musicdsp.org source code archive, no-frills version

Please notice that this version does NOT contain the comments you can find in the fancy version...
You can find the fancy version on http://www.musicdsp.org.

Contents

- (Allmost) Ready-to-use oscillators
- 16-to-8-bit first-order dither
- 18dB/oct resonant 3 pole LPF with tanh() dist
- 1st and 2nd order pink noise filters
- 2 Wave shaping things
- 303 type filter with saturation
- 3rd order Spline interpollation
- 5-point spline interpollation
- Alias-free waveform generation with analog filtering
- Alien Wah
- All-Pass Filters, a good explanation
- Allocating aligned memory
- Bandlimited sawtooth synthesis
- Bandlimited waveform generation
- Bandlimited waveform generation with hard sync
- Bandlimited waveforms synopsis.
- Bandlimited waveforms...
- Base-2 exp
- Biquad C code
- Bit quantization/reduction effect
- Calculate notes (java)
- Class for waveguide/delay effects
- Clipping without branching
- Coefficients for Daubechies wavelets 1-38
- Constant-time exponent of 2 detector
- Conversions on a PowerPC
- Cubic interpollation

- Cubic polynomial envelopes
- Delay time calculation for reverberation
- Denormal DOUBLE variables, macro
- Denormal numbers
- Denormal numbers, the meta-text
- DFT
- Digital RIAA equalization filter coefficients
- Discrete Summation Formula (DSF)
- Dither code
- Dithering
- Double to Int
- Early echo's with image-mirror technique
- ECE320 project: Reverberation w/ parameter control from PC
- Envelope detector
- Exponential parameter mapping
- Fast binary log approximations
- Fast exp2 approximation
- Fast in-place Walsh-Hadamard Transform
- Fast log2
- Fast sine and cosine calculation
- Fast sine wave calculation
- Fast square wave generator
- FFT
- FFT classes in C++ and Object Pascal
- Float to int
- Float-to-int, coverting an array of floats
- Formant filter
- Gaussian dithering
- Gaussian White noise
- Gaussian White Noise
- Guitar feedback
- Hermite interpollation
- Inverted parabolic envelope
- Lock free fifo
- Look ahead limiting
- Lowpass filter for parameter edge filtering
- LP and HP filter

- Magnitude and phase plot of arbitrary IIR function, up to 5th order
- MATLAB-Tools for SNDAN
- Measuring interpollation noise
- Millimeter to DB (faders...)
- Moog VCF
- Moog VCF, variation 1
- Moog VCF, variation 2
- Noise Shaping Class
- Notch filter
- One pole LP and HP
- One pole, one zero LP/HP
- One zero, LP/HP
- Parallel combs delay calculation
- Phase modulation Vs. Frequency modulation
- Phaser code
- Pink noise filter
- Polyphase Filters
- pow(x,4) approximation
- Prewarping
- Pseudo-Random generator
- Pulsewidth modulation
- RBJ-Audio-EQ-Cookbook
- Reading the compressed WA! parts in gigasampler files
- Resonant filter
- Resonant IIR lowpass (12dB/oct)
- Resonant low pass filter
- Reverberation techniques
- Saturation
- SawSin
- Simple peak follower
- Sin, Cos, Tan approximation
- Sine calculation
- smsPitchScale Source Code
- Soft saturation
- Square Waves
- State variable
- State Variable Filter (Double Sampled, Stable)

- Time domain convolution with O(n^log2(3))
- Time domain convolution with O(n^log2(3))
- tone detection with Goertzel
- Variable-hardness clipping function
- Various Biquad filters
- Waveform generator using MinBLEPS
- WaveShaper
- Waveshaper
- Waveshaper
- Waveshaper (simple description)
- Waveshaper :: Gloubi-boulga
- Wavetable Synthesis
- Weird synthesis
- Zoelzer biquad filters

(Allmost) Ready-to-use oscillators

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 folows... (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];
}
```

16-to-8-bit first-order dither

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;
}
```

18dB/oct resonant 3 pole LPF with tanh() dist

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

Type : Pink noise
References : Posted by umminger@umminger.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]

2 Wave shaping things

References : Posted by Frederic Petrot

Notes :
Makes nice saturations effects that can be easilly 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)
```

303 type filter with saturation

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!!

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 asymmetry 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*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 declick

       outs   aout*ipanl, aout*ipanr  ; Output the sound

       endin

-- Hans Mikelson <hljmm@charter.net>
```

3rd order Spline interpollation

References : Posted by Dave from Muon Software, originally from Josh Scholar

Notes :
(from Joshua Scholar about Spline interpollation 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 nyquest 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 prefilitered for our target nyquest frequency and reasonably oversampled relative to our nyquest 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 linear 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 ThirdInterp(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 )))));
}
```

5-point spline interpollation

Type : interpollation
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)))));
};
```

Alias-free waveform generation with analog filtering

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) = exp(p*x)   if t >= 0,
       0          if t < 0

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 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>
```

Alien Wah

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."

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 ideea 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);
}
```

All-Pass Filters, a good explanation

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 postive
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:

a = -g
b = 1-g^2

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:

```
mid(T) = in(T) + g*mid(T-delay);
out(T) = (1/g-g)*mid(T) - (1/g)*in(T);
```

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:

```
         1
--------------------
1 - g e^(-i w delay)
```

Adding the mixing on the second code line gives the flat-magnitude frequency response of the whole allpass system:

```
|       1/g - g               |
| -------------------- - 1/g | | = 1
| 1 - g e^(-i w delay)        |
```

The phase response is a wavy one: (formula not double-checked!)

```
        (g^2-1) sin(delay w)
atan ------------------------
     (g^2+1) cos(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) :

$$H = \frac{1/g - g}{1 - g\,e^{-iwT}} - \frac{1}{g}$$

$$H = \frac{1 - g^2 - (1 - g\,e^{-iwT})}{(1 - g\,e^{-iwT})\,g}$$

$$H = \frac{-g + e^{-iwT}}{1 - g\,e^{-iwT}}$$

$$H = e^{-iwT} \cdot \frac{1 - g\,e^{iwT}}{1 - g\,e^{-iwT}}$$

Le numerateur et le denominateur sont conjugues. d'ou :

$|H| = 1$

et

$$\arg(H) = -wT - 2 \cdot \arctan\frac{g \cdot \sin(wT)}{1 - g \cdot \cos(wT)}$$

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 :

$$H(z) = b \cdot \frac{z^{-n} \cdot \sum_i a(i) \cdot z^i}{\sum_i \mathrm{conj}(a(i)) \cdot z^{-i}}$$

where $|b| = 1$ and of course n should be large enough that your filter is causal.

-- Frederick Umminger <fumminger@my-Deja.com>

Allocating aligned memory

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);
}
```

Bandlimited sawtooth synthesis

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)

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    = 3.14159265358979323846264338327950 29L;
double twopi = 6.28318530717958647692528676655900 58L;

/* 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

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(partials!=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 coeffecient 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

References : Posted by Emanuel Landeholm
Linked file : http://www.algonet.se/~e-san/hardsync.tar.gz

Bandlimited waveforms synopsis.

References : Joe Wright
Linked file : waveforms.txt (this linked file is included below)

Notes :
(see linkfile)

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
are 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 formular (which has infinite harmonics):

s(t) = (t*f0) mod 1  (t=time, f0=frequency of note)

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

s(n) = (f0*n*Ts) mod 1 (n=sample, Ts equals 1/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-triaval and below is discussed three
techniques to solve the problem - wavetables, approximation through
processing of |sinewave| 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(2PI*f0*k*Ts)) where f0*n < Ts/2

The wavetable can then be played back, pitched up or down subject that pitched f*n < Ts/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 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 abs(sin(n)) and 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:

BLIT(x) = (m/p) * (sin(PI*x*m/p)/(m*sin(PI*x/p))

x=sample number ranging from 1 to period
p=period in samples (fs/f0). Although this should theorectically not be an interger, I found pratically speaking it needs to be.
m=2*((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 (int)p or (int)p +1

As an extension, we also have a bipolar version:

BP-BLIT k (x)=BLIT(x) - BLIT(x+k) (where k is in the range [0,period])

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

Rect(n) = Sum(i=0,n) (BP-BLIT k0 (i) - C4)

C4 is a DC offset which for the BP-BLIT is zero. This gives a nice iteration for rect(n):

Rect(n) = Rect(n-1) + BP-BLIT k0 (n) where k0 is the pulse width between [0,period] or in practive [1,period-1]

A triangle wave is given by:

Tri(n) = Sum(i=0,n) (Rect(k) - C6)

Tri(n) = Tri(n-1) + Rect(n) - C6

C6 = k0/period

The triangle must also be scaled:

Tri(b) = Tri(n-1) + g(f,d)*(Rect(n) - C6)

where g(f,d) = 0.99 / (period* d * (d-1)) d=k0/period

Theorietcally 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/(periodg*(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 hestitate to contact me.

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

Bandlimited waveforms...

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=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 postion 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);
```

Base-2 exp

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);
}
```

Biquad C code

References : Posted by Tom St Denis
Linked file : biquad.c (this linked file is included below)

Notes :
Implementation of the RBJ cookbook, in C.

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 &lt;math.h&gt;
#include &lt;stdlib.h&gt;

#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-&gt;a0 * sample + b-&gt;a1 * b-&gt;x1 + b-&gt;a2 * b-&gt;x2 -
        b-&gt;a3 * b-&gt;y1 - b-&gt;a4 * b-&gt;y2;

    /* shift x1 to x2, sample to x1 */
    b-&gt;x2 = b-&gt;x1;
    b-&gt;x1 = sample;

    /* shift y1 to y2, result to y1 */
    b-&gt;y2 = b-&gt;y1;
    b-&gt;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-&gt;a0 = b0 /a0;
    b-&gt;a1 = b1 /a0;
    b-&gt;a2 = b2 /a0;
    b-&gt;a3 = a1 /a0;
    b-&gt;a4 = a2 /a0;

    /* zero initial samples */
    b-&gt;x1 = b-&gt;x2 = 0;
    b-&gt;y1 = b-&gt;y2 = 0;

    return b;
}
/* crc==3062280887, version==4, Sat Jul  7 00:03:23 2001 */
```

Bit quantization/reduction effect

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)));
}
```

Calculate notes (java)

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 somwhere 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));
    }
}
```

Class for waveguide/delay effects

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 neighbourgh 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;
};
```

Clipping without branching

Type : Min, max and clip
References : Posted by Laurent de Soras

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);
}
```

Coefficients for Daubechies wavelets 1-38

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 logarithimically 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
 *  |XXXXXXXXXXXXXXXX  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] = {
 7.07106781186547524400844362104849039284835937688474036588339e-01,
 7.07106781186547524400844362104849039284835937688474036588339e-01};

const double Daub2[4] = {
 4.82962913144534143374871599864448683816952419504202275201171 5e-01,
 8.36516303737807905575293780916873203459370388348439293495341 4e-01,
 2.24143868042013381025972762240400355467883518184271761387168 3e-01,
-1.29409522551260381174449418812024164174534450659965256907001 6e-01};

const double Daub3[6] = {
 3.32670552950082615998511589139005630012923399245068359708470 5e-01,
 8.06891509311092576494493604088713490519297394994823618165092 0e-01,
 4.59877502118491570095151942147616720808110177431492306643386 7e-01,
-1.35011020010254588696389906699374480562219845223781191975686 2e-01,
-8.54412738820266616928191691817733115361976389880866297635174 8e-02,
 3.52262918857095366027406647155100293277583879174316103989340 6e-02};
```

```
const double Daub4[8] = {
 2.30377813308896500863291183044070850001615248248309297791096 8e-01,
 7.14846570552915647089921955273992603707608401099308175845011 0e-01,
 6.30880767929858907881716338300615220203222922677195117405747 3e-01,
-2.79837694168598542114137471800753854119873202244917528400335 8e-02,
-1.87034811719093084079570672789081419584544174374580091205777 0e-01,
 3.08413818355607636272193625349590501703148217200340334182121 9e-02,
 3.28830116668851997354075135492443886645419411375497125972727 8e-02,
-1.05974017850690321048832085240272291810999649063764198348497 4e-02};

const double Daub5[10] = {
 1.60102397974192914480723748020420733650544124625057832772569 9e-01,
 6.03829269797189670540119306525062107507422163101698698796928 3e-01,
 7.24308528437772927728071244102218640768756218232007372576733 5e-01,
 1.38428145901320731505397146339024697314105791173956102269465 2e-01,
-2.42294887066382031862571379474616361991490808062618598391372 6e-01,
-3.22448695846383746484797550621349283135649841637984722543426 8e-02,
 7.75714938400457135231304893886018198062309945201252798321014 6e-02,
-6.24149021279827427419051911292019297076355716568760732341743 5e-03,
-1.25807519990819994685097399317757929492045916260978502016923 2e-02,
 3.33572528547377127799818341581735574763652474230531509970642 8e-03};

const double Daub6[12] = {
 1.11540743350109463621323917240923439042539591984421675908236 0e-01,
 4.94623890398453085677204176877855588637786382896274362353183 4e-01,
 7.51133908021095350678934498439731685580254783338261200973042 0e-01,
 3.15250351709197629085989654810926396649519923517294524440416 3e-01,
-2.26264693965439820076314500660903465670540153972896994014348 7e-01,
-1.29766867567261935562289605876585460845233749223581470159931 0e-01,
 9.75016055873230491023435525381253423398307474952551427989319 3e-02,
 2.75228655303057286255408395041932136573875878304345432149420 2e-02,
-3.15820393174860295650790806998486690574795323731484233751146 4e-02,
 5.53842201161496139251918398046501220611026277386496429547652 4e-04,
 4.77725751094551063963597524682070705023050121658143429759325 4e-03,
-1.07730108530847956485262160958720003523523360933441968981858 0e-03};

const double Daub7[14] = {
 7.78520540850091790199635219578937483791830529279556843870293 7e-02,
 3.96539319481917306539000390936842856358715114933328740111049 9e-01,
 7.29132090846235119916943070339282051717966061190136378269771 5e-01,
 4.69782287405193122471591160974451738681791305678735953239252 9e-01,
-1.43906003928564975405068362213046001795273570549908483440175 3e-01,
-2.24036184993874982638140420233250964475783089677324655266509 5e-01,
 7.13092192668302647508765705011290482271132745141231465957511 3e-02,
 8.06126091510830719129224803593819058582382096562948905813921 8e-02,
-3.80299369350144135795920616018580358544619693846786989828312 2e-02,
-1.65745416306668806541076748917026547920450439482071370523927 2e-02,
 1.25509985560998406129898860341877795728947404604871003841181 8e-02,
 4.29577972921366521132129122819732222823535039694240974294636 6e-04,
```

```
-1.80164070404749091526826291273955096258565146964109062532 3864e-03,
 3.53713799974520248446295836306425431095906005952004001252 4275e-04};

const double Daub8[16] = {
 5.44158422431040099550094052029993550359955429473305039772 9280e-02,
 3.12871590914299970659162375505717721949731974037022918569 8712e-01,
 6.75630736297289806807800767047183149986911590633636422776 6759e-01,
 5.85354683654206712771265520045098194430326667805336905570 7175e-01,
-1.58291052563493056673805478764663041577447115450282655973 5335e-02,
-2.84015542961546926516203132374164732468435012487145179359 9204e-01,
 4.72484573913282770360590009825894986194801128877007464408 4096e-04,
 1.28747426620478458857029287509708384302260157555648879557 7000e-01,
-1.73693010018075461696161488680959831141308652948839431697 7315e-02,
-4.40882539307947515067637232389635018975183919011099647275 0391e-02,
 1.39810279173982816487229305726334514423955953293434716914 6368e-02,
 8.74609404740577671638274324647564018040214708114067674268 6747e-03,
-4.87035299345157431042218155710982401663497851215700376473 6208e-03,
-3.91740373376947046298080357323776267522935007389049372449 2694e-04,
 6.75449406450569366369547573879299121848963001355843210361 7077e-04,
-1.17476784124769533730628231698890944408669395031150392762 0013e-04};

const double Daub9[18] = {
 3.80779473638783465886976588795511844877171449627841747664 7192e-02,
 2.43834674612590353732041581649284415526361108560923136142 9088e-01,
 6.04823123690111111903076867434236170895956271189611756533 3713e-01,
 6.57288078051300538078212639045173214030585866924591885443 6034e-01,
 1.33197385825007576190954945899795553692178076843366113615 4346e-01,
-2.93273783279174908806403195242198731043896162858990682572 5112e-01,
-9.68407832229764605135081335376966022482545810459909967947 1267e-02,
 1.48540749338106380135072717506042302479125857728060306076 1649e-01,
 3.07256814793333792123174007203788271410580502467074478150 3060e-02,
-6.76328290613299736756422748297190159257879087135373990074 8331e-02,
 2.50947114831451957587189749988554331517627199370963332183 4164e-04,
 2.23616621236790972053737827026909524185564668830885375472 1816e-02,
-4.72320475775139727792570784824246540572951491262793801875 8526e-03,
-4.28150368246342983449679500231453187648118181146328837486 0455e-03,
 1.84764688305622647661912949112567705112108135960031816073 2515e-03,
 2.30385763523195967205216392824542169294066205246371197226 0006e-04,
-2.51963188942710136974988684287860660728218154347802821413 4265e-04,
 3.93473203162715994806898830658915070778247705551701350735 9938e-05};

const double Daub10[20] = {
 2.66700579005555535866174487713085827719249829085128993277 9975e-02,
 1.88176800077691489020892973679093994270254675864039348434 8595e-01,
 5.27201188931725586481744827959508192498140268084022344531 8549e-01,
 6.88459039453603565741871782549235853977136404240733953727 9681e-01,
 2.81172343660577460748726998445589287624388885902615041383 1543e-01,
-2.49846424327315379416101897920779100056466973713207371501 3121e-01,
-1.95946274377377043504299254319098131876677647638277847439 6781e-01,
```

```
 1.27369340335793260082677233201400977078617748042224599556309 7e-01,
 9.30573646035723511603522898354527322694291799894692586806397 4e-02,
-7.13941471663970871453360930760506476729261198370215091752375 6e-02,
-2.94575368218758128582832376014183919938820051606494879769654 e-02,
 3.32126740593410017397636531821591289797833741326709604332351 e-02,
 3.60655356695616965542329141713340329951735051861899476273061 2e-03,
-1.07331754833305750443181141065136444811154878114392321337033 3e-02,
 1.39535174705290116578931844795770756766054285568855242672111 7e-03,
 1.99240529518505611715874224264064321176255536551410528006793 6e-03,
-6.85856694959711626561370981926571419662504333678692051621190 3e-04,
-1.16466855129285450951480971025899189152746185434759736281923 5e-04,
 9.35886703200695913340501303422285439968845621529727644352187 3e-05,
-1.32642028945212448124366753122668330574924096060582975640067 4e-05};

const double Daub11[22] = {
 1.86942977614710840254357293956197572896777445592195854328669 2e-02,
 1.44067021150624512795191584936100114302371896755623960431885 2e-01,
 4.49899764356045334768894037385360367780689537864893347459965 5e-01,
 6.85686774916200511120938631696309793594020496456770349505158 9e-01,
 4.11964368947907462925939648571066730743040041018784531569724 2e-01,
-1.62275245027490362240582726998551154074426432421213020964966 7e-01,
-2.74230846817946961202100945283526662864808952177517822190577 8e-01,
 6.60435881966831919006145788812630265675314216894079154111345 7e-02,
 1.49812012466378496406656261704419329858827242026748465379690 9e-01,
-4.64799551166841872716172258902374457722326096684826074745032 0e-02,
-6.64387856950252052789921553697120319181956689607973962285857 4e-02,
 3.13350902190460760309479840830314453635810568088003196493644 5e-02,
 2.08409043601810630229481125565649101515776183273471569112669 2e-02,
-1.53648209062015994261981160995882274401432649577300012020584 8e-02,
-3.34085887301444560609080861798240610193065835949919084565673 1e-03,
 4.92841765605904112317073974170827369028554772991580241839745 8e-03,
-3.08592858815143165175459072627895330718021660507848858192156 2e-04,
-8.93023250666264613390082462264865398987951987862072879313335 8e-04,
 2.49152523552823498871221687266680108822119930285542538197139 2e-04,
 5.44390746993684716735785687957683219193667852560079397804368 8e-05,
-3.46349841869849955412808515997404321450648804823345803594360 1e-05,
 4.49427427723651009541564828231013091641049798738375346057174 1e-06};

const double Daub12[24] = {
 1.31122579572295175067460908889332806566551064193132500774828 0e-02,
 1.09566272821185154605704505024890542607568050306677404638365 7e-01,
 3.77355135214212657092821260487920614901094170605752633470583 9e-01,
 6.57198722579307089302761128664116983425020328998841214139428 1e-01,
 5.15886478427815608756032648054303270067769308703609005612764 7e-01,
-4.47638856537746266676274731154016652928454363150592413907170 4e-02,
-3.16178453752785536864802935347803109850883903254736438957420 3e-01,
-2.37792572560697276839975460913322578455336655833174115248261 2e-02,
 1.82478605927579679854043611618924171029477144809630269832901 1e-01,
 5.35956967435215032827627672976833228886266518419270582163634 2e-03,
```

```
-9.643212009650708202650320534322484127430880143045220514346402e-02,
 1.084913025582218438089010237748152188661630567603334659322512e-02,
 4.154627749508444073927094681906574864513532221388374861287078e-02,
-1.221864906974828071998798266471567712982466093116558175344811e-02,
-1.284082519830068329466034471894728496206109832314097633275225e-02,
 6.711499008795509177767027068215672450648112185856456740379455e-03,
 2.248607240995237599950865211267234018343199786146177099262010e-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,
 7.294893365677716380902830610477661983325929026879873553627963e-02,
-1.058076181879343264509667304196464849478860754801236658232360e-01,
-2.648840647534369463963912248034785726419604844297697016264224e-02,
 5.613947710028342886214501998387331119988378792543100244737056e-02,
 2.379972254059078811465170958554208358094394612051934868475139e-03,
-2.383142071032364903206403067757391342529227176362262740772 98e-02,
 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,
```

```
 2.18670687758906521491747591821751705176577432127043205903027 3e-01,
-2.71688552278748041414219247618117109460488246568333081431189 6e-01,
-2.18033529993276044755555881270231191197524066947060475274712 7e-01,
 1.38395213864806591073993969002157371398990046322968611905911 9e-01,
 1.39989016584460701249294316227116344032822155561432618133368 3e-01,
-8.67484115681696890456082206672779538297914953951750365749296 4e-02,
-7.15489555040461307358414511517380799095806967312953809999091 3e-02,
 5.52371262592160441161883406053340339791383363251167215767110 7e-02,
 2.69814083079129169739903140321519334337576659580727423328434 9e-02,
-3.01853515403906351871482262348913757378157540665865262488375 6e-02,
-5.61504953035695913321837136769149863745729720392581038769868 0e-03,
 1.27894932663334089615733070578407929937490386157205831348153 4e-02,
-7.46218989268384937181716073918178097195818798881330290043548 7e-04,
-3.84963886802218744578634931609555177409681850828570049305891 5e-03,
 1.06169108560676184303256674938841117303394158214783086389393 9e-03,
 7.08021154235527858644297769761712898347186346418159537167009 4e-04,
-3.86831947312954482107666339805731442732890210784216537990146 8e-04,
-4.17772457703725973526797953983925892838972659013273013105432 3e-05,
 6.87550425269750960387343702162803160189037068765187527988272 7e-05,
-1.03372091845707739466140734259481458626927250949074485069144 3e-05,
-4.38970490178139411525404256136716982932308536080082571815104 9e-06,
 1.72499467536781276988571269274179852358789470986735657691071 7e-06,
-1.78713996831135907633419293847083934388299030997695944699402 2e-07};

const double Daub15[30] = {
 4.53853736157889888145939491021169634666367124378878699791651 3e-03,
 4.67433948927662718917096933484357577657915170021494351311319 7e-02,
 2.06023863986995731539891500947630721930613850564193090270204 7e-01,
 4.92631771708139623606775707402994637261722156513093240216016 0e-01,
 6.45813140357424358176420912010691799643260828749404618107148 9e-01,
 3.39002535454731527691264114383577391875676949179355466933669 0e-01,
-1.93204139609145428706399053432147174630409003914286382793775 4e-01,
-2.88882596566965646248412500982233298131143563043534259497129 2e-01,
 6.52829528487728169228310791986957488203917428559614412596510 1e-02,
 1.90146714007122982348489311658602051795950125817433669687815 6e-01,
-3.96661765557909444838436675189620066838174282068373680544974 5e-02,
-1.11120936037231693365671032467405860885862376216591412050565 7e-01,
 3.38771439235076862085481784443352377086474468741126536946319 5e-02,
 5.47805505845076126891379031258187910860941599742276856424484 5e-02,
-2.57670073284399625859452575426982639220364163482534013839683 6e-02,
-2.08100501696930816778848342467700016205465795136489904099616 6e-02,
 1.50839180278359023632927446017032273624489282330562771623396 8e-02,
 5.10100036040754316970886018556531472480106652734422205552663 1e-03,
-6.48773456031574499518168314921869081695584563938882640792896 7e-03,
-2.41756490761624281166722532630017960522994699581453522332941 1e-04,
 1.94332398038221154176491233254108744101142486557953140145230 2e-03,
-3.73482354137616992009809421364541461138763096803025662574022 6e-04,
-3.59565244362468812164962007590980885819420245408409030562748 0e-04,
 1.55896489920599747947165824122710881625556705962549591522860 3e-04,
```

```
 2.57926991553189368092586241761685591294404236876734070916011 9e-05,
-2.81332962660478136475532477707847866579144387629378890426725 5e-05,
 3.36298718173757980312484521042017747213484665586407818718630 4e-06,
 1.81127040794057708376851091228584116057708592533750785059029 0e-06,
-6.31688232588166442120159729951765765416613791512119551041664 1e-07,
 6.13335991330575202905629946028978860198919045088539651217384 5e-08};

const double Daub16[32] = {
 3.18922092534773802976954756464595868706708675013142876787587 8e-03,
 3.49077143236733464103014722402302000921824143050398414614005 4e-02,
 1.65064283488853117899125273056113481158483500234272324021359 2e-01,
 4.30312722846003813740392542435768462063397047803698677392464 6e-01,
 6.37356332083788898631985241299603053649859594081419812596775 1e-01,
 4.40290256886356900039086916357167928852780303513527257878988 4e-01,
-8.97510894024896428571871807744259743065924744558266014962471 8e-02,
-3.27063310527917704646290567568911964175722891822881242814172 3e-01,
-2.79182081330282766826451959502687320433997121917473604153547 9e-02,
 2.11190693947104288720968016326883790092849142616767943925104 2e-01,
 2.73402637527160413648524575720161796542902781950713022023150 0e-02,
-1.32388305563810390450047414775649337509228781770602797879854 9e-01,
-6.23972275247487176567450339412002586544465631167876099076145 8e-03,
 7.59242360442763158214849874394142246153040594610094335194031 3e-02,
-7.58897436885773763849489086463699579658697514499092540009716 0e-03,
-3.68883976917301423335266632089455431471874842970673083106406 8e-02,
 1.02976596409559694116500058007661690052885626580366220885414 7e-02,
 1.39937688598287310295045187367032972640984029172786898849010 0e-02,
-6.99001456341391667028424953651728833805785619964646907811575 9e-03,
-3.64427962149838993216900054093362938705533397335310866884121 5e-03,
 3.12802338120626883166120255985467876782147190619360811745036 0e-03,
 4.07896980849712836241747032340609578243195297231054671507139 7e-04,
-9.41021749359567588926645395363587540775474721673448050925027 3e-04,
 1.14241520038722392644022809955566294583968434493647265287709 1e-04,
 1.74787245225338180380175863766074687498602472861539989797195 3e-04,
-6.10359662141093583516236915052221281195725998196591914396172 2e-05,
-1.39456689882088934519907831199840198232527356919867533540870 7e-05,
 1.13366086612762585875884876288653699751947106820375366175784 3e-05,
-1.04357134231160650152545473726261540488747893063567647154603 2e-06,
-7.36365678545120551209969571972556364658544554584166332743356 9e-07,
 2.30878408685754586640541273294200612130630673586665552537254 4e-07,
-2.10933963010074309700057262360348990683629758459160530774534 9e-08};

const double Daub17[34] = {
 2.24180700103731285353596267707443691406219188056037073325053 1e-03,
 2.59853937036060433891486459172078831547394452487824129439994 8e-02,
 1.31214903307824406577550623185906996014429360925997853006700 4e-01,
 3.70350724152641150449254819072188644947707887689680382365042 5e-01,
 6.10996615684622818188667886767937208273709389335872629137178 3e-01,
 5.18315764056937839325453852808596804621681719771841640243990 4e-01,
 2.73149704032936350043125071914758648035046981896456300367294 2e-02,
```

```
-3.28320748363961736090966534072506176758159769815155802467913 0e-01,
-1.26599752215882702874467911093382550505396626010408616210372 8e-01,
 1.97310589565010992785404704478193014255142241413564691712228 4e-01,
 1.01135489177470272150969985643343480219662254549966487610943 7e-01,
-1.26815691778286311094857112866233168038479218591501706573213 7e-01,
-5.70914196316769272891123947865138232416116086984534705399014 4e-02,
 8.11059866541608850796588574855542920102436419095449919402067 8e-02,
 2.23123361781037959533913605953481375623224211409368924402086 9e-02,
-4.69224383892697373330089705921140050713876812549803060287843 9e-02,
-3.27095553581929378165536022217749445206952595806160939280927 5e-03,
 2.27336765839462703184561624478844896990671374133833949802486 4e-02,
-3.04298998135463706859248263790720607863339545722509658828788 1e-03,
-8.60292152032285483171370641324365991792673628427173061192098 6e-03,
 2.96799669152609487280648506000803826995946384654837899504419 5e-03,
 2.30120524215354562430205986903842360424197668018944747606476 4e-03,
-1.43684530480297612622289040298038490350367453072993580956143 4e-03,
-3.28132519409837971395444401752011507581240244272874970019565 1e-04,
 4.39465427768643677838567752731784163228924931973889217946591 0e-04,
-2.56101095665484588272989121094992022166408206153190965517841 3e-05,
-8.20480320245339183909548257628218986613627304963676433868959 3e-05,
 2.31868137987459508448206820570627757210669517409189533853073 4e-05,
 6.99060098507675127320454970085537862776275858590205796402748 1e-06,
-4.50594247722298819410226820637831212971357260071649994491841 6e-06,
 3.01654960999455741560520759487993976347616870521764689770270 6e-07,
 2.95770093331685675497990525881615136787034562892431730735463 9e-07,
-8.42394844600268017878707129692287706841031094222279962259313 3e-08,
 7.26749296856160811087976744140903503415858171978979108889204 6e-09};

const double Daub18[36] = {
 1.57631021844076043154074492993977774767075371099166036368442 9e-03,
 1.92885317241463770592139171582905241995466702528849757223671 4e-02,
 1.03588465822423596224191049193725359647069655522024167297622 4e-01,
 3.14678941337031699057199825565257993178670619048937450949130 7e-01,
 5.71826807766607223481858937090062341939367374313093056129532 4e-01,
 5.71801654888651335289111999406596502566804788281852506075939 5e-01,
 1.47223111969928141575097727108107231255786410735570138780167 7e-01,
-2.93654040736558744247903099498115072393571072903505323966175 2e-01,
-2.16480934005142971123767862566827147143793723566949240838869 2e-01,
 1.49533975565377789350930173891366720880481669189376561026194 3e-01,
 1.67081312763257404514931813995013474532420564635398808315225 0e-01,
-9.23318841508462806042937255865945973143184800014456961207450 8e-02,
-1.06752246659828485593220058161498486138526640462411208391770 2e-01,
 6.48872162119054428194757795514191146312938211663414784613714 9e-02,
 5.70512477385368841209076884649962226059622612043103852460067 6e-02,
-4.45261419029823247155614355974465349297147789143983359275503 4e-02,
-2.37332103958600010327520958266521611019751933071349023307156 5e-02,
 2.66707059264705902998790863167202034320789599993607281336347 1e-02,
 6.26216795430570748523609314449788250199032520474501319026805 2e-03,
-1.30514809466120017727763644760080716975519105450757166660613 3e-02,
```

```
 1.18630033858117465730174159216181908454489941745231740518561 5e-04,
 4.94334360546673813066552951680297483429963831336647776529520 3e-03,
-1.11873266699249707280065885523865018231806048258497014551268 7e-03,
-1.34059629833610662951756722825158360982304452468598664032394 2e-03,
 6.28465682965145712561944988542083821755102279630158287434965 2e-04,
 2.13581561910340688403905281434192602587320032599646652254344 0e-04,
-1.98648552311747948579824541636248955492779788026401787613960 5e-04,
-1.53591712353472467506977033587671719370047242702151323658728 8e-07,
 3.74123788074003818109220813803539395230429261579398503073136 3e-05,
-8.52060253744669520391925491165552302243759695622637651230591 7e-06,
-3.33263447888582188878245203334103682731150590779649843982933 7e-06,
 1.76871298362761545587632873075537517641250135911405881545310 0e-06,
-7.69163268988517614600015287853959840581739758815652511676990 8e-08,
-1.17609876702823169845098235656129256134757977769539695352814 1e-07,
 3.06883586304517480093547829493397537245017978789457449293057 0e-08,
-2.50793445494859826719517318314712673180631714486827581994140 3e-09};

const double Daub19[38] = {
 1.10866976318171057109915419520971516424529967777343593213545 5e-03,
 1.42810984507643973743988915295019923474566344216366595787071 5e-02,
 8.12781132654595506529630678490162483984497997102862036649772 6e-02,
 2.64388431740896784674810038028942687386237780721192071841738 5e-01,
 5.24436377464654915336057597548406462604463364104807211639316 0e-01,
 6.01704549127537894886707713592180262053656563958596329331393 1e-01,
 2.60894952651038829287245667531052832417267310130190773992521 3e-01,
-2.28091394215482646374632577605463720709378723708642590953482 2e-01,
-2.85838631755826241854597569502898423721735609558833514992211 9e-01,
 7.46522697081032663676343311187881900586586614973190965636539 9e-02,
 2.12349743306278488809060856705982419707707420087883944841690 8e-01,
-3.35185419023028786816938841878573150697784507523896681981403 2e-02,
-1.42785695038736574977960273162611281299849770615242850862756 2e-01,
 2.75843506256286687501474352016219865537447459696342308076281 8e-02,
 8.69067555558122324884764542880844303478520800246819275964035 2e-02,
-2.65012362501230408990183584367638736107506801768674780817134 5e-02,
-4.56742262772309080564544421429579601793893573211563005088010 9e-02,
 2.16237674095850471303298425717237235431809706785875254257102 0e-02,
 1.93755498891761276463709435445799981449688509587582554640696 3e-02,
-1.39883886785351416325040123524866252191681386745309583680836 6e-02,
-5.86692228101217472658449343605437377381460834080875817737276 5e-03,
 7.04074736710524315301451120740062010940168989766538307822939 8e-03,
 7.68954359257548355974913914867395516347794708603940612954642 2e-04,
-2.68755180070158200395736385507039863653403892098247829017026 7e-03,
 3.41808653458595776565165729046380813521421484881951725779403 1e-04,
 7.35802520505435207026048190539728187518317579277990485818949 4e-04,
-2.60676135678628005731831513089752279038393936207356340861354 7e-04,
-1.24600791734158775344978440890165399031734141334198090475759 2e-04,
 8.71127046721992296541686238819112826841293389328208351772944 3e-05,
 5.10595048707388605304922280993423157368736799210628266938926 4e-06,
-1.66401762971549445462067771989919863033367560881201810873914 4e-05,
```

```
 3.01096431629652633969533445472594363264579893816242716885 1382e-06,
 1.53193147669119306393183238108663603120312303272347746362 4141e-06,
-6.86275565776914270188355461348673285445274075277139241175 8418e-07,
 1.44708829879784454207821986329161542055167357407136783431 6167e-08,
 4.63693777578260422343085772821094889887174829108596229664 9320e-08,
-1.11640206703582581639050476914247258646497579928447368224 6076e-08,
 8.66684883899761935032301354078212462728974219027305931912 2840e-10};

const double Daub20[40] = {
 7.79953613666846321586199481888937097051072203923286388003 1127e-04,
 1.05493946249503983245448097301564149823196146873323669129 9796e-02,
 6.34237804590815149758734658266878513640652331572966635364 3372e-02,
 2.19942113551397045008033597253720939212130676101088220929 8252e-01,
 4.72696185310901696371024146510144623075780414117172784583 4637e-01,
 6.10493238938593820163151566008420190685862892469544889882 4748e-01,
 3.61502298739331062919560266526863174496708472307967789413 6358e-01,
-1.39212088011483872580697054515553051826494491543780831481 3582e-01,
-3.26786800434034967403112283790537066671664558748002174442 5550e-01,
-1.67270883090770075751717499730429705400374430362047939400 6890e-02,
 2.28291050819916322972842912664822308643754723725029083563 9880e-01,
 3.98502464577712021979058107652217418110402757695442768445 6660e-02,
-1.55458750707267955931530787056246437435999609175228515707 7477e-01,
-2.47168273386135840158799229916992226291515141334931351368 5587e-02,
 1.02291719174442557886101368101686608388838138523308151658 3444e-01,
 5.63224685730743550695324698821520986156680066440278593859 1145e-03,
-6.17228996246804597331865833408328355820927876200704182325 0642e-02,
 5.87468181181182649130067974208199716720974344695690184195 9711e-03,
 3.22942995307695817588544086061721911756455860503597960107 3235e-02,
-8.78932492390156134875365036670069591650303093928383096815 1332e-03,
-1.38105261371519200781960642386035659049690428572473035660 2106e-02,
 6.72162730225945683533685052140542556052002523791570836200 2910e-03,
 4.42054238704579096305822952667351408880899947811558115346 8068e-03,
-3.58149425960962277755616963835823837576519424862389103494 0330e-03,
-8.31562172822556919248258519937323095692448422113573997339 0038e-04,
 1.39255961932313632390525499934796728376054414739753053114 2397e-03,
-5.34975984399769505175971637721368003618579605908735317207 3952e-05,
-3.85104748699217606065028850147571646326623303593702230364 9838e-04,
 1.01532889736702905079748878530605652252997926757200399090 1472e-04,
 6.77428082837772955801118440672797822129579665220081983946 4354e-05,
-3.71058618339471286422722127121640841695822526498061282261 7745e-05,
-4.37614386218399681037309582252860760690062059258576219054 2483e-06,
 7.24124828767362010284310587749718156546872575738700713955 5885e-06,
-1.01199401001888615034047541375684910319739506943108500570 9201e-06,
-6.84707959700055689416333478757515975910909133009296399036 4192e-07,
 2.63392422627000108412905779199436712155576968661674716226 2697e-07,
 2.01432202355051269432475784594402604790441413663377695839 2681e-10,
-1.81484324829969597321060525822702408145853111076208337131 0917e-08,
 4.05612705551832766099146230616888024627380574113178257963 252e-09,
-2.99883648961931956640776707837270538573246005268562192317 8375e-10};
```

```
const double Daub21[42] = {
 5.48822509852683708677633667599252142675067305458824552383477 5e-04,
 7.77663905235478375433878739808879986251077905955562370487923 4e-03,
 4.92477715381772749139985337834005696810448316159832069365795 4e-02,
 1.81359625440381515626037872276462419093195151070805051651918 1e-01,
 4.19687944939362773094685060908926633997360154303629487177265 3e-01,
 6.01506094935003897562988066402095595306654259389612670534612 2e-01,
 4.44590451927600340364329099452360101615134274308987847847896 2e-01,
-3.57229196172552904592291417800530718903676254714396657806683 8e-02,
-3.35664089530529509483297886711436306998757528225609835149973 1e-01,
-1.12397071568450981351500498134030690164182421246419797349029 5e-01,
 2.11564527680872392384678164523846865943086273624889612852937 3e-01,
 1.15233298439687104199343441168173042810316001659455894468796 7e-01,
-1.39940424932547224924775876483977690322650365750207167024530 4e-01,
-8.17759429808638288738730363419379054252257067023455615756678 6e-02,
 9.66003903237242207023218970037253968162778332224982984227551 7e-02,
 4.57234057492287923925120294473123542103482871075338119134518 6e-02,
-6.49775048937323206333231110600861668574892941945224954469096 7e-02,
-1.86538592021185153409324441200814126613120809300721713923217 0e-02,
 3.97268354278504417519746440075612681829991899248258786699970 7e-02,
 3.35775639033811084253260476637620076079166995410667993314472 3e-03,
-2.08920536779790794878523547974621237172821986652521113534370 7e-02,
 2.40347092080543476238063216978568954591052566739631355067965 2e-03,
 8.98882438197191187534946339839546411441781794973891110137231 2e-03,
-2.89133434858890124737526871801588261084467593111746349555195 8e-03,
-2.95837403893283128075077022821551095983017026417695571982751 0e-03,
 1.71660704063062413849450628256923012633330853353550279923533 3e-03,
 6.39418500512030214643254376705286543609999438764735945224934 7e-04,
-6.90671117082101650726893922889378479051827074431352554871406 5e-04,
-3.19640627768043719370883422080464034763698490127094808833910 2e-05,
 1.93664650416508061532369668985600491057977756850421878202902 7e-04,
-3.63552025008633830944285500618637075220633142987113659692713 7e-05,
-3.49966598498744795397407949004659724027626804440962572268984 9e-05,
 1.53548250927604928312423349864605047209648232929971914110712 8e-05,
 2.79033053981448704610616958269176791628379394602592238755691 7e-06,
-3.09001716454569919715855593685269732598586458841816798268540 0e-06,
 3.16609544236703055660388900983395444005854535577778178200027 8e-07,
 2.99213663046485279440129460753681368277129235250632809612585 7e-07,
-1.00040087903059733204546060051662197167936396516624921106375 5e-07,
-2.25401497467333013156318485145682599161791554964330875482815 9e-09,
 7.05803354123112185902094797690390468546451282573123049514422 6e-09,
-1.47195419765036526518954960081669877821324706138947027733717 3e-09,
 1.03880557102370655303537313876037270349294261751881612257005 0e-10};

const double Daub22[44] = {
 3.86263231491098215852435890061546036887785200957689968076731 6e-04,
 5.72185463133453912080978340348449333355536159138620812918383 3e-03,
 3.80699372364110849476987304639182557444772706895344839045633 5e-02,
```

```
 1.48367540890111428501440444871024983738583637316821561642703 0e-01,
 3.67728683446037478861469081845237282743053564969646272033489 7e-01,
 5.78432731009524427142118183173544410638509995790865714559010 4e-01,
 5.07901090622163901839152332539071683656871319249871156271128 2e-01,
 7.37245011836301516557013901653065311372517241210495535036811 4e-02,
-3.12726580428296191803322622262178853707845253599354544071698 8e-01,
-2.00568406104887093932436124404220017413290584486823744713038 2e-01,
 1.64093188106766481860622322628688571255438531741222883670588 8e-01,
 1.79973187992891303725215429531308316838784079142498842275776 2e-01,
-9.71107984091147096927420917969173325145673513799420155292679 9e-02,
-1.31768137686683410751364851814683834547787502235208835752383 8e-01,
 6.80763143927322155673920214700458055936744255064138818188602 3e-02,
 8.45573763668260750336281365935678649435763580519741090587707 8e-02,
-5.13642542974441324572794998401888470790944176847709194458458 4e-02,
-4.65308118275067134787583360784697999782577127797654808090442 3e-02,
 3.69708466206980205761531889298858182563789669687636134335438 0e-02,
 2.05867076275653604406024971067665680728167145160963298148713 9e-02,
-2.34800013444931886856014285451936498736388233375475381979138 1e-02,
-6.21378284936465849906933612380760829312290045050844042010446 2e-03,
 1.25647252183433740688701783549560446381538299321429608817222 1e-02,
 3.00137398507643595122912925558825574690493704297931605448518 3e-04,
-5.45569198615671707659535316307167910786876239536723472659227 3e-03,
 1.04426073918602532335075565918473406080743217261168941374502 9e-03,
 1.82701049565727908011259743685015711023533677206296104115460 7e-03,
-7.70690988123119623288037272295551978165576991363456575733973 9e-04,
-4.23787399839180079953194776800397697819743830253352866182575 8e-04,
 3.28609414213678734198375847140593540582332307282961924852369 7e-04,
 4.34589990453200337904699262557507609282380966593357557871069 6e-05,
-9.40522363481576042184519009835267364788129898004051209159994 3e-05,
 1.13743496621259317273614427486663921033982020313567050528725 0e-05,
 1.73737569575618935616356507450540590685974660586777200232050 9e-05,
-6.16672931646757837215225166842297915216958730721270898176896 6e-06,
-1.56517913199516015930742699357820473337811274257992650383209 5e-06,
 1.29518205731887757388971123234506814780039572192568256639493 6e-06,
-8.77987987336128627688811704615304905391724376047581678922676 4e-08,
-1.28333622875175441781969393211406488707509603026474807997673 6e-07,
 3.76122874933736236615671164818774339916423939780362902261286 2e-08,
 1.68017140492298888555433118369128024596229024765443811480711 2e-09,
-2.72962314663297608344932736173910475444322190331774576893884 6e-09,
 5.33593882166748990516978322703680453325301111788658630543561 5e-10,
-3.60211348433955470379480781093930184729910697023781433410427 4e-11};

const double Daub23[46] = {
 2.71904194128288841419267360970330235709833600392092395892475 7e-04,
 4.20274889318383353839003437252351147234521556361100340798470 1e-03,
 2.93100036578841151473620401892948042787431746067607995951513 1e-02,
 1.20515531783971933630605389561189908900427433689170906795803 5e-01,
 3.18450813852865236341652774846047215279057503140983041725964 0e-01,
 5.44931147873520428267424067242198438750414992483454449546679 3e-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,
 3.122876449818144997419144765125750522437659393621577492535411e-03,
-2.465014005163512031940473100375377210862560761576109755841161e-04,
-1.061231228886651321139357625683805642193648671030425010215075e-03,
 3.194204927099011503676530359692366990929679170022583007683112e-04,
 2.567624520078737205563856675376636092314813400664190770435450e-04,
-1.500218503490340967673163290447832236259277810659068637402668e-04,
-3.378894834120903434270962452674534330903724108906662510305045e-05,
 4.426071203109246077621875303440935335701832843654692827539837e-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,
```

```
 2.80985553233711883344262608511540294184295947592927888328140 9e-01,
-1.87271406885156237698188715977579146906026577844166784030793 4e-01,
-3.17943078999362737545394848979770755089808778916002518266429 9e-01,
 4.77661368434472818795019832303136086634910499403555320078863 1e-03,
 2.39237388780310855197326829194582482221485813451231771581561 6e-01,
 4.25287296414838325814736447217064523268434323548695154053389 3e-02,
-1.71175351370346889689763851508057239394916594233555639791766 6e-01,
-3.87771735779200162017759472619957268844648803375077102019028 3e-02,
 1.21016303469224236231263731114906228665937703904600680152382 6e-01,
 2.09801137091448153498088382732601706312163726272844778360551 8e-02,
-8.21616542080016670229146600616418946091681674862996819802889 8e-02,
-4.57843624181922163799751633976506882526015916989396789487727 2e-03,
 5.13016200399808791555533488139868895884307849459514039487388 4e-02,
-4.94470942812562829981592003264955081187788721928275117479821 1e-03,
-2.82131070949018909811389536190069922888690099541275919767405 8e-02,
 7.66172188164658589732989990430876440538465840461366981784343 0e-03,
 1.30499708710857358305249406788371753304310185712865323378339 6e-02,
-6.29143537001818778072184358116934390086429863408574386150976 7e-03,
-4.74656878632311380047779695951355840173225280090598238501724 5e-03,
 3.73604617828252334517905216081033286872512635649315572862557 2e-03,
 1.15376493683948150485828249520227198445441004668280537515756 6e-03,
-1.69645681897482439427453463641211624308031260132232564274158 9e-03,
-4.41618485614152006336595890007940673763624368213836356187775 0e-05,
 5.86127059318310993371673545027289403542579234780651567869576 5e-04,
-1.18123323796955474061302122775656896680689230845722101625796 1e-04,
-1.46007981776261683892430181808272903631453947681102325567066 6e-04,
 6.55938863930563408530373856045506197436935453827131607150269 8e-05,
 2.18324146046655836336504403298425770979118764096350938054930 7e-05,
-2.02288829261269768286085998720045570261485559541226751055865 9e-05,
 1.34115775080911471931993755318602366058108415182859322289366 3e-08,
 3.90110033859770261040901412902422385312791153000976679335249 2e-06,
-8.98025314393840772414992666998079116637838801329388771840479 6e-07,
-4.03250775687997162409898324735898342523609211038772431524464 6e-07,
 2.16633965327857463917639397851024633547894669739640035928141 2e-07,
-5.05764541979250030849250892434324897931750786652068841756760 6e-10,
-2.25574038817608610736882167494717580400532315344317052652027 7e-08,
 5.15777678967199963895077426631320871501541969964333378462636 3e-09,
 4.74837582425623111809445354979917582452655999433322745673743 3e-10,
-4.02465864458437977425149957446819511860169871355429494175655 9e-10,
 6.99180115763823097413269643350962593402167779345373222554295 1e-11,
-4.34278250380371024725903755288674945795105312420381418581129 7e-12};

const double Daub25[50] = {
 1.34802979347018899457848924715935605537046065650888147126861 1e-04,
 2.25695959185477952012139104962805614927001686066666192813074 7e-03,
 1.71867412540401553381718691495484890224119400244469622101313 1e-02,
 7.80358628721326755975065932048140366842205219925713916838608 4e-02,
 2.31693507886021819990062151805708910494621688151207536162421 4e-01,
 4.59683415146094593789697386453965994401026085804994739609327 7e-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,
 6.675216449401860666895983072443984697329752470942906490126865e-02,
-7.708411105657419356208567671699032054872853174701595359329826e-02,
-3.717396286112250887598137324046870459877639250821705817221557e-02,
 5.361790939877949960629041419546536897037332284703545849594129e-02,
 1.554260592910229163981295854603203625062268043511894295387375e-02,
-3.404232046065334099320628584033729153497903561399447916116575e-02,
-3.079836794847036661636693963570288706232460663070983852354326e-03,
 1.892280447662762841086581178691039363674755753459524525886478e-02,
-1.989425782202736494289461896386235348901617760816745484282494e-03,
-8.860702618046368399013064252456569691996123318336053102786 98e-03,
 2.726936258738495739871469244610042793734119359765762028996059e-03,
 3.322707773973191780118197357194829286271392998979276105842863e-03,
-1.842484290203331280837780430014195744813667655929909114672154e-03,
-8.999774237462950491085382524008429604309720852269895692000702e-04,
 8.772581936748274843488806190175921376284150686011179612908221e-04,
 1.153212440466300456460181455345639872216326644527860903202733e-04,
-3.098800990984697989530544245356271119416614147098459162436317e-04,
 3.543714523276059005284289830559259809540337561365927850248007e-05,
 7.904640003965528255137496303166001735463107762646364003487560e-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,
-2.611598556111770864259843089151782206922842627174274274741722e-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,
```

```
 6.22747440251496048419358170510741593769053864101330974598396 2e-02,
 1.95039438716770099424589150836932469470382052248978912590861 2e-01,
 4.13292962278356368611610868666654708284674122804223273147614 7e-01,
 5.73669043034222260319555714785302206075839266408663339652034 5e-01,
 4.39158311789166232193147756579410563381536338408459055988949 3e-01,
 1.77407678098668572782353356203155689322657131988141767649259 5e-03,
-3.26384593691780021638534083005534995344774500576941628717749 7e-01,
-1.74839961289392504266483568360658421524858234543881634617004 2e-01,
 1.81291832311122696070545976602543091871623358416798294204442 4e-01,
 1.82755409589672374653753383203328683968993192470976056794559 5e-01,
-1.04323900285927043914800913720274765842096814433010851017929 0e-01,
-1.47977193275254493578231454636945818824394777292298006407120 5e-01,
 6.98231861132923651375659168395020895511060321237941233470114 5e-02,
 1.06482405249808630323659379771534440583601500292931929171577 7e-01,
-5.34485616814831914949357726939007421396023701309943943113208 6e-02,
-6.86547596040359152545472525871535128094743582335401114085800 1e-02,
 4.22321857963720354120657090275328824779085776006789445611492 7e-02,
 3.85357159711118642583214456736232814299488539525543886796878 1e-02,
-3.13781103630677548424464477633759443509409696433640279807236 0e-02,
-1.77609035683581835409429862588405817035412904425995101918273 2e-02,
 2.07349201799638247588779007306898422451507766551710339989885 4e-02,
 5.82958055531888797193931574759661303847956194308529107278735 9e-03,
-1.17854979061930289372862446840213807250422652754032546384739 0e-02,
-5.28738399262681443919863076521796980496631997103800399398448 0e-04,
 5.60194723942380485320651423994047478897718846045205346277032 4e-03,
-9.39058250473828964616569867507064176581079086351433920520599 8e-04,
-2.14553028156762098030540140343222166884798029560074891374890 2e-03,
 8.38348805654361604638192405455405210493778437943543642669056 0e-04,
 6.16138220457434419370378901269641156121468238827167321419773 1e-04,
-4.31955707426180746671290191348194347852199161160743397179460 2e-04,
-1.06057474828380388996615080355183740255386681619165995934705 3e-04,
 1.57479523860749359054776566659081125808771569973771458390360 e-04,
-5.27779549303786897629356663601562760924884745764652524627103 6e-06,
-4.10967399639147781632650243899746653282263938511909023096525 2e-05,
 1.07422154087219503127358440924506062310493133093827393648459 3e-05,
 7.00007868296498673485910249521068480964365747425392107493468 4e-06,
-3.88740016185679518758779041070655057603360309795406507402312 8e-06,
-4.65046322064026263923114594453609297344659602746983386000161 8e-07,
 7.93921063370995208837345925506736079337028478868297906512281 0e-07,
-1.07900423757867141192296158384571612606065821394384037516265 4e-07,
-8.90446637016859076905298336272156720275059191474101683507125 7e-08,
 3.40779562129073000867382107214820587991557116806912418558069 e-08,
 2.16932825985032310698622229652593009993587386102631078808622 1e-09,
-3.77601047853232432818404366755657638563984646033789496313862 1e-09,
 6.78004724582863666830580819260709151760534947867744246858082 5e-10,
 1.00230319104652691350928184413625800403417730967326953341864 4e-10,
-5.84040818534117146846549244779981926290531757684742697075770 0e-11,
 9.13051001637179624392323292665025257023905481593948390005668 1e-12,
-5.25187122424443503781050345256427982853900707167872428571746 4e-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,
 2.272732884141708265275037216925482827043581894357907763081103e-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,
 6.856635609684880675273184141746359000591385833807880272568038e-03,
 1.342626877303679609082208800217479591902967766971379107017011e-03,
-3.332854469520006162763300141047111065412307706449049389557931e-03,
 1.457529625931728587128588244152604734177322144376309490881599e-04,
 1.301177450244135139135787970279897042994109161268159963884641e-03,
-3.418351226915427611946547437228006377896519777431057005796358e-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,
-3.472468147394389269364673179891460601330730511237974736379548e-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,
-3.295790122476585807069953975043096139541415768606924980926275e-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,
-4.333336861608628393863254980828284403766309203453808666888800e-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,
```

```
-1.00432604133342260178184856043212092063464869278235785547 3103e-05,
 1.24790031757483414605238169275279604705244326598223242264 2017e-06,
 1.84036373451776919168437930903927781035062030533090053640 4818e-06,
-6.67021547995489258874745045808522588009688269939725677496 7304e-07,
-1.75746117320984277990367626497191863587090698328139293981 2547e-07,
 1.49066001353536217098934006503306195196093395438863350726 4360e-07,
-8.26238731562655696596642924360098489965003970483108098865 8278e-09,
-1.78413869087571007719171394144126324656073841021362454611 6655e-08,
 5.04404705638343644463125284005786200226408772067680858037 3667e-09,
 6.94454032894622695297670471867769752541005140505566257553 0111e-10,
-6.07704124722901022476024530559630780383005353383684938468 0534e-10,
 8.49222001105638210546120607724037702440440463894759129976 1197e-11,
 1.86736726378339041896387914617545237694045358579142884100 4699e-11,
-8.36549047125880079934928979439790890076705408521600819737 2193e-12,
 1.18885053340590152084232174902108949720394068888236451845 5403e-12,
-6.36777235471485733563269209226725426636893459097369382094 2617e-14};

const double Daub29[58] = {
 3.31896627984152476181354635981807544134916997592243998884 3475e-05,
 6.40951680304443454083370672912059632208306171693500498737 4676e-04,
 5.70212651777337543476084399862350749491455146496812645516 8657e-03,
 3.07735802214083767671670733651675181471331201834471915092 3618e-02,
 1.11370116951740530476218616637032777019132577234219071511 8617e-01,
 2.80653455970982937696888126277048060650092009239853422961 5289e-01,
 4.89758804762199314359270593299357353923583961005533162024 0518e-01,
 5.51374432758375195122374607167013599246698439123342966388 6536e-01,
 2.89105238335829163460569111358626406151318015835446095246 9246e-01,
-1.54028734459900054246629377950337014173133998291928095123 0240e-01,
-3.30040948917588052029508377948701261195931053962962712461 3719e-01,
-5.57068000729408578151454193171579578430941023572621440035 0351e-02,
 2.36105236153025941598311073405462677064946835732836242683 0433e-01,
 1.12419174873188376476974067053588054307681781686151866789 8467e-01,
-1.60877988594187736077161546553185233308515994015996839359 0303e-01,
-1.07845949938721420107788195735470791378624115393426431658 9273e-01,
 1.14472295893818257973413593006005328626782279764039338690 3440e-01,
 8.32207471624497579029734883503253735789192053600262778494 1129e-02,
-8.51254926156355023283231133142080458188123544886283450728 1486e-02,
-5.50274895253257232092454145062665006770734472534484109987 3446e-02,
 6.34791645842118663357778931469897236108161199479414011930 2163e-02,
 3.05315432727041364663732821209394103059213322523172896404 7047e-02,
-4.51879812777883451597970447530440569139009032747497208979 0857e-02,
-1.29171425542667946296647396255541066038767118242807657068 6472e-02,
 2.94704318717476411102812231994990366763878637952051989915 4373e-02,
 2.64832730767816791554239756347974911967376828699013605157 7167e-03,
-1.70412245736066896923419674340761517909952920611869304474 1086e-02,
 1.73788033272051116443002782434535480161137341926459006809 7416e-03,
 8.46972549356075228777296166110471079130649637335423712699 8903e-03,
-2.55080712778947265914507224772473563718359094251185825535 4005e-03,
-3.47379898968110063064979025507623397095772166682019562059 8374e-03,
```

```
 1.87712092572365013317933815434487347723056734066854801635 8e-03,
 1.08705394222606296673894439784449841794552363005341114818 2206e-03,
-1.00077832708568054105569670776006287092589701453034826279 4137e-03,
-2.00071136307677980829630111079602647016311020284889474431 6755e-04,
 4.11128345474276703342474054300404150005488966066536749012 9376e-04,
-2.29201804121449989738229827143808457706517023610385918113 4525e-05,
-1.29304484008072060916146693967822685244047531274471437949 9074e-04,
 3.64502606856277496766546421660275076169098483080553417855 7146e-05,
 2.91334475016904121849578725192957101577543696765294538621 7480e-05,
-1.65732839530661628986339638785488051297686140987069002969 5161e-05,
-3.59364480402518763806691518973195045003462939252254296247 7168e-06,
 4.75060924645255285019711756475936319495351831742840024162 9683e-06,
-3.02905459205281828647422829430714179205379169585505856329 9597e-07,
-8.97570175063628073451165194168181876789505228733247153751 0510e-07,
 2.63389838699769655390096770411147347536801961236892259939 4214e-07,
 9.38719741109586302648441060128487681229255486380065329231 8725e-08,
-6.28615692201078616676850325287059095316686773944810280439 2389e-08,
 1.07659190661919613738520197502878513960767031982126680356 6785e-09,
 7.76897885477006223889596463939132455161170129359405593534 6266e-09,
-1.89399538617198414777461107661894601133749879060903162669 7228e-09,
-3.42680086326308900181101227888986420055034256638640567689 3537e-10,
 2.40709945350934296239981199192933072518662658289109046223 9366e-10,
-2.94058925076453258288847397463827366424468254129783598630 6504e-11,
-7.83250973362781703235655658281949479488413143381084884470 9881e-12,
 3.15276241337031042379753987689386162141838202466870449262 0948e-12,
-4.28565487006834410189818507337630768687538625954118096734 7399e-13,
 2.21919131158830296093466170006802372773781291800601101918 4982e-14};

const double Daub30[60] = {
 2.33861617273142147147440727989489196001166114635658042540 0538e-05,
 4.66637950428550933666200011105536514084898756388219903532 2085e-04,
 4.30079716504806951004501675740282740849348297478228696650 0398e-03,
 2.41308326715883789519491998795831194397672500511356126233 4092e-02,
 9.12383040670157067932157555508589970856450019108075159564 2650e-02,
 2.42020670940214099446759965834291951231819403268789843622 9538e-01,
 4.50487821853317836698135180289833641531494437574069950655 4771e-01,
 5.57572232912836430407808252099985041349257164575478537462 9734e-01,
 3.66242683371627979314487115136908953301629923499258474162 9624e-01,
-6.61836707759373150190974104181372647491121254447489544139 5148e-02,
-3.32966975020855606919684932059885050587749456126861350639 2514e-01,
-1.41968513330082931021902626740375825495427060282502011148 3505e-01,
 1.99462121580664303242899006211123022352322608813136432877 4921e-01,
 1.77829873244836736128025092133042504626028970097117675036 2566e-01,
-1.14558219432707781489151877861367224340495754911439374917 3137e-01,
-1.57236817959993812687819737888650155325171191061767339812 4611e-01,
 7.27786589703644269989354432660524423524871380455671560441 6632e-02,
 1.22747746045009377869157879769815009162435336524821290732 5446e-01,
-5.38064654582570767602201505183730430033864598461563923793 0800e-02,
-8.76586900363836604802657205369902835384698230485134247989 3827e-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,
 6.196717564977244383592534999284255315694546230739551683085460e-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,
 7.252145535890469015723401169934327900622894130695550273452916e-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,
 1.543997570847620046003616417646988780670333040868954794039905e-13,
-7.737942630954405708679963277418806436871098329050829841696327e-15};

const double Daub31[62] = {
 1.648013386456140748122177817418358316441195236228590958603489e-05,
 3.394122037769956699157160165352942212213928231154233571163033e-04,
 3.236884068627721218296626722969122583381316688100671696308l3e-03,
 1.885369161298591269159568944275763468999829139547989648553486e-02,
 7.433609301164788697908776495388047669378919816041031344650271e-02,
 2.070128744852353286198055444111916450619762837756134323019573e-01,
 4.091922000374278563928213235836188963704298775635493549519369e-01,
```

```
 5.51139840914275498359048457707466313207499226388681032442161 7e-01,
 4.29468808206137295543041314879900835457340853841433131223664 5e-01,
 2.71692124973694642230535473263426187340167909209599282719830 8e-02,
-3.10955118319507518692656028581100471539867822933352263420200 8e-01,
-2.17978485523563352169354450722010563163954743590311274713393 4e-01,
 1.40178288765273268165625320699307389542288151138015263344109 6e-01,
 2.24966711473737093369729790506688607830749013641530262401833 0e-01,
-4.99263491604682397700057939973013869307454390323409279793648 4e-02,
-1.86962360895715449437457719625838300920865507618765384707916 7e-01,
 1.54369884294889340965299533528123623184529354857116688321902 3e-02,
 1.45089500931993198151894290785487905912887287311692150415667 4e-01,
-8.13983227346923686352770871556658855000668054915234484014685 1e-03,
-1.07612773323495632666860551164801395238030195359044710607561 4e-01,
 1.09412974523649692572523790063780266950483574355546681179636 9e-02,
 7.53536117432814069552828975110913394137670198441945263868622 6e-02,
-1.48800266181048220269955598750342928910080197991004691325730 6e-02,
-4.86190754648543300353760338583119010939126354204451604887111 3e-02,
 1.61541715659859111361945386458670166563586916619386565196059 1e-02,
 2.80476193667561690686192721165915497704939228147911376469778 5e-02,
-1.42762752777635194330980014075674608721501619477557907059900 4e-02,
-1.39005529392665288075589888893444767173237351902867020112481 6e-02,
 1.05176394873718408912863344124499164333103382510203190885865 2e-02,
 5.51616357331099256656128976224116021447662266276463718181655 0e-03,
-6.52085237587461255332546968262853007921029377454113138175169 5e-03,
-1.42826422321890989140051603868784229217721129229504923892106 8e-03,
 3.39306677671593192841935879696061241109734741979235589691554 6e-03,
-6.39790110601460049288120231430729007799297275501649406287520 1e-05,
-1.45904174198516094311451522159808022384523925519005562190168 1e-03,
 3.43139829690473443811840108492950591220822968462985753000914 7e-04,
 4.99881617563722261489691240667951323196672244003279902497950 2e-04,
-2.39658346940294961528564668806947614026078170800617491253566 0e-04,
-1.24341161725022866940917980738339919987964117799345358880772 6e-04,
 1.08958435041676688273865183375263420635844130888086918441667 0e-04,
 1.50133572744453299707165193763098344275829768808771152144122 9e-05,
-3.63125515786008616426131377317216299110734869808316448916583 7e-05,
 4.03452023518427883975274149954609877899392634483173607440976 5e-06,
 8.79530134269298776544061803067834942736702258121185585745822 0e-06,
-3.03514236589150963006900785294705722076088721524950351278302 3e-06,
-1.36906023094294078205048975198712395507440478217716347127928 5e-06,
 9.81001542204437157395097608805806438494614618811090532167380 2e-07,
 5.32725065697491542697744095978308059377601213006317068830912 7e-08,
-1.97592512917020624815212115669659050130380318723192851386704 6e-07,
 3.61682651733100480524756721840579859132978812233727495617231 5e-08,
 2.32830971382140964430853888858932992114194853967810668077708 2e-08,
-1.06152960215025230650040426615082396240267378048496553827054 1e-08,
-6.47431168795986139870258153934195443874792625567160565709580 7e-10,
 1.40856815102517742707654780494458530133208710812572781319437 4e-09,
-2.52404395415335330618364370293321830861797946718484845656583 7e-10,
-7.34893003248626390476691391965362437958648743791517510640734 8e-11,
```

```
 3.69210880887112941160418919625967764044091936947826372889960 2e-11,
-3.32700896712597992991063624633715085164207979487111604118727 9e-12,
-1.32433491724396316387827434560946571729442662805346015184370 5e-12,
 4.44546709629193216329841185209301145962603756043917891761159 2e-13,
-5.55944205057901433764137573008353452151381816482755676375654 3e-14,
 2.69938287976266564729549392880138717392131457659850550785550 4e-15};

const double Daub32[64] = {
 1.16146330213501488556746410076065933295143142012104899630559 1e-05,
 2.46656690638090335273910421127466713447016944388644912467399 6e-04,
 2.43126191957226610078042307190595812781196967805597148806057 4e-03,
 1.46810463814191356354780900640219483110766200134342189348808 6e-02,
 6.02574991203353708174545197552796703185167738407899726192002 4e-02,
 1.75750783639438898818929991575334850520837639965186466139758 8e-01,
 3.67509628597349636199534033914323412520607956040686859596802 5e-01,
 5.34317919340953832290111785855218642552977470029058749592167 9e-01,
 4.77809163733948403355513081441479413035405375367550928793474 1e-01,
 1.20630538265617826953809871066526129939150730834201378889122 2e-01,
-2.66698181476675553548978408786986502422654260553408037150740 5e-01,
-2.77421581558427215333815332030340166668129450614329196765566 6e-01,
 6.47133548055162383100009009516766491844865915772015532156081 1e-02,
 2.48310642356880173606485215722286758879189817011410130099976 0e-01,
 2.46624448396974044170147933480872321480261493808125892063530 2e-02,
-1.92102344708546898434136527824799052586312389114778342606899 0e-01,
-4.89951171846717385335594322557637741839428015694598689941747 5e-02,
 1.45232079475286646083883074405194483232699834205314842631234 1e-01,
 4.44049081999397402264061953404660357108653154446842151914362 9e-02,
-1.09456113116089383102772277434326923275517113062389004161942 0e-01,
-2.96278725084477049120445237905121550504906864555107077936784 3e-02,
 8.08741406384839574409083159042632769081885467183642327541281 3e-02,
 1.41061515161066077286973880293174015027526938246379903101390 5e-02,
-5.69263140624784355047841627115853796055527009795333056765236 4e-02,
-2.38026446493257383444317836208650384732813499459195413587978 9e-03,
 3.70514579235446801043763345801303089801549690560942400445095 3e-02,
-4.14590766082721878146070042886261106126732810865364965363427 6e-03,
-2.16628228363911934763477851694748559859902936751803386960170 2e-02,
 6.16752731068567511257905968952010500474436728241292173981116 4e-03,
 1.10174007154068811653280611956434571247305176907971240790864 8e-02,
-5.41156825727579120858150241075238305060004594227564768536137 0e-03,
-4.64921675118441152865809498450490017298919012890588760254139 6e-03,
 3.62722464068786496012212298439170478234354838537532126025198 8e-03,
 1.46895510046846777252881178284048063916658282257719107926054 3e-03,
-1.96474055582177825418364754065674645009272585812659598490730 4e-03,
-2.21167872957909791627809758691495683419674913861040310277271 0e-04,
 8.67305851845055534392566238956353989059654965568338628779962 4e-04,
-1.02453731060739618694965679681297206229079612291593035663412 2e-04,
-3.05965442382691175047926116155257450073909133212150463442257 7e-04,
 1.05391546173982811470090519209110414107608360268637441014660 3e-04,
 8.10367832913483838982809189633415622422782136249162604495042 8e-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,
 4.384387799404743695532369498484275796871474868920335879980023e-10,
-5.881091462634605628881794361152305108432139465417759716875076e-10,
 8.904723796221605490455387579189371137903330749397374037644960e-11,
 3.263270741332907875981844980104948375955551273115386408552080e-11,
-1.430918765169202320188022211739750594608742928641485026836608e-11,
 1.075610653501062115165734990153347111902874668945095034791947e-12,
 5.361482229611801638107331379599434078296259332654994508124989e-13,
-1.663800489433402369889818192962259823988673359967722467427927e-13,
 2.000715303810524954375796020597627467104635766752154321244151e-14,
-9.421019139535078421314655362291088223782497046057523323473331e-16};

const double Daub33[66] = {
 8.186358314175091939858945975190102731733968885547217619434602e-06,
 1.791016153702791479424389068736094134247294413108336017758506e-04,
 1.822709435164084208084617771787691709255513374281497713580568e-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,
 1.219678564037346149389134584371009777591763921148126952722200e-01,
 9.478808805061595889263191779090571160237408179346345390888721e-02,
-9.114696835133148913093153757138373418923462847746880902676089e-02,
-7.030248505405615921453280814171665167171986608963193275084895e-02,
 7.019114394099653254998935842432841393915841096633514680190145e-02,
 4.573456189389667743139040427641638967843459421665709740086516e-02,
-5.347125133582228919431110824663168583260050383336359554980188e-02,
```

```
-2.52485829774764992925839220783772479393772734617729468470037 8e-02,
 3.86870607602449648174867503185252804730332381625015079309183 2e-02,
 1.07032658200195494265453496813772776969816885318607188873631 1e-02,
-2.57287617547329733612321139227830187568776083771020457962826 5e-02,
-2.16775861735360732478329865717283020389643384841806162243672 7e-03,
 1.53169541158576654834744226643187406022930478719158943096753 8e-02,
-1.59428878241460476863785644611139272405983693445518983750024 4e-03,
-7.95354038705793924045930540653811622067849524030259267758277 3e-03,
 2.38906240816590857593581597343972898815183609475368996610840 5e-03,
 3.48080095340571199941146100242922738593794225477852425743627 8e-03,
-1.86071821445579591207448215071056782431722820389700012972996 7e-03,
-1.20430925760465887691664498009732737289200858604709571963682 9e-03,
 1.07438069635129135507389923494171908047387702059520919770665 1e-03,
 2.72730584733693721174928235835019646173359529056954004581732 9e-04,
-4.90832900759035147448779225406654068372494875738210465249745 8e-04,
 4.39316625176618575505900529695812984409406352432471817525467 3e-06,
 1.78043189825124535183172802320006958692851366138262211696999 2e-04,
-4.16043851627370930623436880793393236056778769291888311888373 6e-05,
-4.92956442341730183431023148262157412740995092158306255948368 6e-05,
 2.42333539881689036562118837992204104607380881918202402658977 0e-05,
 9.07080575782845380020367746492150817846825668543821181857504 0e-06,
-8.86612136675773616917603443236429813418692909827465102282076 0e-06,
-3.60751610287977163123035111859506933019615545910558934286662 5e-07,
 2.28837127614152730548139554599376301056596866757776816420179 2e-06,
-4.42692340795287014798400212934180918562276835398355067075510 6e-07,
-3.98579129198594407694262651173922075316938746098429001918551 4e-07,
 1.82244333257105343746712899800279823396911223655321529163930 3e-07,
 3.37797270373085437751620666348186909937615425989721278414477 9e-08,
-3.98783819851888072281950285081493636919738439256197031934966 3e-08,
 3.67286357683818134050556375937916909971771264528344877939032 0e-09,
 5.11121185734745383954936659399875889113092102837457621325602 7e-09,
-1.67139267725193249517321961410441184189154560152178455979301 2e-09,
-2.49640210524619364807351926937019733117640537153840429874501 3e-10,
 2.42683310230568230989130288336123229766409948551460179034427 9e-10,
-3.04957445394586343036129693145514150012817015164320693754792 8e-11,
-1.42023685988993679243707784494041274934322564448777084054329 0e-11,
 5.50941472076552454875267363119771444781874098592908106490752 4e-12,
-3.34348121895327876598253272268998472517075819356617456649219 9e-13,
-2.15248838683330261852060354568599475332947827580599373709521 4e-13,
 6.21474024717439831557621469957723069302130785467355721465275 1e-14,
-7.19651054536332241403365447077907059231660078069755836108315 1e-15,
 3.28937367841630636862556410878209564403641540190251881297879 8e-16};

const double Daub34[68] = {
 5.77051063273028562746606779680932911732470891904790081773802 5e-06,
 1.29947620067953003783348481539056940036943265820772272040508 4e-04,
 1.36406139005904999820001444939687743959168043561083736941133 9e-03,
 8.81988940388497880318276456309587933533097793954163086280475 7e-03,
 3.90488413517859413890502621959156920404381657794151701963191 6e-02,
```

```
 1.24152482111376808195444989821096917270819967242863537805128 5e-01,
 2.87765059233714562933425661808771887255856012099965127799183 9e-01,
 4.78478746279371062146861070612051946626801032903134584333610 4e-01,
 5.30555099656463177313326022399079444560569903050365238279560 0e-01,
 2.90366329507274951045594518619953011575566497793456412882265 0e-01,
-1.28246842174437167291237774704855842761277493294374862865082 4e-01,
-3.31525301508386941771554846308753734503582888642634539725687 6e-01,
-1.03891915515640471828726050692586797059644861864700669838859 6e-01,
 2.16907220187427595061001866709932246561940803025653419781978 4e-01,
 1.66601750412207443731157433450926136668299370057348853457789 0e-01,
-1.27337358223801156284386263698869389010879362996654169580724 7e-01,
-1.60924927177866806301479949042964919661462885726738297695860 7e-01,
 7.79918469379481073826534953183201508709688227733396847372639 9e-02,
 1.34125960271136128480239991397738799935828090070858246262553 9e-01,
-5.44829680641390463663267138314064255426586594868615727101728 6e-02,
-1.02947596992814085234207382368909049824549605684547356906666 7e-01,
 4.35760946496312972642848661092580072713772413637066942124660 9e-02,
 7.31852354367956055554622133545204568075799894749388312493456 7e-02,
-3.70128384178624496035640212555419004075007900912746165578492 7e-02,
-4.74385596452777624722068141098385137788975601871642735800829 6e-02,
 3.07397465739593445993122651384413434630556292846699320816460 3e-02,
 2.72283507563541961009583989580585885520274589771811773149653 4e-02,
-2.36717379228263648504678643809494042745607952804355556686711 0e-02,
-1.31439800166571608610582750612628704134268057840400735943961 2e-02,
 1.64093741998651925211226149553740959236315630987447331005747 1e-02,
 4.71364926099980990591887612543748885623587402707775500453920 5e-03,
-1.00455067083615191743914686114643100036485840118133713489142 1e-02,
-6.19474884515387283901435662183550185732234544523480934743109 8e-04,
 5.33495076875993603217027019598392151156553910079190695290139 8e-03,
-7.69212797506783697598949090056102984488728533580434947499360 7e-04,
-2.39945394353705586393312482768808195270178059988306756050187 0e-03,
 8.58995987436366195544489847574653658349752210745929171890005 8e-04,
 8.75199906407868873261057005522433973376030477332722847625564 7e-04,
-5.52735576214419797551641529673512446055063228376368835964988 8e-04,
-2.32673214023353163542886321283394224559736108570856752823073 3e-04,
 2.65077239755805781975581130907100254382214566093301695773593 7e-04,
 2.66005001845344190304682846802558908640312618079846434780167 8e-05,
-9.91469777078013460358035075886937847180275183760846197102256 7e-05,
 1.35311722724964958125188737641448622512734635204220914131556 2e-05,
 2.84495141969780737650308000194376593060124222518389365854003 2e-05,
-1.05765749425795062384831630475521812023325347931757433740962 2e-05,
-5.71082651099830393827505007433340030551245141998364659176231 8e-06,
 4.16987175854702839831676165998492880436202364362974135879974 4e-06,
 4.97971810142130774808185763647176105742921926553161860296014 7e-07,
-1.11630653481700842859799507075176508038326165811265694852695 4e-06,
 1.44819570833318512706118061815000952675865864123110490170356 1e-07,
 2.02599066666785921669053688569372554534493323543230764920549 7e-07,
-7.52670174041258941117748179784104428166255578596941539836901 9e-08,
-1.99034650153173691586618044833761496757074421115824151458912 1e-08,
```

```
 1.74042333293606807649705127444514716019078384785440983648966 2e-08,
-8.66574426136872221586474116624538588881856757114595853193693 9e-10,
-2.31650194699548275158229424013601006741508449902575311794100 1e-09,
 6.44637821032340231310121489450023118160652021157958113244254 8e-10,
 1.30041031860941524888040325930046772063118912097892837715223 3e-10,
-9.90477453763240901547953033397912454018319917459137776284522 7e-11,
 1.00420873546176986483651642899830677803114365010184236162233 0e-11,
 6.08012535400016725405902592991559129111575173428858456313163 6e-12,
-2.10787910891530154628537039544377886467627523512604459968327 1e-12,
 9.79945115821159772790117852052638869214058604116362425299180 5e-14,
 8.57919405179973317979311229865260051148658121652868348214310 6e-14,
-2.31708370390640848107825708190308952323402042309217526192551 5e-14,
 2.58733838193569955581353816314498668883414257120715287914473 1e-15,
-1.14894475448059012824481579431260624514788815801882349093628 0e-16};

const double Daub35[70] = {
 4.06793406114855902666524711020608457105120147712197261221800 5e-06,
 9.42146947557674063160302753311663022445104973605090336145875 9e-05,
 1.01912268037509810931931467275148508020255760746719921377808 5e-03,
 6.80729288431913201197133397901562511349405064279739781762532 6e-03,
 3.12362885114907145306339121076935306818708899949589325705117 9e-02,
 1.03404455861478378993878775492927918398555332279606351704914 0e-01,
 2.51307378994493312851325197148890504286677976101474019281690 2e-01,
 4.43592739224035437818391048944849459478203903280795629482610 5e-01,
 5.37008427509166102867069023171697454758003493236105360772388 7e-01,
 3.60345640518047327874445857398871842253811421789079227062156 3e-01,
-4.38838818739340411134347939409722431210034901193202886509862 5e-02,
-3.23822864912116121214730280799317671562548032723551253059316 0e-01,
-1.81786976766727832578835026452819167684149336946084912353861 6e-01,
 1.66041357490780919543843332747094794053809791452529806447778 5e-01,
 2.17299289321089297767549345619955911403632635851767210697295 6e-01,
-6.52628713106775389215489591133110828400738073886565242030423 3e-02,
-1.91919589298593952876078680079863619851649595792479882050087 6e-01,
 1.93095446660183509194773458593810994464743524348496705777511 0e-02,
 1.55292480396237114420675376071256699398731937896523118647763 0e-01,
-4.75268083411135044528811099803097914371086468904190216711911 8e-03,
-1.20585522643393554507658948070495772263532445681232215043798 9e-01,
 4.73422917264194876329398031499221329397177069548061678982838 4e-03,
 8.99135475707295441786537419526196298364404899821823390048185 6e-02,
-9.31855894990392483787500282361750422724656215267189457950437 8e-03,
-6.33560374404434661209888753402054570573167171805796480200667 1e-02,
 1.32285495850365552445592984760511071964874689049735680828930 2e-02,
 4.12546930647050921274975081429912665615150480584541799465141 7e-02,
-1.43668397842200718210402517321401279778890489429171637349352 5e-02,
-2.41694978016602674029488068173108409126453316881674622753703 0e-02,
 1.27664567156567441940391801874243271497365659822793982494003 5e-02,
 1.22894360081187108616196762581429705061110020002389837794915 1e-02,
-9.57779789235709998147309703713518608283233882793489733491642 e-03,
-5.08599164923342988179763658357892119467539380776115454973354 7e-03,
```

```
 6.13775458674052108959680188363192122114571254504251998764 1234e-03,
 1.42808879407076210735558587066984213260915904062589509007 0111e-03,
-3.35764438092238322956773256529866563903734858596112707550 7937e-03,
 7.61596943517273654676964992389531745153470306601611625730 0160e-06,
 1.54963746970236297556171924653978771720443863799782493578 7688e-03,
-3.34669216425085496160852612152459690804110991836130628220 1310e-04,
-5.86481031899181753217580922413145673836710103569418822340 8841e-04,
 2.64832881996128903930281012269971096604856536804757521869 3134e-04,
 1.70001228366124904358469019471676777120420774262574630852 2935e-04,
-1.36588307226116160255992671474474642256750917744359404570 9653e-04,
-2.97699596284850974394422586648851966858524265598065664654 4319e-05,
 5.30414312291331022253831798068637469600560553347568558748 6683e-05,
-2.43700152682778986099042947854055675269438969343266883107 3769e-06,
-1.57244207727028169366328896640586121569280597273798198612 1447e-05,
 4.30804786171673119135049343793751322073745041013287803216 3179e-06,
 3.35334586287130988939087716804613365737710568161870835526 6688e-06,
-1.89592961769315328849389105187544443975331854810599816657 4535e-06,
-3.90393173328730616665751946849451192076076738839782577532 6745e-07,
 5.30236861690476091707435263391574325076960063582922960081 2520e-07,
-3.70030837820512453798640264491887914989403591010648908251 2364e-08,
-9.99039694453490075578172847756124076219144342231824912886 6740e-08,
 3.00818865071906692823026891866171827450495504502255021705 1301e-08,
 1.08490273378993482526656024010044988470274930332657174732 3086e-08,
-7.45811655289303763119240761126278859350598863836584040936 7117e-09,
 5.89795131038436157547035586116202250117249193783771296986 5619e-11,
 1.03082334548543338381170048148855742200521016806916377973 0908e-09,
-2.43354557375167293616887725040594081722736793723028980125 1648e-10,
-6.40793825650188901843060832323597440621919317691828466497 3727e-11,
 4.00053662725374451074278820135409300647171041667100224430 2586e-11,
-3.12563935710855754059809822867815076852812156539137626562 7294e-12,
-2.56706547615508144920464385242840153028351968563825607475 2850e-12,
 8.01508853368790092194860541878932482611561641634339108128 8979e-13,
-2.59795432889384808431519820509438914570668012920899863880 2995e-14,
-3.39772085679626743195678382565906959694033513010087191232 9556e-14,
 8.62403743472008920268033766369277682810714650060805832406 135e-15,
-9.29801252932418542092155566471986350184831509911672518437 0339e-16,
 4.01462871233348865431856916461422030804602109117818465425 0982e-17};

const double Daub36[72] = {
 2.86792518275594633463047947302923861553551177589426271105 4705e-06,
 6.82602867854635869174862910220960536224034426650503598179 1715e-05,
 7.60215109966848828586979267710608210014127505489238937919 8545e-04,
 5.24029737740988436620160352439299569604217493719443523500 3941e-03,
 2.48905656448279648488592733395911557940302334704472973925 5255e-02,
 8.56520925952640908386471699552111148643759475037785652477 2704e-02,
 2.17756953097900814963794591571999974624896970565062553341 5876e-01,
 4.06433697708255346740779399025038444590315163076855814212 5382e-01,
 5.32266895260728691477744474864146202721355472315390690112 9337e-01,
 4.17875335600969786362063455937423645522227530299693117826 5919e-01,
```

```
 4.39751975293486299386218289835876378311074555923898217969013 2e-02,
-2.94421039589114571110071596989875894072245888737784463344367 5e-01,
-2.46807036978125527052479827862269844656652071823031388908601 6e-01,
 9.81142041631147705051840137140156803894343732229991351404972 8e-02,
 2.46537277608974211052970911180959543465641876289815270662135 6e-01,
 7.27851509579222900968768229946038287864313902666895888442964 1e-03,
-1.99337205608649619860336340009478414271416225679218257054103 6e-01,
-4.58614007463927163914512622877483174300297137399832960457439 4e-02,
 1.54106236627642884177631630042065487588384281941362339535826 2e-01,
 5.02761800735384286203681697280988409676170603601974831689091 3e-02,
-1.18803754310135631680181693138354744607315295104444422444950 1e-01,
-3.98808535755131758409169996792404403410037425707586426093410 2e-02,
 9.11567822580165440633605928130671515105890305537052203184377 1e-02,
 2.50387214495684898991948429670984686056918099304038362198054 6e-02,
-6.82090166368175112488043634426553869058035810871454076312511 9e-02,
-1.13191003168174279438180808217369502212305628082161135457788 3e-02,
 4.85130835478090853861626766231573563229298974901326120704636 7e-02,
 1.42497266176539160314780260737854239632342965766000975565240 4e-03,
-3.19807206776396965447029351374234460117273968827425164187377 8e-02,
 3.98404019871700485739717948679008232131429136665615121342906 8e-03,
 1.90635947806253593287757616436819827485810851369683272888920 9e-02,
-5.65781324505881838042401697351671457049916143497576179837902 0e-03,
-9.99026347328137234800174380648917266546568505697565249750377 2e-03,
 5.02298910666582900469981922079653883039394599468728979246554 1e-03,
 4.41348483535057525191861678028777558547101255684803730102599 9e-03,
-3.48454144540488331120954139542853573269766197181872728600302 8e-03,
-1.50307406629664374954936365536341187985807020274081405496460 3e-03,
 1.99079377185173727040429324570187818660089943951347582330591 4e-03,
 2.77681279571202606815238420760514038349024275692193650194038 9e-04,
-9.46340382326110196460491805944791304772548213006349224277987 8e-04,
 8.61456575899270203261387915940233090963473720457860639940310 7e-05,
 3.69350728496751050262004034188223668774956341443343284256751 1e-04,
-1.15511889584352709684837699941310239519197635093666657381879 9e-04,
-1.13189946808466567172739192292441146793845074356510697809945 6e-04,
 6.69474119693059025710423174928378625155556677339819999033769 8e-05,
 2.37510668366086077716195083238034136225750376149058089661767 8e-05,
-2.73139082465433791292234641472204540477993582583438425002319 2e-05,
-1.18347105998561594278318276235236091730434803494741298660832 2e-06,
 8.37221819816078843262805604321749155219885735843211227525331 0e-06,
-1.58614578243457749550261463156621183972287949282791179070949 8e-06,
-1.87081160285918071376297228115495352805625745190038109747696 8e-06,
 8.31142127970777852816359740593537588685502959215042454450071 8e-07,
 2.54842352255657783121851905284438747881986653190285452354470 9e-07,
-2.45537765843423269913587828679457851538713819424769320184626 3e-07,
 2.75324907333951225408507645670024192949272045788907605845107 2e-09,
 4.79904346545099200993452686765049768354571685860611978632755 9e-08,
-1.15609368881700840675691394917520845208376536882544248222609 3e-08,
-5.61278434332779139747411435709436855798241389580298081481336 9e-09,
 3.13884169578242401835156795215841500357138069923614775223900 1e-09,
```

```
 1.09081555371375181096471305880044867606847567361134956640571 6e-10,
-4.51254577856324963442520085608849019500407780606297806779602 0e-10,
 8.96241820385961198706596832029592967977469346579136761004477 3e-11,
 3.03742909811253522180001360957629719606178692773455663569641 6e-11,
-1.59971668926135714320039692240944851539864848979504446804642 0e-11,
 8.87684628721737421352439968289556405594988605074832181841116 1e-13,
 1.07096935711401700242443347162119757905992726172784637596837 8e-12,
-3.02928502697487726889613458976947385466975879744679575732986 2e-13,
 5.54226318263980423523168586102899515869439722390729526918033 6e-15,
 1.33807138629910589602557876145847295529476331076637117836378 3e-14,
-3.20462854340174986043931663884857971178917644432013435525375 0e-15,
 3.33997198481869321313257877771250367001445941116783921149523 7e-16,
-1.40327417537319061748982320916801392256435349544348743124261 0e-17};

const double Daub37[74] = {
 2.02206086249839212181503833533633351464174415618614893795880e-06,
 4.94234375062813200471428611743445449948573794779139786719591 0e-05,
 5.66241837706672401376839437324943916351865484049360357514473 7e-04,
 4.02414036825728677070214012489377244795225684247889154809270 3e-03,
 1.97622861538795915324405550220501746153858947570561841489689 3e-02,
 7.05848259771816083203036189079300765996348392531213274186867 1e-02,
 1.87326331862064944802884349174760157676190165688828883819202 3e-01,
 3.68440972400306140944583861696494113267028772475472942520404 7e-01,
 5.18167040855622887310451966753443720538710957926571807117417 8e-01,
 4.62207553661605714550544840152817207005076853450427869422936 3e-01,
 1.30878963233020172605770120101764960103438107089327558689807 5e-01,
-2.46180429761083413286901858114572071036543391458468069169371 7e-01,
-2.94375915262661772280821957593267373367429077223564469136742 7e-01,
 1.96715004523593897707776864874005238028815650722264718730189 4e-02,
 2.51523254360268693343522409507816629144292399261159382755271 0e-01,
 8.18060283872186233902907698265241169600004553371672602766214 7e-02,
-1.81962291778608000740882425652522521644444314386875261128426 0e-01,
-1.08451713823301784555407881234187656851483517634163978355854 3e-01,
 1.29929646959853752784252889525918865312060231862094450297972 6e-01,
 1.01780296838814179747094822850586561748004828798317658160796 4e-01,
-9.66075406166843903091540504595577271598858537477128229131549 6e-02,
-8.23302119065574086740407366092037941498830249201878377470202 8e-02,
 7.50476199483601793357900507259424543507167445288214822858386 5e-02,
 5.95674108715299524543558904252010806687711476821627250368439 8e-02,
-5.92568156326589709515380672496592433407755517428143618951223 9e-02,
-3.82538294793842488201110888509044211680299419361188473813337 3e-02,
 4.58079441512683324663325615611038180584813815878473449698177 8e-02,
 2.09728005925975488331376946903639329446149774908392116235422 9e-02,
-3.35235840641009699435866287591324306723478629600923894992058 2e-02,
-8.83349389041023239406418799062556325710742910913072629152864 8e-03,
 2.26186515445994735657143165895880291206110560821282867532345 2e-02,
 1.69047238348442374366395285909070563651280716153695401840008 1e-03,
-1.37639819628947843385798548609707033978622513672806700059118 7e-02,
 1.51930577883339921848126184459950740856329510223596407654433 4e-03,
```

```
 7.387757452855583640107787619408806919082115520707105052944171e-03,
-2.248053187003824706127276829147166466869908326245810952521710e-03,
-3.394523276408398601988475786247462646314228994098320665709345e-03,
 1.816871343801423525477184531347879515909226877688306010517914e-03,
 1.263934258117477182626760951047019242187910977671449470318766e-03,
-1.111484865318630197259018233162929628309920117691177260742614e-03,
-3.280788470880198419407186455190899535706232295554613820907245e-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,
 4.854731396996411681769911684430785681028852413859386141424939e-07,
 1.002121399297177629772998172241869405763288457224082581829033e-06,
-3.494948603445727645895194867933547164628229076947330682199174e-07,
-1.509885388671583553484927666148474078148724554849968758642331e-07,
 1.109031232216439389999036327867142640916239658806376290861690e-07,
 5.350657515461434290618742656970344024396382191417247602674540e-09,
-2.252193836724805775389816424695618411834716065179297102428180e-08,
 4.224485706362419268050011630338101126995607958955688879525896e-09,
 2.793974465953982659829387370821677112004867350709951380622807e-09,
-1.297205001469435139867686007585972538983682739297235604327668e-09,
-1.031411129096974965677950646498153071722880698222864687038596e-10,
 1.946164894082315021308714557636277980079559327508927751052218e-10,
-3.203398244123241367987902201268363088933939831689591684670080e-11,
-1.398415715537641487959551682557483348661602836709278513081908e-11,
 6.334955440973913249611879065201632922100533284261000819747915e-12,
-2.096363194234800541614775742755555713279549381264881030843258e-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,
 1.563724934757215617277490102724080070486270026632620664785632e-02,
 5.788994361285925649727664279317241952513246287766481213301801e-02,
 1.600719935641106973482800861166599685169395465055048951307626e-01,
```

```
 3.30775781411014651149363753440461175480076867704157703075730 6e-01,
 4.96591175311718097659917114771870893935241483895172608756441 9e-01,
 4.93356078517100797572848534699731706496951362359435909111580 4e-01,
 2.13050571355578513828674335345856245125562466595116044512230 7e-01,
-1.82867667708335890797554850794623913521822318504141063292481 5e-01,
-3.21675637808997862848347172540691636192984194052818905900254 8e-01,
-6.22665060478243222664336016047876584756586210104559718031049 0e-02,
 2.32125963835353108502870810428599499867161556366285807926299 6e-01,
 1.49985119618717019958640345378892730729822602826260302863575 8e-01,
-1.41795685973059621671005314452233027639259105537583065451908 0e-01,
-1.59912565158244361828853321452353493780420884438610263917769 3e-01,
 8.56381215561510574161221781436916531348712964553600185027698 7e-02,
 1.41414734073382680088468311937917059409260617491575528349615 3e-01,
-5.65864586307273814568178765784332064681550941063511423494790 2e-02,
-1.14731170710744375239414401945894277971566548923016995020102 2e-01,
 4.30958954330476428813787122361603062424656868359540879207860 2e-02,
 8.72043982620397501191071416415445676207378612423308847185586 8e-02,
-3.66051034028742956737207103950677237256793871094343283890824 7e-02,
-6.17662087084131599360473670561324624189749778237337911398117e-02,
 3.19898775315378063081838113636685902613703545057663113417687 5e-02,
 4.00549811051159482095208708624111430903857737936673295964854 8e-02,
-2.68914938808945143855085176771596731341789039328723670007207 1e-02,
-2.31141340205493168085691355358562124892530386554020335718076 8e-02,
 2.09046452556552434021598236535134209467026149152683167268224 4e-02,
 1.12904972786859648427008148776154423285111589144984396715165 7e-02,
-1.47018820653986821370898640281660504564848122466243511408824 5e-02,
-4.13130665603108927412323110332674572318813454852093815799570 2e-03,
 9.21478503219718051203153487018173400352286164590389450430228 6e-03,
 5.62571574840353200574156559488114875706670343721452210174094 1e-04,
-5.07131450921834809393506141750566300200682132395875264964032 9e-03,
 7.16982182106401925778416536489491562188854149677337043588958 5e-04,
 2.40069778189097318389230691408259214398414055021013013953519 3e-03,
-8.44862666553777500906893785146585697325136301092400331464361 2e-04,
-9.42461407722737796401594227178009828391023063990801877858891 0e-04,
 5.81075975053286366202032106367819663340955570698147672398831 2e-04,
 2.81763925038067074601804896753560819012352318061296106260367 2e-04,
-3.03102046072661199360062902032978468249647710647042778774785 5e-04,
-4.55568269668420274688683005987764360677217149927938344795290e-05,
 1.26204335016617070538234653713181770136152238790491733595870 5e-04,
-1.15540910383371719262847904798346095338195934264237417582286 3e-05,
-4.17514164854039779729632506577571130919741192628941246828080 1e-05,
 1.33417614992135038254750345728606092221807003133013760142732 4e-05,
 1.03735918404559979563225833501006510352495984496609487021768 7e-05,
-6.45673042846961916037991043961757542098697239413712195380623 6e-06,
-1.55084435011860257585338014852591299940129247318553439574037 1e-06,
 2.14996026993966520778954819979077059689025240507639488560603 8e-06,
-8.48708758607259307186980526608942662960647987698222184083309 8e-08,
-5.18773373887414442600847468337854236806631000060282309600918 7e-07,
 1.39637754550835548122796158105996118451987250249346201026463 3e-07,
```

```
 8.4003510468959655269335871767812795079530806692593187229105 23e-08,
-4.8847579374592867620821854116087639640410103921019148549181 57e-08,
-5.4242748002872985111266841748544149284475217106644764109739 81e-09,
 1.0347045392748584809240464909528039373282395372229081594510 39e-08,
-1.4363294877951357068545398569792759111836284765216362516608 49e-09,
-1.3491977539834488218503817708897863012467413043079349559971 11e-09,
 5.2611325573575984945357666387726245721003322091989796590770 82e-10,
 6.7323364901893086857406269641826231597597675367248440301645 51e-11,
-8.2782565225381347273306929381589911153353846117958747675217 31e-11,
 1.1016929345994545511508326221602242312801953629194985409136 58e-11,
 6.2915373170395085815809136208591408358528863089895841981661 74e-12,
-2.4847892375636428570433612145027607236114685918332626758522 42e-12,
 2.6264965040652520704882828764705253798514295383894815764546 18e-14,
 1.8086612362745305822670848463439593770859220190678081456352 63e-13,
-4.2498178195714630069666163715542065728631225627449167965564 74e-14,
-4.5633971621273731091016916430479237477965634491940756218544 91e-16,
 2.0450996767889889078022725644023100953986410928193671672529 52e-15,
-4.4053070424834613424490271398383016110068352854550501558428 65e-16,
 4.3045968395587900162518674771227915088496976880581690531344 63e-17,
-1.7161524510887441887324042817379642777130260872242482355410 71e-18};
```

Constant-time exponent of 2 detector

References : Posted by Brent Lehman (mailbjl[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;
```

Conversions on a PowerPC

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
```

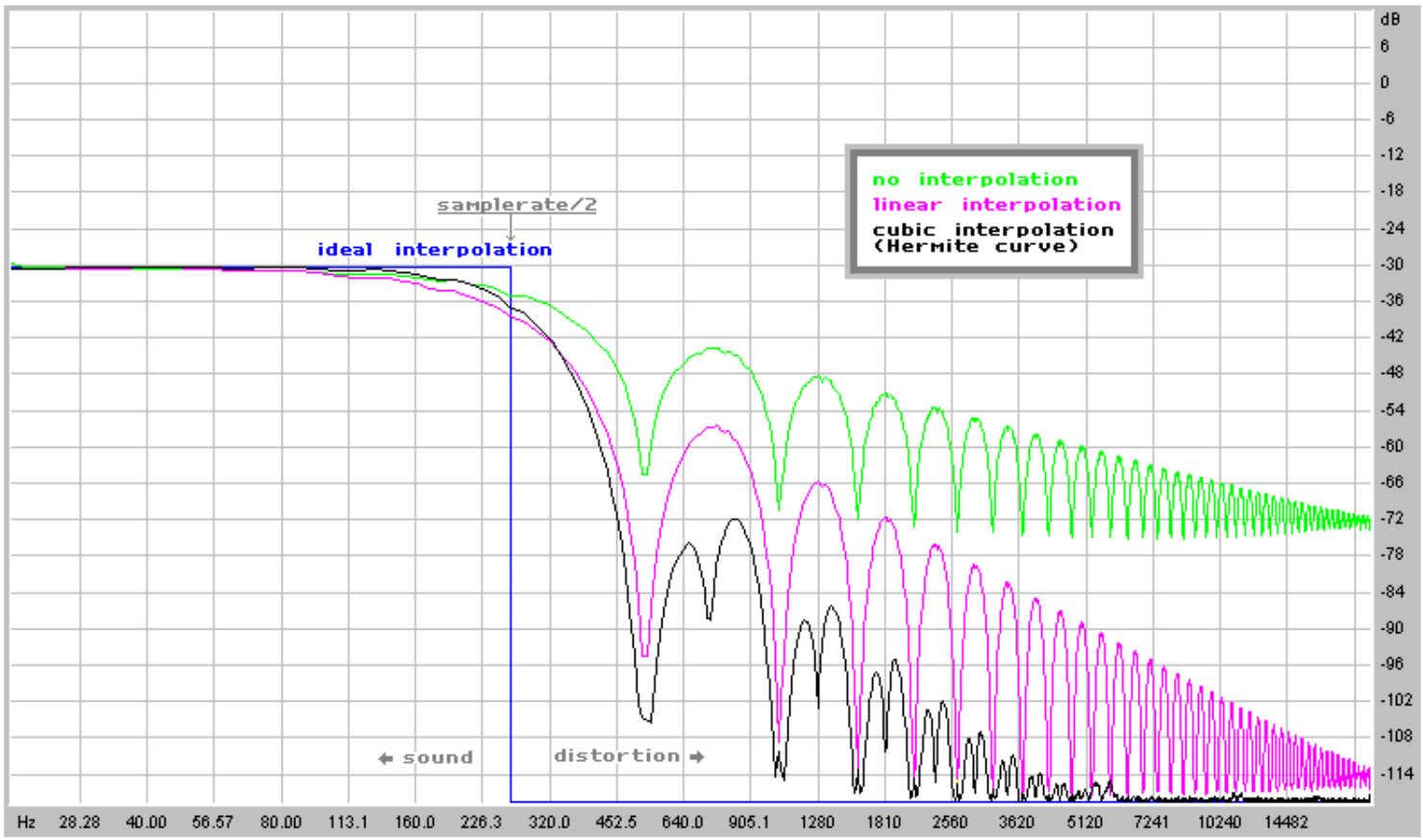
Cubic interpollation

Type : interpollation
References : Posted by Olli Niemitalo
Linked file : other001.gif (this linked file is included below)

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;
```

Linked files



Cubic polynomial envelopes

Type : envellope 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
}
```

Delay time calculation for reverberation

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 intial 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..
```

Denormal DOUBLE variables, macro

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;
}
```

Denormal numbers

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

---

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 occured, just trace your code and
look up on the STAT FPU register ... if 0x0002 flag is set then a denormal
operation occured (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 distrubuted 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 becomming 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

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 descibes how you can get rid of them too!

(see linked file)

DFT

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;
}
```

Digital RIAA equalization filter coefficients

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
```

Discrete Summation Formula (DSF)

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;
}
```

Dither code

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

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

Type : pointer cast (round to zero, or 'trunctate')
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 000000000h,042380000h
```

```
fld    AFloatingPoint Number
fadd   QWORD PTR __double2fixmagic
fstp   TEMP
movsx  eax,TEMP+2
```

Early echo's with image-mirror technique

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)

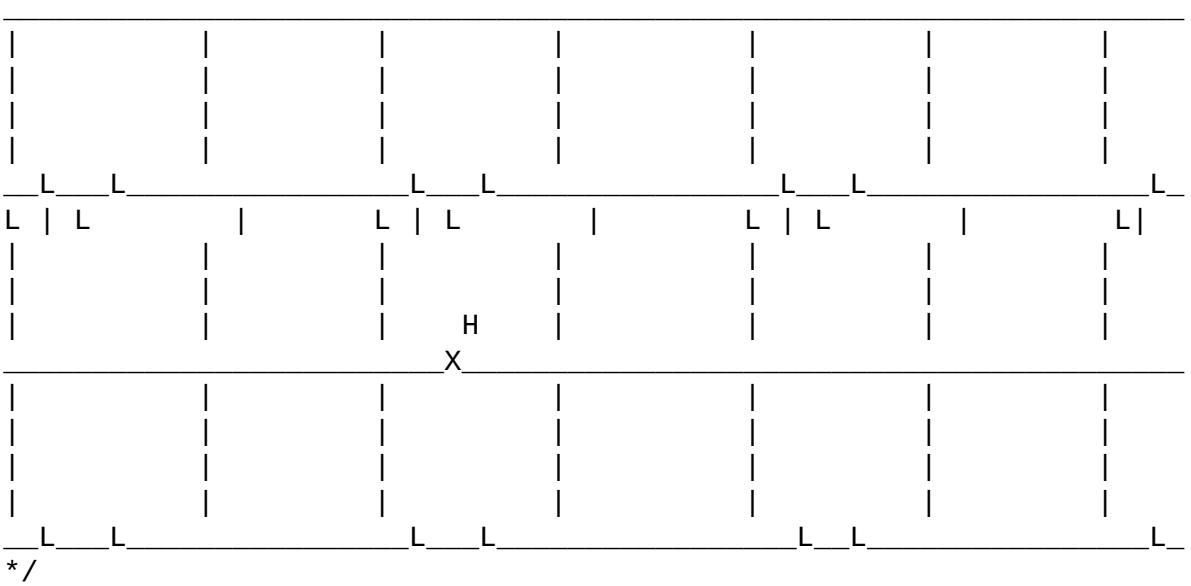
Notes :
(see linked files)
Donald Schulz's code for computing early echoes using the image-mirror method. There's an english and a german version.

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=?
```




```
*/

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 gemaess 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));
```

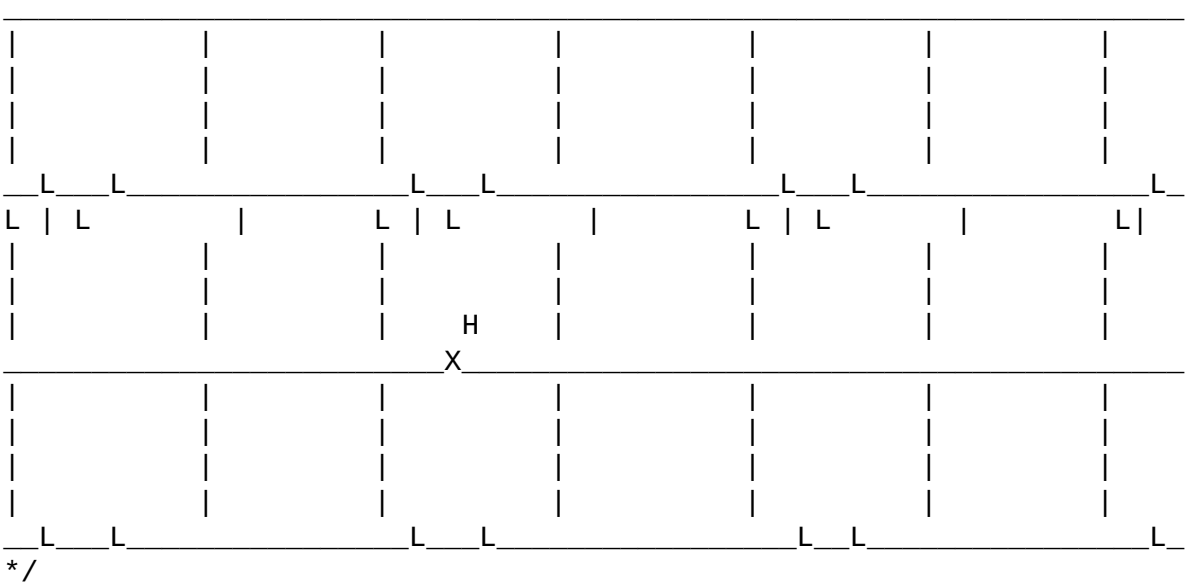
```
    /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHZ-> *106 */
    if ((distance*147)<early_length)
    {
      /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
      winkel=atan(y_pos/x_pos);
      if (y_pos>0)   /* Klang kommt aus vorderem Bereich: */
      {  /* 0=links=>Verstaerkung=1 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 gemaess 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];     /* Geweighten 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)
    {
      /* Einfallsangle 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)
    {
      /* Einfallsangle 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)
    {
      /* Einfallsangle 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)
      {
        /* Einfallsangle 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_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

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_l equ     $1000           ; max er delayline length 4096/48=85.3ms
dl_co_l equ     $1000           ; max co delayline length 85.3ms
dl_ap_l equ     $200            ; max ap delayline length 512/48=10.67ms

        org     p:$1000
dl_er1  dsm     dl_er_l ; er1 delayline
dl_er2  dsm     dl_er_l ; er2 delayline
dl_er3  dsm     dl_er_l ; er3 delayline
        org     y:$8000
dl_co1  dsm     dl_co_l ; co1 delayline
dl_co2  dsm     dl_co_l ; co2 delayline
dl_co3  dsm     dl_co_l ; co3 delayline
dl_co4  dsm     dl_co_l ; co4 delayline
dl_co5  dsm     dl_co_l ; co5 delayline
dl_co6  dsm     dl_co_l ; co6 delayline
        org     y:$F000
dl_ap1  dsm     dl_ap_l ; ap1 delayline
```

```
dl_ap2  dsm     dl_ap_l ; ap2 delayline
dl_ap3  dsm     dl_ap_l ; ap3 delayline

;---------------------------
; Memory usage
;---------------------------
; P:$0000 -- $0200      core file
; P:$0200 -- $0300      progam
;
; 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_l,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_l,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_l,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_l-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 ...
;       co1 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
```

Envelope detector

References : Posted by Bram

Notes :
Basicaly 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 ;)
}
```

Exponential parameter mapping

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;
}
```

Fast binary log approximations

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;
}
```

Fast exp2 approximation

References : Posted by Laurent de Soras

Notes :
Partial approximation of exp2 in fixed-point arithmetic. It is exactly :
[0 ; 1[-> [0.5 ; 1[
f : x |-> 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 in-place Walsh-Hadamard Transform

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)]);
            }
        }
    }
}
```

Fast log2

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);
}
```

Fast sine and cosine calculation

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]
```

Fast sine wave calculation

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
```

Fast square wave generator

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 propably of not much use for syntheis. 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;
```

FFT

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 assebly
     mov eax,i
         mov ecx,sizze
     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:
// intput 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,dl;
        double xr,yr,xi,yi,wr,wi,dr,di,ang;
        double *l1, *l2, *end, *ol2;

     astep=1;
         end=data+size+size;
     for(dl=size;dl>1;dl>>=1,astep+=astep){
                                          l1=data;
```

```
                                        l2=data+dl;
                        for(;l2<end;l1=l2,l2=l2+dl){
                                                        ol2=l2;
                              for(angle=0;l1<ol2;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:
    // intput 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,dl;
          double xr,yr,xi,yi,wr,wi,dr,di,ang;
          double *l1, *l2, *end, *ol2;

       astep=size>>1;
           end=data+size+size;
       for(dl=2;astep>0;dl+=dl,astep>>=1){
                                        l1=data;
                                        l2=data+dl;
                        for(;l2<end;l1=l2,l2=l2+dl){
                                                        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+i];im=data[i+i+1];
                    data[i+i]=data[j+j];data[i+i+1]=data[j+j+1];
                    data[j+j]=re;data[j+j+1]=im;
                         }
        }*/

l=size-1;
m=size>>1;
for (i=0,j=0; i<l ; i++){
          if (i<j){
                                re=data[j+j]; im=data[j+j+1];
                                data[j+j]=data[i+i]; data[j+j+1]=data[i+i+1];
                                data[i+i]=re; data[i+i+1]=im;
                                }
          k=m;
          while (k<=j){
                                j-=k;
```

```
                                k>>=1;
                                }
            j+=k;
        }
    }

    ////////////////////////////////////////////////////
    // used by realfft
    // 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 realfft
    // 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_dit_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(;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++;
                }
                //the remaining filterings
                j=1;
                while(l3<end){
                        l0=l3;l1=l0+dl2;l2=l0+dl;l3=l1+dl;
                        c=2*cos(p4*bruun_reverse(j,size));
                        for(i=0;i<dl2;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;
                        }
                 }
                     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

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)

Float to int

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
}
```

Float-to-int, coverting an array of floats

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 endianess confuses...
    mov     ecx,0x4b004b00    ; damn endianess 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. Unfortumately "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 ?
    jb      saturate
    pop     eax
    mov     [ebx],eax    ; restore end flag
    }
}
```

Formant filter

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;
    }
```

Gaussian dithering

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):

dither = (RND+RND+...+RND) / N.
           \           /
            \_________/
              N times

If your RND goes from 0 to 1, then this code is applicable:

dither = (RND+RND+...+RND - 0.5*N) / N.
```

Gaussian White noise

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));
}
```

Gaussian White Noise

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 );
```

Guitar feedback

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.
```

Hermite interpollation

References : Posted by various

Notes :
These are all different ways to do the same thing : hermite interpollation. 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);
}
```

Inverted parabolic envelope

Type : envellope 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;
}
```

Lock free fifo

References : Posted by Timo
Linked file : LockFreeFifo.h (this linked file is included below)

Notes :
Simple implementation of a lock free FIFO.

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;
};
```

Look ahead limiting

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);
}
```

Lowpass filter for parameter edge filtering

References : Olli Niemitalo
Linked file : filter001.gif (this linked file is included below)

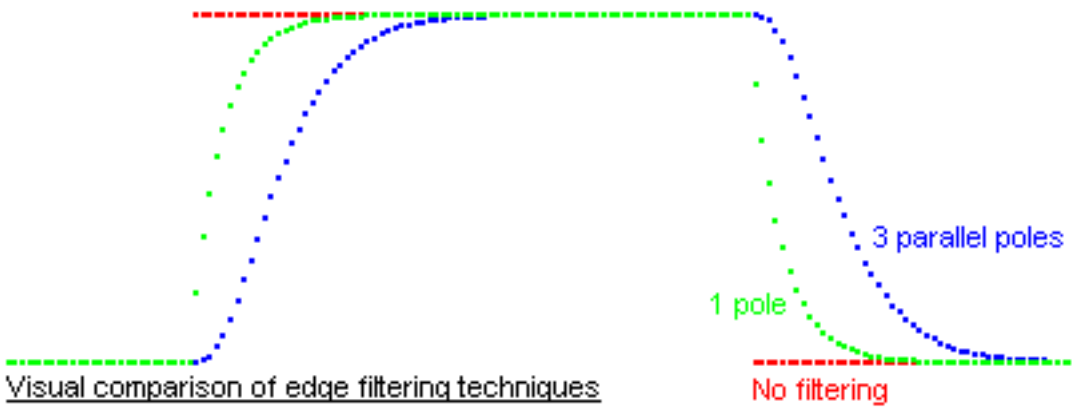
Notes :
use this filter to smooth sudden parameter changes
(see linkfile!)

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;
        }
}
```

Linked files



Visual comparison of edge filtering techniques

LP and HP filter

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;
```

Magnitude and phase plot of arbitrary IIR function, up to 5th order

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))/
```

```
              (b0*b0 + b1*b1 + b2*b2 + b3*b3 + b4*b4 + b5*b5 +
              2*((b0*b1 + b1*b2 + b3*(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);
}
```

MATLAB-Tools for SNDAN

References : Posted by Markus Sapp
Linked file : other001.zip

Notes :
(see linkfile)

Measuring interpollation noise

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. log10 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.. log10(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.

Millimeter to DB (faders...)

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;
}
```

Moog VCF

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 tunning)
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;
```

Moog VCF, variation 1

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);
```

Moog VCF, variation 2

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;
}
```

Noise Shaping Class

Type : Dithering with 9th order noise shaping
References : Posted by cshei[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)

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;
}
```

Notch filter

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;
  }
```

One pole LP and HP

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
coeficient 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
coeficient approximation: p = (2*cutoff/samplerate)^2
```

One pole, one zero LP/HP

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;
```

One zero, LP/HP

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
```

Parallel combs delay calculation

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_ ,...,T_n are the delay lines in increasing order, the formula can be derived by setting T_(k-1)/T_k = Constant and T_n/(2*T_1) = 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.

Phase modulation Vs. Frequency modulation

References : Posted by Bram
Linked file : SimpleOscillator.h (this linked file is included below)

Notes :
This code shows what the difference is betwee 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 interpollation 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);
                }
        }
```

```
}
```

Phaser code

References : Posted by Ross Bencina
Linked file : phaser.cpp (this linked file is included below)

Notes :
(see linked file)

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 Towsend 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 Towsend 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 )
        , _zm1( 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 + _zm1 * _fb ))))));
        _zm1 = y;

        return inSamp + y * _depth;
    }
private:
        class AllpassDelay{
    public:
        AllpassDelay()
                : _a1( 0.f )
            , _zm1( 0.f )
            {}

        void Delay( float delay ){ //sample delay time
                _a1 = (1.f - delay) / (1.f + delay);
        }

        float Update( float inSamp ){
                float y = inSamp * -_a1 + _zm1;
                _zm1 = y * _a1 + inSamp;

            return y;
        }
    private:
        float _a1, _zm1;
    };

    AllpassDelay _alps[6];

    float _dmin, _dmax; //range
    float _fb; //feedback
    float _lfoPhase;
    float _lfoInc;
    float _depth;

    float _zm1;
};
```

Pink noise filter

References : Posted by Paul Kellett
Linked file : pink.txt (this linked file is included below)

Notes :
(see linked file)

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/
```

Polyphase Filters

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)

Linked files

```
CAllPassFilter::CAllPassFilter(const double coefficient)
{
	a=coefficient;

	x0=0.0;
	x1=0.0;
	x2=0.0;

	y0=0.0;
	y1=0.0;
	y2=0.0;

};


CAllPassFilter::~CAllPassFilter()
{
};


double CAllPassFilter::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
                ,0.3325011117394731
                ,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;
};
```

---

pow(x,4) approximation

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-=0x3F800000l; /* un-bias */
  l<<=2;          /* **4 */
  l+=0x3F800000l; /* bias */
  *lp=l;

  /* compiler will read this from memory since & operator had been used */
  return in;
}
```

Prewarping

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*w*T) and we evaluate the analog Ha(s) at s=j*W . the following will confirm the jw 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*W = (2/T) * (exp(j*w*T) - 1) / (exp(j*w*T) + 1)

j*W = (2/T) * (exp(j*w*T/2) - exp(-j*w*T/2)) / (exp(j*w*T/2) + exp(-j*w*T/2))

= (2/T) * (j*2*sin(w*T/2)) / (2*cos(w*T/2))

= j * (2/T) * tan(w*T/2)

or

analog W = (2/T) * tan(w*T/2)

so when the real input frequency is w, 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 w*T approaches pi (Nyquist) the digital filter behaves as the analog filter does as W -> inf. 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 -inf dB faster than an equivalent analog at the same frequencies.

Pseudo-Random generator

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;
}
```

Pulsewidth modulation

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%.

RBJ-Audio-EQ-Cookbook

Type : Biquads for all puposes!
References : Robert Bristow-Johnson (a.k.a. RBJ)
Linked file : http://www.harmony-central.com/Computer/Programming/Audio-EQ-Cookbook.txt
Linked file : http://www.harmony-central.com/Effects/Articles/EQ_Coefficients/EQ-Coefficients.pdf

Notes :
(see linkfile)
A superb collection of filters used in a lot of (commercial) plugins and effects.
There's also a very interesting paper linked about biquad EQ filters.

Reading the compressed WA! parts in gigasampler files

References : Paul Kellett
Linked file : gigxpand.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

Resonant filter

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;
```

Resonant IIR lowpass (12dB/oct)

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;
}
```

Resonant low pass filter

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 roloff), 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;
}


/*
 * ------------------------------------------------------------
 * bilinear()
 *
 * Transform the numerator and denominator coefficients
 * of s-domain biquad section into corresponding
 * z-domain coefficients.
 *
 *      Store the 4 IIR coefficients in array pointed by coef
 *      in following order:
 *             beta1, beta2    (denominator)
 *             alpha1, alpha2  (numerator)
 *
 * Arguments:
 *             a0-a2   - s-domain numerator coefficients
 *             b0-b2   - s-domain denominator coefficients
 *             k               - filter gain factor. initially set to 1
 *                               and modified by each biquad section in such
 *                               a way, as to make it the coefficient by
 *                               which to multiply the overall filter gain
 *                               in order to achieve a desired overall filter
gain,
 *                               specified in initial value of k.
 *             fs             - sampling rate (Hz)
 *             coef    - array of z-domain coefficients to be filled in.
 *
 * Return:
 *             On return, set coef z-domain coefficients
```

```
 * ------------------------------------------------------------
 */
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 */
)
{
    double ad, bd;

                 /* alpha (Numerator in s-domain) */
    ad = 4. * a2 * fs * fs + 2. * a1 * fs + a0;
                 /* beta (Denominator in s-domain) */
    bd = 4. * b2 * fs * fs + 2. * b1* fs + b0;

                 /* update gain constant for this section */
    *k *= ad/bd;

                 /* Denominator */
    *coef++ = (2. * b0 - 8. * b2 * fs * fs)
                           / bd;         /* beta1 */
    *coef++ = (4. * b2 * fs * fs - 2. * b1 * fs + b0)
                           / bd; /* beta2 */

                 /* Nominator */
    *coef++ = (2. * a0 - 8. * a2 * fs * fs)
                           / ad;         /* alpha1 */
    *coef = (4. * a2 * fs * fs - 2. * a1 * fs + a0)
                           / ad;   /* alpha2 */
}


/*
 * ------------------------------------------------------------
 * Transform from s to z domain using bilinear transform
 * with prewarp.
 *
 * Arguments:
 *      For argument description look at bilinear()
 *
 *      coef - pointer to array of floating point coefficients,
 *                     corresponding to output of bilinear transofrm
 *                     (z domain).
 *
 * Note: frequencies are in Hz.
 * ------------------------------------------------------------
 */
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 */
{
                /* 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 roloff. 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 secions 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 secion is:

```
       a2 * s^2 + a1 * s + a0
H(s) = ----------------------
       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:

```
       0 + 0 + 1                  0 + 0 + 1
--€---------------------- * --------------------------
1 + ((1/Q) * 0.765367) + 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:

```
       1 + alpha1 * z^(-1) + alpha2 * z^(-2)
H(z) = -------------------------------------
       1 + beta1 * z^(-1) + beta2 * 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 achive. For our purposes the gain = 1.

We call szxform() function 2 times becase 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
* ------------------------------------------------------------
```

Reverberation techniques

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.

Saturation

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.
Neigther 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)));
    }
}
```

SawSin

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 peak follower

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;
}
```

Sin, Cos, Tan approximation

References : Posted by jan[AT]rpgfan[DOT]demon[DOT]co[DOT]uk
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.

Code :

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;
}
//----------------------------------------------------------------------
```

Sine calculation

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];
}
```

smsPitchScale Source Code

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
```

Soft saturation

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
```

Square Waves

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.

State variable

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
```

State Variable Filter (Double Sampled, Stable)

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   = MIN(0.25, 2.0*sin(PI*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;
}
```

Time domain convolution with O(n^log2(3))

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^log2(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 = A1 + A2, and B = B1 +B2.

Now we can write:

(1) A*B = (A1+A2)*(B1+B2) = A1*B1 + A2*B1 + A1*B2 + A2*B2

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) (A1 + A2 * z^-n) * (B1 + B2 * z^-n)

This is a convolution of two blocks of size n: We shift A2 by n leftwards so it completely overlaps A1, then we add them.
We do the same thing with B1 and B2. Then we convolute the two resulting blocks.

now let's transform this term:

(3) (A1 + A2 * z^-n) * (B1 + B2 * z^-n)

= A1*B1 + A1*B2*z^-n + A2*z^-n*B1 + A2*z^ n*B2*z^-n
= A1*B1 + (A1*B2 + A2*B1)*z^-n + A2*B2*z^-2n

(4) (A1 + A2 * z^-n) * (B1 + B2 * z^-n) - A1*B1 - A2*B2*z^-2n

= (A1*B2 + A2*B1)*z^-n

Now we convolute both sides of the equation (4) by z^n:

(5) (A1 + A2 * z^-n)*(B1 + B2 * z^-n)*z^n - A1*B1*z^n - A2*B2*z^-n

= (A1*B2 + A2*B1)

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 = (A1+A2)*(B1+B2) = A1*B1 + A2*B1 + A1*B2 + A2*B2

= A1*B1
+ (A1 + A2 * z^-n)*(B1 + B2 * z^-n)*z^n - A1*B1*z^n - A2*B2*z^-n
+ A2*B2

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^log2(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 substract 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 substract 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];
}
```

Time domain convolution with O(n^log2(3))

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];
```

```
            }
        }
    }
```

tone detection with Goertzel

Type : Goertzel
References : Posted by espenr@ii.uib.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.
*   Basiclly 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;
}
```

Variable-hardness clipping function

References : Posted by Laurent de Soras
Linked file : laurent.gif (this linked file is included below)

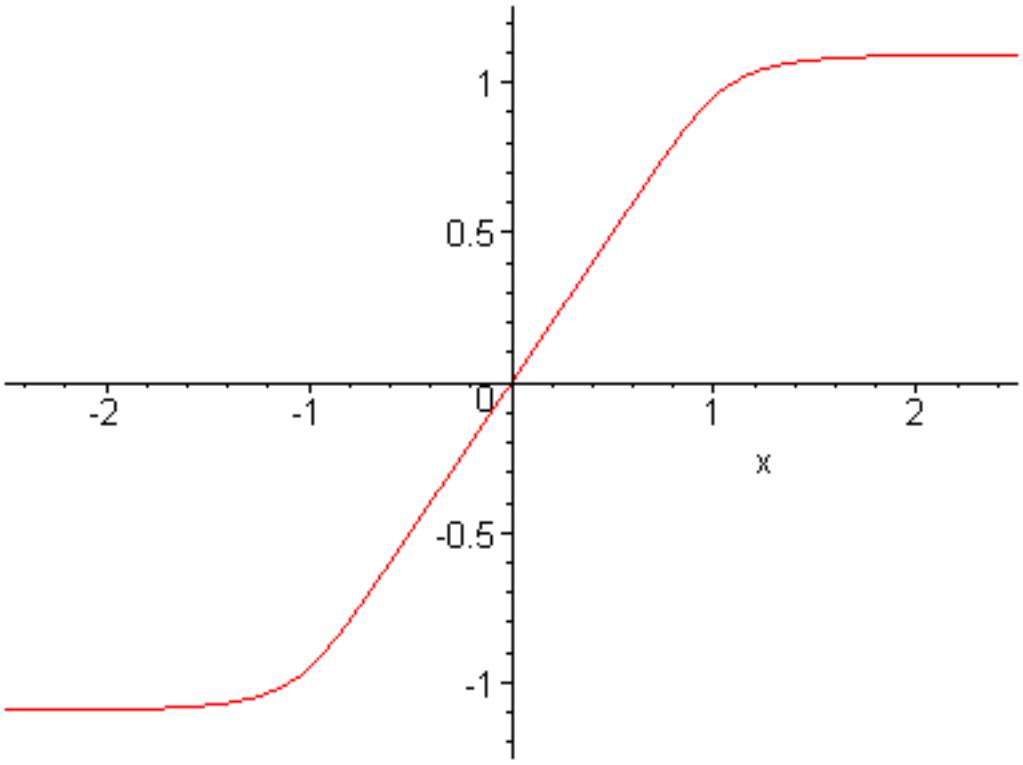
Notes :
k >= 1 is the "clipping hardness". 1 gives a smooth clipping, and a high value gives hardclipping.

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));
```

Linked files



Various Biquad filters

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

Linked files

```
/*
 * Presence and Shelve 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,aa1,ab1;

  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;

  aa1 = a*ta1;
  *a0 = ta0 + aa1 + asq*ta2;
  *a1 = 2.0*a*(ta0+ta2)+(1.0+asq)*ta1;
  *a2 = asq*ta0 + aa1 + ta2;

  ab1 = a*tb1;
  b0 = tb0 + ab1 + asq*tb2;
  *b1 = 2.0*a*(tb0+tb2)+(1.0+asq)*tb1;
  *b2 = asq*tb0 + ab1 + 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);
}
```

Waveform generator using MinBLEPS

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 :

WaveShaper

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

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
```

Waveshaper

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))
```

Waveshaper (simple description)

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 -> 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;
}
```

Waveshaper :: Gloubi-boulga

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));
```

Wavetable Synthesis

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

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...

Zoelzer biquad filters

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 Zolzer 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
```