul_sign(b,a); // b = b*sign(a)
```

The speed increase with all these functions greatly depends on the predictability of the branches. If the branch is highly predictable (a lot of positive numbers, then a lot of negative numbers, without mixing them), then an if/else solution is pretty fast. If the branch is unpredictable (random numbers, or audio similar to white noise) then bit operations should perform significantly better on today's most CPUs with multi-level pipelines.

-- Peter Schoffhauzer

from : scoofy [[a t]] inf.elte.hu

comment : Sorry, there is a bug in the above code. Correctly:

```
// equivalent to dest *= sgn(source)
inline void mul_sign_nozero(float &dest, float const &source) {
    int sign_mask = ((int&dest) ^ (int&source) & 0x80000000); // XOR and mask
    (int&dest) &= 0x7FFFFFFF; // clear sign bit
}
```

```
(int&)dest |= sign_mask; // set sign bit if necessary  
}
```

[Float to int](#) (click this to go back to the index)

[References](#) : Posted by Ross Bencina

[Notes](#) :

intel only

[Code](#) :

```
int truncate(float flt)
{
    int i;
    static const double half = 0.5f;
    _asm
    {
        fld flt
        fsub half
        fistp i
    }
    return i
}
```

[Comments](#)

[from](#) : jaha [[a t]] smartelectronix.com

[comment](#) : Note: Round nearest doesn't work, because the Intel FPU uses Even-Odd rounding in order to conform to the IEEE floating-point standard: when the FPU is set to use the round-nearest rule, values whose fractional part is exactly 0.5 round toward the nearest *even* integer. Thus, 1.5 rounds to 2, 2.5 rounds to 2, 3.5 rounds to 4. This is quite disastrous for the FLOOR/CEIL functions if you use the strategy you describe.

[from](#) : sugonaut [[a t]] yahoo.com

[comment](#) : This version below seems to be more accurate on my Win32/P4. Doesn't deal with the Intel FPU issue. A faster solution than c-style casts though. But you're not always going to get the most accurate conversion. Like the previous comment; 2.5 will convert to 2, but 2.501 will convert to 3.

```
int truncate(float flt)
```

```
{
    int i;
    _asm
    {
        fld flt
        fistp i
    }
    return i
}
```

[from](#) : kuma [[a t]] nowhere.com

[comment](#) : if you use msvc, just use the /Qlfist compile-switch

[Float to int \(more intel asm\)](#) (click this to go back to the index)

References : Posted by Laurent de Soras (via flipcode)

Notes :

[Found this on flipcode, seemed worth posting here too, hopefully Laurent will approve :) -- Ross]

Here is the code I often use. It is not a `_ftol` replacement, but it provides useful functions. The current processor rounding mode should be "to nearest" ; it is the default setting for most of the compilers.

The `[fadd st, st (0) / sar i,1]` trick fixes the "round to nearest even number" behaviour. Thus, `round_int (N+0.5)` always returns `N+1` and `floor_int` function is appropriate to convert floating point numbers into fixed point numbers.

Laurent de Soras
Audio DSP engineer & software designer
Ohm Force - Digital audio software
<http://www.ohmforce.com>

Code :

```
inline int round_int (double x)
{
    int    i;
    static const float round_to_nearest = 0.5f;
    __asm
    {
        fld     x
        fadd   st, st (0)
        fadd   round_to_nearest
        fistp  i
        sar   i, 1
    }
    return (i);
}

inline int floor_int (double x)
{
    int    i;
    static const float round_toward_m_i = -0.5f;
    __asm
    {
        fld     x
        fadd   st, st (0)
        fadd   round_toward_m_i
        fistp  i
        sar   i, 1
    }
    return (i);
}

inline int ceil_int (double x)
{
    int    i;
    static const float round_toward_p_i = 0.5f;
    __asm
    {
        fld     x
        fadd   st, st (0)
        fsubr  round_toward_p_i
        fistp  i
        sar   i, 1
    }
    return (-i);
}
```

Comments

from : [daniel.schaack\[-dot\]-basementarts.de](mailto:daniel.schaack@basementarts.de)

comment : cool trick, but using round to zero FPU mode is still the best methode (2 lines):

```
__asm
{
    fld x
    fistp y
}
```

from : [ejd \[\[a t \]\] dork.com](mailto:ejd@t.dork.com)

comment : If anyone is using NASM, here is how I implemented the first function, if anyone is interested. I did this after sitting down for a while with the intel manuals. I've not done any x86-FPU stuff before, so this may not be the "best" code. The other functions are easily implemented by modifying this one.

bits 32
global dsp_round

HALF dq 0.5

```
align 4
dsp_round:
fld qword[HALF]
fld qword[esp+4]
```

```
fadd st0
fadd st1
```

```
fistp qword[eax]
mov eax, [eax]
sar eax, 1
ret
```

from : ejd [[a t]] dork.com

comment : Whoops. I've really gone and embarrassed myself this time. The code I posted is actually broken -- I don't know what I was thinking. I'll post the fixed version a bit later today. My appologies.

from : hammer [[a t]] utcnetwork.jp

comment : Will this method also work for other processor types like AMD and CELERON?

from : scoofy [[a t]] inf.elte.hu

comment : It works with AMD and Celeron too (and as I know, probably with all x87 FPU's)

Float to integer conversion (click this to go back to the index)

References : Posted by Peter Schoffhauzer

Notes :

The fld x/fistp i instructions can be used for fast conversion of floating point numbers to integers. By default, the number is rounded to the nearest integer. However, sometimes that's not what we need.

Bits 12 and 11 of the FPU control word determine the way the FPU rounds numbers. The four rounding states are:

00 = round to nearest (this is the default)
01 = round down (towards -infinity)
10 = round up (towards +infinity)
11 = truncate up (towards zero)

The status word is loaded/stored using the fldcw/fstcw instructions. After setting the rounding mode as desired, the float2int() function will use that rounding mode until the control mode is reset. The ceil() and floor() functions set the rounding mode for every instruction, which is very slow. Therefore, it is a lot faster to set the rounding mode (down or up) before processing a block, and use float2int() instead.

References: SIMPLY FPU by Raymond Filiatreault

<http://www.website.masmforum.com/tutorials/fptute/index.html>

MSVC (and probably other compilers too) has functions defined in <float.h> for changing the FPU control word: _control87/_controlfp. The equivalent instructions are:

```
_controlfp(_RC_CHOP, _MCW_RC); // set rounding mode to truncate
_controlfp(_RC_UP, _MCW_RC); // set rounding mode to up (ceil)
_controlfp(_RC_DOWN, _MCW_RC); // set rounding mode to down (floor)
_controlfp(_RC_NEAR, _MCW_RC); // set rounding mode to near (default)
```

Note that the FPU rounding mode affects other calculations too, so the same code may give different results.

Alternatively, single precision floating point numbers can be truncated or rounded to integers by using SSE instructions cvtss2si, cvtss2si, cvtps2pi or cvtpps2pi, where SSE instructions are available (which means probably on all modern CPUs). These are not discussed here in detail, but I provided the function truncateSSE which always truncates a single precision floating point number to integer, regardless of the current rounding mode.

(Also I think the SSE rounding mode can differ from the rounding mode set in the FPU control word, but I haven't tested it so far.)

Code :

```
#ifndef __FPU_ROUNDING_H_
#define __FPU_ROUNDING_H_

static short control_word;
static short control_word2;

// round a float to int using the actual rounding mode
inline int float2int(float x) {
    int i;
    __asm {
        fld x
        fistp i
    }
    return i;
}

// set rounding mode to nearest
inline void set_round2near() {
    __asm {
        fstcw control_word // store fpu control word
        mov dx, word ptr [control_word]
        and dx, 0xf9ff // round towards nearest (default)
        mov control_word2, dx
        fldcw control_word2 // load modified control word
    }
}

// set rounding mode to round down
inline void set_round_down() {
    __asm {
        fstcw control_word // store fpu control word
        mov dx, word ptr [control_word]
        or dx, 0x0400 // round towards -inf
        and dx, 0xf7ff
        mov control_word2, dx
        fldcw control_word2 // load modified control word
    }
}

// set rounding mode to round up
inline void set_round_up() {
    __asm {
        fstcw control_word // store fpu control word
        mov dx, word ptr [control_word]
        or dx, 0x0800 // round towards +inf
        and dx, 0xfbff
        mov control_word2, dx
        fldcw control_word2 // load modified control word
    }
}
```



```

}
}

// set rounding mode to truncate
inline void set_truncate() {
    __asm {
        fstcw control_word // store fpu control word
        mov dx, word ptr [control_word]
        or dx, 0x0c00 // rounding: truncate
        mov control_word2, dx
        fldcw control_word2 // load modified control word
    }
}

// restore original rounding mode
inline void restore_cw() {
    __asm fldcw control_word
}

// truncate to integer using SSE
inline long truncateSSE(float x) {
    __asm cvttss2si eax, x
}

#endif

```

Comments

from : jrocnucl [[a t]] hotmail.com

comment : I've seen a lot of talk about using the function lrintf() when converting from float to int, regarding bypassing some 'slow' opcodes in what standard compilers put in for something like:

```
int myint = (int) myfloat;
```

in other words the following code would be faster:

```
int myint = lrintf(myfloat);
```

this lrintf function is available on GNU C/C++

Float-to-int, covering an array of floats (click this to go back to the index)

References : Posted by Stefan Stenzel

Notes :

intel only

Code :

```
void f2short(float *fptr,short *iptr,int n)
{
_asm {
    mov     ebx,n
    mov     esi,fptr
    mov     edi,iptr
    lea     ebx,[esi+ebx*4]    ; ptr after last
    mov     edx,0x80008000    ; damn endianness confuses...
    mov     ecx,0x4b004b00    ; damn endianness confuses...
    mov     eax,[ebx]         ; get last value
    push   eax
    mov     eax,0x4b014B01
    mov     [ebx],eax         ; mark end
    mov     ax,[esi+2]
    jmp     startf2slp

; Pad with nops to make loop start at address divisible
; by 16 + 2, e.g. 0x01408062, don't ask why, but this
; gives best performance. Unfortunately "align 16" does
; not seem to work with my VC.
; below I noted the measured execution times for different
; nop-paddings on my Pentium Pro, 100 conversions.
; saturation: off pos neg

    nop           ;355 546 563 <- seems to be best
;   nop           ;951 547 544
;   nop           ;444 646 643
;   nop           ;444 646 643
;   nop           ;944 951 950
;   nop           ;358 447 644
;   nop           ;358 447 643
;   nop           ;358 544 643
;   nop           ;543 447 643
;   nop           ;643 447 643
;   nop           ;1047 546 746
;   nop           ;545 954 1253
;   nop           ;545 547 661
;   nop           ;544 547 746
;   nop           ;444 947 1147
;   nop           ;444 548 545
in_range:
    mov     eax,[esi]
    xor     eax,edx
saturate:
    lea     esi,[esi+4]
    mov     [edi],ax
    mov     ax,[esi+2]
    add     edi,2
startf2slp:
    cmp     ax,cx
    je     in_range
    mov     eax,edx
    js     saturate         ; saturate neg -> 0x8000
    dec     eax             ; saturate pos -> 0x7FFF
    cmp     esi,ebx        ; end reached ?
    jnb    saturate
    pop     eax
    mov     [ebx],eax     ; restore end flag
}
}
```

Comments

from : magnus[at]smartelectronix.com

comment :

```
_asm {
mov ebx,n
mov esi,fptr
mov edi,iptr
lea ebx,[esi+ebx*4] ; ptr after last
mov edx,0x80008000 ; damn endianness confuses...
mov ecx,0x4b004b00 ; damn endianness confuses...
mov eax,[ebx] ; get last value
```

I think the last line here reads outside the buffer.

Gaussian dithering (click this to go back to the index)

Type : Dithering

References : Posted by Aleksey Vaneev (picoder[AT]mail[DOT]ru)

Notes :

It is a more sophisticated dithering than simple RND. It gives the most low noise floor for the whole spectrum even without noise-shaping. You can use as big N as you can afford (it will not hurt), but 4 or 5 is generally enough.

Code :

Basically, next value is calculated this way (for RND going from -0.5 to 0.5):

$$\text{dither} = \frac{\text{RND} + \text{RND} + \dots + \text{RND}}{N \text{ times}} / N.$$

If your RND goes from 0 to 1, then this code is applicable:

$$\text{dither} = (\text{RND} + \text{RND} + \dots + \text{RND} - 0.5 * N) / N.$$

Gaussian random numbers (click this to go back to the index)

Type : random number generation

References : Posted by tobybear[AT]web[DOT]de

Notes :

```
// Gaussian random numbers
// This algorithm (adapted from "Natur als fraktale Grafik" by
// Reinhard Scholl) implements a generation method for gaussian
// distributed random numbers with mean=0 and variance=1
// (standard gaussian distribution) mapped to the range of
// -1 to +1 with the maximum at 0.
// For only positive results you might abs() the return value.
// The q variable defines the precision, with q=15 the smallest
// distance between two numbers will be  $1/(2^q \text{ div } 3) = 1/10922$ 
// which usually gives good results.
```

```
// Note: the random() function used is the standard random
// function from Delphi/Pascal that produces *linear*
// distributed numbers from 0 to parameter-1, the equivalent
// C function is probably rand().
```

Code :

```
const q=15;
      c1=(1 shl q)-1;
      c2=(c1 div 3)+1;
      c3=1/c1;

function GRandom:single;
begin
  result:=(2*(random(c2)+random(c2)+random(c2))-3*(c2-1))*c3;
end;
```

Hermite Interpolator (x86 ASM) (click this to go back to the index)

Type : Hermite interpolator in x86 assembly (for MS VC++)

References : Posted by robert[DOT]bielik[AT]rbcaudio[DOT]com

Notes :

An "assembled" variant of Laurent de Soras hermite interpolator. I tried to do calculations as parallel as I could muster, but there is almost certainly room for improvements. Right now, it works about 5.3 times (!) faster, not bad to start with...

Parameter explanation:

frac_pos: fractional value [0.0f - 1.0f] to interpolator

pntr: pointer to float array where:

pntr[0] = previous sample (idx = -1)

pntr[1] = current sample (idx = 0)

pntr[2] = next sample (idx = +1)

pntr[3] = after next sample (idx = +2)

The interpolation takes place between pntr[1] and pntr[2].

Regards,

/Robert Bielik

RBC Audio

Code :

```
const float c_half = 0.5f;
```

```
__declspec(naked) float __hermite(float frac_pos, const float* pntr)
{
    __asm
    {
        push ecx;
        mov ecx, dword ptr[esp + 12];
        //////////////////////////////////////
        add ecx, 0x04; // ST(0) ST(1) ST(2) ST(3) ST(4) ST(5) ST(6) ST(7)
        fld dword ptr [ecx+4]; // x1
        fsub dword ptr [ecx-4]; // x1-xm1
        fld dword ptr [ecx]; // x0 x1-xm1
        fsub dword ptr [ecx+4]; // v x1-xm1
        fld dword ptr [ecx+8]; // x2 v x1-xm1
        fsub dword ptr [ecx]; // x2-x0 v x1-xm1
        fxch st(2); // x1-m1 v x2-x0
        fmul c_half; // c v x2-x0
        fxch st(2); // x2-x0 v c
        fmul c_half; // 0.5*(x2-x0) v c
        fxch st(2); // c v 0.5*(x2-x0)
        fst st(3); // c v 0.5*(x2-x0) c
        fadd st(0), st(1); // w v 0.5*(x2-x0) c
        fxch st(2); // 0.5*(x2-x0) v w c
        faddp st(1), st(0); // v+.5(x2-x0) w c
        fadd st(0), st(1); // a w c
        fadd st(1), st(0); // a b_neg c
        fmul dword ptr [esp+8]; // a*frac b_neg c
        fsubp st(1), st(0); // a*f-b c
        fmul dword ptr [esp+8]; // (a*f-b)*f c
        faddp st(1), st(0); // res-x0/f
        fmul dword ptr [esp+8]; // res-x0
        fadd dword ptr [ecx]; // res
        pop ecx;
        ret;
    }
}
```

Comments

from : dmj[at]smartelectronix

comment : This code produces a nasty buzzing sound. I think there might be a bug somewhere but I haven't found it yet.

from : hplus-musicdsp[at]mindcontrol.org

comment : I agree; the output, when plotted, looks like it has overlaid rectangular shapes on it.

from : robert.bielik [[a t]] rbcaudio.com

comment : AHH! True! A bug has sneaked in! Change the row that says:

```
fsubp st(1), st(0); // a*f-b c
```

to:

```
fsubp st(1), st(0); // a*f-b c
```

and it should be much better. Although I noticed that a good optimization by the compiler generates nearly as fast a code, but only nearly. This is still about 10% faster.

from : dwerner [[a t]] experimentalscene.com

comment : I have tested the four hermite interpolation algorithms posted to musicdsp.org plus the assembled version of

Laurent de Soras' code by Robert Bielik and found that on a Pentium 4 with full optimization (targeting the

Pentium 4 and above, but not using code that won't work on older processors) with MS VC++ 7 that the second function is the fastest.

Function	Percent Total	Time	Return Value
hermite1:	18.90%,	375ms,	0.52500004f
hermite2:	16.53%,	328ms,	0.52500004f
hermite3:	17.34%,	344ms,	0.52500004f
hermite4:	19.66%,	390ms,	0.52500004f
hermite5:	27.57%,	547ms,	0.52500004f

- Daniel Werner
<http://experimentalscene.com/>

Hermite interpolation (click this to go back to the index)

References : Posted by various

Notes :

These are all different ways to do the same thing : hermite interpolation. Try'm all and benchmark.

Code :

```
// original
inline float hermite1(float x, float y0, float y1, float y2, float y3)
{
    // 4-point, 3rd-order Hermite (x-form)
    float c0 = y1;
    float c1 = 0.5f * (y2 - y0);
    float c2 = y0 - 2.5f * y1 + 2.f * y2 - 0.5f * y3;
    float c3 = 1.5f * (y1 - y2) + 0.5f * (y3 - y0);

    return ((c3 * x + c2) * x + c1) * x + c0;
}

// james mccartney
inline float hermite2(float x, float y0, float y1, float y2, float y3)
{
    // 4-point, 3rd-order Hermite (x-form)
    float c0 = y1;
    float c1 = 0.5f * (y2 - y0);
    float c3 = 1.5f * (y1 - y2) + 0.5f * (y3 - y0);
    float c2 = y0 - y1 + c1 - c3;

    return ((c3 * x + c2) * x + c1) * x + c0;
}

// james mccartney
inline float hermite3(float x, float y0, float y1, float y2, float y3)
{
    // 4-point, 3rd-order Hermite (x-form)
    float c0 = y1;
    float c1 = 0.5f * (y2 - y0);
    float y0my1 = y0 - y1;
    float c3 = (y1 - y2) + 0.5f * (y3 - y0my1 - y2);
    float c2 = y0my1 + c1 - c3;

    return ((c3 * x + c2) * x + c1) * x + c0;
}

// laurent de soras
inline float hermite4(float frac_pos, float xm1, float x0, float x1, float x2)
{
    const float c = (x1 - xm1) * 0.5f;
    const float v = x0 - x1;
    const float w = c + v;
    const float a = w + v + (x2 - x0) * 0.5f;
    const float b_neg = w + a;

    return (((a * frac_pos) - b_neg) * frac_pos + c) * frac_pos + x0;
}
```

Comments

from : theguylle

comment : great sources but what is Hermite ?

if you don't describe what is your code made for, you will made a great sources but I don't know why?

cheers Paul

from : bram [[a t]] musicdsp.org

comment : hermite is interpolation.

have a look around the archive, you'll see that the word 'hermite' is in more than one item ;-)

-bram

from : ronaldowf [[a t]] sanepar.com.br

comment : Please, would like to know of hermite code it exists in delphi.

thankful

Ronaldo

Cascavel/Paraná/Brasil

from : m.magrini [[a t]] NOSPAMbad-sector.com

comment : Please,

add, at least, the meaning of each parameter (I mean x, y0, y1,y2, y3)....

m.

from : musicdsp [[a t]] [remove this]dsparsons.co.uk

comment : Ronaldo, it doesn't take much to translate these to Delphi - for float, either use single or double to your preference!

Looking at the codes, it seems quite clear that the parameters follow a pattern of: Sample Position between middle two samples, then the sample before current, current sample, current sample +1, current sample +2.

HTH
DSP

from : antiprosynthesis [[a t]] hotmail.com

comment : What are all these variables standing for? Not very clear :|

from : George

comment : parameters are allright.

$x_{m1} \rightarrow x[n-1]$

$x_0 \rightarrow x[n]$

$x_1 \rightarrow x[n+1]$

$x_2 \rightarrow x[n+2]$

fractional position stands for a fraction between 0 and 1 to interpolate

from : paranoias-poison-door [[a t]] gmail.youknowwhatandtakeoutthehyphens

comment : Couldn't #2 be sped up a hair by commenting out

float c0 = y1;

and then replacing c0 with y1 in the return line? Or do the compilers do that kind of thing automatically when they optimize?

from : asynth [[a t]] io.com

comment : "Couldn't #2 be sped up a hair"
It gets optimized out.

[Linear interpolation](#) (click this to go back to the index)

Type : Linear interpolators for oversampled audio

References : Posted by scoofy[AT]inf[DOT]elte[DOT]hu

Notes :
Simple, fast linear interpolators for upsampling a signal by a factor of 2,4,8,16 or 32. Not very usable on their own since they introduce aliasing (but still better than zero order hold). These are best used with already oversampled signals.

-- Peter Schoffhauzer

```
Code :
#ifdef __LIN_INTERPOLATOR_H_
#define __LIN_INTERPOLATOR_H_

/*****
 * Linear interpolator class
 *****/

class interpolator_linear
{
public:
    interpolator_linear() {
        reset_hist();
    }

    // reset history
    void reset_hist() {
        dl = 0.f;
    }

    // 2x interpolator
    // out: pointer to float[2]
    inline void process2x(float const in, float *out) {
        out[0] = dl + 0.5f*(in-dl); // interpolate
        out[1] = in;
        dl = in; // store delay
    }

    // 4x interpolator
    // out: pointer to float[4]
    inline void process4x(float const in, float *out) {
        float y = in-dl;
        out[0] = dl + 0.25f*y; // interpolate
        out[1] = dl + 0.5f*y;
        out[2] = dl + 0.75f*y;
        out[3] = in;
        dl = in; // store delay
    }

    // 8x interpolator
    // out: pointer to float[8]
    inline void process8x(float const in, float *out) {
        float y = in-dl;
        out[0] = dl + 0.125f*y; // interpolate
        out[1] = dl + 0.25f*y;
        out[2] = dl + 0.375f*y;
        out[3] = dl + 0.5f*y;
        out[4] = dl + 0.625f*y;
        out[5] = dl + 0.75f*y;
        out[6] = dl + 0.875f*y;
        out[7] = in;
        dl = in; // store delay
    }

    // 16x interpolator
    // out: pointer to float[16]
    inline void process16x(float const in, float *out) {
        float y = in-dl;
        out[0] = dl + (1.0f/16.0f)*y; // interpolate
        out[1] = dl + (2.0f/16.0f)*y;
        out[2] = dl + (3.0f/16.0f)*y;
        out[3] = dl + (4.0f/16.0f)*y;
        out[4] = dl + (5.0f/16.0f)*y;
        out[5] = dl + (6.0f/16.0f)*y;
        out[6] = dl + (7.0f/16.0f)*y;
        out[7] = dl + (8.0f/16.0f)*y;
        out[8] = dl + (9.0f/16.0f)*y;
        out[9] = dl + (10.0f/16.0f)*y;
        out[10] = dl + (11.0f/16.0f)*y;
        out[11] = dl + (12.0f/16.0f)*y;
        out[12] = dl + (13.0f/16.0f)*y;
        out[13] = dl + (14.0f/16.0f)*y;
        out[14] = dl + (15.0f/16.0f)*y;
        out[15] = in;
        dl = in; // store delay
    }

    // 32x interpolator
```

```

// out: pointer to float[32]
inline void process32x(float const in, float *out) {
    float y = in-d1;
    out[0] = d1 + (1.0f/32.0f)*y; // interpolate
    out[1] = d1 + (2.0f/32.0f)*y;
    out[2] = d1 + (3.0f/32.0f)*y;
    out[3] = d1 + (4.0f/32.0f)*y;
    out[4] = d1 + (5.0f/32.0f)*y;
    out[5] = d1 + (6.0f/32.0f)*y;
    out[6] = d1 + (7.0f/32.0f)*y;
    out[7] = d1 + (8.0f/32.0f)*y;
    out[8] = d1 + (9.0f/32.0f)*y;
    out[9] = d1 + (10.0f/32.0f)*y;
    out[10] = d1 + (11.0f/32.0f)*y;
    out[11] = d1 + (12.0f/32.0f)*y;
    out[12] = d1 + (13.0f/32.0f)*y;
    out[13] = d1 + (14.0f/32.0f)*y;
    out[14] = d1 + (15.0f/32.0f)*y;
    out[15] = d1 + (16.0f/32.0f)*y;
    out[16] = d1 + (17.0f/32.0f)*y;
    out[17] = d1 + (18.0f/32.0f)*y;
    out[18] = d1 + (19.0f/32.0f)*y;
    out[19] = d1 + (20.0f/32.0f)*y;
    out[20] = d1 + (21.0f/32.0f)*y;
    out[21] = d1 + (22.0f/32.0f)*y;
    out[22] = d1 + (23.0f/32.0f)*y;
    out[23] = d1 + (24.0f/32.0f)*y;
    out[24] = d1 + (25.0f/32.0f)*y;
    out[25] = d1 + (26.0f/32.0f)*y;
    out[26] = d1 + (27.0f/32.0f)*y;
    out[27] = d1 + (28.0f/32.0f)*y;
    out[28] = d1 + (29.0f/32.0f)*y;
    out[29] = d1 + (30.0f/32.0f)*y;
    out[30] = d1 + (31.0f/32.0f)*y;
    out[31] = in;
    d1 = in; // store delay
}

private:
    float d1; // previous input
};

#endif

```

[Lock free fifo](#) (click this to go back to the index)

References : Posted by Timo

Linked file : [LockFreeFifo.h](#) (this linked file is included below)

Notes :

Simple implementation of a lock free FIFO.

Comments

from : joshscholar_REMOVETHIS [[a t]] yahoo.com

comment : There is a good algorithm for a lock free (but multiprocessor safe) FIFO. But the given implementation is flawed in a number of ways. This code is not reliable. Two problems on the surface of it:

1. I can easily see that it's possible for two threads/processors to return the same item from the head if the timing is right.
2. there's no interlocked instructions to make sure that changes to the shared variables are globally visible
3. there's not attempt in the code to make sure that data is read in an atomic way, let alone changed in one...

The code is VERY naive

I do have code that works, but it's not so short that will post it in a comment. If anyone needs it they can email me

from : Timo

comment : This is only supposed to work on uniprocessor machines with `_one_` reading and `_one_` writing thread assuming that the assignments to read and write `idx` are simple `mov` instructions (i.e. atomic). To be sure you'd need to write the update parts in hand-coded asm; never know what the compiler comes up with. The context of this code was omitted (i.e. Bram posted my written in 1 min sketch in a discussion on IRC on a lock-free fifo, not production code).

Linked files

```
#include <vector>
#include <exception>
```

```
using std::vector;
using std::exception;
```

```
template<class T> class LockFreeFifo
{
public:
    LockFreeFifo (unsigned bufsz) : readidx(0), writeidx(0), buffer(bufsz)
    {}
```

```
    T get (void)
    {
        if (readidx == writeidx)
            throw runtime_error ("underrun");
```

```
        T result = buffer[readidx];
```

```
        if ((readidx + 1) >= buffer.size())
            readidx = 0;
        else
            readidx = readidx + 1;
```

```
        return result;
    }
```

```
    void put (T datum)
```

```
    {
        unsigned newidx;

        if ((writeidx + 1) >= buffer.size())
            newidx = 0;
        else
            newidx = writeidx + 1;
```

```
        if (newidx == readidx)
            throw runtime_error ("overrun");
```

```
        buffer[writeidx] = datum;
```

```
        writeidx = newidx;
    }
```

```
private:
```

```
    volatile unsigned readidx, writeidx;
    vector<T> buffer;
};
```

Matlab Time Domain Impulse Response Inverter/Divider (click this to go back to the index)

References : Posted by arcane[AT]arcanemethods[DOT]com

Notes :

Matlab code for time domain inversion of an impulse response or the division of two of them (transfer function.) The main teoplitz function is given both as a .m file and as a .c file for Matlab/w MEX compilation. The latter is much faster.

Code :

```
function inv=invimplms(den,n,d)
% syntax inv=invimplms(den,n,d)
%     den - denominator impulse
%     n   - length of result
%     d   - delay of result
%     inv - inverse impulse response of length n with delay d
%
% Levinson-Durbin algorithm from Proakis and Manolokis p.865
%
% Author: Bob Cain, May 1, 2001 arcane[AT]arcanemethods[DOT]com

m=xcorr(den,n-1);
m=m(n:end);
b=[den(d+1:-1:1);zeros(n-d-1,1)];
inv=toepsolve(m,b);
```

```
function quo=divimplms(num,den,n,d)
%Syntax quo=divimplms(num,den,n,d)
%     num - numerator impulse
%     den - denominator impulse
%     n   - length of result
%     d   - delay of result
%     quo - quotient impulse response of length n delayed by d
%
% Levinson-Durbin algorithm from Proakis and Manolokis p.865
%
% Author: Bob Cain, May 1, 2001 arcane@arcanemethods.com
```

```
m=xcorr(den,n-1);
m=m(n:end);
b=xcorr([zeros(d,1);num],den,n-1);
b=b(n:-1:1);
quo=toepsolve(m,b);
```

```
function hinv=toepsolve(r,q)
% Solve Toeplitz system of equations.
%     Solves R*hinv = q, where R is the symmetric Toeplitz matrix
%     whos first column is r
%     Assumes all inputs are real
%     Inputs:
%     r - first column of Toeplitz matrix, length n
%     q - rhs vector, length n
%     Outputs:
%     hinv - length n solution
%
% Algorithm from Roberts & Mullis, p.233
%
% Author: T. Krauss, Sept 10, 1997
%
% Modified: R. Cain, Dec 16, 2004 to remove a pair of transposes
%           that caused errors.
```

```
n=length(q);
a=zeros(n+1,2);
a(1,1)=1;
```

```
hinv=zeros(n,1);
hinv(1)=q(1)/r(1);
```

```
alpha=r(1);
c=1;
d=2;
```

```
for k=1:n-1,
    a(k+1,c)=0;
    a(1,d)=1;
    beta=0;
    j=1:k;
    beta=sum(r(k+2-j).*a(j,c))/alpha;
    a(j+1,d)=a(j+1,c)-beta*a(k+1-j,c);
    alpha=alpha*(1-beta^2);
    hinv(k+1,1)=(q(k+1)-sum(r(k+2-j).*hinv(j,1)))/alpha;
    hinv(j)=hinv(j)+a(k+2-j,d)*hinv(k+1);
    temp=c;
    c=d;
    d=temp;
end
```

-----What follows is the .c version of toepsolve-----

```
#include <math.h>
#include "mex.h"

/* function hinv = toepsolve(r,q);
 * TOEPSOLVE Solve Toeplitz system of equations.
 * Solves R*hinv = q, where R is the symmetric Toeplitz matrix
 * whos first column is r
 * Assumes all inputs are real
 * Inputs:
 *   r - first column of Toeplitz matrix, length n
 *   q - rhs vector, length n
 * Outputs:
 *   hinv - length n solution
 *
 * Algorithm from Roberts & Mullis, p.233
 *
 * Author: T. Krauss, Sept 10, 1997
 *
 * Modified: R. Cain, Dec 16, 2004 to replace unnecessary
 *           n by n matrix allocation for a with an n by 2 rotating
 *           buffer and to more closely match the .m function.
 */
void mexFunction(
    int nlhs,
    mxArray *plhs[],
    int nrhs,
    const mxArray *prhs[]
)
{
    double (*a)[2],*hinv,alpha,beta;
    int c,d,temp,j,k;

    double eps = mxGetEps();
    int n = (mxGetN(prhs[0])>mxGetM(prhs[0])) ? mxGetN(prhs[0]) : mxGetM(prhs[0]) ;
    double *r = mxGetPr(prhs[0]);
    double *q = mxGetPr(prhs[1]);

    a = (double (*)[2])mxMalloc((n+1)*2,sizeof(double));
    if (a == NULL) {
        mexErrMsgTxt("Sorry, failed to allocate buffer.");
    }
    a[0][0]=1.0;

    plhs[0] = mxCreateDoubleMatrix(n,1,0);
    hinv = mxGetPr(plhs[0]);
    hinv[0] = q[0]/r[0];

    alpha=r[0];
    c=0;
    d=1;

    for (k = 1; k < n; k++) {
        a[k][c] = 0;
        a[0][d] = 1.0;
        beta = 0.0;
        for (j = 1; j <= k; j++) {
            beta += r[k+1-j]*a[j-1][c];
        }
        beta /= alpha;
        for (j = 1; j <= k; j++) {
            a[j][d] = a[j][c] - beta*a[k-j][c];
        }
        alpha *= (1 - beta*beta);
        hinv[k] = q[k];
        for (j = 1; j <= k; j++) {
            hinv[k] -= r[k+1-j]*hinv[j-1];
        }
        hinv[k] /= alpha;
        for (j = 1; j <= k; j++) {
            hinv[j-1] += a[k+1-j][d]*hinv[k];
        }
        temp=c;
        c=d;
        d=temp;
    } /* loop over k */

    mxFree(a);

    return;
}
```

[MATLAB-Tools for SNDAN](#) (click this to go back to the index)

References : Posted by Markus Sapp

Linked file : [other001.zip](#)

Notes :
(see linkfile)

MIDI note/frequency conversion (click this to go back to the index)

Type :-

References : Posted by tobybear[AT]web[DOT]de

Notes :

I get often asked about simple things like MIDI note/frequency conversion, so I thought I could as well post some source code about this. The following is Pascal/Delphi syntax, but it shouldn't be a problem to convert it to almost any language in no time.

Uses for this code are mainly for initializing oscillators to the right frequency based upon a given MIDI note, but you might also check what MIDI note is closest to a given frequency for pitch detection etc.

In realtime applications it might be a good idea to get rid of the power and log2 calculations and generate a lookup table on initialization.

A full Pascal/Delphi unit with these functions (including lookup table generation) and a simple demo application can be downloaded here: http://tobybear.phreque.com/dsp_conv.zip

If you have any comments/suggestions, please send them to: tobybear@web.de

Code :

```
// MIDI NOTE/FREQUENCY CONVERSIONS

const notes:array[0..11] of string= ('C ','C#','D ','D#','E ','F ','F#','G ','G#','A ','A#','B ');
const base_a4=440; // set A4=440Hz

// converts from MIDI note number to frequency
// example: NoteToFrequency(12)=32.703
function NoteToFrequency(n:integer):double;
begin
  if (n>=0)and(n<=119) then
    result:=base_a4*power(2,(n-57)/12)
  else result:=-1;
end;

// converts from MIDI note number to string
// example: NoteToName(12)='C 1'
function NoteToName(n:integer):string;
begin
  if (n>=0)and(n<=119) then
    result:=notes[n mod 12]+inttostr(n div 12)
  else result:='---';
end;

// converts from frequency to closest MIDI note
// example: FrequencyToNote(443)=57 (A 4)
function FrequencyToNote(f:double):integer;
begin
  result:=round(12*log2(f/base_a4))+57;
end;

// converts from string to MIDI note
// example: NameToNote('A4')=57
function NameToNote(s:string):integer;
var c,i:integer;
begin
  if length(s)=2 then s:=s[1]+' '+s[2];
  if length(s)<>3 then begin result:=-1;exit end;
  s:=uppercase(s);
  c:=-1;
  for i:=0 to 11 do
    if notes[i]=copy(s,1,2) then
      begin
        c:=i;
        break
      end;
  try
    i:=strtoint(s[3]);
    result:=i*12+c;
  except
    result:=-1;
  end;
  if c<0 then result:=-1;
end;
```

Comments

from : tobybear [[a t]] web.de

comment : For the sake of completeness, here is octave fraction notation and pitch class notation:

```
// converts from MIDI note to octave fraction notation
// the integer part of the result is the octave number, where
// 8 is the octave starting with middle C. The fractional part
// is the note within the octave, where 1/12 represents a semitone.
// example: NoteToOct(57)=7.75
function NoteToOct(i:integer):double;
begin
  result:=3+(i div 12)+(i mod 12)/12;
```

end;

```
// converts from MIDI note to pitch class notation
// the integer part of the number is the octave number, where
// 8 is the octave starting with middle C. The
fractional part
// is the note within the octave, where a 0.01 increment is a
// semitone.
// example: NoteToPch(57)=7.09
function NoteToPch(i:integer):double;
begin
  result:=3+(i div 12)+(i mod 12)*0.01;
end;
```

from : kaleja [[a t]] estarcion.com

comment : I thought most sources gave A-440Hz = MIDI note 69. MIDI 60 = middle C = ~262Hz, A-440 = "A above middle C". Not so?

from : DFL

comment : Kaleja is correct. Here is some C code:

```
double MIDtoFreq( char keynum ) {
  return 440.0 * pow( 2.0, ((double)keynum - 69.0) / 12.0 );
}
```

you can double-check the table here:

http://tomscarff.tripod.com/midi_analyser/midi_note_frequency.htm

Millimeter to DB (faders...) (click this to go back to the index)

References : Posted by James McCartney

Notes :

These two functions reproduce a traditional professional mixer fader taper.

MMtoDB converts millimeters of fader travel from the bottom of the fader for a 100 millimeter fader into decibels. DBtoMM is the inverse.

The taper is as follows from the top:

The top of the fader is +10 dB

100 mm to 52 mm : -5 dB per 12 mm

52 mm to 16 mm : -10 dB per 12 mm

16 mm to 4 mm : -20 dB per 12 mm

4 mm to 0 mm : fade to zero. (in these functions I go to -200dB which is effectively zero for up to 32 bit audio.)

Code :

```
float MMtoDB(float mm)
{
    float db;

    mm = 100. - mm;

    if (mm <= 0.) {
        db = 10.;
    } else if (mm < 48.) {
        db = 10. - 5./12. * mm;
    } else if (mm < 84.) {
        db = -10. - 10./12. * (mm - 48.);
    } else if (mm < 96.) {
        db = -40. - 20./12. * (mm - 84.);
    } else if (mm < 100.) {
        db = -60. - 35. * (mm - 96.);
    } else db = -200.;
    return db;
}

float DBtoMM(float db)
{
    float mm;
    if (db >= 10.) {
        mm = 0.;
    } else if (db > -10.) {
        mm = -12./5. * (db - 10.);
    } else if (db > -40.) {
        mm = 48. - 12./10. * (db + 10.);
    } else if (db > -60.) {
        mm = 84. - 12./20. * (db + 40.);
    } else if (db > -200.) {
        mm = 96. - 1./35. * (db + 60.);
    } else mm = 100.;

    mm = 100. - mm;

    return mm;
}
```

Comments

from : Christian [[a t]] savioursofsoul.de

comment : Pascal Translation...

```
function MMtoDB(Milimeter:Single):Single;
var mm: Single;
begin
    mm:=100-Milimeter;
    if mm = 0 then Result:=10
    else if mm < 48 then Result:=10-5/12*mm;
    else if mm < 84 then Result:=-10-10/12*(mm-48);
    else if mm < 96 then Result:=-40-20/12*(mm-84);
    else if mm < 100 then Result:=-60-35*(mm-96);
    else Result:=-200.;
end;
```

```
function DBtoMM(db:Single):Single;
begin
    if db>=10 then result:=0;
    else if db>-10 then result:=-12/5*(db-10);
    else if db>-40 then result:=48-12/10*(db+10);
    else if db>-60 then result:=84-12/20*(db+40);
    else if db>-200 then result:=96-1/35*(db+60);
    else result:=100.;
    Result:=100-Result;
end;
```


[Motorola 56300 Disassembler](#) (click this to go back to the index)

Type : disassembler

References : Posted by [chris_townsend\[AT\]digidesign\[DOT\]com](#)

Linked file : [Disassemble56k.zip](#)

Notes :

This code contains functions to disassemble Motorola 56k opcodes. The code was originally created by Stephen Davis at Stanford. I made minor modifications to support many 56300 opcodes, although it would nice to add them all at some point. Specifically, I added support for CLB, NORMF, immediate AND, immediate OR, multi-bit ASR/ASL, multi-bit LSL/LSR, MAX, MAXM, BRA, Bcc, BSR, BScC, DMAC, MACsu, MACuu, and conditional ALU instructions.

Comments

from : jawoll

comment : nice! let's disassemble virus c, nord lead 3, ...

from : elijahr [[a t]] gmail.com

comment : Very nice. How would one get ahold of the OS for one of these synths, to disassemble it? I've got a Nord Micro, with a single 56303... question is.... how to get the OS from the flash?

Noise Shaping Class (click this to go back to the index)

Type : Dithering with 9th order noise shaping

References : Posted by csheif[AT]indiana.edu

Linked file : [NS9dither16.h](#) (this linked file is included below)

Notes :

This is an implemetation of a 9th order noise shaping & dithering class, that runs quite fast (it has one function that uses Intel x86 assembly, but you can replace it with a different rounding function if you are running on a non-Intel platform). `_aligned_malloc` and `_aligned_free` require the MSVC++ Processor Pack, available from www.microsoft.com. You can replace them with "new" and "delete," but allocating aligned memory seems to make it run faster. Also, you can replace ZeroMemory with a memset that sets the memory to 0 if you aren't using Win32. Input should be floats from -32768 to 32767 (processS will clip at these points for you, but clipping is bad when you are trying to convert floats to shorts). Note to reviewer - it would probably be better if you put the code in a file such as NSDither.h and have a link to it - it's rather long.

(see linked file)

Comments

from : mail[ns] [[a t]] mutagene.net
comment : I haven't tried this class out, but it looks like there's a typo in the unrolled loop -- shouldn't it read "c[2]*EH[HistPos+2]"? This might this also account for the 3 clock cycle improvement on a P-III.

```
// This arrangement seems to execute 3 clock cycles faster on a P-III
samp -= c[8]*EH[HistPos+8] + c[7]*EH[HistPos+7] + c[6]*EH[HistPos+6] +
c[5]*EH[HistPos+5] + c[4]*EH[HistPos+4] + c[3]*EH[HistPos+3] + c[2]*EH[HistPos+1] +
c[1]*EH[HistPos+1] + c[0]*EH[HistPos];
```

from : chaikov [AT] sakrament [DOT] com

comment : Great class! But I found one more mistake: function `my_mod(9, 9)` gives 9 instead of 0 (`9 % 9 = 0`).

```
HistPos = my_mod((HistPos+8), order);
EH[HistPos+9] = EH[HistPos] = output - samp;
```

-> `HistPos + 9 = 18` (max EH index is 17) which leads to out of array boundary and crash.

So `my_mod` should look like:

```
__inline my_mod(int x, int n)
{
    if(x >= n) x-=n;
    return x;
}
```

Linked files

```
#pragma once
```

```
#include <malloc.h>
#include <rounding.h>
```

```
// F-weighted
//static const float or3Fc[3] = {1.623f, -0.982f, 0.109f};
static const float or9Fc[9] = {2.412f, -3.370f, 3.937f, -4.174f, 3.353f, -2.205f, 1.281f, -0.569f,
0.0847f};
```

```
// modified-E weighted
//static const float or2MEc[2] = {1.537f, -0.8367f};
//static const float or3MEc[3] = {1.652f, -1.049f, 0.1382f};
//static const float or9MEc[9] = {1.662f, -1.263f, 0.4827f, -0.2913f, 0.1268f, -0.1124f, 0.03252f,
-0.01265f, -0.03524f};
```

```
// improved-E weighted
//static const float or5IEc[5] = {2.033f, -2.165f, 1.959f, -1.590f, 0.6149f};
//static const float or9IEc[9] = {2.847f, -4.685f, 6.214f, -7.184f, 6.639f, -5.032f, 3.263f, -
1.632f, 0.4191f};
```

```
// Simple 2nd order
//static const float or2Sc[2] = {1.0f, -0.5f};
```

```
// Much faster than C's % operator (anyway, x will never be > 2*n in this case so this is very
simple and fast)
// Tell the compiler to try to inline as much as possible, since it makes it run so much faster
since these functions get called at least once per sample
__inline my_mod(int x, int n)
{
    if(x > n) x-=n;
    return x;
}
```

```

}

__inline int round(float f)
{
    int r;
    __asm {
        fld f
        fistp r
    }
    return r;
}

__inline short ltos(long l)
{
    return (short)((l==(short)l) ? l : (l>>31)^0x7FFF);
}

__inline float frand()
{
    // Linear Congruential Method - got it from the book "Algorithms," by Robert Sedgewick
    static unsigned long a = 0xDEADBEEF;

    a = a * 140359821 + 1;
    return a * (1.0f / 0xFFFFFFFF);
}

class NS9dither16
{
public:
    NS9dither16();
    ~NS9dither16();
    short processS(float samp);
    int processI(float samp);
    void reset();

private:
    int order;
    int HistPos;

    float* c; // Coeffs
    float* EH; // Error History
};

__inline NS9dither16::NS9dither16()
{
    order = 9;

    //c=new float[order]; // if you don't have _aligned_malloc
    c = (float*)_aligned_malloc(order*sizeof(float), 16);
    CopyMemory(c, or9Fc, order*sizeof(float));

    //EH = new float[2*order]; // if you don't have _aligned_malloc
    EH = (float*)_aligned_malloc(2*order*sizeof(float), 16);
    ZeroMemory(EH, 2*order*sizeof(float));

    // Start at top of error history - you can make it start anywhere from 0-8 if you really want to
    HistPos=8;
}

__inline NS9dither16::~NS9dither16()
{
    //if(c) delete [] c; // if you don't have _aligned_free
    //if(EH) delete [] EH; // if you don't have _aligned_free
    if(c) _aligned_free(c); // don't really need "if," since it is OK to free null pointer, but still...
    if(EH) _aligned_free(EH);
}

__inline void NS9dither16::reset()
{
    ZeroMemory(EH, 2*order*sizeof(float));
    // Start at top of error history - you can make it start anywhere from 0-8 if you really want to
    HistPos=8;
}

// Force inline because VC++.NET doesn't inline for some reason (VC++ 6 does)
__forceinline short NS9dither16::processS(float samp)
{
    int output;

    /*for(int x=0; x<order; x++)
    {
        //samp -= c[x] * EH[(HistPos+x) % order];
        samp -= c[x] * EH[HistPos+x];
    }
}

```

```

}*/
// Unrolled loop for faster execution
/*samp -= c[0]*EH[HistPos] + c[1]*EH[HistPos+1] + c[2]*EH[HistPos+2] +
  c[3]*EH[HistPos+3] + c[4]*EH[HistPos+4] + c[5]*EH[HistPos+5] +
  c[6]*EH[HistPos+6] + c[7]*EH[HistPos+7] + c[8]*EH[HistPos+8];*/
// This arrangement seems to execute 3 clock cycles faster on a P-III
samp -= c[8]*EH[HistPos+8] + c[7]*EH[HistPos+7] + c[6]*EH[HistPos+6] +
  c[5]*EH[HistPos+5] + c[4]*EH[HistPos+4] + c[3]*EH[HistPos+3] +
  c[2]*EH[HistPos+1] + c[1]*EH[HistPos+1] + c[0]*EH[HistPos];

output = round(samp + (frand() + frand() - 1));

//HistPos =(HistPos+8) % order; // The % operator is really slow
HistPos = my_mod((HistPos+8), order);
// Update buffer (both copies)
EH[HistPos+9] = EH[HistPos] = output - samp;

return ltos(output);
}

__forceinline int NS9dither16::processI(float samp)
{
int output;

/*for(int x=0; x<order; x++)
{
//samp -= c[x] * EH[(HistPos+x) % order];
samp -= c[x] * EH[HistPos+x];
}*/
// Unrolled loop for faster execution
/*samp -= c[0]*EH[HistPos] + c[1]*EH[HistPos+1] + c[2]*EH[HistPos+2] +
  c[3]*EH[HistPos+3] + c[4]*EH[HistPos+4] + c[5]*EH[HistPos+5] +
  c[6]*EH[HistPos+6] + c[7]*EH[HistPos+7] + c[8]*EH[HistPos+8];*/
// This arrangement seems to execute 3 clock cycles faster on a P-III
samp -= c[8]*EH[HistPos+8] + c[7]*EH[HistPos+7] + c[6]*EH[HistPos+6] +
  c[5]*EH[HistPos+5] + c[4]*EH[HistPos+4] + c[3]*EH[HistPos+3] +
  c[2]*EH[HistPos+1] + c[1]*EH[HistPos+1] + c[0]*EH[HistPos];

output = round(samp + (frand() + frand() - 1));

//HistPos =(HistPos+8) % order; // The % operator is really slow
HistPos = my_mod((HistPos+8), order);
// Update buffer (both copies)
EH[HistPos+9] = EH[HistPos] = output - samp;

return output;
}

```

Nonblocking multiprocessor/multithread algorithms in C++ (click this to go back to the index)

Type : queue, stack, garbage collection, memory allocation, templates for atomic algorithms and types

References : Posted by joshscholar[AT]yahoo[DOT]com

Linked file : [ATOMIC.H](#)

Notes :

see linked file...

Comments

from : eigamx [[a t]] xhanmailx.net

comment : This code has a problem with operation exceeding 4G times. If you do more then 4G of Put and Get with the MPQueue, "AtomicUInt & Index(int i) { return data[i & (len-1)]; } will cause BUG.

Modular operation with length of 2^n is OK, but not for other numbers.

My email does not have any "x" letters.

[Piecewise quadratic approximate exponential function](#) (click this to go back to the index)

Type : Approximation of base-2 exponential function

References : Posted by Johannes M.-R.

Notes :

The code assumes round-to-zero mode, and ieee 754 float.

To achieve other bases, multiply the input by the logarithmus dualis of the base.

Code :

```
inline float fpow2(const float y)
{
    union
    {
        float f;
        int i;
    } c;

    int integer = (int)y;
    if(y < 0)
        integer = integer-1;

    float frac = y - (float)integer;

    c.i = (integer+127) << 23;
    c.f *= 0.33977f*frac*frac + (1.0f-0.33977f)*frac + 1.0f;

    return c.f;
}
```


please add it as a comment to the Denormalization preventer entry (no comments are allowed now) thanks (click this to go back to the index)

Type : quantization

References : Posted by scoofy[AT]inf[DOT]elte[DOT]hu

Notes :

You can zero out denormals by adding and subtracting a small number.

```
void kill_denormal_by_quantization(float &val)
{
    static const float anti_denormal = 1e-18;
    val += anti_denormal;
    val -= anti_denormal;
}
```

Reference: Laurent de Soras' great article on denormal numbers:

<http://ldesoras.free.fr/doc/articles/denormal-en.pdf>

I tend to add DC because it is faster than quantization. A slight DC offset (0.000000000000000001) won't hurt. That's -360 decibels...

Comments

from : scoofy [[a t]] inf.elte.hu

comment : Sorry I got an error message when I tried to post a comment that's why I submitted this. Why did this got into the archive? Someone please remove this entry. Thanks.

from : scoofy [[a t]] inf.elte.hu

comment : Are comments allowed now?

from : Arif [[a t]] mail.---

comment : Who is this idiot? Atleast integer had some interesting social critique.

[pow\(x,4\) approximation](#) (click this to go back to the index)

[References](#) : Posted by Stefan Stenzel

[Notes](#) :

Very hacked, but it gives a rough estimate of x^{**4} by modifying exponent and mantissa.

[Code](#) :

```
float p4fast(float in)
{
    long *lp,l;

    lp=(long *)(&in);
    l=*lp;

    l-=0x3F800001; /* un-bias */
    l<<=2;        /* **4 */
    l+=0x3F800001; /* bias */
    *lp=l;

    /* compiler will read this from memory since & operator had been used */
    return in;
}
```

[rational tanh approximation](#) (click this to go back to the index)

Type : saturation

References : Posted by cschueler

Notes :
This is a rational function to approximate a tanh-like soft clipper. It is based on the pade-approximation of the tanh function with tweaked coefficients.

The function is in the range $x=-3..3$ and outputs the range $y=-1..1$. Beyond this range the output must be clamped to $-1..1$.

The first to derivatives of the function vanish at -3 and 3 , so the transition to the hard clipped region is C2-continuous.

```
Code :
float rational_tanh(x)
{
    if( x < -3 )
        return -1;
    else if( x > 3 )
        return 1;
    else
        return x * ( 27 + x * x ) / ( 27 + 9 * x * x );
}
```

Comments

from : mdsp

comment : nice one

BTW if you google about "pade-approximation" you'll find a nice page with many solutions for common functions.

there's exp, log, sin, cos, tan, gaussian...

from : scoofy [[a t]] inf.elte.hu

comment : Works fine. If you want only a little overdrive, you don't even need the clipping, just the last line for faster processing.

```
float rational_tanh_noclip(x)
{
    return x * ( 27 + x * x ) / ( 27 + 9 * x * x );
}
```

The maximum error of this function in the $-4.5 .. 4.5$ range is about 2.6%.

from : scoofy [[a t]] inf.elte.hu

comment : By the way this is the fastest tanh() approximation in the archive so far.

from : cschueler

comment : Yep, I thought so.

That's why I thought it would be worth sharing.

Especially fast when using SSE you can do a 4-way parallel implementation, with MIN/MAX and the RCP instruction.

from : scoofy [[a t]] inf.elte.hu

comment : Yep, but the RCP increases the noise floor somewhat, giving a quantized sound, so I'd refrain from using it for high quality audio.

[Reading the compressed WA! parts in gigasampler files \(click this to go back to the index\)](#)

References : Paul Kellett

Linked file : [gigxpannd.zip](#)

Notes :

(see linkfile)

Code to read the .WA! (compressed .WAV) parts of GigaSampler .GIG files.

For related info on reading .GIG files see <http://www.linuxdj.com/evo>

Real basic DSP with Matlab (+ GUI) ... (click this to go back to the index)

Type : Like effects racks, made with Matlab !

References : Posted by guillaume[DOT]carniato[AT]meletu[DOT]univ-valenciennes[DOT]fr

Linked file : <http://www.xenggeng.fr.st/ici/guitou/Matlab Music.zip>

Notes :

You need Matlab v6.0 or more to run this stuff...

Code :

take a look at <http://www.xenggeng.fr.st/ici/guitou/Matlab Music.zip>

I'm now working on a Matlab - sequencer, which will certainly use 'Matlab Music'. I'm interested in integrating WaveWarp in this project; it's a toolbox that allow you to make real time DSP with Matlab.

If you're ok to improve this version (add effects, improve effects quality, anything else...) let's go ! Email me if you're interested in developing this beginner work...

Comments

from : harrison [[a t]] media.mit.edu

comment : The link is dead. Is the file posted somewhere else?

from : nobody [[a t]] nobody.com

comment : Quel idee fantastique - un description plus simple de DSP, mais malheureusement le link ne marche pas. Est-ce qu'il y a un link neuf?

[real value vs display value](#) (click this to go back to the index)

Type : Macro

References : Posted by [emil\[AT\]arpanet\[DOT\]no](mailto:emil[at]arpanet[DOT]no)

Notes :

REALVAL converts the vst param at given ranges to a display value.
VSTVAL does the opposite.

a = start

b = end

Code :

```
#define REALVAL(a, b, vstval) (a + (vstval)*(b-a))  
#define VSTVAL(a, b, realval) ((realval-a)/(b-a))
```

Comments

from : [bekkah \[\[a t \] \] web.de](mailto:bekkah[at]web.de)

comment : Why da hell do you use Makros???

BTW: I'll post my mapper class in a few days here, which does all this in a much more convenient way.

from : [kaleja \[\[a t \] \] estarcion.com](mailto:kaleja[at]estarcion.com)

comment : See <http://www.u-he.com/vstsource/archive.php?classid=2#16> for my solutions to this problem.

[Really fast x86 floating point sin/cos](#) (click this to go back to the index)

References : Posted by [rerdavies\[AT\]msn\[DOT\]com](mailto:rerdavies@msn.com)

Linked file : [sincos.zip](#)

Notes :

Frightful code from the Intel Performance optimization front. Not for the squeamish.

The following code calculates sin and cos of a floating point value on x86 platforms to 20 bits precision with 2 multiplies and two adds. The basic principle is to use $\sin(x+y)$ and $\cos(x+y)$ identities to generate the result from lookup tables. Each lookup table takes care of 10 bits of precision in the input. The same principle can be used to generate sin/cos to full (! Really. Full!) 24-bit float precision using two 8-bit tables, and one 10 bit table (to provide guard bits), for a net speed gain of about 4x over `fsin/fcos`, and 8x if you want both sin and cos. Note that microsoft compilers have trouble keeping doubles aligned properly on the stack (they must be 8-byte aligned in order not to incur a massive alignment penalty). As a result, this class should NOT be allocated on the stack. Add it as a member variable to any class that uses it.

e.g.

```
class CSomeClass {
    CQuickTrig m_QuickTrig;
    ...
    mQuickTrig.QuickSinCos(dAngle,fSin,fCos);
    ...
}
```

Code :

(see attached file)

Reasonably accurate/fastish tanh approximation (click this to go back to the index)

References : Posted by Fuzzpilz

Notes :

Fairly obvious, but maybe not obvious enough, since I've seen calls to `tanh()` in code snippets here.

It's this, basically:

$$\begin{aligned}\tanh(x) &= \sinh(x)/\cosh(x) \\ &= (\exp(x) - \exp(-x))/(\exp(x) + \exp(-x)) \\ &= (\exp(2x) - 1)/(\exp(2x) + 1)\end{aligned}$$

Combine this with a somewhat less accurate approximation for `exp` than usual (I use a third-order Taylor approximation below), and you're set. Useful for waveshaping.

Notes on the `exp` approximation:

It only works properly for input values above `x`, but since `tanh` is odd, that isn't a problem.

$$\exp(x) = 1 + x + x^2/(2!) + x^3/(3!) + \dots$$

Breaking the Taylor series off after the third term, I get

$$1 + x + x^2/2 + x^3/6.$$

I can save some multiplications by using

$$6 + x * (6 + x * (3 + x))$$

instead; a division by 6 becomes necessary, but is lumped into the additions in the `tanh` part:

$$(a/6 - 1)/(a/6 + 1) = (a - 6)/(a + 6).$$

Accuracy:

I haven't looked at this in very great detail, but it's always \leq the real `tanh` (\geq for $x < 0$), and the greatest deviation occurs at about ± 1.46 , where the result is ca. .95 times the correct value.

This is still faster than `tanh` if you use a better approximation for the exponential, even if you simply call `exp`.

There are probably additional ways of improving parts of this, and naturally if you're going to use it you'll want to figure out whether your particular application offers additional ways of simplifying it, but it's a good start.

Code :

```
/* single precision absolute value, a lot faster than fabsf() (if you use MSVC++ 6 Standard - others'
implementations might be less slow) */
float sabs(float a)
{
  int b=*((int *)&a)&0x7FFFFFFF;
  return *((float *)&b);
}
```

```
/* approximates tanh(x/2) rather than tanh(x) - depending on how you're using this, fixing that could well be
wasting a multiplication (though that isn't much, and it could be done with an integer addition in sabs
instead) */
float tanh2(float x)
{
  float a=sabs(x);
  a=6+a*(6+a*(3+a));
  return ((x<0)?-1:1)*(a-6)/(a+6); /* instead of using <, you can also check directly whether the sign bit is
set ((*((int *)&x))&0x80000000), but it's not really worth it */
}
```

Comments

from : fuzzpilz [[a t]] gmx.net

comment : Not sure why this didn't occur to me earlier, but you can easily save another two adds as follows:

```
a*=(6+a*(3+a));
return ((x<0)?-1:1)*a/(a+12);
```

from : fuzzpilz [[a t]] gmx.net

comment : AFAIK intrinsics aren't supported by VC6 Standard, but limited to Professional and Enterprise. Might be wrong, though, in which case I am a silly person. (no time to check now)

from : Laurent de Soras

comment : You can optimise it a bit more by using the fact that $\tanh(x) = 1 - 2 / (\exp(2x) + 1)$

from : kaleja [[a t]] estarcion.com

comment : You shouldn't need the `sabs()` on VC6 - you just need to add:

```
#pragma intrinsic( fabs )
```

before calling `fabsf()`, and it should go optimally fast.

from : Christian [[a t]] savioursofsoul.de

comment : Delphi Code:

```
// approximates tanh(x/2) rather than tanh(x) - depending on how you're using
// this, fixing that could well be wasting a multiplication
function tanh2(x:single):Single;
var a : single;
begin
  a:=f_abs(x);
  a:=a*(12+a*(6+a*(3+a)));
  if (x<0)
  then result:=-a/(a+24)
  else result:= a/(a+24);
end;
```

from : Christian [[a t]] savioursofsoul.de

comment : Laurent de Soras wrote:

"You can optimise it a bit more by using the fact that $\tanh(x) = 1 - 2 / (\exp(2*x) + 1)$ "

It's not faster, because you'll need 3 more cycles. The routine would then look like this:

```
function tanh2(x:single):Single;
var a : single;
begin
  a:=f_abs(x);
  a:=24+a*(12+a*(6+a*(3+a)));
  if (x<0)
  then result:= (-1+24/a)
  else result:= (1-24/a);
end;
```

from : didid [[a t]] skynet.be

comment : I must have missed this one..

but why is the comparison needed?

a simpler version would be:

```
a:=Abs(x)
Result:=x*(6+a*(3+a))/(a+12)
```

no?

So in asm:

```
function Tanh2(x:Single):Single;
const c3 :Single=3;
      c6 :Single=6;
      c12:Single=12;
Asm
  FLD  x
  FLD  ST(0)
  FABS
  FLD  c3
  FADD ST(0),ST(1)
  FMUL ST(0),ST(1)
  FADD c6
  FMULP ST(2),ST(0)
  FADD c12
  FDIVP ST(1),ST(0)
End;
```

..but almost all the CPU is wasted by the division anyway

from : didid [[a t]] skynet.be

comment : wait.. has anyone tested this function?

Here's a test plot:

<http://www.flstudio.com/gol/tanh.gif>

Red=TanH

Green=the approximation suggested in this thread

Blue=another approximation that does:

```
function TanH3(x:Single):Single;
Begin
  Result:=x - x*x*x/3 + 2*x*x*x*x*x/15;
end;
```

If I didn't do anything wrong, the green one is VERY far from TanH. Blue is both closer & computationally more efficient.

But ok, this plot is for a normalized 0..1. When you go above, the blue like goes crazy.

But now considering that -1..1 is what matters the most for what we do, the input could still be clipped.

from : didid [[a t]] skynet.be
comment : forget all this :)

it's all embarrassing bullshit and I obviously need some sleep :)

from : didid [[a t]] skynet.be
comment : Ok ignore my above crap that I can't delete, I swear that this one does work :)

First I hadn't seen that this function was assuming x^2 , so my graph was scaled by 2..

Second, the other algo (blue line) is still not to be neglected (because no FDIV) when the input is in the -1..1 range, and it does work.

Third, I'm suggesting here a version without the comparison/branching, but still, the CPU difference is neglectable because of the FDIV.

Here it is (this one does NOT assume a pre-multiplied x)..

plain code:

```
function Tanh2(x:Single):Single;
var a,b:Single;
begin
x:=x*2;
a:=abs(x);
b:=(6+a*(3+a));
Result:=(x*b)/(a*b+12);
end;
```

asm:

```
function Tanh2(x:Single):Single;
const c3 :Single=3;
      c6 :Single=6;
      c12 :Single=12;
      Mul2:Single=2;
Asm
  FLD  x
  FMUL  Mul2
  FLD  ST(0)
  FABS          // a
  FLD  c3
  FADD  ST(0),ST(1)
  FMUL  ST(0),ST(1)
  FADD  c6          // b
  FMUL  ST(2),ST(0) // x*b
  FMULP ST(1),ST(0) // a*b
  FADD  c12
  FDIVP ST(1),ST(0)
End;
```

from : Christian [[a t]] savioursofsoul.de
comment : Any suggestions about improving the 3DNow Divide-Operation??? I simply hate my code...

```
procedure Transistor2_3DNow(pinp,pout : PSingle; Samples:Integer;Scalar:Single);
const ng : Array [0..1] of Integer = ($FFFFFFF,$FFFFFFF);
      pg : Array [0..1] of Integer = ($80000000,$80000000);
      c2 : Array [0..1] of Single = (2,2);
      c3 : Array [0..1] of Single = (3,3);
      c6 : Array [0..1] of Single = (6,6);
      c12 : Array [0..1] of Single = (12,12);
      c24 : Array [0..1] of Single = (24,24);
asm
shr ecx,1
femms
movd  mm1, Scalar.Single
punpckldq mm1, mm1
movq  mm0, c2
pfmul mm0, mm1

movq  mm3, c3
movq  mm4, c6
movq  mm5, c12
movq  mm6, c24

@Start:
movq  mm1, [eax] //mm1=input
```

```

pfmul mm1, mm0 //mm1=a
movq mm2, mm1 //mm1=a, mm2=a

pand mm2, ng //mm1=a, mm2=|a|

pfadd mm3, c3 //mm1=a, mm2=|a|, mm3=|a|+3
pfmul mm3, mm2 //mm1=a, mm2=|a|, mm3=|a|*(|a|+3)
pfadd mm3, c6 //mm1=a, mm2=|a|, mm3=6+|a|*(3+|a|)
pfmul mm3, mm2 //mm1=a, mm2=|a|, mm3=|a|*(6+|a|*(3+|a|))
pfadd mm3, c12 //mm1=a, mm2=|a|, mm3=b=12+|a|*(6+|a|*(3+|a|))
pfmul mm1, mm3 //mm1=a*b, mm2=|a|, mm3=a*(12+|a|*(6+|a|*(3+|a|)))
pfmul mm2, mm3 //mm1=a*b, mm2=|a|*b
pfadd mm2, c24 //mm1=a*b, mm2=|a|*b+24

```

```

movq mm3, mm2
pfrcp mm4, mm3
punpckldq mm3, mm3
pfrcpit1 mm3, mm4
pfrcpit2 mm3, mm4
movq mm4, mm2
punpckhdq mm4, mm4
pfrcp mm5, mm4
pfrcpit1 mm4, mm5
pfrcpit2 mm4, mm5
punpckldq mm4, mm5
pfmul mm1, mm4
movq [edx], mm1
add eax, 8
add edx, 8
loop @Start
femms
end;

```

from : didid [[a t]] skynet.be

comment : mmh why the loop? You can't process more than 2 Tanh in parallel in this filter, can you?
What CPU gain did you get btw?

Anyway, sucks that 3DNow doesn't provide division.. SSE does, though.. DIVPS (or DIVPD to get a double accuracy in this case) would work here. Only problem is that on an AMD I usually get better performances out of 3DNow than SSE/SSE2.

from : Christian [[a t]] savioursofsoul.de

comment : The loop works perfectly well, but it's of course for Tanh() processing of a whole block instead of inside the moog filter.

The thing, that 3DNow doesn't provide division really sucks. Anyway, this way i will save a small amount of performance, but it's not huge. But i believe one can optimize the 12 lines of division further more. Also data prefetching might help a little. Or restructuring, because on AMD the order does matter!

I'll SSE/SSE2 the code tonight. I think SSE gives a good performance boost, but SSE2 precision would be needed, if the thing is inside the moog filter (IIR Filter coefficients should allways stay 64bit to avoid digital artifacts).

Cheers,

Christian

from : Christian [[a t]] savioursofsoul.de

comment : Here's the Analog Devices "Sharc" DSP translation of the tanh function (inline processing of register f0):

```

f11 = 2.;
f0 = f0 * f11;
f11 = abs f0;
f3 = 3.;
f12 = f11+f3;
f12 = f11*f2;
f3 = 6.;
f12 = f12+f3;
f0 = f0*f12;
f12 = f11*f12;
f7 = 12.;
f12 = f12+f3;
f11 = 2.;
f0 = recip f12, f7=f0;
f12=f0*f12;
f7=f0*f7, f0=f11-f12;
f12=f0*f12;
f7=f0*f7, f0=f11-f12;
f12=f0*f12;
rts(db);
f7=f0*f7, f0=f11-f12;
f0=f0*f7;

```

it can be optimized further more, but hey...

from : Gene

comment : $\tanh(x/2) \sim x/(abs(x)+3/(2+x*x))$

better...

from : Gene

comment : $\tanh(x/2) \sim x/(abs(x)+2/(2.12-1.44*abs(x)+x*x))$

Maximum normalized difference 0.0063 from real $\tanh(x/2)$ - good enough now.

[resampling](#) (click this to go back to the index)

Type : linear interpolated aliased resampling of a wave file

References : Posted by mail[AT]mroc[DOT]de

Notes :

som resampling stuff. the code is heavily used in MSynth, but do not lough about ;-)

perhaps, prefiltering would reduce aliasing.

Code :

```
signed short* pSample = ...;
unsigned int sampleLength = ...;

// stretch sample to length of one bar...
float playPosDelta = sampleLength / ( ( 240.0f / bpm ) * samplingRate );

// requires for position calculation...
float playpos1 = 0.0f;
unsigned int iter = 0;

// required for interpolation...
unsigned int i1, i2;

float* pDest = ...;
float* pDestStop = pDest + len;
for( float *pt=pDest;pt<pDestStop;++pt )
{
    // linear interpolation...
    i1 = (unsigned int)playpos;
    i2 = i1 + 1;
    (*pt) = ((pSample[i2]-pSample[i1]) * (playpos - i1) + pSample[i1]);

    // position calculation preventing float sumation error...
    playpos1 = (++iter) * playposIncrement;
}
...
```

Comments

from : Gog

comment : Linear interpolation normally introduces a lot of artefacts. An easy way to improve upon this is to use the hermite interpolator instead. The improvement is _dramatic_!

from : pete [[a t]] bannister25.plus.com

comment : i1 = (unsigned int)playpos;
i2 = i1 + 1;

would this be better as:

```
i1 = (unsigned int) floor(playpos);
i2 = (unsigned int) ceil(i1 + playposIncrement);
```

?

if you are actually decimating rather than interpolating (which is what would give aliasing), then the second interpolation point in the input could potentially be more than i1 + 1.

from : pete [[a t]] bannister25.plus.com

comment : no, sorry it wouldn't :%|

from : mail [[a t]] mroc.de

comment : yes, a more sophisticated interpolation would improve the sound and prefiltering would terminate the aliasing. but everything with hi runtime overhead.

Saturation (click this to go back to the index)

Type : Waveshaper

References : Posted by Bram

Notes :
when the input is below a certain threshold (t) these functions return the input, if it goes over that threshold, they return a soft shaped saturation.
Neither claims to be fast ;-)

```
Code :
float saturate(float x, float t)
{
    if(fabs(x)<t)
        return x
    else
    {
        if(x > 0.f);
        return t + (1.f-t)*tanh((x-t)/(1-t));
        else
        return -(t + (1.f-t)*tanh((-x-t)/(1-t)));
    }
}
```

or

```
float sigmoid(x)
{
    if(fabs(x)<1)
        return x*(1.5f - 0.5f*x*x);
    else
        return x > 0.f ? 1.f : -1.f;
}
```

```
float saturate(float x, float t)
{
    if(abs(x)<t)
        return x
    else
    {
        if(x > 0.f);
        return t + (1.f-t)*sigmoid((x-t)/((1-t)*1.5f));
        else
        return -(t + (1.f-t)*sigmoid((-x-t)/((1-t)*1.5f)));
    }
}
```

Comments

from : terry [[a t]] yahoo.com
comment : But My question is
BUT HAVE YOU TRIED YOUR CODE!!!!!!!!!!!!!!!!!!!!????
I think no, 'cos give a compiling error.
the right (for sintax) version is this:

```
float sigmoid(float x)
{
    if(fabs(x)<1)
        return x*(1.5f - 0.5f*x*x);
    else
        return x > 0.f ? 1.f : -1.f;
}
```

```
float saturate(float x, float t)
{
    if(abs(x)<t)
        return x;
    else
    {
        if(x > 0.f)
            return t + (1.f-t)*sigmoid((x-t)/((1-t)*1.5f));
        else
            return -(t + (1.f-t)*sigmoid((-x-t)/((1-t)*1.5f)));
    }
}
```

from : imbeachhunt [[a t]] hotmail.com
comment : except for the missing parenthesis of course =)
the first line of saturate should be either

```
if(fabs(x)) return x;
```

or

```
if(abs(x)) return x;
```

depending on whether you're looking at the first or second saturate function (in the orig post)

[Sin\(x\) Aproximation \(with SSE code\)](#) (click this to go back to the index)

[References](#) : Posted by [williamk\[AT\]wusik\[DOT\]com](#)

Notes :

Sin Aproximation: $\sin(x) = x + (x * (-x * x / 6))$;

This is very handy and fast, but not precise. Below you will find a simple SSE code.

Remember that all movaps command requires 16 bit aligned variables.

Code :

SSE code for computing only ONE value (scalar)

Replace all "ss" with "ps" if you want to calculate 4 values. And instead of "movps" use "movaps".

```
movss xmm1, xmm0      ; xmm0 = x
mulss xmm1, Filter_GenVal[k_n1] ; * -1
mulss xmm1, xmm0      ; -x * x
divss xmm1, Filter_GenVal[k_6]  ; / 6
mulss xmm1, xmm0
addss xmm0, xmm1
```

Comments

from : [kaleja \[\[a t \] \] estarcion.com](#)

comment : Divides hurt. Change your constant 6 to a constant (1.0/6.0) and change divss to mulss.

from : [little%20eeky \[\[a t \] \] aol.com](#)

comment : error about 7.5% by +/- pi/2
you can improve this considerably by
fitting cubic at points -pi/2, 0, pi/2 i.e:
 $\sin(x) = x - x^3 / 6.7901358$

Sin, Cos, Tan approximation (click this to go back to the index)

References : <http://www.wild-magic.com>

Linked file : [approx.h](#) (this linked file is included below)

Notes :

Code for approximation of cos, sin, tan and inv sin, etc.
Surprisingly accurate and very usable.

[edit by bram]

this code is taken literally from

<http://www.wild-magic.com/SourceCode.html>

Go have a look at the MgcMath.h and MgcMath.cpp files in their library...

[/edit]

Comments

from : asynth [[a t]] io.com

comment : It'd be nice to have a note on the domain of these functions. I assume Sin0 is meant to be used about zero and Sin1 about 1. But a note to that effect would be good.

Thanks,

james mccartney

from : mcodespam [[a t]] gmx.net

comment : Sin0 is faster but less accurate than Sin1, same for the other pairs. The domains are:

Sin/Cos [0, pi/2]

Tan [0,pi/4]

InvSin/Cos [0, 1]

InvTan [-1, 1]

This comes from the original header file.

Linked files

```
//-----  
Real Math::FastSin0 (Real fAngle)  
{  
    Real fASqr = fAngle*fAngle;  
    Real fResult = 7.61e-03f;  
    fResult *= fASqr;  
    fResult -= 1.6605e-01f;  
    fResult *= fASqr;  
    fResult += 1.0f;  
    fResult *= fAngle;  
    return fResult;  
}  
//-----  
Real Math::FastSin1 (Real fAngle)  
{  
    Real fASqr = fAngle*fAngle;  
    Real fResult = -2.39e-08f;  
    fResult *= fASqr;  
    fResult += 2.7526e-06f;  
    fResult *= fASqr;  
    fResult -= 1.98409e-04f;  
    fResult *= fASqr;  
    fResult += 8.3333315e-03f;  
    fResult *= fASqr;  
    fResult -= 1.666666664e-01f;  
    fResult *= fASqr;  
    fResult += 1.0f;  
    fResult *= fAngle;  
    return fResult;  
}  
//-----  
Real Math::FastCos0 (Real fAngle)  
{  
    Real fASqr = fAngle*fAngle;  
    Real fResult = 3.705e-02f;  
    fResult *= fASqr;  
    fResult -= 4.967e-01f;  
    fResult *= fASqr;  
    fResult += 1.0f;  
    return fResult;  
}  
//-----  
Real Math::FastCos1 (Real fAngle)
```

```

{
  Real fASqr = fAngle*fAngle;
  Real fResult = -2.605e-07f;
  fResult *= fASqr;
  fResult += 2.47609e-05f;
  fResult *= fASqr;
  fResult -= 1.3888397e-03f;
  fResult *= fASqr;
  fResult += 4.16666418e-02f;
  fResult *= fASqr;
  fResult -= 4.999999963e-01f;
  fResult *= fASqr;
  fResult += 1.0f;
  return fResult;
}
//-----
Real Math::FastTan0 (Real fAngle)
{
  Real fASqr = fAngle*fAngle;
  Real fResult = 2.033e-01f;
  fResult *= fASqr;
  fResult += 3.1755e-01f;
  fResult *= fASqr;
  fResult += 1.0f;
  fResult *= fAngle;
  return fResult;
}
//-----
Real Math::FastTan1 (Real fAngle)
{
  Real fASqr = fAngle*fAngle;
  Real fResult = 9.5168091e-03f;
  fResult *= fASqr;
  fResult += 2.900525e-03f;
  fResult *= fASqr;
  fResult += 2.45650893e-02f;
  fResult *= fASqr;
  fResult += 5.33740603e-02f;
  fResult *= fASqr;
  fResult += 1.333923995e-01f;
  fResult *= fASqr;
  fResult += 3.333314036e-01f;
  fResult *= fASqr;
  fResult += 1.0f;
  fResult *= fAngle;
  return fResult;
}
//-----
Real Math::FastInvSin (Real fValue)
{
  Real fRoot = Math::Sqrt(1.0f-fValue);
  Real fResult = -0.0187293f;
  fResult *= fValue;
  fResult += 0.0742610f;
  fResult *= fValue;
  fResult -= 0.2121144f;
  fResult *= fValue;
  fResult += 1.5707288f;
  fResult = HALF_PI - fRoot*fResult;
  return fResult;
}
//-----
Real Math::FastInvCos (Real fValue)
{
  Real fRoot = Math::Sqrt(1.0f-fValue);
  Real fResult = -0.0187293f;
  fResult *= fValue;
  fResult += 0.0742610f;
  fResult *= fValue;
  fResult -= 0.2121144f;
  fResult *= fValue;
  fResult += 1.5707288f;
  fResult *= fRoot;
  return fResult;
}
//-----
Real Math::FastInvTan0 (Real fValue)
{
  Real fVSqr = fValue*fValue;
  Real fResult = 0.0208351f;
  fResult *= fVSqr;
  fResult -= 0.085133f;
  fResult *= fVSqr;

```

```
fResult += 0.180141f;
fResult *= fVSqr;
fResult -= 0.3302995f;
fResult *= fVSqr;
fResult += 0.999866f;
fResult *= fValue;
return fResult;
}
//-----
Real Math::FastInvTan1 (Real fValue)
{
  Real fVSqr = fValue*fValue;
  Real fResult = 0.0028662257f;
  fResult *= fVSqr;
  fResult -= 0.0161657367f;
  fResult *= fVSqr;
  fResult += 0.0429096138f;
  fResult *= fVSqr;
  fResult -= 0.0752896400f;
  fResult *= fVSqr;
  fResult += 0.1065626393f;
  fResult *= fVSqr;
  fResult -= 0.1420889944f;
  fResult *= fVSqr;
  fResult += 0.1999355085f;
  fResult *= fVSqr;
  fResult -= 0.3333314528f;
  fResult *= fVSqr;
  fResult += 1.0f;
  fResult *= fValue;
  return fResult;
}
//-----
```