Musicdsp.org Documentation

Too many to list

Jun 25, 2021

Contents:

Welcome to musicdsp.org.

Musicdsp.org is a collection of algorithms, thoughts and snippets, gathered for the music dsp community. Most of this data was gathered by and for the people of the splendid Music-DSP mailing list at http://sites.music.columbia.edu/cmc/music-dsp/

Important: Please help us edit these pages at https://github.com/bdejong/musicdsp

---

- Special thanks: http://www.fxpansion.com for sponsoring a server to host the archive for many, many years
- Special thanks: http://www.dspdimension.com for pointing http://www.musicdsp.com to this site too

CHAPTER 1

Synthesis

1.1 (Allmost) Ready-to-use oscillators

- **Author or source:** Ross Bencina, Olli Niemitalo, . . .
- **Type:** waveform generation
- **Created:** 2002-01-17 01:01:39

Listing 1: notes

```
Ross Bencina: original source code poster
Olli Niemitalo: UpdateWithCubicInterpolation
```

Listing 2: 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
        /* //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];
}
```

1.2 AM Formantic Synthesis

- **Author or source:** Paul Sernine
- **Created:** 2006-07-05 20:14:14

Listing 3: notes

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

Various formantic waveforms are precalculated and put in tables, they correspond to
different formant widths.
The runtime uses many intances (here 4) of these and pitch shifts them with double
carriers (to preserve the harmonicity of the signal).

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

Paul
```

Listing 4: code

```
// FormantsAM.cpp

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

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

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

```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    //bilinear interpolation.
    return (1-fI)*(TF[i00] + fP*(TF[i00+1]-TF[i00]))
         +    fI*(TF[i10] + fP*(TF[i10+1]-TF[i10]));
}

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

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

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

    //The f0/fn coefficients stand for a -3dB/oct spectral enveloppe
    float out=
           a1*(f0/f1)*formant(p0,100*un_f0)*porteuse(f1*un_f0,p0)
```

(continues on next page)

(continued from previous page)

```
        +0.7f*a2*(f0/f2)*formant(p0,120*un_f0)*porteuse(f2*un_f0,p0)
        +     a3*(f0/f3)*formant(p0,150*un_f0)*porteuse(f3*un_f0,p0)
        +     a4*(f0/f4)*formant(p0,300*un_f0)*porteuse(f4*un_f0,p0);

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

1.2.1 Comments

- **Date**: 2007-04-24 12:04:12
- **By**: Baltazar

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

- **Date**: 2007-08-14 11:30:14
- **By**: phoenix-69

```
Very funny sound !
```

- **Date**: 2008-08-19 20:51:30
- **By**: Wait.

```
What header files are you including?
```

1.3 Alias-free waveform generation with analog filtering

- **Author or source:** Magnus Jonsson
- **Type:** waveform generation
- **Created:** 2002-01-15 21:25:29
- **Linked files:** `synthesis001.txt`.

Listing 5: notes

```
(see linkfile)
```

1.4 Another LFO class

- **Author or source:** mdsp
- **Created:** 2003-08-26 14:56:14
- **Linked files:** `LFO.zip`.

Listing 6: notes

```
This LFO uses an unsigned 32-bit phase and increment whose 8 Most Significant Bits␣
↪adress
a Look-up table while the 24 Least Significant Bits are used as the fractionnal part.
Note: As the phase overflow automatically, the index is always in the range 0-255.

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

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

Modifications:
We could use phase on 64-bit or change the proportion of bits used by the index and␣
↪the
fractionnal part.
```

1.4.1 Comments

- **Date**: 2004-09-07 13:52:17
- **By**: ku.oc.enydranos@aja

```
This type of oscillator is know as a numerically controlled oscillator(nco) or phase␣
↪accumulator sythesiser. Integrated circuits that implement it in hardware are␣
↪available such as the AD7008 from Analog Devices.
The frequency resolution is very high and is = (SampleRate)/32^2. So if clocked at 44.
↪1Khz the frequency resolution would be 0.00001026Hz!
As you said the output waveform can be whatever shape you choose to put in the lookup␣
↪table. The phase register is already in saw tooth form.

Regards,
Tony
```

- **Date**: 2004-12-22 08:40:40
- **By**: moc.yddaht@yddaht

```
It works great!
Here's a Delphi version I just knocked up. Both VCL and KOL supported.

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

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

(continues on next page)

(continued from previous page)

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

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

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



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

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

implementation

{ TPaLfo }
{$IFDEF KOL}
function NewPaLfo(aSamplerate:Single):PPaLfo;
begin
  New(Result,Create);
  with Result^ do
  begin
    FPhase:=0;
```

(continues on next page)

(continued from previous page)

```
    Finc:=0;
    FSamplerate:=aSamplerate;
    SetWaveform(triangle);
    FRate:=1;
  end;
end;
{$ELSE}
constructor TPaLfo.create(SampleRate: Single);
begin
    inherited create;
    FPhase:=0;
    Finc:=0;
    FSamplerate:=aSamplerate;
    SetWaveform(triangle);
    FRate:=1;
end;
{$ENDIF}


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

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

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

(continues on next page)

(continued from previous page)

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

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

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

end.
```

- **Date**: 2004-12-22 12:43:17
- **By**: moc.yddaht@yddaht

```
Oops,

This one is correct:


code:
unit PALFO;
//
// purpose: LUT based LFO
```

(continues on next page)

(continued from previous page)

```
//  author: © 2004, Thaddy de Koning
// Remarks: Translated from c++ sources by Remy Mueller, www.musicdsp.org

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

const
 k1Div24lowerBits = 1/(1 shl 24);

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

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



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

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

(continues on next page)

(continued from previous page)

```
implementation

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


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

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

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

(continues on next page)

(continued from previous page)

```
          end;
          FTable[256] := 0;
        end;
  sawtooth:
      begin
          for i:=0 to 255 do
            FTable[i] := 2*(i/255) - 1;
          FTable[256] := -1;
      end;
  square:
      begin
          for i:=0 to 127 do
          begin
            FTable[i]     :=  1;
            FTable[i+128] := -1;
          end;
          FTable[256] := 1;
      end;
  exponent:
      begin
          // symetric exponent similar to triangle
          for i:=0 to 127 do
        begin
            FTable[i] := 2 * ((exp(i/128) - 1) / (exp(1) - 1)) - 1  ;
            FTable[i+128] := 2 * ((exp((128-i)/128) - 1) / (exp(1) - 1)) - 1  ;
          end;
          FTable[256] := -1;
      end;
  end;
end;

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



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

end.
```

1.5 Another cheap sinusoidal LFO

- **Author or source:** moc.cinohp-e@ofni
- **Created:** 2004-05-14 18:32:57

Listing 7: notes

```
Some pseudo code for a easy to calculate LFO.

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

PJ
```

Listing 8: code

```
r = the rate 0..1

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

1.5.1 Comments

- **Date**: 2004-06-10 21:32:22
- **By**: moc.regnimmu@psd-cisum

```
Slick! I like it!

Sincerely,
Frederick Umminger
```

- **Date**: 2005-02-23 22:24:03
- **By**: es.tensse@idarozs.szalab

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

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

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

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

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

- **Date**: 2006-08-02 22:10:54

- **By**: uh.etle.fni@yfoocs

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

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

- **Date**: 2006-08-03 10:25:10
- **By**: uh.etle.fni@yfoocs

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

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

- **Date**: 2013-11-10 16:51:38
- **By**: ten.akionelet@isangi

```
/* this goes from -1 to +1 */
#include <iostream>
#include <math.h>

using namespace std;
int main(int argc, char *argv[]) {

    //r = the rate 0..1
    float r = 0.5f;
    float p = 0.f;
    float result=0.f;
    //--------------
    for(int i=1;i<=2048;i++){
    p += r;
    if(p > 1) p -= 2;
    result = 4*p*(1-fabs(p));
    cout << result;
    cout <<"\r";
    }
    }
```

1.6 Antialiased square generator

- **Author or source:** Paul Sernine
- **Type:** 1st April edition
- **Created:** 2006-04-01 12:46:23

Listing 9: notes

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

Listing 10: code

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

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

- **Author or source:** uh.doilop.cak@egamer
- **Created:** 2003-01-02 20:27:18

Listing 11: code

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

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

My analogue waveforms are generated in a _very_ high sampling rate (actually it's 1.
↪4112 GHz for 44.1 kHz waveforms, using 32x oversampling).  Using this sample-rate,␣
↪the amplitude of the aliasing harmonics are negligible (the base analogue waveforms␣
↪has exponentially decreasing harmonics amplitudes).

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

(continues on next page)

(continued from previous page)

```

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

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

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

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

1.7.1 Comments

- **Date**: 2003-01-23 13:26:38
- **By**: moc.xinortceletrams@xelA

```
Why can't you use fft/ifft
to synthesis directly wavetables of 2048,1024,..?
It'd be not so
"time consuming" comparing to FIR filtering.
Further cubic interpolation still might give you audible
distortion in some cases.
--Alex.
```

- **Date**: 2003-02-02 19:24:23
- **By**: uh.doilop.cak@egamer

```
What should I use instead of cubic interpolation? (I had already some aliasing problems with cubic interpolation, but that can be solved by oversampling 4x the realtime signal generation)
Is this theory of generating waves from wavetables of 4096, 2084, ... 8, 4, 2 samples wrong?
```

- **Date**: 2003-02-19 17:12:42
- **By**: moc.xinortceletrams@xelA

```
I think tablesize should not vary
depending on tone (4096,2048...)
and you'd better stay with the same table size for all notes (for example 4096, 4096...).

To avoid interpolation noise
(it's NOT caused by aliasing)
try to increase wavetable size
and be sure that waveform spectrum has
steep roll off
(don't forget Gibbs phenomena as well).
```

- **Date**: 2004-08-24 08:04:28

- **By**: es.tensse@idarozs.szalab

```
you say that the higher harmonics (that cause the sharp overshoot at step response)
↪are dampened.
How ? Or is it a result of the filtering ?
```

- **Date**: 2005-04-03 07:10:58
- **By**: uh.doilop.scakam@egamer

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

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

1.8 Audiable alias free waveform gen using width sine

- **Author or source:** moc.emagno@mortslhad.mikaoj
- **Type:** Very simple
- **Created:** 2004-04-07 09:37:32

Listing 12: notes

```
Warning, my english abilities is terribly limited.

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

The psuedo code describes it more.

// Druttis
```

Listing 13: code

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

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

t := time (0 to ...)
ph := phase shift (0 to 1)
```

(continues on next page)

(continued from previous page)

```
fm := freq mod (0 to 1)

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

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

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

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

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

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

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

// Druttis
```

1.8.1 Comments

- **Date**: 2003-02-05 03:10:00
- **By**: es.pp.ecafkrad@sitturd

```
Um, reading it I can see a big flaw...

afact(f) = 1 - sin(f*2PI) is not correct!

should be

afact(f) = 1 - sqrt(f * 2 / sr)

where sr := samplingrate
f should be exceed half sr
```

- **Date**: 2003-02-22 16:49:50
- **By**: moc.ecrofmho@tnerual

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

afact (f) = 1 - sqrt (f  * 2)

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

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

However I haven't checked the formula.
```

- **Date**: 2003-06-25 08:54:21
- **By**: ed.xmg@909

```
Hi Lauent,
I'm new to that DSP stuff and can't get the key to
what'S the meaning of afact? - Can you explain please!? - Thanks in advice!
```

- **Date**: 2004-04-16 14:09:26
- **By**: es.ollehc@sitturd

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

- **Date**: 2004-04-16 14:14:34
- **By**: es.ollehc@sitturd

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

1.9 Band Limited waveforms my way

- **Author or source:** Anton Savov (gb.liam@ottna)
- **Type:** classic Sawtooth example
- **Created:** 2009-06-22 18:09:08

Listing 14: notes

```
This is my <ugly> C++ code for generating a single cycle of a Sawtooth in a table
normaly i create my "fundamental" table big enough to hold on around 20-40Hz in the
current Sampling rate
also, i create the table twice as big, i do "mip-maps" then
so the size should be a power of two, say 1024 for 44100Hz = 44100/1024 = ~43.066Hz
then the mip-maps are with decreasing sizes (twice) 512, 256, 128, 64, 32, 16, 8, 4,
↪and 2

if the "gibbs" effect is what i think it is - then i have a simple solution
here is my crappy code:
```

Listing 15: code

```
int sz = 1024; // the size of the table
int i = 0;
float *table; // pointer to the table
double scale = 1.0;
double pd; // phase
double omega = 1.0 / (double)(sz);

while (i < sz)
{
    double amp = scale;
    double x = 0.0; // the sample
    double h = 1; // harmonic number (starts from 1)
    double dd; // fix high frequency "ring"
    pd = (double)(i) / (double)(sz); // calc phase
```

(continues on next page)

(continued from previous page)

```
    double hpd = pd; // phase of the harmonic
    while (true) // start looping for this sample
    {
            if ((omega * h) < 0.5) // harmonic frequency is in range?
            {
                    dd = cos(omega * h * 2 * pi);
                    x = x + (amp * dd * sin(hpd * 2 * pi));
                    h = h + 1;
                    hpd = pd * h;
                    amp = 1.0 / h;
            }
            else { break; }
    }
    table[i] = x;
    ++i;
}

the peaks are around +/- 0.8
a square can be generated by just changing h = h+2; the peaks would be +/- 0.4

any bugs/improvements?
```

1.9.1 Comments

- **Date**: 2009-06-22 18:37:34
- **By**: gb.liam@ottna

```
excuse me, there is a typo

amp = scale / h;
```

- **Date**: 2009-09-20 10:34:18
- **By**: antto mail bg

```
for even smoother edges:
dd = cos(sin(omega * h * pi) * 0.5 * pi);
no visual ringing, smooth waveform
```

- **Date**: 2009-09-25 04:15:31
- **By**: antto mail bg

```
to get +/- 1.0 amplitude: scale = 1.25
```

1.10 Bandlimited sawtooth synthesis

- **Author or source:** moc.ailet@mlohednal.leuname
- **Type:** DSF BLIT
- **Created:** 2002-03-29 18:06:44
- **Linked files:** `synthesis002.txt`.

Listing 16: 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)
```

1.10.1 Comments

- **Date**: 2003-02-26 00:58:41
- **By**: moc.liamtoh@hsats_wobniar

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

this produces a nearly perfect saw shape in almost all cases
```

- **Date**: 2011-05-31 05:25:01
- **By**: pj.oc.liamtoh@evawtuah

```
The square wave type will be able to be generated from this source.
Please teach if it is possible.
```

1.11 Bandlimited waveform generation

- **Author or source:** Joe Wright
- **Type:** waveform generation
- **Created:** 2002-01-17 01:06:49
- **Linked files:** `bandlimited.cpp`.
- **Linked files:** `bandlimited.pdf`.

Listing 17: notes

```
(see linkfile)
```

1.11.1 Comments

- **Date**: 2012-02-10 16:26:11
- **By**: ed.redienhcsslin@psdcisum

```
The code to reduce the gibbs effect causes aliasing if a transition is made from␣
↪wavetable A with x partials to wavetable B with y partials.

The aliasing can clearly be seen in a spectral view.

The problem is, that the volume modification for partial N is different depending on␣
↪the number of partials the wavetable row contains
```

1.12 Bandlimited waveforms synopsis.

- **Author or source:** Joe Wright
- **Created:** 2002-02-11 17:37:20
- **Linked files:** `waveforms.txt`.

Listing 18: notes

```
(see linkfile)
```

1.12.1 Comments

- **Date**: 2005-11-15 20:07:11
- **By**: dflatccrmadotstanforddotedu

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

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

1.13 Bandlimited waveforms...

- **Author or source:** Paul Kellet
- **Created:** 2002-01-17 00:56:04

Listing 19: 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.
```

Listing 20: 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

```

(continues on next page)

(continued from previous page)

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

1.13.1 Comments

- **Date**: 2004-09-23 00:07:02
- **By**: es.ollehc@evawenis

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

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

Even a small clue would be nice.
Cheers,
A
```

- **Date**: 2012-01-01 23:17:35
- **By**: ku.oc.oohay@ekolbdiurd

```
this sounds quite nice, maybe going to use it an LV 2 plugin
```

- **Date**: 2016-01-17 13:24:11
- **By**: pvdmeer [atorsomething] gmail [point] com

```
this is really amazing, and easily hacked into a lut-based algo. i'll try windowing it too, but it already looks like aliasing is well within acceptable levels.
```

1.14 Butterworth

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** LPF 24dB/Oct
- **Created:** 2006-07-16 11:39:35

Listing 21: code

```
First calculate the prewarped digital frequency:

K = tan(Pi * Frequency / Samplerate);

Now calc some intermediate variables: (see 'Factors of Polynoms' at http://en.wikipedia.org/wiki/Butterworth_filter, especially if you want a higher order like 48dB/Oct)
a = 0.76536686473 * Q * K;
b = 1.84775906502 * Q * K;

```

(continues on next page)

(continued from previous page)

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

Calculate the first biquad:

A0 = (K+a+1);
A1 = 2*(1-K);
A2 =(a-K-1);
B0 = K;
B1 = 2*B0;
B2 = B0;

Calculate the second biquad:

A3 = (K+b+1);
A4 = 2*(1-K);
A5 = (b-K-1);
B3 = K;
B4 = 2*B3;
B5 = B3;

Then calculate the output as follows:

Stage1 = B0*Input + State0;
State0 = B1*Input + A1/A0*Stage1 + State1;
State1 = B2*Input + A2/A0*Stage1;

Output = B3*Stage1 + State2;
State2 = B4*Stage1 + A4/A3*Output + State2;
State3 = B5*Stage1 + A5/A3*Output;
```

1.14.1 Comments

- **Date**: 2006-07-18 18:44:09
- **By**: uh.etle.fni@yfoocs

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

Highpass version should be:

A0 = (K+a+1);
A1 = 2*(1-K);
A2 =(a-K-1);
B0 = 1;
B1 = -2;
B2 = 1;

Calculate the second biquad:

A3 = (K+b+1);
A4 = 2*(1-K);
A5 = (b-K-1);
B3 = 1;
```

(continues on next page)

(continued from previous page)

```
B4 = -2;
B5 = 1;

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

Stage1 = Input + State0;
State0 = B1*Input + A1/A0*Stage1 + State1;
State1 = Input + A2/A0*Stage1;

Output = Stage1 + State2;
State2 = B4*Stage1 + A4/A3*Output + State2;
State3 = Stage1 + A5/A3*Output;

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

Cheers,
Peter
```

- **Date**: 2006-07-21 11:15:06
- **By**: uh.etle.fni@yfoocs

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

inv_A0 = 1.0/A0;
A1A0 = A1 * inv_A0;
A2A0 = A2 * inv_A0;

inv_A3 = 1.0/A3;
A4A3 = A4 * inv_A3;
A5A3 = A5 * inv_A3;

(The above should be faster than writing

A1A0 = A1/A0;
A2A0 = A2/A0;
A4A3 = A4/A3;
A5A3 = A5/A3;

but I think some compilers do this optimization automatically.)

Then the lowpass process loop becomes

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

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

Much faster, isn't it?
```

- **Date**: 2006-07-31 22:48:39

- **By**: ed.luosfosruoivas@naitsirhC

```
Once you figured it out, it's even possible to do higher order butterworth shelving␣
↪filters. Here's an example of an 8th order lowshelf.

First we start as usual prewarping the cutoff frequency:

K = tan(fW0*0.5);

Then we settle up the Coefficient V:

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

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

for i = 0 to 3
{
cm = cos(PI*(i*2+1) / (2*8) );

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

- **Date**: 2006-08-01 23:35:12
- **By**: uh.etle.fni@yfoocs

```
Hmm... interesting. I guess the phase response/group delay gets quite funky, which is␣
↪generally unwanted for an equalizer.

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

I guess the next will be Chebyshev shelving filters ;)

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

Peter
```

- **Date**: 2006-08-02 02:19:57
- **By**: ed.luosfosruoivas@naitsirhC

```
The 1/ is of course an error here. It's left of my own implementation, where I divide␣
↪directly. Also I think A and B is exchanged.

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

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

- **Date**: 2006-08-02 12:32:01

- **By**: uh.etle.fni@yfoocs

```
Different authors tend to mix up A and B coeffs.

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

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

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

- **Date**: 2006-08-02 14:17:19
- **By**: ed.luosfosruoivas@naitsirhC

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

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

- **Date**: 2006-08-02 20:29:36
- **By**: uh.etle.fni@yfoocs

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

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

- **Date**: 2010-03-05 18:45:57
- **By**: moc.xocmdj@xocmdj

```
State2 = B4*Stage1 + A4/A3*Output + State2;
should read
State2 = B4*Stage1 + A4/A3*Output + State3;
```

1.15 C# Oscilator class

- **Author or source:** neotec
- **Type:** Sine, Saw, Variable Pulse, Triangle, C64 Noise
- **Created:** 2007-01-08 10:49:36

Listing 22: notes

```
Parameters:

Pitch: The Osc's pitch in Cents [0 - 14399] startig at A -> 6.875Hz
Pulsewidth: [0 - 65535] -> 0% to 99.99%
Value: The last Output value, a set to this property 'syncs' the Oscilator
```

Listing 23: code

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

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

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

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

    public int Value
    {
        get
```

(continues on next page)

(continued from previous page)

```
        {
            return this._Value;
        }
        set
        {
            this._Value = 0;
            this.OscNow = 0;
        }
    }

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

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

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

    public SynthOscilator()
    {
        if (WaveSteps.Length == 0)
            this.CalcSteps();

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

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

        this.ShiftRegister = 0x7ffff8;

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    }

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

        double s = Math.PI / 32768.0;

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

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

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

            SineTable[i] = t;
        }
    }

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

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

(continues on next page)

(continued from previous page)

```
                ret = SynthTools.SineTable[osc];
                break;
            default:
                break;
        }

        this.OscNow += this.OscStep;

        if (this.OscNow > 0xfffffff)
        {
            int bit0 = ((this.ShiftRegister >> 22) ^ (this.ShiftRegister >> 17)) &
↪0x1;
            this.ShiftRegister <<= 1;
            this.ShiftRegister &= 0x7fffff;
            this.ShiftRegister |= bit0;
        }

        this.OscNow &= 0xfffffff;

        this._Value = ret - 32768;

        return this._Value;
    }

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

1.16 C++ gaussian noise generation

- **Author or source:** ku.oc.latigidpxe@luap
- **Type:** gaussian noise generation
- **Created:** 2004-05-20 09:12:55

Listing 24: notes

```
References :
Tobybears delphi noise generator was the basis. Simply converted it to C++.
Link for original is:
http://www.musicdsp.org/archive.php?classid=0#129
The output is in noise.
```

Listing 25: code

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

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

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

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

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

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

1.16.1 Comments

- **Date**: 2009-07-10 17:39:39
- **By**: moc.enon@enon

```
What's the difference between the much simpler noise generator:

randSeed = (randSeed * 196314165) + 907633515;      out=((int)randSeed)*0.
↪0000000004656612873077392578125f;

and this one? they both sound the same to my ears...
```

- **Date**: 2011-07-22 11:07:12
- **By**: moc.nwonknu@nwonknu

```
How can you change the variance (sigma)?
```

- **Date**: 2013-06-12 12:30:33
- **By**: moc.enon@lubeb

```
This is NOT a good code to generate Gaussian Noice. Look into:
  (random * c2) + (random * c2) + (random * c2)
It is all nonsense! The reason of adding three numbers it the Central Limit Theorem␣
↪to aproximate Gaussian distribution. But the random numbers inside must differ,␣
↪which is not the case. The code on original link http://www.musicdsp.org/archive.
↪php?classid=0#129 is correct.
```

1.17 Chebyshev waveshaper (using their recursive definition)

- **Author or source:** mdsp
- **Type:** chebyshev
- **Created:** 2005-01-10 18:03:29

Listing 26: notes

```
someone asked for it on kvr-audio.

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

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

Listing 27: code

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

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

1.17.1 Comments

- **Date**: 2005-01-10 18:10:12
- **By**: mdsp

```
BTW you can achieve an interresting effect by feeding back the ouput in the input. it␣
↪adds a kind of interresting pitched noise to the signal.

I think VirSyn is using something similar in microTERA.
```

- **Date**: 2005-01-11 19:33:03
- **By**: ku.oc.oodanaw.eiretcab@nad

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

Dan
```

- **Date**: 2005-01-12 11:48:00
- **By**: mdsp

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

if you express your input as a sum of sinusoids of frequencies [f0 f1 f2 ...] and use␣
↪the chebyshev polynom of order 2 you won't have 2*[f0 f1 f2...] as the resulting␣
↪frequencies.
As it's a nonlinear function you can't use the superposition theorem anymore.

beware that chebyshev polynoms are sensitive to
the range of your input. Your sinusoid has to have a gain exaclty equal to 1 in order␣
↪to work as expected.

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

1.18 Cubic polynomial envelopes

- **Author or source:** Andy Mucho
- **Type:** envellope generation
- **Created:** 2002-01-17 00:59:16

Listing 28: 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..
```

Listing 29: code

```
time = 32
startlevel = 0
midlevel = 100
endlevel = 120
```

(continues on next page)

(continued from previous page)

```
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
}
```

1.18.1 Comments

- **Date**: 2004-01-13 12:31:55
- **By**: ti.otinifni@reiruoceht

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

bye bye


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

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

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


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

- **Date**: 2006-10-08 17:50:48
- **By**: if.iki@xemxet

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

(continues on next page)

(continued from previous page)

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

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

Check out forward difference in mathworld for more info.
```

1.19 DSF (super-set of BLIT)

- **Author or source:** David Lowenfels
- **Type:** matlab code
- **Created:** 2003-04-02 23:59:24

Listing 30: notes

```
Discrete Summation Formula ala Moorer

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

If using for BLIT synthesis for virtual analog:
N = maxN;
a = attn_at_Nyquist ^ (1/maxN); %hide top harmonic popping in and out when sweeping
frequency
beta = pi/2;
num = 1 - a^N * cos(N*theta) - a*( cos(theta) - a^N * cos(N*theta - theta) ); %don't␣
↪waste
time on beta

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

Listing 31: code

```
function output = dsf( freq, a, H, samples, beta)
%a = rolloff coeffecient
%H = number of harmonic overtones (fundamental not included)
%beta = harmonic phase shift

samplerate = 44.1e3;
freq = freq/samplerate; %normalize frequency

bandlimit = samplerate / 2; %Nyquist
maxN = 1 + floor( bandlimit / freq ); %prevent aliasing
N = min(H+2,maxN);

theta = 2*pi * phasor(freq, samples);

```

(continues on next page)

(continued from previous page)

```
epsilon = 1e-6;
a = min(a, 1-epsilon); %prevent divide by zero

num = sin(beta) - a*sin(beta-theta) - a^N*sin(beta + N*theta) + a^(N+1)*sin(beta+(N-
↪1)*theta);
den = (1 + a * ( a - 2*cos(theta) ));

output = 2*(num ./ den - 1) * freq; %subtract by one to remove DC, scale by freq to␣
↪normalize
output = output * maxN/N;          %OPTIONAL: rescale to give louder output as␣
↪rolloff increases

function out = phasor(normfreq, samples);
out = mod( (0:samples-1)*normfreq , 1);
out = out * 2 - 1;                 %make bipolar
```

1.19.1 Comments

- **Date**: 2003-04-03 15:05:42
- **By**: David Lowenfels

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

1.20 Direct pink noise synthesis with auto-correlated generator

- **Author or source:** RidgeRat <ten.knilhtrae@6741emmartl>
- **Type:** 16-bit fixed-point
- **Created:** 2007-02-11 20:15:42

Listing 32: notes

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

Listing 33: code

```
// Pink noise class using the autocorrelated generator method.
// Method proposed and described by Larry Trammell "the RidgeRat" --
// see http://home.earthlink.net/~ltrammell/tech/newpink.htm
// There are no restrictions.
//
// ------------------------------------------------------------------
//
// This is a canonical, 16-bit fixed-point implementation of the
// generator in 32-bit arithmetic. There are only a few system
// dependencies.
//
```

(continues on next page)

(continued from previous page)

```
//   -- access to an allocator 'malloc' for operator new
//   -- access to definition of 'size_t'
//   -- assumes 32-bit two's complement arithmetic
//   -- assumes long int is 32 bits, short int is 16 bits
//   -- assumes that signed right shift propagates the sign bit
//
// It needs a separate URand class to provide uniform 16-bit random
// numbers on interval [1,65535]. The assumed class must provide
// methods to query and set the current seed value, establish a
// scrambled initial seed value, and evaluate uniform random values.
//
//
// ----------- header -----------------------------------------------
// pinkgen.h

#ifndef  _pinkgen_h_
#define  _pinkgen_h_  1

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

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

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

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

    // Include a UNoise component
    URand     ugen;

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

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

#include  "pinkgen.h"

// Static class data
long int const PinkNoise::pA[5] =
```

(continues on next page)

(continued from previous page)

```
    { 14055, 12759, 10733, 12273, 15716 };
short int const PinkNoise::pPSUM[5] =
    { 22347, 27917, 29523, 29942, 30007 };

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

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

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

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

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

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

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

// Evaluate next randomized 'pink' number with uniform CPU loading.
short int   PinkNoise::pinkrand( )
```

(continues on next page)

(continued from previous page)

```
{
    short int  randu = ugen.urand() & 0x7fff;      // U[0,32767]
    short int  randv = (short int) ugen.urand();  // U[-32768,32767]

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

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

short int    pink_signal[1024];

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

1.21 Discrete Summation Formula (DSF)

- **Author or source:** Stylson, Smith and others… (Alexander Kritov)
- **Created:** 2002-02-10 12:43:30

Listing 34: 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
```

Listing 35: 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
```

(continues on next page)

(continued from previous page)

```
    return (sin(fi) - s3 - s2 +s1)/s4;
}
```

1.21.1 Comments

- **Date**: 2002-11-08 11:21:19
- **By**: dfl[*AT*]ccrma.stanford.edu

```
According to Stilson + Smith, this should be

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

Could be a typo though?
```

- **Date**: 2003-03-14 17:01:46
- **By**: Alex

```
yepp..
```

- **Date**: 2003-03-20 04:20:51
- **By**: TT

```
So what is wrong about "double" up there?
For DSF, do we have to update the phase (fi input) at every sample?
Another question is what's the input x supposed to represent? Thanks!
```

- **Date**: 2003-04-01 01:45:47
- **By**: David Lowenfels

```
input x should be the phase, and fi is the initial phase I guess? Seems redundant to me.
There is nothing wrong with the double, there is a sign typo in the original posting.
```

- **Date**: 2007-02-14 18:04:44
- **By**: moc.erehwon@ydobon

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

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

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

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

- **Date**: 2008-11-02 11:47:07

- **By**: mysterious T

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

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

1.22 Drift generator

- **Author or source:** ti.oohay@odrasotniuq
- **Type:** Random
- **Created:** 2004-09-23 16:07:34

Listing 36: notes

```
I use this drift to modulate any sound parameter of my synth.
It is very effective if it slightly modulates amplitude or frequency of an FM modulator.
It is based on an incremental random variable, sine-warped.
I like it because it is "continuous" (as opposite to "sample and hold"), and I can set
variation rate and max variation.
It can go to upper or lower constraint (+/- max drift) but it gradually decreases rate of
variation when approaching to the limit.
I use it exactly as an LFO (-1.f .. +1.f)
I use a table for sin instead of sin() function because this way I can change random
distribution, by selecting a different curve (different table) from sine...

I hope that it is clear ... (sigh... :-)
Bye!!!
P.S. Thank you for help in previous submission ;-)
```

Listing 37: code

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

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

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

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

1.22.1 Comments

- **Date**: 2004-09-24 17:37:38
- **By**: ti.oohay@odrasotniuq

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

1.23 Easy noise generation

- **Author or source:** moc.psd-nashi@liam
- **Type:** White Noise
- **Created:** 2006-02-23 22:40:20

Listing 38: notes

```
Easy noise generation,
in .hpp,
b_noise = 19.1919191919191919191919191919191919191919;

alternatively, the number 19 below can be replaced with a number of your choice, to
↪get
that particular flavour of noise.

Regards,
Ove Karlsen.
```

Listing 39: code

```
    b_noise = b_noise * b_noise;
    int i_noise = b_noise;
    b_noise = b_noise - i_noise;

    double b_noiseout;
    b_noiseout = b_noise - 0.5;

    b_noise = b_noise + 19;
```

1.23.1 Comments

- **Date**: 2006-07-16 18:24:22
- **By**: moc.liamg@saoxyz

```
This is quite a good PRNG! The numbers it generates exhibit a slight a pattern
↪(obviously, since it's not very sophisticated) but they seem quite usable! The real
↪FFT spectrum is very flat and "white" with just one or two aberrant spikes while
↪the imaginary spectrum is almost perfect (as is the case with most PRNGs). Very
↪nice! Either that or I need more practice with MuPad...
```

- **Date**: 2007-01-16 12:16:24
- **By**: moc.liamtoh@neslrakevofira

```
Alternatively you can do:

            double b_noiselast = b_noise;
            b_noise = b_noise + 19;
            b_noise = b_noise * b_noise;
            b_noise = b_noise + ((-b_noise + b_noiselast) * 0.5);
            int i_noise = b_noise;
            b_noise = b_noise - i_noise;

This will remove the patterning.
```

- **Date**: 2007-01-16 16:56:19
- **By**: moc.erehwon@ydobon

```
>>b_noise = b_noise + ((-b_noise + b_noiselast) * 0.5);

That seems to reduce to just:

b_noise=(b_noise+b_noiselast) * 0.5;
```

- **Date**: 2007-01-18 22:04:19
- **By**: mymail@com

```
Hi, is this integer? Please do not disturb the forum, rather send me an email.

B.i.T
```

- **Date**: 2007-02-01 16:21:12
- **By**: moc.liamtoh@neslrakevofira

```
The line is written like that, so you can change 0.5, to for instance 0.19.
```

- **Date**: 2007-02-01 16:52:21
- **By**: moc.erehwon@ydobon

```
>>The line is written like that, so you can change 0.5, to for instance 0.19.

OK. Why would I do that? What's that number control?
```

- **Date**: 2007-02-03 15:51:46
- **By**: moc.liamtoh@neslrakevofira

```
It controls the patterning. I usually write my algorithms tweakable.

You could try even lower aswell, maybe 1e-19.
```

1.24 Fast Exponential Envelope Generator

- **Author or source:** Christian Schoenebeck
- **Created:** 2005-03-03 14:44:11

Listing 40: notes

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

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

Listing 41: code

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

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

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

1.24.1 Comments

- **Date**: 2005-03-03 21:16:41
- **By**: moc.erawknuhc@knuhcnezitic

```
is there a typo in the runtime equation? or am i missing something in the␣
↪implementation?
```

- **Date**: 2005-03-04 15:13:56
- **By**: schoenebeck (@) software (minus) engineering.org

```
Why should there be a typo?

Here is my benchmark code btw:
http://stud.fh-heilbronn.de/~cschoene/studienarbeit/benchmarks/exp.cpp
```

- **Date**: 2005-03-04 16:20:41
- **By**: moc.erawknuhc@knuhcnezitic

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

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

- **Date**: 2005-03-04 19:16:29
- **By**: schoenebeck (@) software (minus) engineering.org

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

- **Date**: 2005-03-05 10:26:44
- **By**: ten.ooleem@ooleem

```
This is not working for long envelopes because of numerical accury problems. Try␣
↪calculating is over 10 seconds @ 192KHz to see what I mean: it drifts.
I have an equivalent system that permits to have linear to log and to exp curves with␣
↪a simple parameter. I may submit it one of these days...

Sebastien Metrot
--
http://www.usbsounds.com
```

- **Date**: 2005-03-05 13:48:12
- **By**: schoenebeck (@) software (minus) engineering.org

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

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

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

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

- **Date**: 2005-03-09 11:44:31
- **By**: ten.ooleem@ooleem

```
I believe you are seeing unrealistic results with this test because on x86 the fpu's␣
↪internal format is 80bits and your compiler probably optimises this cases quite␣
↪easily. Try doing the same test, calculating the same envelope, but by breaking the␣
↪calculation in blocks of 256 or 512 samples at a time and then storing in memory␣
↪the temp values for the next block. In this case you may see diferent results and a␣
↪much bigger drift (that's my experience with the same algo).
Anyway my algo is a bit diferent as it permits to change the curent type with a␣
↪parameter, this makes the formula looks like
```

(continues on next page)

(continued from previous page)

```
value = value * coef + contant;
May be this leads to more calculation errors :).
```

- **Date**: 2005-03-09 13:33:43
- **By**: schoenebeck (@) software (minus) engineering.org

```
And again... no! :)

Replace the C equation by:

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

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

Calculated sample points: 5764845
Demanded duration: 30.000000 s
Actual duration: 30.025234 s

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

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

- **Date**: 2005-03-23 22:42:06
- **By**: m (at) mindplay (dot) dk

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

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

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

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

(continues on next page)

(continued from previous page)

```
        ...
    }
}

...

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

endLevel = 10 ^ dB/20;

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

- **Date**: 2005-03-31 14:45:51
- **By**: schoenebeck (@) software (minus) engineering.org

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

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

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

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

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

- **Date**: 2006-03-10 01:53:44
- **By**: na

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

- **Date**: 2006-03-10 02:23:29
- **By**: na[eldar # starman # ee]

```
or coeff = exp(log(levelEnd/levelBegin) /
             (releaseTime * sampleRate) );
not sure but it looks computationally more expensive
```

- **Date**: 2006-11-26 15:44:04
- **By**: hc.xmg@.i.l.e

```
what's about?
coeff = 1.0f + (log(levelEnd) - log(levelBegin)) /
                     (releaseTime * sampleRate - 1);
```

- **Date**: 2006-11-26 15:55:12
- **By**: hc.xmg@.i.l.e

```
sorry for the double post. and i'm now almost sure, that it should be:
coeff = 1.0f + (log(levelEnd) - log(levelBegin)) /
                     (releaseTime * sampleRate + 1);
```

1.25 Fast LFO in Delphi...

- **Author or source:** Dambrin Didier (moc.tcerideciffo-e@log)
- **Created:** 2003-07-15 09:01:18
- **Linked files:** `LFOGenerator.zip`.

Listing 42: notes

```
[from Didier's mail...]
[see attached zip file too!]

I was working on a flanger, & needed an LFO for it. I first used a Sin(), but it was␣
↪too
slow, then tried a big wavetable, but it wasn't accurate enough.
I then checked the alternate sine generators from your web site, & while they're good,
they all can drift, so you're also wasting too much CPU in branching for the drift␣
↪checks.
So I made a quick & easy linear LFO, then a sine-like version of it. Can be useful for
LFO's, not to output as sound.
If has no branching & is rather simple. 2 Abs() but apparently they're fast. In all␣
↪cases
faster than a Sin()
It's in delphi, but if you understand it you can translate it if you want.
It uses a 32bit integer counter that overflows, & a power for the sine output.
If you don't know delphi, $ is for hex (h at the end in c++?), Single is 32bit float,
integer is 32bit integer (signed, normally).
```

Listing 43: code

```
unit Unit1;

interface

uses
```

(continues on next page)

(continued from previous page)

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

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

var
  Form1: TForm1;

implementation

{$R *.DFM}

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

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


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

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

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

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


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

// loop
```

(continues on next page)

(continued from previous page)

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

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

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

end.
```

1.25.1 Comments

- **Date**: 2004-04-29 09:18:58
- **By**: ed.luosfosruoivas@naitsirhC

```
LFO Class...

unit Unit1;

interface

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

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

(continues on next page)

(continued from previous page)

```
    property Min:Single read FMin write SetMin;
    property Max:Single read FMax Write SetMax;
    property LFO:TLFOType read fType Write SetType;
  end;

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

var
  Form1: TForm1;

implementation

{$R *.DFM}

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

procedure TLFO.SetType(tt: TLFOType);
begin
 fType:=tt;
 if fType = lfoSine then
  begin
   fPosMul:=(Sqrt(fScale*2))/$80000000;
   fHalfScale:=(Sqrt(fScale*2))/2;
  end
 else
  begin
   fPosMul:=fScale/$80000000;
  end;
end;

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

procedure TLFO.SetMax(v: Single);
begin
 fMax:=v;
```

(continues on next page)

(continued from previous page)

```
 fScale:=fMax-((fMin+fMax)/2);
end;

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

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

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

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


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

end.
```

- **Date**: 2005-06-01 23:36:15
- **By**: ed.luosfosruoivas@naitsirhC

```
Ups, i forgot something:

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

(continues on next page)

(continued from previous page)

```
...

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

procedure TLFO.SetSpeed(v:Single);
begin
 fSpeed:=v;
 iSpeed:=Round($100000000/fSpeed);
end;

...
```

1.26 Fast SIN approximation for usage in e.g. additive synthesizers

- **Author or source:** neotec
- **Created:** 2008-12-09 11:56:33

Listing 44: notes

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

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

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

Listing 45: code

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

(continues on next page)

(continued from previous page)

```

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

1.26.1 Comments

- **Date**: 2008-12-09 12:11:11
- **By**: neotec

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

UINT32 phase = 0;
UINT32 step = frequency * powf(2.0f, 32.0f) / samplerate;

Then it's just:
...
out = fastsin(phase);
phase += step;
...
```

- **Date**: 2008-12-14 20:04:10
- **By**: moc.toohay@bob

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

1.27 Fast Whitenoise Generator

- **Author or source:** ed.bew@hcrikdlef.dreg
- **Type:** Whitenoise
- **Created:** 2006-02-23 22:39:56

Listing 46: notes

```
This is Whitenoise... :o)
```

Listing 47: code

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

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

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

1.27.1 Comments

- **Date**: 2006-07-18 17:34:00
- **By**: uh.etle.fni@yfoocs

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

- **Date**: 2006-11-29 20:50:44
- **By**: ed.bew@hcrikdlef.dreg

```
As I said! :-)
Take care
```

- **Date**: 2006-11-30 00:57:31
- **By**: moc.erehwon@ydobon

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

- **Date**: 2006-11-30 15:02:54
- **By**: uh.etle.fni@yfoocs

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

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

Peter
```

- **Date**: 2006-11-30 17:55:02
- **By**: moc.erehwon@ydobon

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

I'm currently using this great stuff:

vellocet.com/dsp/noise/VRand.html
```

- **Date**: 2006-12-15 17:12:19
- **By**: moc.liamg@palbmert

```
I compiled it, but I get some grainyness that a unisgned long LC algorithm does not
↪give me...  am I the only one?

pa
```

- **Date**: 2006-12-17 18:12:42
- **By**: uh.etle.fni@yfoocs

```
Did you do everything right? It works here.
```

- **Date**: 2006-12-19 21:24:04
- **By**: ed.bew@hcrikdlef.dreg

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

Links:
en.wikipedia.org/wiki/Linear_feedback_shift_register
en.wikipedia.org/wiki/MOS_Technology_SID
www.cc65.org/mailarchive/2003-06/3156.html
```

- **Date**: 2007-03-13 00:39:39
- **By**: —.liam@firA

```
SID noise! cool.
```

- **Date**: 2021-06-25 11:43:00
- **By**: TaleTN

```
I still seem to run into this noise generator from time to time, so I thought I'd
↪provide some extra info here:

The seed provided above will result in a sequence with a period of 3/4 * 2^29, and
↪with 268876131 unique output values in the [-2147483635, 2147483642] range. This is
↪probably more than enough to generate white noise at any reasonable sample rate,
↪but you can easily increase/max out the period and range, simply by using different
↪seed values.

If you instead use g_x1 = 0x70f4f854 and g_x2 = 0xe1e9f0a7, then this will result in
↪a sequence with a period of 3/4 * 2^32, with 1896933636 unique output values in the
↪[-2147483647, 2147483647] range. This is probably the best you can do with a word
↪size of 32 bits. Also note that only the highest bit will actually have the max
↪period, lower bits will have increasingly shorter periods (just like with a Linear
↪Congruential Generator).
```

1.28 Fast sine and cosine calculation

- **Author or source:** Lot's or references... Check Julius O. SMith mainly
- **Type:** waveform generation
- **Created:** 2002-01-17 00:54:32

Listing 48: 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]
```

1.28.1 Comments

- **Date**: 2003-04-05 10:52:49
- **By**: DFL

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

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

- **Date**: 2003-04-05 17:02:22
- **By**: gro.tucetontsap@kcitgib

```
After a while it may drift, so you should resync it as follows:
const float tmp=1.5f-0.5f*(s[1]*s[1]+s[0]*s[0]);
s[0]*=tmp; s[1]*=tmp;

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

'Tick
```

- **Date**: 2003-04-08 09:19:40
- **By**: DFL

```
Just to expalin the above "resync" equation
(3-x)/2 is an approximation of 1/sqrt(x)
So the above is actually renormalizing the complex magnitude.
[ sin^2 (x) + cos^2(x) = 1 ]
```

- **Date**: 2003-05-15 08:26:22

- **By**: nigel

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

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

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

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

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

- **Date**: 2003-11-03 00:14:34
- **By**: moc.liamtoh@sisehtnysorpitna

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

- **Date**: 2004-12-22 16:52:20
- **By**: hplus

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

- **Date**: 2007-07-24 20:33:12
- **By**: more on that topic…

```
... can be found in Jon Datorro, Effect Design, Part 3, a paper that can be easily␣
↪found in the web.

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

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

- **Date**: 2007-07-24 20:34:22
- **By**: ed.corm@liam

```
More on that can be found in Jon Datorro, Effect Design, Part 3, a paper that can be␣
↪easily found in the web.

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

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

- **Date**: 2008-09-21 20:16:40
- **By**: moc.foohay@bob

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

Thanks
```

1.29 Fast sine wave calculation

- **Author or source:** James McCartney in Computer Music Journal, also the Julius O. Smith paper
- **Type:** waveform generation
- **Created:** 2002-01-17 00:52:33

Listing 49: 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.
```

Listing 50: 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
```

1.29.1 Comments

- **Date**: 2003-04-22 15:05:21

- **By**: moc.liamtoh@trahniak

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

- **Date**: 2006-10-24 22:34:59
- **By**: moc.oi@htnysa

```
looks like the formula has gotten munged.
w = freq * twopi / samplerate
```

1.30 Fast square wave generator

- **Author or source:** Wolfgang (ed.tfoxen@redienhcsw)
- **Type:** NON-bandlimited osc...
- **Created:** 2002-02-10 12:46:22

Listing 51: 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.
```

Listing 52: 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;
```

1.30.1 Comments

- **Date**: 2003-08-03 01:35:08
- **By**: moc.jecnal@psdcisum

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

- **Date**: 2009-04-12 15:33:37

- **By**: moc.liamtoh@18_ogag_leafar

```
In response to lancej, yo can declare a union with a float and a int and operate the␣
↪floatas as here using the int part of the union.

If I remeber correctly the value for -1.f = 0xBF800000 and the value for 1.f =␣
↪0x3F800000, note the 0x80000000 difference between them that is the sign.
```

1.31 Gaussian White Noise

- **Author or source:** uh.atsopten@egamer
- **Created:** 2002-08-07 16:23:28

Listing 53: notes

```
SOURCE:

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

Listing 54: 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 );
```

1.31.1 Comments

- **Date**: 2002-08-28 02:05:50
- **By**: gro.sdikgninnips@nap

```
The previous one seems better for me, since it requires only a rand, half log and␣
↪half sqrt per sample.
Actually, I used that one, but I can't remember where I found it, too. Maybe on Knuth
↪'s book.
```

1.32 Gaussian White noise

- **Author or source:** Alexey Menshikov
- **Created:** 2002-08-01 01:13:47

Listing 55: notes

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

 - Alexey Menshikov
```

Listing 56: 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));
}
```

1.32.1 Comments

- **Date**: 2004-01-29 15:41:35
- **By**: davidchristenATgmxDOTnet

```
White Noise does !not! consist of uniformly distributed values. Because in white␣
↪noise, the power of the frequencies are uniformly distributed. The values must be␣
↪normal (or gaussian) distributed. This is achieved by the Box-Muller Transformation.
↪ This function is the polar form of the Box-Muller Transformation. It is faster and␣
↪numeriacally more stable than the basic form. The basic form is coded in the other␣
↪(second) post.
Detailed information on this topic:
http://www.taygeta.com/random/gaussian.html
http://www.eece.unm.edu/faculty/bsanthan/EECE-541/white2.pdf

Cheers David
```

- **Date**: 2007-09-06 04:09:52
- **By**: moc.dlrownepotb@wahs.a.kcin

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

- **Date**: 2010-07-17 20:35:18
- **By**: ed.rab@oof

```
@nick: Way to late, but y2 will always be initialized as in the first run "pass" is 0
↪(i.e. false). The C# compiler just can't prove it.
```

- **Date**: 2011-09-03 20:43:59
- **By**: moc.liamg@htnysa

```
David is wrong. The distribution of the sample values is irrelevant. 'white' simply
↪describes the spectrum. Any series of sequentially independent random values --
↪whatever their distribution -- will have a white spectrum.
```

1.33 Generator

- **Author or source:** Paul Sernine
- **Type:** antialiased sawtooth
- **Created:** 2006-02-23 22:38:56

Listing 57: notes

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

Listing 58: code

```
/* clair.c         Examen Partiel 2b
   T.Rochebois
   02/03/98
*/
#include <stdio.h>
#include <math.h>
main()
{
  double phase=0,dphase,freq,compensation;
  double aw0=0,aw1=0,ax0=0,ax1=0,ay0=0,ay1=0,az0=0,az1=0,sortie;
  short aout;
  int sr=44100;        //sample rate (Hz)
  double f_debut=55.0;//start freq (Hz)
  double f_fin=sr/6.0;//end freq (Hz)
  double octaves=log(f_fin/f_debut)/log(2.0);
  double duree=50.0;   //duration (s)
  int i;
  FILE* f;
  f=fopen("saw.pcm","wb");
  for(i=0;i<duree*sr;i++)
  {
```

(continues on next page)

(continued from previous page)

```
    //exponential frequency sweep
    //Can be replaced by anything you like.
    freq=f_debut*pow(2.0,octaves*i/(duree*sr));
    dphase=freq*(2.0/sr);      //normalised phase increment
    phase+=dphase;             //phase incrementation
    if(phase>1.0) phase-=2.0; //phase wrapping (-1,+1)

    //polynomial calculation (extensive continuity at -1 +1)
    //        7        1  3     1    5
    //P(x) = --- x - -- x   + --- x
    //       360      36       120
    aw0=phase*(7.0/360.0 + phase*phase*(-1/36.0 + phase*phase*(1/120.0)));
    // quad differentiation (first order high pass filters)
    ax0=aw1-aw0; ay0=ax1-ax0; az0=ay1-ay0; sortie=az1-az0;
    //compensation of the attenuation of the quad differentiator
    //this can be calculated at "control rate" and linearly
    //interpolated at sample rate.
    compensation=1.0/(dphase*dphase*dphase*dphase);
    // compensation and output
    aout=(short)(15000.0*compensation*sortie);
    fwrite(&aout,1,2,f);
    //old memories of differentiators
    aw1=aw0; ax1=ax0; ay1=ay0; az1=az0;
  }
  fclose(f);
}
```

1.33.1 Comments

- **Date**: 2006-03-14 09:02:47
- **By**: rf.liamtoh@57eninreS_luaP

```
More infos and discussions in the KVR thread:
http://www.kvraudio.com/forum/viewtopic.php?t=123498
```

- **Date**: 2006-05-05 17:09:03
- **By**: moc.asile@nobnob

```
nice but i prefer the fishy algo, it generates less alias.
bonaveture rosignol
```

1.34 Inverted parabolic envelope

- **Author or source:** James McCartney
- **Type:** envellope generation
- **Created:** 2002-01-17 00:57:43

Listing 59: 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;
}
```

1.34.1 Comments

- **Date**: 2002-04-11 17:20:10
- **By**: ti.orebil@erognekark

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

- **Date**: 2002-04-13 23:51:49
- **By**: moc.liamtoh@r0x0r0xe

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

- **Date**: 2005-01-17 07:39:28
- **By**: asynth(at)io(dot)com

```
This is not an approximation of a parabola, it IS a parabola.
This entry has become corrupted since it was first posted. Should be:

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

1.35 Matlab/octave code for minblep table generation

- **Author or source:** ude.drofnats.amrcc@lfd

- **Created:** 2005-11-15 22:27:53

Listing 60: notes

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

Enjoy!

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

Listing 61: code

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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


*********************************************


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

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

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

*********************************************

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
*********************************************

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

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

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

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

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

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

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

%sinc = num ./ den;

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

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

1.36 PADsynth synthesys method

- **Author or source:** moc.oohay@xfbusddanyz
- **Type:** wavetable generation
- **Created:** 2005-11-15 22:29:01

Listing 62: notes

```
Please see the full description of the algorithm with public domain c++ code here:
http://zynaddsubfx.sourceforge.net/doc/PADsynth/PADsynth.htm
```

Listing 63: code

```
It's here:
http://zynaddsubfx.sourceforge.net/doc/PADsynth/PADsynth.htm
You may copy it (everything is public domain).
Paul
```

1.36.1 Comments

- **Date**: 2005-11-23 15:49:26
- **By**: moc.yddaht@yddaht

```
Impressed at first hearing! Well documented.
```

- **Date**: 2005-11-25 08:18:58
- **By**: moc.yticliam@qe.n

```
1. Isn't this plain additive synthesis
2. Isn't this the algorithm used by the waldorf microwave synths?
```

- **Date**: 2005-11-30 09:48:02
- **By**: mok.00hay@em

```
1. Nope. This is not a plain additive synthesis. It's a special kind :-P Read the doc␣
↪again :)
2. No way.. this is NOT even close to waldord microwave synths. Google for this :)
```

1.37 PRNG for non-uniformly distributed values from trigonometric identity

- **Author or source:** moc.liamg@321tiloen
- **Type:** pseudo-random number generator
- **Created:** 2009-09-02 07:16:14

Listing 64: notes

```
a method, which generates random numbers in the [-1,+1] range, while having a␣
↪probability
density function with less concentration of values near zero for sin().
you can use an approximation of sin() and/or experiment with such an equation for
different distributions. using tan() will accordingly invert the pdf graph i.e. more
concentration near zero, but the output range will be also affected.

extended read on similar methods:
http://www.stat.wisc.edu/~larget/math496/random2.html

regards
lubomir
```

Listing 65: code

```
//init
x=y=1;

//sampleloop
y=sin((x+=1)*y);
```

1.38 Parabolic shaper

- **Author or source:** rf.eerf@aipotreza
- **Created:** 2008-03-13 10:41:18

Listing 66: notes

```
This function can be used for oscillators or shaper.
it can be driven by a phase accumulator or an audio input.
```

Listing 67: code

```
Function Parashape(inp:single):single;
var fgh,tgh:single;
begin
fgh    := inp ;
fgh    := 0.25-f_abs(fgh) ;
tgh    := fgh ;
tgh    := 1-2*f_abs(tgh);
fgh    := fgh*8;
result := fgh*tgh ;
end;
// f_abs is the function of ddsputils unit.
```

1.39 Phase modulation Vs. Frequency modulation

- **Author or source:** Bram
- **Created:** 2002-08-05 15:31:25
- **Linked files:** `SimpleOscillator.h`.

Listing 68: 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 <b>EXAMPLE</b>

See linked file.
```

1.40 Phase modulation Vs. Frequency modulation II

- **Author or source:** James McCartney
- **Created:** 2003-12-01 08:24:26

Listing 69: notes

```
The difference between FM & PM in a digital oscillator is that FM is added to the
frequency before the phase integration, while PM is added to the phase after the phase
integration. Phase integration is when the old phase for the oscillator is added to␣
↪the
current frequency (in radians per sample) to get the new phase for the oscillator. The
equivalent PM modulator to obtain the same waveform as FM is the integral of the FM
modulator. Since the integral of sine waves are inverted cosine waves this is no␣
↪problem.
In modulators with multiple partials, the equivalent PM modulator will have different
relative partial amplitudes. For example, the integral of a square wave is a triangle
wave; they have the same harmonic content, but the relative partial amplitudes are
different. These differences make no difference since we are not trying to exactly
recreate FM, but real (or nonreal) instruments.

The reason PM is better is because in PM and FM there can be non-zero energy produced␣
↪at 0
Hz, which in FM will produce a shift in pitch if the FM wave is used again as a␣
↪modulator,
however in PM the DC component will only produce a phase shift. Another reason PM is
better is that the modulation index (which determines the number of sidebands␣
↪produced and
which in normal FM is calculated as the modulator amplitude divided by frequency of
modulator) is not dependant on the frequency of the modulator, it is always equal to␣
↪the
amplitude of the modulator in radians. The benefit of solving the DC frequency shift
problem, is that cascaded carrier-modulator pairs and feedback modulation are␣
↪possible.
The simpler calculation of modulation index makes it easier to have voices keep the␣
↪same
harmonic structure throughout all pitches.

The basic mathematics of phase modulation are available in any text on electronic
communication theory.

Below is some C code for a digital oscillator that implements FM,PM,and AM. It␣
↪illustrates
the difference in implementation of FM & PM. It is only meant as an example, and not␣
↪as an
efficient implementation.
```

Listing 70: code

```
/* Example implementation of digital oscillator with FM, PM, & AM */

#define PI 3.14159265358979
#define RADIANS_TO_INDEX (512.0 / (2.0 * PI))

typedef struct{      /* oscillator data */
    double freq;   /* oscillator frequency in radians per sample */
    double phase;  /* accumulated oscillator phase in radians */
    double wavetable[512]; /* waveform lookup table */
} OscilRec;


/* oscil - compute 1 sample of oscillator output whose freq. phase and
*    wavetable are in the OscilRec structure pointed to by orec.
*/
double oscil(orec, fm, pm, am)
    OscilRec *orec;  /* pointer to the oscil's data */
    double fm; /* frequency modulation input  in radians per sample */
    double pm; /* phase modulation input      in radians */
    double am; /* amplitude modulation input  in any units you want */
{
    long tableindex;            /* index into wavetable */
    double instantaneous_freq;  /* oscillator freq  + freq  modulation */
    double instantaneous_phase; /* oscillator phase + phase modulation */
    double output;              /* oscillator output */

    instantaneous_freq  = orec->freq  + fm; /* get instantaneous freq */
    orec->phase += instantaneous_freq;      /* accumulate phase */
    instantaneous_phase = orec->phase + pm; /* get instantaneous phase */

    /* convert to lookup table index */
    tableindex = RADIANS_TO_INDEX * instantaneous_phase;
    tableindex &= 511; /* make it mod 512 === eliminate multiples of 2*k*PI */

    output = orec->wavetable[tableindex] * am; /* lookup and mult by am input */

    return (output);  /* return oscillator output */
}
```

1.40.1 Comments

- **Date**: 2011-03-08 11:29:32
- **By**: ed.redienhcsslin@mapsvulipsdcisum

```
As the PM/FM is "iterative", won't this code produce different results at different␣
↪sampling rates? How can this be prevented?

Any advice is highly appreciated!
```

1.41 Pseudo-Random generator

- **Author or source:** Hal Chamberlain, "Musical Applications of Microprocessors" (Phil Burk)
- **Type:** Linear Congruential, 32bit
- **Created:** 2002-01-17 03:11:50

Listing 71: 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.
```

Listing 72: 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;
}
```

1.42 PulseQuad

- **Author or source:** moc.ellehcim-erdna@ma
- **Type:** Waveform
- **Created:** 2007-01-08 10:50:55

Listing 73: notes

```
This is written in Actionscript 3.0 (Flash9). You can listen to the example at
http://lab.andre-michelle.com/playing-with-pulse-harmonics
It allows to morph between a sinus like quadratic function and an ordinary pulse width
with adjustable pulse width. Note that the slope while morphing is always zero at the␣
↪edge
points of the waveform. It is not just distorsion.
```

Listing 74: code

```
http://lab.andre-michelle.com/swf/f9/pulsequad/PulseQuad.as
```

1.43 Pulsewidth modulation

- **Author or source:** Steffan Diedrichsen
- **Type:** waveform generation

- **Created:** 2002-01-15 21:29:52

Listing 75: 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%.
```

1.44 Quick & Dirty Sine

- **Author or source:** MisterToast
- **Type:** Sine Wave Synthesis
- **Created:** 2007-01-08 10:50:10

Listing 76: notes

```
This is proof of concept only (but code works--I have it in my synth now).

Note that x must come in as 0<x<=4096. If you want to scale it to something else (like
0<x<=2*M_PI), do it in the call. Or do the math to scale the constants properly.

There's not much noise in here. A few little peaks here and there. When the signal is␣
↪at
-20dB, the worst noise is at around -90dB.

For speed, you can go all floats without much difference. You can get rid of that␣
↪unitary
negate pretty easily, as well. A couple other tricks can speed it up further--I went␣
↪for
clarity in the code.

The result comes out a bit shy of the range -1<x<1. That is, the peak is something␣
↪like
0.999.

Where did this come from? I'm experimenting with getting rid of my waveform tables,␣
↪which
require huge amounts of memory. Once I had the Hamming anti-ringing code in, it looked
like all my waveforms were smooth enough to approximate with curves. So I started with
sine. Pulled my table data into Excel and then threw the data into a curve-fitting
application.

This would be fine for a synth. The noise is low enough that you could easily get away
with it. Ideal for a low-memory situation. My final code will be a bit harder to
understand, as I'll break the curve up and curve-fit smaller sections.
```

Listing 77: code

```
float xSin(double x)
{
    //x is scaled 0<=x<4096
    const double A=-0.015959964859;
    const double B=217.68468676;
```

(continues on next page)

(continued from previous page)

```
    const double C=0.000028716332164;
    const double D=-0.0030591066066;
    const double E=-7.3316892871734489e-005;
    double y;

    bool negate=false;
    if (x>2048)
    {
            negate=true;
            x-=2048;
    }
    if (x>1024)
            x=2048-x;
    if (negate)
            y=-((A+x)/(B+C*x*x)+D*x-E);
    else
            y=(A+x)/(B+C*x*x)+D*x-E;
    return (float)y;
}
```

1.44.1 Comments

- **Date**: 2007-01-08 18:39:26
- **By**: moc.dniftnacuoyerehwemos@tsaot

```
Improved version:

float xSin(double x)
{
    //x is scaled 0<=x<4096
    const double A=-0.40319426317E-08;
    const double B=0.21683205691E+03;
    const double C=0.28463350538E-04;
    const double D=-0.30774648337E-02;
    double y;

    bool negate=false;
    if (x>2048)
    {
            negate=true;
            x-=2048;
    }
    if (x>1024)
            x=2048-x;
    y=(A+x)/(B+C*x*x)+D*x;
    if (negate)
            return (float)(-y);
    else
            return (float)y;
}
```

- **Date**: 2007-04-15 23:54:56
- **By**: ten.zo@1otniped

```
%This is Matlab code. you can convert it to C
%All it take to make a high quality sine
%wave is 1 multiply and one subtract.
%You first have to initialize the 2 unit delays
% and the coefficient

Fs = 48000;      %Sample rate
oscfreq = 1000.0; %Oscillator frequency in Hz
c1 = 2 * cos(2 * pi * oscfreq / Fs);
%Initialize the unit delays
d1 = sin(2 * pi * oscfreq / Fs);
d2 = 0;
%Initialization done here is the oscillator loop
% which generates a sinewave
for j=1:100
   output = d1;        %This is the sine value
   fprintf(1, '%f\n', output);
   %one multiply and one subtract is all it takes
   d0 = d1 * c1 - d2;
   d2 = d1;   %Shift the unit delays
   d1 = d0;
end
```

- **Date**: 2008-02-09 20:54:10
- **By**: moc.liamg@iratuusala.osuuj

```
Hi,

Can I use this code in a GPL2 or GPL3 licensed program (a soft synth project called
↪Snarl)? In other words, will you grant permission for me to re-license your code?
↪And what name should I write down as copyright holder in the headers?

Thanks,
Juuso Alasuutari
```

- **Date**: 2009-06-23 03:12:13
- **By**: by moc.dniftnacuoyerehwemos@tsaot

```
Juuso,

Absolutely!

Toast
```

1.45 Quick and dirty sine generator

- **Author or source:** moc.liamtoh@tsvreiruoc
- **Type:** sine generator
- **Created:** 2004-01-06 19:57:44

Listing 78: notes

```
this is part of my library, although I've seen a lot of sine generators, I've never␣
↪seen
the simplest one, so I try to do it,
tell me something, I've try it and work so tell me something about it
```

Listing 79: code

```
PSPsample PSPsin1::doOsc(int numCh)
{

    double x=0;
    double t=0;

    if(m_time[numCh]>m_sampleRate)  //re-init cycle
            m_time[numCh]=0;

    if(m_time[numCh]>0)
    {
            t =(double)(((double)m_time[numCh])/(double)m_sampleRate);

            x=(m_2PI *(double)(t)*m_freq);
    }
    else
            x=0;


    PSPsample r=(PSPsample) sin(x+m_phase)*m_amp;

    m_time[numCh]++;

    return  r;

}
```

1.45.1 Comments

- **Date**: 2004-01-08 13:51:26
- **By**: moc.sulp.52retsinnab@etep

```
isn't the sin() function a little bit heavyweight?  Since this is based upon slices␣
↪of time, would it not be much more processor efficient to use a state variable␣
↪filter that is self oscillating?


The operation:
t =(double)(((double)m_time[numCh])/(double)m_sampleRate);

also seems a little bit much, since t could be calculated by adding an interval value,
↪ which would eliminate the divide (needs more clocks).  The divide would then only␣
↪need to be done once.
An FDIV may take 39 clock cycles minimum(depending on the operands), whilst an FADD␣
↪is far faster (3 clocks).  An FMUL is comparable to an add, which would be a␣
↪predominant instruction if using the SVF method.
```

(continues on next page)

(continued from previous page)

```
FSIN may take between 16-126 clock cylces.

(clock cycle info nabbed from: http://www.singlix.com/trdos/pentium.txt)
```

- **Date**: 2004-01-09 21:19:37
- **By**: moc.hclumoidua@bssor

```
See also the fun with sinusoids page:
http://www.audiomulch.com/~rossb/code/sinusoids/
```

- **Date**: 2014-11-18 07:37:27
- **By**: moc.oohay@trawets.tna

```
For audio generation, sines are expensive i think, they are so perfect and take up
↪more processing. it's rare to find a synth that sounds nicer with a sine compared
↪to a parabol wave. My favourite parabolic wave is simply triangle wave with x*x
↪with one of the half periods flipped. x*x is a very fast!!!
```

- **Date**: 2014-12-06 12:06:58
- **By**: ed.xmg@retsneum.ellak

```
hmm... x*x second half flipped...
very cool ! i'll give it a try!!
```

1.46 RBJ Wavetable 101

- **Author or source:** Robert Bristow-Johnson
- **Created:** 2005-05-04 20:34:05
- **Linked files:** `Wavetable-101.pdf`.

Listing 80: notes

```
see linked file
```

1.47 Randja compressor

- **Author or source:** randja
- **Type:** compressor
- **Created:** 2009-09-29 07:37:05

Listing 81: notes

```
I had found this code on the internet then made some improvements (speed) and now I
↪post
it here for others to see.
```

Listing 82: code

```
#include <cmath>
#define max(a,b)  (a>b?a:b)

class compressor
{

    private:
            float   threshold;
            float   attack, release, envelope_decay;
            float   output;
            float   transfer_A, transfer_B;
            float   env, gain;

    public:
    compressor()
    {
            threshold = 1.f;
            attack = release = envelope_decay = 0.f;
            output = 1.f;

            transfer_A = 0.f;
            transfer_B = 1.f;

            env = 0.f;
            gain = 1.f;
    }

    void set_threshold(float value)
    {
            threshold = value;
            transfer_B = output * pow(threshold,-transfer_A);
    }


    void set_ratio(float value)
    {
            transfer_A = value-1.f;
            transfer_B = output * pow(threshold,-transfer_A);
    }


    void set_attack(float value)
    {
            attack = exp(-1.f/value);
    }


    void et_release(float value)
    {
            release = exp(-1.f/value);
            envelope_decay = exp(-4.f/value); /* = exp(-1/(0.25*value)) */
    }


    void set_output(float value)
```

(continues on next page)

(continued from previous page)

```
    {
            output = value;
            transfer_B = output * pow(threshold,-transfer_A);
    }


    void reset()
    {
            env = 0.f; gain = 1.f;
    }


    __forceinline void process(float *input_left, float *input_right,float *output_left, float *output_right,       int frames)
    {
            float det, transfer_gain;
            for(int i=0; i<frames; i++)
            {
                    det = max(fabs(input_left[i]),fabs(input_right[i]));
                    det += 10e-30f; /* add tiny DC offset (-600dB) to prevent denormals */

                    env = det >= env ? det : det+envelope_decay*(env-det);

                    transfer_gain = env > threshold ? pow(env,transfer_A)*transfer_B:output;

                    gain = transfer_gain < gain ?
                                                    transfer_gain+attack *(gain-transfer_gain):
                                                    transfer_gain+release*(gain-transfer_gain);

                    output_left[i] = input_left[i] * gain;
                    output_right[i] = input_right[i] * gain;
            }
    }


    __forceinline void process(double *input_left, double *input_right,     double *output_left, double *output_right,int frames)
    {
            double det, transfer_gain;
            for(int i=0; i<frames; i++)
            {
                    det = max(fabs(input_left[i]),fabs(input_right[i]));
                    det += 10e-30f; /* add tiny DC offset (-600dB) to prevent denormals */

                    env = det >= env ? det : det+envelope_decay*(env-det);

                    transfer_gain = env > threshold ? pow(env,transfer_A)*transfer_B:output;

                    gain = transfer_gain < gain ?
                                                    transfer_gain+attack *(gain-transfer_gain):
```

(continues on next page)

(continued from previous page)

```
                                                    transfer_gain+release*(gain-
↪transfer_gain);

                        output_left[i] = input_left[i] * gain;
                        output_right[i] = input_right[i] * gain;
                }
        }

};
```

1.47.1 Comments

- **Date**: 2010-08-30 21:04:37
- **By**: moc.oohay@xofirgomsnart

```
env = det >= env ? det : det+envelope_decay*(env-det);

I have found it is good to add either a timed peak hold-before-release or something␣
↪like a "peak magnetism" with an attack/release time that is within a cycle of the␣
↪lowest expected frequency in the side chain...often ~80Hz is a good reference point.
↪.. for example here is a snippet to illustrate the idea...please keep in mind this␣
↪will not work as show because of some obvious things needing to be added...
//add these variables to your compressor object...
class compressor {
float patk;  //"seek" slope
float ipatk;
float peakdet;
float target;
float envelope;  //the final value used for gain control
int hold;  //time to hold peak
int holdcnt;

patk = SAMPLE_RATE/80.0f;
ipatk = 1.0f - patk;
.
.
.
}
///now the envelope following function
float compressor::envelope_tracker(float input)
{
float rect = fabs(input);
holdcnt++;
if(rect>peakdet) {
 peakdet = rect;
 if(holdcnt>hold) //the hold is really optional
 {
 target = rect;
 holdcnt = 0;
 }
}
peakdet*=ipatk;  //sort of like capacitor+diode discharge.

if(envelope<target) envelope+=patk;
```

(continues on next page)

(continued from previous page)

```
else envelope -= patk;  //sort of like a heat seeking missile ;)
//for the seek function this is really nothing more than a feedback control system,␣
↪so a more elaborate higher order system could be used...
//the linear interpolation is good to use because
///it is simple while the main attack & release
//functions filter off the corners.
//In either case it is an improvement to a leaky
//peak detector.

}

Another thing I saw in the form of an analog circuit was a "round robin" peak␣
↪detector which had 4 peak detectors being reset by a JFET to ground and was clocked␣
↪by simple binary counter IC.  The full cycle from 0 to 15 on the binary counter was␣
↪set at about 80Hz, then the JFET gates were reset every /4 of that period.  That␣
↪way the envelope tracker pretty much grabbed every peak during the 1/80 period,␣
↪then the "release" time was 1/80th second.  This could be implemented digitally␣
↪very easily ;)
```

- **Date**: 2012-12-14 15:24:30
- **By**: moc.liamg@etteresemv

```
Hi randja,

I am trying to use your compressor.
Could you explain in what unity shall be given the values of threshold, ratio,␣
↪release, attack and output ?
Have you a good set of values for these parameters to make it work ?

Thanks in advance
```

- **Date**: 2013-07-17 21:22:21
- **By**: moc.elpmaxe@ylper-on

```
@vmeserette:

To quote the original code that randja "found" on the Internet and then "made some␣
↪improvements" (he only added __forceinline):
"
How to use it:
--------------
free_comp.cpp and free_comp.hpp implement a simple C++ class called free_comp.
(People are expected to know what to do with it! If not seek help on a beginner
programming forum!)
The free_comp class implements the following methods:
    void    set_threshold(float value);
            Sets the threshold level; 'value' must be a _positive_ value
            representing the amplitude as a floating point figure which should be
            1.0 at 0dBFS

    void    set_ratio(float value);
            Sets the ratio; 'value' must be in range [0.0; 1.0] with 0.0
            representing a oo:1 ratio, 0.5 a 2:1 ratio; 1.0 a 1:1 ratio and so on

    void    set_attack(float value);
```

(continues on next page)

(continued from previous page)

```
            Sets the attack time; 'value' gives the attack time in _samples_

    void    set_release(float value);
            Sets the release time; 'value' gives the release time in _samples_

    void    set_output(float value);
            Sets the output gain; 'value' represents the gain, where 0dBFS is 1.0
            (see set_threshold())

    void    reset();
            Resets the internal state of the compressor (gain reduction)

    void    process(float *input_left, float *input_right,
                                float *output_left, float *output_right,
                                int frames, int skip);
    void    process(double *input_left, double *input_right,
                                double *output_left, double *output_right,
                                int frames, int skip);
            Processes a stereo stream of length 'frames' from either two arrays of
            floats or arrays of doubles 'input_left' and 'input_right' then puts
            the processed data in 'output_left' and 'output_right'.
            'input_{left,right}' and 'output_{left,right}' may be the same location
            in which case the algorithm will work in place. '{input,output}_left'
            and '{input,output}_right' can also point to the same data, in which
            case the algorithm works in mono (although if you process a lot of mono
            data it will yield more performance if you modify the source to make the
            algorithm mono in the first place).
            The 'skip' parameter allows for processing of interleaved as well as two
            separate contiguous streams. For two separate streams this value should
            be 1, for interleaved stereo it should be 2 (but it can also have other
            values than that to process specific channels in an interleaved audio
            stream, though if you do that it is highly recommended to study the
            source first to check whether it yields the expected behaviour or not).
" - Source: http://outsim.co.uk/forum/download/file.php?id=7296
```

1.48 Rossler and Lorenz Oscillators

- **Author or source:** moc.noicratse@ajelak
- **Type:** Chaotic LFO
- **Created:** 2004-10-10 00:13:58

Listing 83: notes

```
The Rossler and Lorenz functions are iterated chaotic systems - they trace smooth␣
↪curves
that never repeat the same way twice. Lorenz is "unpitched", having no distinct peaks␣
↪in
its spectrum -- similar to pink noise. Rossler exhibits definite spectral peaks␣
↪against a
noisy broadband background.

Time-domain and frequency spectrum of these two functions, as well as other info, can␣
↪be
```

(continues on next page)

(continued from previous page)

```
found at:

http://www.physics.emory.edu/~weeks/research/tseries1.html

These functions might be useful in simulating "analog drift."
```

Listing 84: code

```
Available on the web at:
http://www.tinygod.com/code/BLorenzOsc.zip
```

1.48.1 Comments

- **Date**: 2005-01-10 09:11:12
- **By**: moc.yddaht@yddaht

```
A Delphi/pascal version for VCL, KOL, Kylix and Freepascal on my website:
http://members.chello.nl/t.koning8/loro_sc.pas

Nice work!
```

1.49 SawSin

- **Author or source:** Alexander Kritov
- **Type:** Oscillator shape
- **Created:** 2002-02-10 12:40:59

Listing 85: 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;
}
```

1.50 Simple Time Stretching-Granular Synthesizer

- **Author or source:** Harry-Chris
- **Created:** 2008-12-18 07:08:20

Listing 86: notes

```
Matlab function that implements crude time stretching - granulizing function, by␣
↪overlap
add in time domain.
```

Listing 87: code

```
function y = gran_func(x, w, H,H2, Fs, tr_amount)


% x -> input signal
% w -> Envelope - Window Vector
% H1 -> Original Hop Size
% H2 -> Synthesis Hop Size
% Fs -> Sample Rate
% str_amount -> time stretching factor


M = length(w);

pin = 1;
pend = length(x) - M;


y = zeros(1, floor( str_amount * length(x)) +M);


count = 1;
idx = 1;

while pin < pend

    input = x(pin : pin+M-1) .* w';


    y(idx : idx + M - 1) = y(idx : idx + M - 1) + input;

    pin = pin + H;
    count = count + 1;
    idx = idx + H2;

end
```

1.51 Sine calculation

- **Author or source:** Phil Burk
- **Type:** waveform generation, Taylor approximation of sin()
- **Created:** 2002-01-17 00:57:01

Listing 88: 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,
```

(continues on next page)

(continued from previous page)

```
suitable for FM or other time varying applications where accurate frequency is needed.
↪ The
sine generated is accurate to at least 16 bits.
```

Listing 89: 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];
}
```

1.52 Smooth random LFO Generator

- **Author or source:** Rob Belcham
- **Created:** 2009-06-30 08:31:24

Listing 90: notes

```
I've been after a random LFO that's suitable for modulating a delay line for ages ( e.
↪g
for chorus / reverb modulation) , so after i rolled my own, i thought i'd better make␣
↪it
my first contribution to the music-dsp community.

My aim was to achive a sinusoidal based random but smooth waveform with a frequency
control that has no discontinuities and stays within a -1:1 range. If you listen to␣
↪it, it
sounds quite like brown noise, or wind through a microphone (at rate = 100Hz for␣
↪example)
```

(continues on next page)

(continued from previous page)

```
It's written as a matlab m function, so shouldn't be too hard to port to C.

The oscillator generates a random level stepped waveform with random time spent at each
step (within bounds). These levels are linearly interpolated between and used to drive the
frequency of a sinewave. To achive amplitude variation, at each zero crossing a new random
amplitude scale factor is generated. The amplitude coefficient is ramped to this value
with a simple exponential.

An example call would be,
t = 4; Fs = 44100;
y = random_lfo(100, t*Fs, Fs);
axis([0, t*Fs, -1, 1]);
plot(y)

Enjoy !
```

Listing 91: code

```
% Random LFO Generator
% creates a random sinusoidal waveform with no discontinuities
%   rate = average rate in Hz
%   N = run length in samples
%   Fs = sample frequency in Hz
function y = random_lfo(rate, N, Fs)

step_freq_scale = Fs / (1*rate);
min_Cn = 0.1 * step_freq_scale;
An = 0;
lastA = 0;
Astep = 0;
y = zeros(1,N); % output
x = 0;  % sine phase
lastSign = 0;
amp_scale = 0.6;
new_amp_scale = 0.6;
amp_scale_ramp = exp(1000/Fs)-1;
for (n=1:N)
    if (An == 0) || (An>=Cn)
        % generate a new random freq scale factor
        Cn = floor(step_freq_scale * rand());
        % limit to prevent rapid transitions
        Cn = max(Cn, min_Cn);
        % generate new value & step coefficient
        newA = 0.1 + 0.9*rand();
        Astep = (newA - lastA) / Cn;
        A = lastA;
        lastA = newA;
        % reset counter
        An = 0;
    end
    An = An + 1;
    % generate output
```

(continues on next page)

(continued from previous page)

```
    y(n) = sin(x) * amp_scale;
    % ramp amplitude
    amp_scale = amp_scale + ( new_amp_scale - amp_scale ) * amp_scale_ramp;
    sin_inc = 2*pi*rate*A/Fs;
    A = A + Astep;
    % increment phase
    x = x + sin_inc;
    if (x >= 2*pi)
        x = x - 2*pi;
    end
    % scale at each zero crossing
    if (sign(y(n)) ~= 0) && (sign(y(n)) ~= lastSign)
        lastSign = sign(y(n));
        new_amp_scale = 0.25 + 0.75*rand();
    end;
end;
```

1.53 Square Waves

- **Author or source:** Sean Costello
- **Type:** waveform generation
- **Created:** 2002-01-15 21:26:38

Listing 92: 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.
```

1.53.1 Comments

- **Date**: 2003-04-01 01:28:28

- **By**: dfl@stanford. edu

```
Actually, I don't think this would work...
The proper way to do it is subtract a phase shifted buzz (aka BLIT) at the same␣
↪frequency. This is equivalent to comb filtering, which will notch out the even␣
↪harmonics.
```

- **Date**: 2008-11-08 16:24:18
- **By**: moc.psdallahlav@naes

```
The above comment is correct, and my concept is inaccurate. My technique may have␣
↪produced a signal with the proper harmonic structure, but it has been nearly 10␣
↪years since I wrote the post, so I can't remember what I was working with.

DFL's technique can be implemented with two buzz generators, or with a single buzz␣
↪generator in conjunction with a fractional delay, where the delay controls the␣
↪amount of phase shift.
```

1.54 Trammell Pink Noise (C++ class)

- **Author or source:** ude.drofnats.amrcc@lfd
- **Type:** pink noise generator
- **Created:** 2006-05-06 08:40:38

Listing 93: code

```
#ifndef _PinkNoise_H
#define _PinkNoise_H

// Technique by Larry "RidgeRat" Trammell 3/2006
// http://home.earthlink.net/~ltrammell/tech/pinkalg.htm
// implementation and optimization by David Lowenfels

#include <cstdlib>
#include <ctime>

#define PINK_NOISE_NUM_STAGES 3

class PinkNoise {
public:
  PinkNoise() {
  srand ( time(NULL) ); // initialize random generator
    clear();
  }

  void clear() {
    for( size_t i=0; i< PINK_NOISE_NUM_STAGES; i++ )
      state[ i ] = 0.0;
    }

  float tick() {
    static const float RMI2 = 2.0 / float(RAND_MAX); // + 1.0; // change for range [0,
↪1)
    static const float offset = A[0] + A[1] + A[2];
```

(continues on next page)

(continued from previous page)

```

  // unrolled loop
    float temp = float( rand() );
    state[0] = P[0] * (state[0] - temp) + temp;
    temp = float( rand() );
    state[1] = P[1] * (state[1] - temp) + temp;
    temp = float( rand() );
    state[2] = P[2] * (state[2] - temp) + temp;
    return ( A[0]*state[0] + A[1]*state[1] + A[2]*state[2] )*RMI2 - offset;
  }

protected:
  float state[ PINK_NOISE_NUM_STAGES ];
  static const float A[ PINK_NOISE_NUM_STAGES ];
  static const float P[ PINK_NOISE_NUM_STAGES ];
};

const float PinkNoise::A[] = { 0.02109238, 0.07113478, 0.68873558 }; // rescaled by␣
↪(1+P)/(1-P)
const float PinkNoise::P[] = { 0.3190,  0.7756,  0.9613  };

#endif
```

1.54.1 Comments

- **Date**: 2007-02-09 06:15:59
- **By**: ten.knilhtrae@6741emmartl

```
Many thanks to David Lowenfels for posting this implementation of the early␣
↪experimental version. I recommend switching to the new algorithm form described in
↪'newpink.htm' -- better range to 9+ octaves, better accuracy to +-0.25 dB, and␣
↪leveled computational loading.  So where is MY submission to the archive? Um... ␣
↪well, it's coming... if he doesn't beat me to the punch again and post his code␣
↪first! -- Larry Trammell (the RidgeRat)
```

1.55 Waveform generator using MinBLEPS

- **Author or source:** ku.oc.nomed.nafgpr@ekcol
- **Created:** 2002-08-05 18:44:50
- **Linked files:** `MinBLEPS.zip`.

Listing 94: 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
```

(continues on next page)

(continued from previous page)

```
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.
```

Listing 95: code

```

```

1.55.1 Comments

- **Date**: 2004-07-02 22:31:36
- **By**: moc.oiduaesionetihw@ofni

```
http://www.slack.net/~ant/bl-synth/windowed-impulse/

This page also describes a similar algorithm for generating waves. Could the aliasing␣
↪be due to the fact that the blep only occurs after the discontinuity? On this page␣
↪the blep also occurs in the opposite direction as well, leading up to the␣
↪discontinuity.
```

- **Date**: 2008-02-11 18:42:15
- **By**: kernel[@}audiospillage.com

```
The sawtooth is a nice oscillator but I can't seem to get the square wave to work␣
↪properly.  Anyone else had any luck with this?  Also, it's worth noting that the␣
↪code assumes it is running on a little endian architecture.
```

- **Date**: 2009-07-07 04:45:40
- **By**: moc.enecslatnemirepxe@leinad

```
I have written GPLv3 C++ source code for a MinBLEP oscillator and also public domain␣
↪C++ source code for generating the MinBLEP without MatLab.

http://www.experimentalscene.com/articles/minbleps.php - Article and Code

http://www.experimentalscene.com/source.php - Look in DarkWave / latest version /␣
↪CoreMachines / VCO.cpp
```

1.56 Wavetable Synthesis

- **Author or source:** Robert Bristow-Johnson
- **Created:** 2002-05-07 18:46:18
- **Linked files:** `http://www.harmony-central.com/Synth/Articles/Wavetable_101/Wavetable-101.pdf`.

Listing 96: notes

```
Wavetable sythesis AES paper by RBJ.
```

1.57 Weird synthesis

- **Author or source:** Andy M00cho
- **Created:** 2002-01-17 00:55:09

Listing 97: 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...
```

1.57.1 Comments

- **Date**: 2002-07-18 18:57:00
- **By**: moc.loa@attongamlihp

```
Andy:
I'm curious about your delay line. It's length is
4.8 m.sec.fixed. What are the variables in the two controllable taps and is the 6dB␣
↪filter variable frequency wise?
Phil
```

- **Date**: 2003-01-03 20:01:34
- **By**: moc.oohay@poportcele

```
What you have there is the core of a physical modelling algorithm.  I have done
↪virtually the same thing to model plucked string instruments in Reaktor.  It's
↪amazingly realistic.  See http://joeorgren.com
```

CHAPTER 2

Analysis

2.1 Beat Detector Class

- **Author or source:** rf.eerf@retsaMPSD
- **Created:** 2005-05-12 14:35:52

Listing 1: notes

```
This class was designed for a VST plugin. Basically, it's just a 2nd order LP filter,
followed by an enveloppe detector (thanks Bram), feeding a Schmitt trigger. The rising
edge detector provides a 1-sample pulse each time a beat is detected. Code is self
documented...
Note : The class uses a fixed comparison level, you may need to change it.
```

Listing 2: code

```
// ***** BEATDETECTOR.H *****
#ifndef BeatDetectorH
#define BeatDetectorH

class TBeatDetector
{
private:
  float KBeatFilter;        // Filter coefficient
  float Filter1Out, Filter2Out;
  float BeatRelease;        // Release time coefficient
  float PeakEnv;            // Peak enveloppe follower
  bool BeatTrigger;         // Schmitt trigger output
  bool PrevBeatPulse;       // Rising edge memory
public:
  bool BeatPulse;           // Beat detector output

  TBeatDetector();
```

(continues on next page)

(continued from previous page)

```
  ~TBeatDetector();
  virtual void setSampleRate(float SampleRate);
  virtual void AudioProcess (float input);
};
#endif


// ***** BEATDETECTOR.CPP *****
#include "BeatDetector.h"
#include "math.h"

#define FREQ_LP_BEAT 150.0f    // Low Pass filter frequency
#define T_FILTER 1.0f/(2.0f*M_PI*FREQ_LP_BEAT)  // Low Pass filter time constant
#define BEAT_RTIME 0.02f  // Release time of enveloppe detector in second

TBeatDetector::TBeatDetector()
// Beat detector constructor
{
  Filter1Out=0.0;
  Filter2Out=0.0;
  PeakEnv=0.0;
  BeatTrigger=false;
  PrevBeatPulse=false;
  setSampleRate(44100);
}

TBeatDetector::~TBeatDetector()
{
  // Nothing specific to do...
}

void TBeatDetector::setSampleRate (float sampleRate)
// Compute all sample frequency related coeffs
{
  KBeatFilter=1.0/(sampleRate*T_FILTER);
  BeatRelease=(float)exp(-1.0f/(sampleRate*BEAT_RTIME));
}

void TBeatDetector::AudioProcess (float input)
// Process incoming signal
{
  float EnvIn;

  // Step 1 : 2nd order low pass filter (made of two 1st order RC filter)
  Filter1Out=Filter1Out+(KBeatFilter*(input-Filter1Out));
  Filter2Out=Filter2Out+(KBeatFilter*(Filter1Out-Filter2Out));

  // Step 2 : peak detector
  EnvIn=fabs(Filter2Out);
  if (EnvIn>PeakEnv) PeakEnv=EnvIn;  // Attack time = 0
  else
  {
    PeakEnv*=BeatRelease;
    PeakEnv+=(1.0f-BeatRelease)*EnvIn;
  }

  // Step 3 : Schmitt trigger
```

(continues on next page)

(continued from previous page)

```
  if (!BeatTrigger)
  {
    if (PeakEnv>0.3) BeatTrigger=true;
  }
  else
  {
    if (PeakEnv<0.15) BeatTrigger=false;
  }

  // Step 4 : rising edge detector
  BeatPulse=false;
  if ((BeatTrigger)&&(!PrevBeatPulse))
    BeatPulse=true;
  PrevBeatPulse=BeatTrigger;
}
```

2.1.1 Comments

- **Date**: 2005-05-18 22:59:08
- **By**: moc.yddaht@yddaht

```
// Nice work!
//Here's a Delphi and freepascal version:
unit beattrigger;

interface

type
TBeatDetector = class
private
  KBeatFilter,                // Filter coefficient
  Filter1Out,
  Filter2Out,
  BeatRelease,                // Release time coefficient
  PeakEnv:single;             // Peak enveloppe follower
  BeatTrigger,                // Schmitt trigger output
  PrevBeatPulse:Boolean;      // Rising edge memory
public
  BeatPulse:Boolean;              // Beat detector output
  constructor Create;
  procedure setSampleRate(SampleRate:single);
  procedure AudioProcess (input:single);
end;

function fabs(value:single):Single;

implementation


const
 FREQ_LP_BEAT = 150.0;                    // Low Pass filter frequency
 T_FILTER = 1.0/(2.0 * PI*FREQ_LP_BEAT);  // Low Pass filter time constant
 BEAT_RTIME = 0.02;   // Release time of enveloppe detector in second
```

(continues on next page)

(continued from previous page)

```
constructor TBeatDetector.create;
// Beat detector constructor
begin
  inherited;
  Filter1Out:=0.0;
  Filter2Out:=0.0;
  PeakEnv:=0.0;
  BeatTrigger:=false;
  PrevBeatPulse:=false;
  setSampleRate(44100);
end;


procedure TBeatDetector.setSampleRate (sampleRate:single);
// Compute all sample frequency related coeffs
begin
  KBeatFilter:=1.0/(sampleRate*T_FILTER);
  BeatRelease:= exp(-1.0/(sampleRate*BEAT_RTIME));
end;

function fabs(value:single):Single;
asm
 fld value
 fabs
 fwait
end;

procedure  TBeatDetector.AudioProcess (input:single);
var
 EnvIn:Single;
// Process incoming signal
begin
  // Step 1 : 2nd order low pass filter (made of two 1st order RC filter)
  Filter1Out:=Filter1Out+(KBeatFilter*(input-Filter1Out));
  Filter2Out:=Filter2Out+(KBeatFilter*(Filter1Out-Filter2Out));
  // Step 2 : peak detector
  EnvIn:=fabs(Filter2Out);
  if EnvIn>PeakEnv then PeakEnv:=EnvIn  // Attack time = 0
  else
  begin
    PeakEnv:=PeakEnv*BeatRelease;
    PeakEnv:=PeakEnv+(1.0-BeatRelease)*EnvIn;
  end;
  // Step 3 : Schmitt trigger
  if not BeatTrigger then
  begin
    if PeakEnv>0.3 then BeatTrigger:=true;
  end
  else
  begin
    if PeakEnv<0.15 then BeatTrigger:=false;
  end;

  // Step 4 : rising edge detector
  BeatPulse:=false;
  if (BeatTrigger = true ) and( not PrevBeatPulse) then
    BeatPulse:=true;
```

(continues on next page)

(continued from previous page)

```
  PrevBeatPulse:=BeatTrigger;
end;

end.
```

2.2 Coefficients for Daubechies wavelets 1-38

- **Author or source:** Computed by Kazuo Hatano, Compiled and verified by Olli Niemitalo
- **Type:** wavelet transform
- **Created:** 2002-01-17 02:00:43
- **Linked files:** `daub.h`.

2.3 DFT

- **Author or source:** Andy Mucho
- **Type:** fourier transform
- **Created:** 2002-01-17 01:59:38

Listing 3: 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;
}
```

2.4 Envelope detector

- **Author or source:** Bram
- **Created:** 2002-04-12 21:37:18

Listing 4: 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
```

Listing 5: code

```
//attack and release in seconds
float ga = (float) exp(-1/(SampleRate*attack));
float gr = (float) exp(-1/(SampleRate*release));

float envelope=0;

for(...)
{
  //get your data into 'input'
  EnvIn = std::abs(input);

  if(envelope < EnvIn)
  {
     envelope *= ga;
     envelope += (1-ga)*EnvIn;
  }
  else
  {
     envelope *= gr;
     envelope += (1-gr)*EnvIn;
  }
  //envelope now contains.........the envelope ;)
}
```

2.4.1 Comments

- **Date**: 2005-05-17 13:58:11
- **By**: moc.liamg@sisehtnysorpitna

```
// Slightly faster version of the envelope follower using one multiply form.

// attTime and relTime is in seconds

float ga = exp(-1.0f/(sampleRate*attTime));
float gr = exp(-1.0f/(sampleRate*relTime));

float envOut = 0.0f;

for( ... )
```

(continues on next page)

(continued from previous page)

```
{
    // get your data into 'input'

    envIn = fabs(input);

    if( envOut < envIn )
        envOut = envIn + ga * (envOut - envIn);
    else
        envOut = envIn + gr * (envOut - envIn);

    // envOut now contains the envelope
}
```

2.5 Envelope follower with different attack and release

- **Author or source:** Bram
- **Created:** 2003-01-15 00:21:39

Listing 6: notes

```
xxxx_in_ms is xxxx in milliseconds ;-)
```

Listing 7: code

```
init::

attack_coef = exp(log(0.01)/( attack_in_ms * samplerate * 0.001));
release_coef = exp(log(0.01)/( release_in_ms * samplerate * 0.001));
envelope = 0.0;

loop::

tmp = fabs(in);
if(tmp > envelope)
    envelope = attack_coef * (envelope - tmp) + tmp;
else
    envelope = release_coef * (envelope - tmp) + tmp;
```

2.5.1 Comments

- **Date**: 2003-01-18 20:56:46
- **By**: kd.utd.xaspmak@mj

```
// the expressions of the form:

xxxx_coef = exp(log(0.01)/( xxxx_in_ms * samplerate * 0.001));

// can be simplified a little bit to:

xxxx_coef = pow(0.01, 1.0/( xxxx_in_ms * samplerate * 0.001));
```

- **Date**: 2007-07-01 19:05:26
- **By**: uh.etle.fni@yfoocs

```
Here the definition of the attack/release time is the time for the envelope to fall␣
↪from 100% to 1%.
In the other version, the definition is for the envelope to fall from 100% to 36.7%.␣
↪So in this one
the envelope is about 4.6 times faster.
```

2.6 FFT

- **Author or source:** Toth Laszlo
- **Created:** 2002-02-11 17:43:15
- **Linked files:** `rvfft.ps`.
- **Linked files:** `rvfft.cpp`.

Listing 8: 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)."
```

2.6.1 Comments

- **Date**: 2011-01-22 20:56:46
- **By**: moc.oohay@ygobatem

```
Thank you very much, this was useful, and it worked right out of the box,
so to speak. It's very efficient, and the algorithm is readable. It also includes some
very useful functions.
```

2.7 FFT classes in C++ and Object Pascal

- **Author or source:** Laurent de Soras (Object Pascal translation by Frederic Vanmol)
- **Type:** Real-to-Complex FFT and Complex-to-Real IFFT
- **Created:** 2002-02-14 02:09:26
- **Linked files:** `FFTReal.zip`.

Listing 9: notes

```
(see linkfile)
```

2.8 Fast in-place Walsh-Hadamard Transform

- **Author or source:** Timo H Tossavainen
- **Type:** wavelet transform
- **Created:** 2002-01-17 01:54:52

Listing 10: 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)
```

Listing 11: 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))
```

(continues on next page)

(continued from previous page)

```
    {
        for (int k = 0; k < (1<<i); ++k)
        {
            wht_bfly (data [j + k], data [j + k + (1<<i)]);
        }
    }
  }
}
```

2.9 Frequency response from biquad coefficients

- **Author or source:** moc.feercinos@retep
- **Type:** biquad
- **Created:** 2004-11-29 09:49:47

Listing 12: notes

```
Here is a formula for plotting the frequency response of a biquad filter. Depending on the
coefficients that you have, you might have to use negative values for the b-coefficients.
```

Listing 13: code

```
//w = frequency (0 < w < PI)
//square(x) = x*x

y = 20*log((sqrt(square(a0*square(cos(w))-a0*square(sin(w))+a1*cos(w)+a2)+square(2*a0*cos(w)*sin(w)+a1*(sin(w))))/
            sqrt(square(square(cos(w))-square(sin(w))+b1*cos(w)+b2)+square(2*    cos(w)*sin(w)+b1*(sin(w))))));
```

2.9.1 Comments

- **Date**: 2006-03-16 19:36:32
- **By**: ude.drofnats.amrcc@lfd

```
this formula can have roundoff errors with frequencies close to zero... (especially a problem
with high samplerate filters)

here is a better formula:

from RBJ @ http://groups.google.com/group/comp.dsp/browse_frm/thread/8c0fa8d396aeb444/a1bc5b63ac56b686

 20*log10[|H(e^jw)|] =
 10*log10[ (b0+b1+b2)^2 - 4*(b0*b1 + 4*b0*b2 + b1*b2)*phi + 16*b0*b2*phi^2 ]
-10*log10[ (a0+a1+a2)^2 - 4*(a0*a1 + 4*a0*a2 + a1*a2)*phi + 16*a0*a2*phi^2 ]
```

(continues on next page)

(continued from previous page)

```
where phi = sin^2(w/2)
```

2.10 Java FFT

- **Author or source:** Loreno Heer
- **Type:** FFT Analysis
- **Created:** 2003-11-25 17:38:15

Listing 14: notes

```
May not work correctly ;-)
```

Listing 15: code

```
// WTest.java
/*
    Copyright (C) 2003 Loreno Heer, (helohe at bluewin dot ch)

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

*/
public class WTest{

    private static double[] sin(double step, int size){
            double f = 0;
            double[] ret = new double[size];
            for(int i = 0; i < size; i++){
                    ret[i] = Math.sin(f);
                    f += step;
            }
            return ret;
    }

    private static double[] add(double[] a, double[] b){
            double[] c = new double[a.length];
            for(int i = 0; i < a.length; i++){
                    c[i] = a[i] + b[i];
            }
            return c;
```

(continues on next page)

(continued from previous page)

```
    }

    private static double[] sub(double[] a, double[] b){
        double[] c = new double[a.length];
        for(int i = 0; i < a.length; i++){
            c[i] = a[i] - b[i];
        }
        return c;
    }

    private static double[] add(double[] a, double b){
        double[] c = new double[a.length];
        for(int i = 0; i < a.length; i++){
            c[i] = a[i] + b;
        }
        return c;
    }

    private static double[] cp(double[] a, int size){
        double[] c = new double[size];
        for(int i = 0; i < size; i++){
            c[i] = a[i];
        }
        return c;
    }

    private static double[] mul(double[] a, double b){
        double[] c = new double[a.length];
        for(int i = 0; i < a.length; i++){
            c[i] = a[i] * b;
        }
        return c;
    }

    private static void print(double[] value){
        for(int i = 0; i < value.length; i++){
            System.out.print(i + "," + value[i] + "\n");
        }
        System.out.println();
    }

    private static double abs(double[] a){
        double c = 0;
        for(int i = 0; i < a.length; i++){
            c = ((c * i) + Math.abs(a[i])) / (i + 1);
        }
        return c;
    }

    private static double[] fft(double[] a, int min, int max, int step){
        double[] ret = new double[(max - min) / step];
        int i = 0;
        for(int d = min; d < max; d = d + step){
            double[] f = sin(fc(d), a.length);
            double[] dif = sub(a, f);
            ret[i] = 1 - abs(dif);
            i++;
```

(continues on next page)

(continued from previous page)

```
            }
            return ret;
    }

    private static double[] fft_log(double[] a){
            double[] ret = new double[1551];
            int i = 0;
            for(double d = 0; d < 15.5; d = d + 0.01){
                    double[] f = sin(fc(Math.pow(2,d)), a.length);
                    double[] dif = sub(a, f);
                    ret[i] = Math.abs(1 - abs(dif));
                    i++;
            }
            return ret;
    }

    private static double fc(double d){
            return d * Math.PI / res;
    }

    private static void print_log(double[] value){
            for(int i = 0; i < value.length; i++){
                    System.out.print(Math.pow(2,((double)i/100d)) + "," + value[i] +
↪"\n");
            }
            System.out.println();
    }

    public static void main(String[] args){
            double[] f_0 = sin(fc(440), sample_length); // res / pi =>14005
            //double[] f_1 = sin(.02, sample_length);
            double[] f_2 = sin(fc(520), sample_length);
            //double[] f_3 = sin(.25, sample_length);

            //double[] f = add( add( add(f_0, f_1), f_2), f_3);

            double[] f = add(f_0, f_2);

            //print(f);

            double[] d = cp(f,1000);
            print_log(fft_log(d));
    }

    static double length = .2; // sec
    static int res = 44000; // resoultion (pro sec)
    static int sample_length = res; // resoultion

}
```

2.11 LPC analysis (autocorrelation + Levinson-Durbin recursion)

- **Author or source:** ten.enegatum@liam
- **Created:** 2004-04-07 09:37:51

Listing 16: notes

```
The autocorrelation function implements a warped autocorrelation, so that frequency
resolution can be specified by the variable 'lambda'.  Levinson-Durbin recursion
calculates autoregression coefficients a and reflection coefficients (for lattice
↪filter
implementation) K.  Comments for Levinson-Durbin function implement matlab version of
↪the
same function.

No optimizations.
```

Listing 17: code

```
//find the order-P autocorrelation array, R, for the sequence x of length L and
↪warping of lambda
//wAutocorrelate(&pfSrc[stIndex],siglen,R,P,0);
wAutocorrelate(float * x, unsigned int L, float * R, unsigned int P, float lambda)
{
    double  * dl = new double [L];
    double  * Rt = new double [L];
    double r1,r2,r1t;
    R[0]=0;
    Rt[0]=0;
    r1=0;
    r2=0;
    r1t=0;
    for(unsigned int k=0; k<L;k++)
    {
                    Rt[0]+=double(x[k])*double(x[k]);

                    dl[k]=r1-double(lambda)*double(x[k]-r2);
                    r1 = x[k];
                    r2 = dl[k];
    }
    for(unsigned int i=1; i<=P; i++)
    {
            Rt[i]=0;
            r1=0;
            r2=0;
            for(unsigned int k=0; k<L;k++)
            {
                    Rt[i]+=double(dl[k])*double(x[k]);

                    r1t = dl[k];
                    dl[k]=r1-double(lambda)*double(r1t-r2);
                    r1 = r1t;
                    r2 = dl[k];
            }
    }
    for(i=0; i<=P; i++)
            R[i]=float(Rt[i]);
    delete[] dl;
    delete[] Rt;
}

// Calculate the Levinson-Durbin recursion for the autocorrelation sequence R of
↪length P+1 and return the autocorrelation coefficients a and reflection
↪coefficients K
```

(continues on next page)

(continued from previous page)

```
LevinsonRecursion(unsigned int P, float *R, float *A, float *K)
{
    double Am1[62];

    if(R[0]==0.0) {
            for(unsigned int i=1; i<=P; i++)
            {
                    K[i]=0.0;
                    A[i]=0.0;
            }}
    else {
            double km,Em1,Em;
            unsigned int k,s,m;
            for (k=0;k<=P;k++){
                    A[0]=0;
                    Am1[0]=0; }
            A[0]=1;
            Am1[0]=1;
            km=0;
            Em1=R[0];
            for (m=1;m<=P;m++)                                  //m=2:N+1
            {
                    double err=0.0f;                                    //err = 0;
                    for (k=1;k<=m-1;k++)                    //for k=2:m-1
                            err += Am1[k]*R[m-k];           // err = err + am1(k)*R(m-k+1);
                    km = (R[m]-err)/Em1;                    //km=(R(m)-err)/Em1;
                    K[m-1] = -float(km);
                    A[m]=(float)km;                                     //am(m)=km;
                    for (k=1;k<=m-1;k++)                    //for k=2:m-1
                            A[k]=float(Am1[k]-km*Am1[m-k]);  // am(k)=am1(k)-km*am1(m-k+1);
                    Em=(1-km*km)*Em1;                                   //Em=(1-km*km)*Em1;
                    for(s=0;s<=P;s++)                                   //for s=1:N+1
                            Am1[s] = A[s];                      // am1(s) = am(s)
                    Em1 = Em;                                           //Em1 = Em;
            }
    }
    return 0;
}
```

2.11.1 Comments

- **Date**: 2005-03-31 15:16:20
- **By**: ed.luosfosruoivas@naitsirhC

```
// Blind Object Pascal Translation:
// --------------------------------

unit Levinson;
```

(continues on next page)

(continued from previous page)

```
interface

type
 TDoubleArray = array of Double;
 TSingleArray = array of Single;

implementation

//find the P-order autocorrelation array, R, for the sequence x of length L and␣
↪warping of lambda
procedure Autocorrelate(x,R : TSingleArray; P : Integer; lambda : Single; l: Integer␣
↪= -1);
var dl,Rt      : TDoubleArray;
    r1,r2,r1t : Double;
    k,i       : Integer;
begin
 // Initialization
 if l=-1 then l:=Length(x);
 SetLength(dl,l);
 SetLength(Rt,l);
 R[0]:=0;
 Rt[0]:=0;
 r1:=0;
 r2:=0;
 r1t:=0;

 for k:=0 to l-1 do
  begin
   Rt[0]:=Rt[0]+x[k]*x[k];
   dl[k]:=r1-lambda*(x[k]-r2);
   r1:= x[k];
   r2:= dl[k];
  end;

 for i:=1 to P do
  begin
   Rt[i]:=0;
   r1:=0;
   r2:=0;
   for k:=0 to L-1 do
    begin
     Rt[i]:=Rt[i]+dl[k]*x[k];
     r1t:= dl[k];
     dl[k]:=r1-lambda*(r1t-r2);
     r1:=r1t;
     r2:=dl[k];
    end;
  end;

 for i:=1 to P do R[i]:=Rt[i];
 setlength(Rt,0);
 setlength(dl,0);
end;

// Calculate the Levinson-Durbin recursion for the autocorrelation sequence
// R of length P+1 and return the autocorrelation coefficients a and reflection␣
↪coefficients K
```

(continues on next page)

(continued from previous page)

```
procedure LevinsonRecursion(P : Integer; R,A,K : TSingleArray);
var Am1       : TDoubleArray;
    i,j,s,m   : Integer;
    km,Em1,Em : Double;
    err       : Double;
begin
 SetLength(Am1,62);
 if (R[0]=0.0) then
  begin
   for i:=1 to P do
    begin
     K[i]:=0.0;
     A[i]:=0.0;
    end;
  end
 else
  begin
   for j:=0 to P do
    begin
     A[0]:=0;
     Am1[0]:=0;
    end;
   A[0]:=1;
   Am1[0]:=1;
   km:=0;
   Em1:=R[0];
   for m:=1 to P do
    begin
     err:=0.0;
     for j:=1 to m-1 do err:=err+Am1[j]*R[m-j];
     km:=(R[m]-err)/Em1;
     K[m-1]:=-km;
     A[m]:=km;
     for j:=1 to m-1 do A[j]:=Am1[j]-km*Am1[m-j];
     Em:=(1-km*km)*Em1;
     for s:=0 to P do Am1[s]:=A[s];
     Em1:=Em;
    end;
  end;
end;

end.
```

2.12 Look ahead limiting

- **Author or source:** Wilfried Welti
- **Created:** 2002-01-17 03:08:11

Listing 18: 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.
```

(continues on next page)

(continued from previous page)

```
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
```

Listing 19: 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;
}
```

(continues on next page)

(continued from previous page)

```
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);
}
```

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

- **Author or source:** George Yohng
- **Type:** magnitude and phase at any frequency
- **Created:** 2002-08-01 00:43:57

Listing 20: 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];
```

(continues on next page)

(continued from previous page)

```
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. :)
```

Listing 21: code

```c
/*
 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) +
```

(continues on next page)

(continued from previous page)

```
                (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);
}
```

2.13.1 Comments

- **Date**: 2004-01-02 08:46:35
- **By**: Rob

```
% Actually it is simpler to simply take the zero-padded b and a coefficients and do␣
↪real->complex
% FFT like this (matlab code):

H_complex=fft(b,N)./fft(a,N);
phase=angle(H_complex);
Magn=abs(H_complex);

% This will give you N/2 points from 0 to pi angle freq (or 0 to nyquist freq).
% /Rob
```

- **Date**: 2004-10-01 00:55:09
- **By**: ed.luosfosruoivas@naitsirhC

```
// Here are the formulas if you only have a biquad. But i am not sure, if maybe the
// phase is shifted with pi/2...

20*Log10(
         sqrt(
              (a0*a0+a1*a1+a2*a2+
               2*(a0*a1+a1*a2)*cos(w)+
```

(continues on next page)

(continued from previous page)

```
                 2*(a0*a2)* cos(2*w)
                )
                /
                (
                 1 + b1*b1 + b2*b2 +
                 2*(b1 + b1*b2)*cos(w)+
                 2*b2*cos(2*w)
                )
               )
         )


ArcTan2(
         (
          a0+a1*b1+a2*b2+
         (a0*b1+a1*(1+b2)+a2*b1)*cos(w)+
         (a0*b2+a2)*cos(2*w)
         )
         /
         (
          1+b1*b1+b2*b2+
          2*
          (
           (b1+b1*b2)*cos(w)+ b2*cos(2*w)
          )
         )
         ,
         (
          (
           a1-a0*b1+a2*b1-a1*b2+
           2*(-a0*b2+a2)*cos(w)
          )*sin(w)
          /
          (
           1+b1*b1+b2*b2+
           2*(b1 + b1*b2)*cos(w)+
           2*b2*cos(2*w)
          )
         )
        )
```

- **Date**: 2005-03-28 22:43:17
- **By**: ed.luosfosruoivas@naitsirhC

```
// Recursive Delphi Code with arbitrary order:

unit Plot;

interface

type TArrayOfDouble = Array of Double;

function MagnitudeCalc(f,rate : Double; a,b : TArrayOfDouble): Double;

implementation

```

(continues on next page)

(continued from previous page)

```
uses Math;

function MulVectCalc(const v: TArrayOfDouble; const Z, N : Integer) : Double;
begin
 if N=0
  then result:=0
  else result:=(v[N-1]*v[N-1+Z])+MulVectCalc(v,Z,N-1);
end;

function MagCascadeCalc(const v: TArrayOfDouble; const w : double; N, Order : Integer␣
↪): Double;
begin
 if N=1
  then result:=(MulVectCalc(v,0,Order))
  else result:=((MulVectCalc(v,N-1,1+Order-N)*(2*cos((N-1)*w))+MagCascadeCalc(v, w, N-
↪1, Order )));
end;

function MagnitudeCalc(f,rate : Double; a,b : TArrayOfDouble): Double;
var w : Double;
begin
 w:=(2*f*pi)/rate;
 result:=sqrt(MagCascadeCalc(a, w, Length(a),Length(a))/MagCascadeCalc(b, w,␣
↪Length(b),Length(b)));
end;

end.
```

- **Date**: 2005-07-27 12:39:52
- **By**: ed.luosfosruoivas@naitsirhC

```
function CalcMagPart(w: Double; C : TDoubleArray):Double;
var i,j,l : Integer;
    temp  : Double;
begin
 l:=Length(C);
 temp:=0;
 for j:=0 to l-1
  do temp:=temp+C[j]*C[j];
 result:=temp;
 for i:=1 to l-1 do
  begin
   temp:=0;
   for j:=0 to l-i-1
    do temp:=temp+C[j]*C[j+i];
   result:=Result+2*temp*cos(i*w);
  end;
end;

function CalcMagnitude_dB(const f,rate: Double; const A,B: TDoubleArray): Double;
var w : Double;
begin
 w:=(2*f*pi)/rate;
 result:=10*log10(CalcMagPart(w,A)/CalcMagPart(w,B));
end;
```

(continues on next page)

(continued from previous page)

```
// Here's a really fast function for an arbitrary IIR with high order without stack
↪overflows
// or recursion. And specially for John without sqrt.
```

2.14 Measuring interpollation noise

- **Author or source:** Jon Watte
- **Created:** 2002-01-17 02:00:09

Listing 22: 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.
```

2.15 QFT and DQFT (double precision) classes

- **Author or source:** Joshua Scholar
- **Created:** 2003-05-17 16:17:35
- **Linked files:** `qft.tar_1.gz`.

Listing 23: notes

```
Since it's a Visual C++ project (though it has relatively portable C++) I
guess the main audience are PC users.  As such I'm including a zip file.
Some PC users wouldn't know what to do with a tgz file.

The QFT and DQFT (double precision) classes supply the following functions:

  1. Real valued FFT and inverse FFT functions.  Note that separate arraysare used
     for real and imaginary component of the resulting spectrum.

  2. Decomposition of a spectrum into a separate spectrum of the evensamples
     and a spectrum of the odd samples.  This can be useful for buildingfilter banks.
```

(continues on next page)

(continued from previous page)

```
  3. Reconstituting a spectrum from separate spectrums of the even samples
     and odd samples.  This can be useful for building filter banks.

  4. A discrete Sin transform (a QFT decomposes an FFT into a DST and DCT).

  5. A discrete Cos transfrom.

  6. Since a QFT does it's last stage calculating from the outside in thelast part
     can be left unpacked and only calculated as needed in the case wherethe entire
  spectrum isn't needed (I used this for calculating correlations andconvolutions
  where I only needed half of the results).
  ReverseNoUnpack()
  UnpackStep()
  and NegUnpackStep()
  implement this functionality

  NOTE Reverse() normalizes its results (divides by one half the blocklength), but
  ReverseNoUnpack() does not.

  7. Also if you only want the first half of the results you can call ReverseHalf()

  NOTE Reverse() normalizes its results (divides by one half the blocklength), but
  ReverseHalf() does not.

  8. QFT is less numerically stable than regular FFTs.  With singleprecision␣
↪calculations,
     a block length of 2^15 brings the accuracy down to being barelyaccurate enough.
  At that size, single precision calculations tested sound files wouldoccasionally␣
↪have
  a sample off by 2, and a couple off by 1 per block. Full volume whitenoise would
generate
  a few samples off by as much as 6 per block at the end, beginning and middle.

  No matter what the inputs the errors are always at the same positions in the block.
  There some sort of cancelation that gets more delicate as the block size gets␣
↪bigger.

  For the sake of doing convolutions and the like where the forward transform is
  done only once for one of the inputs, I created a AccurateForward() function.
  It uses a regular FFT algorithm for blocks larger than 2^12, and decomposes into␣
↪even
and
  odd FFTs recursively.

  In any case you can always use the double precision routines to get more accuracy.
  DQFT even has routines that take floats as inputs and return double precision
  spectrum outputs.

As for portability:

1. The files qft.cpp and dqft.cpp start with defines:
#define _USE_ASM

If you comment those define out, then what's left is C++ with no assembly language.

2. There is unnecessary windows specific code in "criticalSection.h"
I used a critical section because objects are not reentrant (each object has
```

(continues on next page)

(continued from previous page)

```
permanent scratch pad memory), but obviously critical sections are operating
system specific.  In any case that code can easily be taken out.


If you look at my code and see that there's an a test built in the examples
that makes sure that the results are in the ballpark of being right. It
wasn't that I expected the answers to be far off, it was that I uncommenting
the "no assembly language" versions of some routines and I wanted to make
sure that they weren't broken.
```

2.16 Simple peak follower

- **Author or source:** Phil Burk
- **Type:** amplitude analysis
- **Created:** 2002-01-17 01:57:19

Listing 24: 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.
```

Listing 25: 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;
}
```

2.16.1 Comments

- **Date**: 2013-01-26 09:48:15
- **By**: moc.liamg@osoromaerfac

```
#ifndef VERY_SMALL_FLOAT
#define VERY_SMALL_FLOAT 1.0e-30F
#endif
```

2.17 Tone detection with Goertzel

- **Author or source:** on.biu.ii@rnepse
- **Type:** Goertzel
- **Created:** 2004-04-07 09:37:10
- **Linked files:** `http://www.ii.uib.no/~espenr/tonedetect.zip`.

Listing 26: 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.
```

Listing 27: 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>
```

(continues on next page)

(continued from previous page)

```

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);
    }
```

(continues on next page)

(continued from previous page)

```

    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;
}
```

2.17.1 Comments

- **Date**: 2004-04-12 22:03:56
- **By**: ed.luosfosruoivas@naitsirhC

```
// yet untested Delphi translation of the algorithm:

function Goertzel(Buffer:array of Single; frequency, samplerate: single):single;
var Skn, Skn1, Skn2 : Single;
    i               : Integer;
    temp1, temp2    : Single;
begin
 skn:=0;
 skn1:=0;
 skn2:=0;
 temp1:=2*PI*frequency/samplerate;
```

(continues on next page)

(continued from previous page)

```
 temp2:=Cos(temp1);
 for i:=0 to Length(Buffer) do
  begin
   Skn2 = Skn1;
   Skn1 = Skn;
   Skn = 2*temp2*Skn1 - Skn2 + Buffer[i];
  end;
 Result:=(Skn - exp(-temp1)*Skn1);
end;

// Maybe someone can use it...
//
// Christian
```

2.18 Tone detection with Goertzel (x86 ASM)

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** Tone detection with Goertzel in x86 assembly
- **Created:** 2004-06-27 22:43:46

Listing 28: notes

```
This is an "assemblified" version of the Goertzel Tone Detector. It is about 2 times
faster than the original code.

The code has been tested and it works fine.

Hope you can use it. I'm gonna try to build a Tuner (as VST-Plugin). I hope, that this
will work :-\ If anyone is intrested, please let me know.

Christian
```

Listing 29: code

```
function Goertzel_x87(Buffer :Psingle; BLength:Integer; frequency: Single;␣
↪samplerate: Single):Single;
asm
 mov ecx,BLength
 mov eax,Buffer
 fld x2
 fldpi
 fmulp
 fmul frequency
 fdiv samplerate
 fld st(0)
 fcos
 fld x2
 fmulp
 fxch st(1)
 fldz
 fsub st(0),st(1)
 fstp st(1)
```

(continues on next page)

(continued from previous page)

```

 fldl2e
 fmul
 fld st(0)
 frndint
 fsub st(1),st(0)
 fxch st(1)
 f2xm1
 fld1
 fadd
 fscale
 fstp st(1)

 fldz
 fldz
 fldz
@loopStart:
 fxch st(1)
 fxch st(2)
 fstp st(0)
 fld st(3)
 fmul st(0),st(1)
 fsub st(0),st(2)
 fld [eax].Single
 faddp
 add eax,4
 loop @loopStart
@loopEnd:

 fxch st(3)
 fmulp st(2), st(0)
 fsub st(0),st(1)
 fstp result
 ffree st(2)
 ffree st(1)
 ffree st(0)
end;
```

2.18.1 Comments

- **Date**: 2005-08-17 17:20:02
- **By**: moc.yddaht@yddaht

```
// Here's a variant on the theme that compensates for harmonics:

Function Goertzel(.Buffer: array of double; frequency, samplerate: double):.double;
var
Qkn, Qkn1, Qkn2, Wkn, Mk: double;
i: integer;
begin
Qkn:=0; Qkn1:=0;
Wkn:=2*.PI*.frequency/samplerate;
Mk:=2*.Cos(.Wkn);
for i:=0 to High(.Buffer) do begin
```

(continues on next page)

(continued from previous page)

```
  Qkn2: = Qkn1; Qkn1: = Qkn;
  Qkn  : = Buffer[.i ] + Mk*.Qkn1 - Qkn2;
end;
Result: = sqrt(.Qkn*.Qkn + Qkn1*.Qkn1 - Mk*.Qkn*.Qkn1);
end;

// Posted on www.delphimaster.ru by Jeer
```

2.19 Vintage VU meters tutorial

- **Author or source:** moc.liamg@321tiloen
- **Created:** 2009-03-10 15:24:04

Listing 30: notes

```
Here is a short tutorial about vintage-styled VU meters:

http://neolit123.blogspot.com/2009/03/designing-analog-vu-meter-in-dsp.html
```

CHAPTER 3

Filters

3.1 1 pole LPF for smooth parameter changes

- **Author or source:** moc.liamg@odiugoiz
- **Type:** 1-pole LPF class
- **Created:** 2008-09-22 20:27:06

Listing 1: notes

```
This is a very simple class that I'm using in my plugins for smoothing parameter␣
↪changes
that directly affect audio stream.
It's a 1-pole LPF, very easy on CPU.

Change the value of variable "a" (0~1) for slower or a faster response.

Of course you can also use it as a lowpass filter for audio signals.
```

Listing 2: code

```
class CParamSmooth
{
public:
    CParamSmooth() { a = 0.99f; b = 1.f - a; z = 0; };
    ~CParamSmooth();
    inline float Process(float in) { z = (in * b) + (z * a); return z; }
private:
    float a, b, z;
};
```

3.1.1 Comments

- **Date**: 2011-09-30 15:46:04
- **By**: moc.oohay@ygobatem

```
I've used this a lot, but here's an important thing: it won't work the same for multiple sample rates, so if you set a to 0.9995 for example, this will be lower if the sample rate is higher than you intended. I fix it by doing this:

/*must compensate this factor for sample rate change*/

float srCompensate;
srCompensate = sr/44100.0f;
float compensated_a;
compensated_a = powf(a, (1.0f/srCompensate));
            b = 1.0f-compensated_a;

Then if you start with a built in value (or range) designed for 44100hz, it will scale up with the sample rate so you will get the same amount of smoothing. I'm not sure if this is mathematically correct, but I came up with it very quickly and it works a charm for me.

Good work, very useful and easy to use filter.
```

- **Date**: 2011-09-30 15:47:29
- **By**: moc.oohay@ygobatem

```
*edit, a won't be LOWER if the sample rate changes, but it won't have the same effect.
```

- **Date**: 2014-12-16 12:14:59
- **By**: moc.liamg@earixela

```
New version, now you can specify the speed response of the parameter in ms. and sampling rate:

class CParamSmooth
{
public:
    CParamSmooth(float smoothingTimeInMs, float samplingRate)
    {
        const float c_twoPi = 6.283185307179586476925286766559f;
```

(continues on next page)

(continued from previous page)

```
        a = exp(-c_twoPi / (smoothingTimeInMs * 0.001f * samplingRate));
        b = 1.0f - a;
        z = 0.0f;
    }

    ~CParamSmooth()
    {

    }

    inline float process(float in)
    {
        z = (in * b) + (z * a);
        return z;
    }

private:
    float a;
    float b;
    float z;
};
```

3.2 1-RC and C filter

- **Author or source:** ac.nortoediv@niarbdam
- **Type:** Simple 2-pole LP
- **Created:** 2004-11-14 22:42:18

Listing 3: notes

```
This filter is called 1-RC and C since it uses these two parameters. C and R␣
↪correspond to
raw cutoff and inverted resonance, and have a range from 0 to 1.
```

Listing 4: code

```
//Parameter calculation
//cutoff and resonance are from 0 to 127

  c = pow(0.5, (128-cutoff)   / 16.0);
  r = pow(0.5, (resonance+24) / 16.0);

//Loop:

  v0 =  (1-r*c)*v0  -  (c)*v1  + (c)*input;
  v1 =  (1-r*c)*v1  +  (c)*v0;

  output = v1;
```

3.2.1 Comments

- **Date**: 2005-01-13 18:25:57
- **By**: yes

```
input is not in 0 - 1 range.

for cutoff i guess 128.

for reso the same ?
```

- **Date**: 2006-08-31 14:28:33
- **By**: uh.etle.fni@yfoocs

```
Nice. This is very similar to a state variable filter in many ways. Relationship␣
↪between c and frequency:

c = 2*sin(pi*freq/samplerate)

You can approximate this (tuning error towards nyquist):

c = 2*pi*freq/samplerate

Relationship between r and q factor:

r = 1/q

This filter has stability issues for high r values. State variable filter stability␣
↪limits seem to work fine here. It can also be oversampled for better stability and␣
↪wider frequency range (use 0.5*original frequency):

//Loop:

v0 = (1-r*c)*v0 - c*v1 + c*input;
v1 = (1-r*c)*v1 + c*v0;
tmp = v1;

v0 = (1-r*c)*v0 - c*v1 + c*input;
v1 = (1-r*c)*v1 + c*v0;
```

(continues on next page)

(continued from previous page)

```
output = (tmp+v1)*0.5;

-- peter schoffhauzer
```

3.3 18dB/oct resonant 3 pole LPF with tanh() dist

- **Author or source:** Josep M Comajuncosas
- **Created:** 2002-02-10 13:17:10
- **Linked files:** `lpf18.zip`.
- **Linked files:** `lpf18.sme`.

Listing 5: notes

```
Implementation in CSound and Sync Modular...
```

3.4 1st and 2nd order pink noise filters

- **Author or source:** moc.regnimmu@regnimmu
- **Type:** Pink noise
- **Created:** 2004-04-07 09:36:23

Listing 6: notes

```
Here are some new lower-order pink noise filter coefficients.

These have approximately equiripple error in decibels from 20hz to 20khz at a 44.1khz
sampling rate.

1st order, ~ +/- 3 dB error (not recommended!)
num = [0.05338071119116  -0.03752455712906]
den = [1.00000000000000  -0.97712493947102]

2nd order, ~ +/- 0.9 dB error
num = [ 0.04957526213389  -0.06305581334498   0.01483220320740 ]
den = [ 1.00000000000000  -1.80116083982126   0.80257737639225 ]
```

3.5 3 Band Equaliser

- **Author or source:** Neil C
- **Created:** 2006-08-29 20:34:25

Listing 7: notes

```
Simple 3 band equaliser with adjustable low and high frequencies ...

Fairly fast algo, good quality output (seems to be accoustically transparent with all
gains set to 1.0)

How to use ...

1. First you need to declare a state for your eq

  EQSTATE eq;

2. Now initialise the state (we'll assume your output frequency is 48Khz)

  set_3band_state(eq,880,5000,480000);

  Your EQ bands are now as follows (approximatley!)

  low  band = 0Hz to 880Hz
  mid  band = 880Hz to 5000Hz
  high band = 5000Hz to 24000Hz

3. Set the gains to some values ...

  eq.lg = 1.5;  // Boost bass by 50%
  eq.mg = 0.75; // Cut mid by 25%
  eq.hg = 1.0;  // Leave high band alone

4. You can now EQ some samples

   out_sample = do_3band(eq,in_sample)


Have fun and mail me if any problems ...  etanza at lycos dot co dot uk


Neil C / Etanza Systems, 2006 :)
```

Listing 8: code

```
First the header file ....
//---------------------------------------------------------------------------
//
//                                3 Band EQ :)
//
// EQ.H - Header file for 3 band EQ
//
// (c) Neil C / Etanza Systems / 2K6
//
// Shouts / Loves / Moans = etanza at lycos dot co dot uk
//
// This work is hereby placed in the public domain for all purposes, including
// use in commercial applications.
//
// The author assumes NO RESPONSIBILITY for any problems caused by the use of
// this software.
//
```

(continues on next page)

(continued from previous page)

```
//----------------------------------------------------------------------------

#ifndef __EQ3BAND__
#define __EQ3BAND__


// ------------
//| Structures |
// ------------

typedef struct
{
  // Filter #1 (Low band)

  double  lf;       // Frequency
  double  f1p0;     // Poles ...
  double  f1p1;
  double  f1p2;
  double  f1p3;

  // Filter #2 (High band)

  double  hf;       // Frequency
  double  f2p0;     // Poles ...
  double  f2p1;
  double  f2p2;
  double  f2p3;

  // Sample history buffer

  double  sdm1;     // Sample data minus 1
  double  sdm2;     //                   2
  double  sdm3;     //                   3

  // Gain Controls

  double  lg;       // low  gain
  double  mg;       // mid  gain
  double  hg;       // high gain

} EQSTATE;


// ---------
//| Exports |
// ---------

extern void   init_3band_state(EQSTATE* es, int lowfreq, int highfreq, int mixfreq);
extern double do_3band(EQSTATE* es, double sample);


#endif // #ifndef __EQ3BAND__
//---------------------------------------------------------------------------

Now the source ...
//----------------------------------------------------------------------------
//
```

(continues on next page)

(continued from previous page)

```
//                                3 Band EQ :)
//
// EQ.C - Main Source file for 3 band EQ
//
// (c) Neil C / Etanza Systems / 2K6
//
// Shouts / Loves / Moans = etanza at lycos dot co dot uk
//
// This work is hereby placed in the public domain for all purposes, including
// use in commercial applications.
//
// The author assumes NO RESPONSIBILITY for any problems caused by the use of
// this software.
//
//----------------------------------------------------------------------------

// NOTES :
//
// - Original filter code by Paul Kellet (musicdsp.pdf)
//
// - Uses 4 first order filters in series, should give 24dB per octave
//
// - Now with P4 Denormal fix :)


//----------------------------------------------------------------------------

// ----------
//| Includes |
// ----------

#include <math.h>
#include "eq.h"


// -----------
//| Constants |
// -----------

static double vsa = (1.0 / 4294967295.0);   // Very small amount (Denormal Fix)


// ---------------
//| Initialise EQ |
// ---------------

// Recommended frequencies are ...
//
//  lowfreq  = 880  Hz
//  highfreq = 5000 Hz
//
// Set mixfreq to whatever rate your system is using (eg 48Khz)

void init_3band_state(EQSTATE* es, int lowfreq, int highfreq, int mixfreq)
{
  // Clear state

```

(continues on next page)

(continued from previous page)

```
  memset(es,0,sizeof(EQSTATE));

  // Set Low/Mid/High gains to unity

  es->lg = 1.0;
  es->mg = 1.0;
  es->hg = 1.0;

  // Calculate filter cutoff frequencies

  es->lf = 2 * sin(M_PI * ((double)lowfreq / (double)mixfreq));
  es->hf = 2 * sin(M_PI * ((double)highfreq / (double)mixfreq));
}


// ---------------
//| EQ one sample |
// ---------------

// - sample can be any range you like :)
//
// Note that the output will depend on the gain settings for each band
// (especially the bass) so may require clipping before output, but you
// knew that anyway :)

double do_3band(EQSTATE* es, double sample)
{
  // Locals

  double  l,m,h;      // Low / Mid / High - Sample Values

  // Filter #1 (lowpass)

  es->f1p0  += (es->lf * (sample   - es->f1p0)) + vsa;
  es->f1p1  += (es->lf * (es->f1p0 - es->f1p1));
  es->f1p2  += (es->lf * (es->f1p1 - es->f1p2));
  es->f1p3  += (es->lf * (es->f1p2 - es->f1p3));

  l          = es->f1p3;

  // Filter #2 (highpass)

  es->f2p0  += (es->hf * (sample   - es->f2p0)) + vsa;
  es->f2p1  += (es->hf * (es->f2p0 - es->f2p1));
  es->f2p2  += (es->hf * (es->f2p1 - es->f2p2));
  es->f2p3  += (es->hf * (es->f2p2 - es->f2p3));

  h          = es->sdm3 - es->f2p3;

  // Calculate midrange (signal - (low + high))

  m          = es->sdm3 - (h + l);

  // Scale, Combine and store

  l         *= es->lg;
  m         *= es->mg;
```

(continues on next page)

(continued from previous page)

```
  h          *= es->hg;

  // Shuffle history buffer

  es->sdm3   = es->sdm2;
  es->sdm2   = es->sdm1;
  es->sdm1   = sample;

  // Return result

  return(l + m + h);
}


//----------------------------------------------------------------------------
```

3.5.1 Comments

- **Date**: 2007-03-28 03:33:04
- **By**: moc.mot@lx_iruy

```
Great Thanks!
I have one problem the below:
  double  f2p0;     // Poles ...
  double  f2p1;
  double  f2p2;
  double  f2p3;
that I want to know the starting value
about f2p0,f2p1,...!
```

- **Date**: 2007-04-14 12:02:27
- **By**: moc.oohay@knuf_red_retavttog_nuarb_semaj

```
yuri:

The invocation of memset() during the initialization method sets all the the members␣
↪of the struct to zero.
```

- **Date**: 2007-05-22 19:05:47
- **By**: moc.liamtoh@cnamlleh

```
This is great -- I want to develop a compressor/limiter/expander and have been␣
↪looking long and hard for bandpass / eq filtering code.  Here it is!

I am sure we could easily expand this into an x band eq.

Thanks!
```

- **Date**: 2007-07-05 06:49:45
- **By**: tom tom

```
Hi !

I've just transposed your code under Delphi.

It works well if the gain is under 1, but if i put gain > 1 i get clipping (annoying sound clips), even at 1.1;

Is it normal ?

I convert my smallint (44100 16 bits) to double before process, and convert the obtained value back to smallint with clipping (if < -32768 i set it to -32768, and if > 32768 i set it to 32768).

What did i do wrong ?

Regards

Tom
```

- **Date**: 2007-07-21 14:24:53
- **By**: ed.xmg@7trebreh

```
Hi.

Maybe the answer is quite easy. The upper limit is 32767 not 32768.

Regards

Herbert
```

- **Date**: 2007-08-23 02:39:23
- **By**: moc.oohay@3617100aggna

```
Hi, Can U send me a full source code for this 3 band state eq from start to end ??
Please !!!!
I really need it for my study in school.
I hope you can send me, to my email.

thanks you.
regard


angga
```

- **Date**: 2009-05-05 16:37:21
- **By**: moc.liamg@2156niahv

```
How can I expand this 3 Band EQ into X Band EQ..?!

Anybody answer me, or email me..
```

- **Date**: 2009-05-22 13:27:00
- **By**: moc.boohay@bob

```
For more bands, you could take the low-pass and repeat the process on that.
```

- **Date**: 2009-05-23 18:46:04
- **By**: moc.yabtsalb@pilihp

```
This is a great little filter, I am using it in an application but when I first␣
↪started playing with it I noticed some problems. The mid range didn't seem to be␣
↪calculated properly, a friend of mine who knows more about dsp than I do took a␣
↪quick look at it and suggested the following change:

  m          = es->sdm3 - (h + l);

Should be:

  m          = sample - (h + l);

I've tested it with this small fix and everything works perfectly now. Just thought I
↪'d bring this to your attention... Thanks for a great code snippet!
```

- **Date**: 2009-05-24 15:27:11
- **By**: moc.boohay@bob

```
What problems were you getting? Doesn't removing the delay cause phase problems?
```

- **Date**: 2009-06-25 10:34:44
- **By**: moc.limagy@ec.rahceb

```
Hi Great Stuff,

how to create 6 band equalizer, is any algorithm for 6 band same like 3 band, please␣
↪help me if any one

thanks in advance
```

- **Date**: 2010-05-06 20:21:20
- **By**: moc.liamg@anesejiw

```
How to extend this to 6 band equalizer?
```

- **Date**: 2010-05-27 21:45:44
- **By**: moc.liamtoh@aanaibas_nairam

```
hello! thanks for your code!!
i tried to use the code in my project of guitar distortions in real time (in C) and i␣
↪could'nt, i'm in linux using jack audio server, and it starts to have x-runs␣
↪everytime i turn on the equalizer. do you any idea of how solving this? (from 5 to␣
↪50 milliseconds o x-runs)
i was thinking of coding it in assembler but i don't know if that would be the␣
↪solution.
excuse me for my english, i'm from argentina and it's been a while since i last wrote␣
↪in this language!
thanks in advance, hoping to see any answer!
mariano
```

- **Date**: 2010-11-03 14:19:25

- **By**: moc.oohay@56_drow

```
Dev c++ can not run it? any suggesstions, how to run it?
```

- **Date**: 2011-02-23 13:24:53
- **By**: moc.liamg@kniniurb.tnecniv

```
This example is exactly what i've been looking for. This little piece of code␣
↪executes faster then the one I have been using before.

FYI,

I will use the code in a 3 band compressor / limiter / clipper for FM broadcasting.

Thanks for sharing.

Best regards,

Vincent Bruinink.
```

- **Date**: 2011-03-21 18:03:15
- **By**: moc.liamg@liamtsil.mtp

```
I've got this filtering audio on iOS by running my sample through do_3band in the␣
↪render callback. However, I'm getting a fair amount of distortion. Here's an␣
↪example with my EQ3Band gains all set to 1.0:

http://www.youtube.com/watch?v=W_6JaNUvUjA

Here's the code for my implementation:

https://github.com/tassock/mixerhost/commit/4b8b87028bfffe352ed67609f747858059a3e89b

I assume others using this aren't having this same distortion issue? If so, what sort␣
↪of audio sample formats are you using (big/little endian, float/integer samples,␣
↪etc). Thanks!
```

- **Date**: 2011-04-30 20:17:58
- **By**: moc.liamg@enohpi.senarab

```
hi,
I got also distorsion even though my all my gains are set to 1. Please help

Lucie
```

- **Date**: 2012-06-05 07:19:56
- **By**: moc.liamg@solbaidcod

```
      I've made an implementation of 3 band Equalizer to read a wave file,apply␣
↪filtering and then save wave  outputfile.
With your code I have a lot of distortion,I think the problem maybe coefficient␣
↪calculation:

es->f1p0  += (es->lf * (sample   - es->f1p0))+vsa ;
  ...
```

(continues on next page)

(continued from previous page)

```


Can anyone resolve distortion?
```

- **Date**: 2013-01-11 16:43:12
- **By**: es.dnargvelk@nahoj

```
Works fine for me on iOS. Maybe you feed a interleaved stereo signal with the same␣
↪EQSTATE instance (you'll need one EQSTATE for each channel)?
```

- **Date**: 2013-04-07 17:50:46
- **By**: ta.erehthgir@liameon

```
It's all about WHERE you init you EQ.  Try a little :) If you don't find out yourself,
↪ I'll help you.
```

- **Date**: 2013-06-18 14:00:45
- **By**: moc.laimtoh@inaam_riamu

```
I added this code to my xcode project. But where i can pass the values and method.␣
↪May be its funny question for you guyz but please help me. Its loking good to me␣
↪but in xcode , i am using avaudio player for play sound and making sound app.
Thanks
```

- **Date**: 2013-07-01 23:07:05
- **By**: moc.liamg@iaznabi

```
The distortion will occur of you are trying to adapt this to stereo and do so by␣
↪simply adding an outer loop per channel. Doing so will cause the filter values to␣
↪compound and cause distortion. Instead, you will need to duplicate all the filter␣
↪values and keep them separate from the other channel.
```

- **Date**: 2013-08-21 14:09:08
- **By**: moc.liamg@nurb.luap

```
Any good x-band equalizer equivalents for C# that I can use?
```

- **Date**: 2014-08-05 22:57:07
- **By**: ed.xmg@retsneum.ellak

```
s'cool thanks!
have portet to c#
works fine!
```

- **Date**: 2015-09-28 07:53:37
- **By**: az.oc.sseccadipar@sook

```
Converted Neil's C Code 3 band equalizer to Delphi class, for those who are␣
↪interested.
```

(continues on next page)

(continued from previous page)

```
1. Create instance of class
   Public
     eq:TEQ;

2. On form create

    eq := TEq.Create;

    //Initialize
    init_3band_state(880,5000,44100,50,-25,0);

    //init_3band_state(lowfreq,highfreq,mixfreq:integer;BassGain,MidGain,
↪HighGain:Double);


3. process: pass Raw 16Bit PCM to eq

eq.Equalize(const Data: Pointer; DataSize: DWORD);

Works like a charm form me

***********************************************
  TEq=Class
  private
    lf,f1p0,f1p1,f1p2,f1p3:double;
    hf,f2p0,f2p1,f2p2,f2p3:double;
    sdm1,sdm2,sdm3:double;
    vsa:double;
    lg,mg,hg:double;
    Function do_3band(sample:Smallint):Smallint;
  public
    constructor create;
    destructor destroy;override;
    procedure init_3band_state(lowfreq,highfreq,mixfreq:integer;BassGain,MidGain,
↪HighGain:Double);
    procedure Equalize(const Data: Pointer; DataSize: DWORD);
  end;


constructor TEQ.create;
begin
  inherited create;
  vsa := (1.0 / 4294967295.0);
  lg := 1.0;
  mg := 1.0;
  hg := 1.0;

end;


destructor TEQ.destroy;
begin
  inherited destroy;
end;

procedure TEQ.init_3band_state(lowfreq,highfreq,mixfreq:integer;BassGain,MidGain,
↪HighGain:Double);
```

(continues on next page)

(continued from previous page)

```
begin

 {(880,5000,44100,1.5,0.75,1.0)
 eq.lg = 1.5; // Boost bass by 50%
 eq.mg = 0.75; // Cut mid by 25%
 eq.hg = 1.0; // Leave high band alone }

 lg := 1+(BassGain/100);
 mg := 1+(MidGain/100);
 hg := 1+(HighGain/100);

// Calculate filter cutoff frequencies

 lf := 2 * sin(PI * (lowfreq / mixfreq));
 hf := 2 * sin(PI * (highfreq / mixfreq));
end;

Function TEQ.do_3band(sample:Smallint):Smallint;
var l,m,h:double;
    res:integer;
begin

// Filter #1 (lowpass)

  f1p0 := f1p0 + (lf * (sample - f1p0)) + vsa;
  f1p1 := f1p1 + (lf * (f1p0 - f1p1));
  f1p2 := f1p2 + (lf * (f1p1 - f1p2));
  f1p3 := f1p3 + (lf * (f1p2 - f1p3));

  l := f1p3;

// Filter #2 (highpass)

  f2p0 := f2p0 + (hf * (sample - f2p0)) + vsa;
  f2p1 := f2p1 + (hf * (f2p0 - f2p1));
  f2p2 := f2p2 + (hf * (f2p1 - f2p2));
  f2p3 := f2p3 + (hf * (f2p2 - f2p3));

  h := sdm3 - f2p3;

// Calculate midrange (signal - (low + high))

  m := sdm3 - (h + l);

// Scale, Combine and store

  l := l * lg;
  m := m * mg;
  h := h * hg;

// Shuffle history buffer

  sdm3 := sdm2;
  sdm2 := sdm1;
  sdm1 := sample;

// Return result
```

(continues on next page)

(continued from previous page)

```
  res := trunc(l+m+h);
  if res > 32767 then res := 32767 else if res < -32768 then res := -32768;

  result := res;
end;

procedure TEQ.Equalize(const Data: Pointer; DataSize: DWORD);
var pSample: PSmallInt;
begin
  pSample := Data;
  while DataSize > 0 do
  begin
    pSample^ := do_3band(pSample^);
    Inc(pSample);
    Dec(DataSize, 2);
  end;
end;
```

3.6 303 type filter with saturation

- **Author or source:** Hans Mikelson
- **Type:** Runge-Kutta Filters
- **Created:** 2002-01-17 02:07:37
- **Linked files:** `filters001.txt`.

Listing 9: 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!!
```

3.7 4th order Linkwitz-Riley filters

- **Author or source:** moc.liamg@321tiloen
- **Type:** LP/HP - LR4
- **Created:** 2009-05-17 19:43:06

Listing 10: notes

```
Original from T. Lossius - ttblue project

Optimized version in pseudo-code.

[! The filter is unstable for fast automation changes in the lower frequency range.
```

(continues on next page)

(continued from previous page)

```
Parameter interpolation and/or oversampling should fix this. !]

The sum of the Linkwitz-Riley (Butterworth squared) HP and LP outputs, will result an all-
pass filter at Fc and flat magnitude response - close to ideal for crossovers.

Lubomir I. Ivanov
```

Listing 11: code

```
//-----------------------------------------
// [code]
//-----------------------------------------

//fc -> cutoff frequency
//pi -> 3.14285714285714
//srate -> sample rate

//================================================
// shared for both lp, hp; optimizations here
//================================================
wc=2*pi*fc;
wc2=wc*wc;
wc3=wc2*wc;
wc4=wc2*wc2;
k=wc/tan(pi*fc/srate);
k2=k*k;
k3=k2*k;
k4=k2*k2;
sqrt2=sqrt(2);
sq_tmp1=sqrt2*wc3*k;
sq_tmp2=sqrt2*wc*k3;
a_tmp=4*wc2*k2+2*sq_tmp1+k4+2*sq_tmp2+wc4;

b1=(4*(wc4+sq_tmp1-k4-sq_tmp2))/a_tmp;
b2=(6*wc4-8*wc2*k2+6*k4)/a_tmp;
b3=(4*(wc4-sq_tmp1+sq_tmp2-k4))/a_tmp;
b4=(k4-2*sq_tmp1+wc4-2*sq_tmp2+4*wc2*k2)/a_tmp;

//================================================
// low-pass
//================================================
a0=wc4/a_tmp;
a1=4*wc4/a_tmp;
a2=6*wc4/a_tmp;
a3=a1;
a4=a0;

//=====================================================
// high-pass
//=====================================================
a0=k4/a_tmp;
a1=-4*k4/a_tmp;
a2=6*k4/a_tmp;
a3=a1;
a4=a0;
```

(continues on next page)

(continued from previous page)

```

//=====================================================
// sample loop - same for lp, hp
//=====================================================
tempx=input;

tempy=a0*tempx+a1*xm1+a2*xm2+a3*xm3+a4*xm4-b1*ym1-b2*ym2-b3*ym3-b4*ym4;
xm4=xm3;
xm3=xm2;
xm2=xm1;
xm1=tempx;
ym4=ym3;
ym3=ym2;
ym2=ym1;
ym1=tempy;

output=tempy;
```

3.7.1 Comments

- **Date**: 2009-05-29 11:09:50
- **By**: moc.liamg@321tiloen

```
LR2 with DFII:

//-------------------------------
// LR2
// fc -> cutoff frequency
// pi -> 3.14285714285714
// srate -> sample rate
//-------------------------------
fpi = pi*fc;
wc = 2*fpi;
wc2 = wc*wc;
wc22 = 2*wc2;
k = wc/tan(fpi/srate);
k2 = k*k;
k22 = 2*k2;
wck2 = 2*wc*k;
tmpk = (k2+wc2+wck2);
//b shared
b1 = (-k22+wc22)/tmpk;
b2 = (-wck2+k2+wc2)/tmpk;
//---------------
// low-pass
//---------------
a0_lp = (wc2)/tmpk;
a1_lp = (wc22)/tmpk;
a2_lp = (wc2)/tmpk;
//----------------
// high-pass
//----------------
a0_hp = (k2)/tmpk;
a1_hp = (-k22)/tmpk;
```

(continues on next page)

(continued from previous page)

```
a2_hp = (k2)/tmpk;

//==========================
// sample loop, in -> input
//==========================
//---lp
lp_out = a0_lp*in + lp_xm0;
lp_xm0 = a1_lp*in - b1*lp_out + lp_xm1;
lp_xm1 = a2_lp*in - b2*lp_out;
//---hp
hp_out = a0_hp*in + hp_xm0;
hp_xm0 = a1_hp*in - b1*hp_out + hp_xm1;
hp_xm1 = a2_hp*in - b2*hp_out;

// the two are with 180 degrees phase shift,
// so you need to invert the phase of one.
out = lp_out + hp_out*(-1);

//result is allpass at Fc
```

- **Date**: 2011-07-13 11:48:20
- **By**: moc.kauqkiuq@evad

```
I've converted this Linkwits Riley 4 into intrinsics. It's set up with the cross over␣
↪point and the sample rate. The function 'ProcessSplit' returns the low and high␣
↪parts. It uses _mm_malloc to align the variables to 16 bytes, as putting them into␣
↪the class as __m128 vars doesn't guarantee alignment.
Enjoy!  :)


//------------------------------------------------
//------------------------------------------------
//------------------------------------------------
//FIL_Linkwitz_Riley4.h

#pragma once

#include <xmmintrin.h>

class   FIL_Linkwitz_Riley4
{
    __m128 *ab;
    __m128 *al;
    __m128 *ah;
    __m128 *xm;
    __m128 *yml;
    __m128 *ymh;

    float a0l;
    float a0h;

public:

    FIL_Linkwitz_Riley4::FIL_Linkwitz_Riley4(float fc, float srate);
    ~FIL_Linkwitz_Riley4();
```

(continues on next page)

(continued from previous page)

```
    void ResetSplit();

    __inline void FIL_Linkwitz_Riley4::ProcessSplit(const float in, float &low, float␣
↪&high)
    {
            __m128 m1;

            m1 = _mm_sub_ps(_mm_mul_ps(*al, *xm), _mm_mul_ps(*ab, *yml));
            low  = a0l * in + m1.m128_f32[0] + m1.m128_f32[1] + m1.m128_f32[2] + m1.
↪m128_f32[3];

            m1 = _mm_sub_ps(_mm_mul_ps(*ah, *xm), _mm_mul_ps(*ab, *ymh));
            high = a0h * in + m1.m128_f32[0] + m1.m128_f32[1] + m1.m128_f32[2] + m1.
↪m128_f32[3];

            *xm  = _mm_shuffle_ps(*xm, *xm, _MM_SHUFFLE(2,1,0,0));
            (*xm).m128_f32[0] = in;
            *yml = _mm_shuffle_ps(*yml, *yml, _MM_SHUFFLE(2,1,0,0));
            (*yml).m128_f32[0] = low;
            *ymh = _mm_shuffle_ps(*ymh, *ymh, _MM_SHUFFLE(2,1,0,0));
            (*ymh).m128_f32[0] = high;
    }


};

//-----------------------------------------------
//-----------------------------------------------
//-----------------------------------------------


// FIL_Linkwitz_Riley4.cpp

#include "FIL_Linkwitz_Riley4.h"
#include <math.h>


FIL_Linkwitz_Riley4::FIL_Linkwitz_Riley4(float fc, float srate)
{
    ab  = (__m128*)_mm_malloc(16, 16);
    al  = (__m128*)_mm_malloc(16, 16);
    ah  = (__m128*)_mm_malloc(16, 16);
    xm  = (__m128*)_mm_malloc(16, 16);
    yml = (__m128*)_mm_malloc(16, 16);
    ymh = (__m128*)_mm_malloc(16, 16);

    float wc  = 2.0f * PI * fc;
    float wc2 = wc*wc;
    float wc3 = wc2*wc;
    float wc4 = wc2*wc2;
    float k  = wc / tanf(PI * fc / srate);
    float k2 = k*k;
    float k3 = k2*k;
    float k4 = k2*k2;
    float sqrt2   = sqrtf(2.0f);
    float sq_tmp1 = sqrt2 *wc3 * k;
    float sq_tmp2 = sqrt2 *wc * k3;
```

(continues on next page)

(continued from previous page)

```
    float a_tmp       = 4.0f * wc2 * k2 + 2.0f * sq_tmp1 + k4 + 2.0f * sq_tmp2 + wc4;

    (*ab).m128_f32[0] = (4.0f *(wc4+sq_tmp1-k4-sq_tmp2))/a_tmp;
    (*ab).m128_f32[1] = (6.0f *wc4-8*wc2*k2+6*k4)/a_tmp;
    (*ab).m128_f32[2] = (4.0f *(wc4-sq_tmp1+sq_tmp2-k4))/a_tmp;
    (*ab).m128_f32[3] = (k4 -2.0f * sq_tmp1 + wc4 - 2.0f * sq_tmp2 + 4.0f * wc2 * k2)␣
↪/ a_tmp;

    //================================================
    // low-pass
    //================================================
    a0l       = wc4/a_tmp;
    (*al).m128_f32[0] = 4.0f * wc4 / a_tmp;
    (*al).m128_f32[1] = 6.0f * wc4 / a_tmp;
    (*al).m128_f32[2] = (*al).m128_f32[0];
    (*al).m128_f32[3] = a0l;

    //======================================================
    // high-pass
    //======================================================
    a0h       = k4 / a_tmp;
    (*ah).m128_f32[0] = -4.0f * k4 / a_tmp;
    (*ah).m128_f32[1] =  6.0f * k4 / a_tmp;
    (*ah).m128_f32[2] = (*ah).m128_f32[0];
    (*ah).m128_f32[3] = a0h;

    ResetSplit();
}

FIL_Linkwitz_Riley4::~FIL_Linkwitz_Riley4()
{
    _mm_free((void*)ab);
    _mm_free((void*)al);
    _mm_free((void*)ah);
    _mm_free((void*)xm);
    _mm_free((void*)yml);
    _mm_free((void*)ymh);
}


void FIL_Linkwitz_Riley4::ResetSplit()
{
    // Reset history...
    *xm  = _mm_set1_ps(0.0f);
    *yml = _mm_set1_ps(0.0f);
    *ymh = _mm_set1_ps(0.0f);
}
//-----------------------------------------------
//-----------------------------------------------
//-----------------------------------------------
```

- **Date**: 2011-07-13 15:00:08
- **By**: moc.kauqkiuq@evad

```
I've no idea why I'm accessing those pointers like that! But never mind. :)
```

- **Date**: 2012-07-04 13:45:19

- **By**: ac.cisum-mutnauq@noidc

```
Your pi value is wrong:
pi -> 3.14285714285714

It should be 3.1415692 ect.
```

- **Date**: 2012-12-31 15:10:16
- **By**: moc.kauqkiuq@evad

```
Or even 3.1415926535!
LOL.
```

- **Date**: 2013-07-29 09:35:15
- **By**: moc.snoitcudorpnrec@mij

```
I don't think this is unstable for changes in frequency.  It's unstable for low␣
↪frequencies.

Here's my implementation.


#include <iostream>
#include <stdio.h>
#include <math.h>
#include <assert.h>

class LRCrossoverFilter { // LR4 crossover filter
private:
    struct filterCoefficents {
        float a0, a1, a2, a3, a4;
    } lpco, hpco;

    float b1co, b2co, b3co, b4co;

    struct {
        float xm1 = 0.0f;
        float xm2 = 0.0f;
        float xm3 = 0.0f;
        float xm4 = 0.0f;
        float ym1 = 0.0f, ym2 = 0.0f, ym3 = 0.0f, ym4 = 0.0f;
    } hptemp, lptemp;

    float coFreqRunningAv = 100.0f;
public:
    void setup(float crossoverFrequency, float sr);
    void processBlock(float * in, float * outHP, float * outLP, int numSamples);
    void dumpCoefficents(struct filterCoefficents x) {
        std::cout << "a0: " << x.a0 << "\n";
        std::cout << "a1: " << x.a1 << "\n";
        std::cout << "a2: " << x.a2 << "\n";
        std::cout << "a3: " << x.a3 << "\n";
        std::cout << "a4: " << x.a4 << "\n";
    }
    void dumpInformation() {
        std::cout << "-----\nfrequency: "<< coFreqRunningAv << "\n";
        std::cout << "lpco:\n";
```

(continues on next page)

(continued from previous page)

```
        dumpCoefficents(lpco);
        std::cout << "hpco:\n";
        dumpCoefficents(hpco);
        std::cout << "bco:\nb1: ";
        std::cout << b1co << "\nb2: " << b2co << "\nb3: " <<  b3co << "\nb4: " <<
↪b4co << "\n";
    }


};


void LRCrossoverFilter::setup(float crossoverFrequency, float sr) {

    const float pi = 3.141f;

    coFreqRunningAv = crossoverFrequency;

    float cowc=2*pi*coFreqRunningAv;
    float cowc2=cowc*cowc;
    float cowc3=cowc2*cowc;
    float cowc4=cowc2*cowc2;

    float cok=cowc/tan(pi*coFreqRunningAv/sr);
    float cok2=cok*cok;
    float cok3=cok2*cok;
    float cok4=cok2*cok2;
    float sqrt2=sqrt(2);
    float sq_tmp1 = sqrt2 * cowc3 * cok;
    float sq_tmp2 = sqrt2 * cowc * cok3;
    float a_tmp = 4*cowc2*cok2 + 2*sq_tmp1 + cok4 + 2*sq_tmp2+cowc4;

    b1co=(4*(cowc4+sq_tmp1-cok4-sq_tmp2))/a_tmp;


    b2co=(6*cowc4-8*cowc2*cok2+6*cok4)/a_tmp;


    b3co=(4*(cowc4-sq_tmp1+sq_tmp2-cok4))/a_tmp;


    b4co=(cok4-2*sq_tmp1+cowc4-2*sq_tmp2+4*cowc2*cok2)/a_tmp;



    //================================================
    // low-pass
    //================================================
    lpco.a0=cowc4/a_tmp;
    lpco.a1=4*cowc4/a_tmp;
    lpco.a2=6*cowc4/a_tmp;
    lpco.a3=lpco.a1;
    lpco.a4=lpco.a0;

    //=====================================================
    // high-pass
    //=====================================================
```

(continues on next page)

(continued from previous page)

```
    hpco.a0=cok4/a_tmp;
    hpco.a1=-4*cok4/a_tmp;
    hpco.a2=6*cok4/a_tmp;
    hpco.a3=hpco.a1;
    hpco.a4=hpco.a0;


}

void LRCrossoverFilter::processBlock(float * in, float * outHP, float * outLP, int␣
↪numSamples) {

    float tempx, tempy;
    for (int i = 0; i<numSamples; i++) {
        tempx=in[i];

        // High pass

        tempy = hpco.a0*tempx +
        hpco.a1*hptemp.xm1 +
        hpco.a2*hptemp.xm2 +
        hpco.a3*hptemp.xm3 +
        hpco.a4*hptemp.xm4 -
        b1co*hptemp.ym1 -
        b2co*hptemp.ym2 -
        b3co*hptemp.ym3 -
        b4co*hptemp.ym4;

        hptemp.xm4=hptemp.xm3;
        hptemp.xm3=hptemp.xm2;
        hptemp.xm2=hptemp.xm1;
        hptemp.xm1=tempx;
        hptemp.ym4=hptemp.ym3;
        hptemp.ym3=hptemp.ym2;
        hptemp.ym2=hptemp.ym1;
        hptemp.ym1=tempy;
        outHP[i]=tempy;

        assert(tempy<10000000);

        // Low pass

        tempy = lpco.a0*tempx +
        lpco.a1*lptemp.xm1 +
        lpco.a2*lptemp.xm2 +
        lpco.a3*lptemp.xm3 +
        lpco.a4*lptemp.xm4 -
        b1co*lptemp.ym1 -
        b2co*lptemp.ym2 -
        b3co*lptemp.ym3 -
        b4co*lptemp.ym4;

        lptemp.xm4=lptemp.xm3; // these are the same as hptemp and could be optimised␣
↪away
        lptemp.xm3=lptemp.xm2;
        lptemp.xm2=lptemp.xm1;
        lptemp.xm1=tempx;
```

(continues on next page)

(continued from previous page)

```
        lptemp.ym4=lptemp.ym3;
        lptemp.ym3=lptemp.ym2;
        lptemp.ym2=lptemp.ym1;
        lptemp.ym1=tempy;
        outLP[i] = tempy;

        assert(!isnan(outLP[i]));
    }
}


int main () {
    LRCrossoverFilter filter;
    float data[2000];
    float lp[2000], hp[2000];

    filter.setup(50.0, 44100.0f);
    filter.dumpInformation();

    for (int i = 0; i<2000; i++) {
        data[i] = sinf(i/100.f);
    }
    filter.processBlock(data, hp, lp, 2000);

}

I'll try and fix it, but this kind of work is new to me, so all suggestions␣
↪appreciated (Including "You Fool, you've copied the code wrong"). cheers!
```

- **Date**: 2013-09-03 22:35:23
- **By**: ku.oc.9f.yrreksirhc@kc

```
I tried this code for a crossover - firstly the SSE intrinsics version then the full␣
↪original version. Both have problems with the HPF output.
With a crossover frequency of 200Hz and a pure sine tone input (any pitch) I get loud␣
↪(-16dBFS)low frequency noise in the HPF output. This noise level reduces as the␣
↪crossover frequency increases but it is unusable in its current state.
Can anyone post a solution for this problem?
Thanks.....Chris
```

- **Date**: 2013-09-04 10:37:34
- **By**: ku.oc.9f.yrreksirhc@kc

```
I also tried the LR2 code, this works better but there is still low frequency noise (-
↪56dBFS & Xover 200Hz) in the HPF output.
Seems there is a fundamental problem with the HPF coefficients in this code :(
The LF Noise for both LR2 and LR4 appears to be a modulating DC offset - maybe that␣
↪can guide the Filter Gurus to identify and solve the problem.
Cheers.....Chris
```

- **Date**: 2020-05-25 01:45:00
- **By**: enummusic

```
In my experience, simply changing all the variables used in␣
↪LRCrossoverFilter::setup() and processBlock() to doubles is sufficient to reduce/
↪eliminate noise, thanks to an idea from here: https://www.musicdsp.org/en/latest/
↪Filters/232-type-lpf-24db-oct.html
"It turns out, that the filter is only unstable if the coefficient/state precision isn
↪'t high enough. Using double instead of single precision already makes it a lot␣
↪more stable."
```

3.8 All-Pass Filters, a good explanation

- **Author or source:** Olli Niemitalo
- **Type:** information
- **Created:** 2002-01-17 02:08:11
- **Linked files:** `filters002.txt`.

3.9 Another 4-pole lowpass…

- **Author or source:** ten.xmg@zlipzzuf
- **Type:** 4-pole LP/HP
- **Created:** 2004-09-06 08:40:52

Listing 12: notes

```
Vaguely based on the Stilson/Smith Moog paper, but going in a rather different␣
↪direction
from others I've seen here.

The parameters are peak frequency and peak magnitude (g below); both are reasonably
accurate for magnitudes above 1. DC gain is 1.

The filter has some undesirable properties - e.g. it's unstable for low peak freqs if
implemented in single precision (haven't been able to cleanly separate it into␣
↪biquads or
onepoles to see if that helps), and it responds so strongly to parameter changes that␣
↪it's
not advisable to update the coefficients much more rarely than, say, every eight␣
↪samples
during sweeps, which makes it somewhat expensive.

I like the sound, however, and the accuracy is nice to have, since many filters are␣
↪not
very strong in that respect.

I haven't looked at the HP again for a while, but IIRC it had approximately the same␣
↪good
and bad sides.
```

Listing 13: code

```
double coef[9];
double d[4];
double omega; //peak freq
double g;     //peak mag

// calculating coefficients:

double k,p,q,a;
double a0,a1,a2,a3,a4;

k=(4.0*g-3.0)/(g+1.0);
p=1.0-0.25*k;p*=p;

// LP:
a=1.0/(tan(0.5*omega)*(1.0+p));
p=1.0+a;
q=1.0-a;

a0=1.0/(k+p*p*p*p);
a1=4.0*(k+p*p*p*q);
a2=6.0*(k+p*p*q*q);
a3=4.0*(k+p*q*q*q);
a4=    (k+q*q*q*q);
p=a0*(k+1.0);

coef[0]=p;
coef[1]=4.0*p;
coef[2]=6.0*p;
coef[3]=4.0*p;
coef[4]=p;
coef[5]=-a1*a0;
coef[6]=-a2*a0;
coef[7]=-a3*a0;
coef[8]=-a4*a0;

// or HP:
a=tan(0.5*omega)/(1.0+p);
p=a+1.0;
q=a-1.0;

a0=1.0/(p*p*p*p+k);
a1=4.0*(p*p*p*q-k);
a2=6.0*(p*p*q*q+k);
a3=4.0*(p*q*q*q-k);
a4=    (q*q*q*q+k);
p=a0*(k+1.0);

coef[0]=p;
coef[1]=-4.0*p;
coef[2]=6.0*p;
coef[3]=-4.0*p;
coef[4]=p;
coef[5]=-a1*a0;
coef[6]=-a2*a0;
coef[7]=-a3*a0;
```

(continues on next page)

(continued from previous page)

```
coef[8]=-a4*a0;

// per sample:

out=coef[0]*in+d[0];
d[0]=coef[1]*in+coef[5]*out+d[1];
d[1]=coef[2]*in+coef[6]*out+d[2];
d[2]=coef[3]*in+coef[7]*out+d[3];
d[3]=coef[4]*in+coef[8]*out;
```

3.9.1 Comments

- **Date**: 2005-04-04 20:39:55
- **By**: ed.luosfosruoivas@naitsirhC

```
Yet untested object pascal translation:

unit T4PoleUnit;

interface

type TFilterType=(ftLowPass, ftHighPass);
     T4Pole=class(TObject)
     private
       fGain     : Double;
       fFreq     : Double;
       fSR       : Single;
     protected
       fCoeffs     : array[0..8] of Double;
       d           : array[0..3] of Double;
       fFilterType : TFilterType;
       procedure SetGain(s:Double);
       procedure SetFrequency(s:Double);
       procedure SetFilterType(v:TFilterType);
       procedure Calc;
     public
       constructor Create;
       function Process(s:single):single;
     published
       property Gain: Double read fGain write SetGain;
       property Frequency: Double read fFreq write SetFrequency;
       property SampleRate: Single read fSR write fSR;
       property FilterType: TFilterType read fFilterType write SetFilterType;
     end;

implementation

uses math;

const kDenorm = 1.0e-25;

constructor T4Pole.Create;
begin
 inherited create;
```

(continues on next page)

(continued from previous page)

```
 fFreq:=1000;
 fSR:=44100;
 Calc;
end;

procedure T4Pole.SetFrequency(s:Double);
begin
 fFreq:=s;
 Calc;
end;

procedure T4Pole.SetGain(s:Double);
begin
 fGain:=s;
 Calc;
end;

procedure T4Pole.SetFilterType(v:TFilterType);
begin
 fFilterType:=v;
 Calc;
end;

procedure T4Pole.Calc;
var k,p,q,b,s : Double;
    a         : array[0..4] of Double;
begin
 fGain:=1;
 if fFilterType=ftLowPass
  then s:=1
  else s:=-1;
 // calculating coefficients:
 k:=(4.0*fGain-3.0)/(fGain+1.0);
 p:=1.0-0.25*k;
 p:=p*p;

 if fFilterType=ftLowPass
  then b:=1.0/(tan(pi*fFreq/fSR)*(1.0+p))
  else b:=tan(pi*fFreq/fSR)/(1.0+p);
 p:=1.0+b;
 q:=s*(1.0-b);

 a[0] := 1.0/(  k+p*p*p*p);
 a[1] := 4.0*(s*k+p*p*p*q);
 a[2] := 6.0*(  k+p*p*q*q);
 a[3] := 4.0*(s*k+p*q*q*q);
 a[4] :=     (  k+q*q*q*q);
 p    := a[0]*(k+1.0);

 fCoeffs[0]:=p;
 fCoeffs[1]:=4.0*p*s;
 fCoeffs[2]:=6.0*p;
 fCoeffs[3]:=4.0*p*s;
 fCoeffs[4]:=p;
 fCoeffs[5]:=-a[1]*a[0];
 fCoeffs[6]:=-a[2]*a[0];
 fCoeffs[7]:=-a[3]*a[0];
```

(continues on next page)

(continued from previous page)

```
 fCoeffs[8]:=-a[4]*a[0];
end;

function T4Pole.Process(s:single):single;
begin
 Result:=fCoeffs[0]*s+d[0];
 d[0]:=fCoeffs[1]*s+fCoeffs[5]*Result+d[1];
 d[1]:=fCoeffs[2]*s+fCoeffs[6]*Result+d[2];
 d[2]:=fCoeffs[3]*s+fCoeffs[7]*Result+d[3];
 d[3]:=fCoeffs[4]*s+fCoeffs[8]*Result;
end;

end.
```

- **Date**: 2015-03-19 19:24:23
- **By**: ed.xmg@0891retep_repus

```
so bad that this filter is so unstable. i tested it and is has a really nice sound.␣
↪but frequencies below 200 hz are not possible. :-(
```

3.10 Bass Booster

- **Author or source:** Johny Dupej
- **Type:** LP and SUM
- **Created:** 2006-08-11 12:47:34

Listing 14: notes

```
This function adds a low-passed signal to the original signal. The low-pass has a␣
↪quite
wide response.

Params:
selectivity - frequency response of the LP (higher value gives a steeper one) [70.0 to
140.0 sounds good]
ratio - how much of the filtered signal is mixed to the original
gain2 - adjusts the final volume to handle cut-offs (might be good to set dynamically)
```

Listing 15: code

```
#define saturate(x) __min(__max(-1.0,x),1.0)

float BassBoosta(float sample)
{
static float selectivity, gain1, gain2, ratio, cap;
gain1 = 1.0/(selectivity + 1.0);

cap= (sample + cap*selectivity )*gain1;
sample = saturate((sample + cap*ratio)*gain2);

return sample;
}
```

3.11 Biquad C code

- **Author or source:** Tom St Denis
- **Created:** 2002-02-10 12:33:52
- **Linked files:** `biquad.c`.

Listing 16: notes

```
Implementation of the RBJ cookbook, in C.
```

3.12 Biquad, Butterworth, Chebyshev N-order, M-channel optimized filters

- **Author or source:** moc.oohay@nniveht
- **Type:** LP, HP, BP, BS, Shelf, Notch, Boost
- **Created:** 2009-11-16 08:46:34

Listing 17: code

```
/*

"A Collection of Useful C++ Classes for Digital Signal Processing"
By Vincent Falco

Please direct all comments to either the music-dsp mailing list or
the DSP and Plug-in Development forum:

    http://music.columbia.edu/cmc/music-dsp/

    http://www.kvraudio.com/forum/viewforum.php?f=33
    http://www.kvraudio.com/forum/

Support is provided for performing N-order Dsp floating point filter
operations on M-channel data with a caller specified floating point type.
The implementation breaks a high order IIR filter down into a series of
cascaded second order stages. Tests conclude that numerical stability is
maintained even at higher orders. For example the Butterworth low pass
filter is stable at up to 53 poles.

Processing functions are provided to use either Direct Form I or Direct
Form II of the filter transfer function. Direct Form II is slightly faster
but can cause discontinuities in the output if filter parameters are changed
during processing. Direct Form I is slightly slower, but maintains fidelity
even when parameters are changed during processing.

To support fast parameter changes, filters provide two functions for
adjusting parameters. A high accuracy Setup() function, and a faster
form called SetupFast() that uses approximations for trigonometric
functions. The approximations work quite well and should be suitable for
most applications.

```

(continues on next page)

(continued from previous page)

```
Channels are stored in an interleaved format with M samples per frame
arranged contiguously. A single class instance can process all M channels
simultaneously in an efficient manner. A 'skip' parameter causes the
processing function to advance by skip additional samples in the destination
buffer in between every frame. Through manipulation of the skip paramter it
is possible to exclude channels from processing (for example, only processing
the left half of stereo interleaved data). For multichannel data which is
not interleaved, it will be necessary to instantiate multiple instance of
the filter and set skip=0.

There are a few other utility classes and functions included that may prove useful.

Classes:

Complex
CascadeStages
    Biquad
            BiquadLowPass
            BiquadHighPass
            BiquadBandPass1
            BiquadBandPass2
            BiquadBandStop
            BiquadAllPass
            BiquadPeakEq
            BiquadLowShelf
            BiquadHighShelf
    PoleFilter
            Butterworth
                    ButterLowPass
                    ButterHighPass
                    ButterBandPass
                    ButterBandStop
            Chebyshev1
                    Cheby1LowPass
                    Cheby1HighPass
                    Cheby1BandPass
                    Cheby1BandStop
            Chebyshev2
                    Cheby2LowPass
                    Cheby2HighPass
                    Cheby2BandPass
                    Cheby2BandStop
EnvelopeFollower
AutoLimiter

Functions:

    zero()
    copy()
    mix()
    scale()

    interleave()
    deinterleave()

Order for PoleFilter derived classes is specified in the number of poles,
except for band pass and band stop filters, for which the number of pole pairs
```

(continues on next page)

(continued from previous page)

```
is specified.

For some filters there are two versions of Setup(), the one called
SetupFast() uses approximations to trigonometric functions for speed.
This is an option if you are doing frequent parameter changes to the filter.

There is an example function at the bottom that shows how to use the classes.

Filter ideas are based on a java applet (http://www.falstad.com/dfilter/)
developed by Paul Falstad.

All of this code was written by the author Vincent Falco except where marked.

--------------------------------------------------------------------------------

License: MIT License (http://www.opensource.org/licenses/mit-license.php)
Copyright (c) 2009 by Vincent Falco

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
 *
 */
/*
    To Do:

    - Shelving, peak, all-pass for Butterworth, Chebyshev, and Elliptic.

            The Biquads have these versions and I would like the others to have them
            as well. It would also be super awesome if higher order filters could
            have a "Q" parameter for resonance but I'm not expecting miracles, it
            would require redistributing the poles and zeroes. But if there's
            a research paper or code out there...I could incorporate it.

    - Denormal testing and fixing

            I'd like to know if denormals are a problem. And if so, it would be nice
            to have a small function that can reproduce the denormal problem. This
            way I can test the fix under all conditions. I will include the function
            as a "unit test" object in the header file so anyone can verify its
            correctness. But I'm a little lost.

    - Optimized template specializations for stages=1, channels={1,2}
```

(continues on next page)

(continued from previous page)

```

            There are some pretty obvious optimizations I am saving for "the end".
            I don't want to do them until the code is finalized.

    - Optimized template specializations for SSE, other special instructions

    - Optimized trigonometric functions for fast parameter changes

    - Elliptic curve based filter coefficients

    - More utility functions for manipulating sample buffers

    - Need fast version of pow( 10, x )
*/

#ifndef __DSP_FILTER__
#define __DSP_FILTER__

#include <cmath>
#include <cfloat>
#include <assert.h>
#include <memory.h>
#include <stdlib.h>

//#define DSP_USE_STD_COMPLEX

#ifdef DSP_USE_STD_COMPLEX
#include <complex>
#endif

#define DSP_SSE3_OPTIMIZED

#ifdef DSP_SSE3_OPTIMIZED
    //#include <xmmintrin.h>
    //#include <emmintrin.h>
    #include <pmmintrin.h>
#endif

namespace Dsp
{
    //--------------------------------------------------------------------------
    // WARNING: Here there be templates
    //--------------------------------------------------------------------------

    //--------------------------------------------------------------------------
    //
    //      Configuration
    //
    //--------------------------------------------------------------------------

    // Regardless of the type of sample that the filter operates on (e.g.
    // float or double), all calculations are performed using double (or
    // better) for stability and accuracy. This controls the underlying
    // type used for calculations:
    typedef double CalcT;

    typedef int             Int32;  // Must be 32 bits
```

(continues on next page)

(continued from previous page)

```
    typedef __int64 Int64;  // Must be 64 bits

    // This is used to prevent denormalization.
    const CalcT vsa=1.0 / 4294967295.0; // for CalcT as float

    // These constants are so important, I made my own copy. If you improve
    // the resolution of CalcT be sure to add more significant digits to these.
    const CalcT kPi         =3.1415926535897932384626433832795;
    const CalcT kPi_2       =1.57079632679489661923;
    const CalcT kLn2    =0.693147180559945309417;
    const CalcT kLn10       =2.30258509299404568402;

    //--------------------------------------------------------------------------

    // Some functions missing from <math.h>
    template<typename T>
    inline T acosh( T x )
    {
            return log( x+::sqrt(x*x-1) );
    }

    //--------------------------------------------------------------------------
    //
    //      Fast Trigonometric Functions
    //
    //--------------------------------------------------------------------------

    // Three approximations for both sine and cosine at a given angle.
    // The faster the routine, the larger the error.
    // From http://lab.polygonal.de/2007/07/18/fast-and-accurate-sinecosine-
↪approximation/

    // Tuned for maximum pole stability. r must be in the range 0..pi
    // This one breaks down considerably at the higher angles. It is
    // included only for educational purposes.
    inline void fastestsincos( CalcT r, CalcT *sn, CalcT *cs )
    {
            const CalcT c=0.70710678118654752440; // sqrt(2)/2
            CalcT v=(2-4*c)*r*r+c;
            if(r<kPi_2)
            {
                    *sn=v+r; *cs=v-r;
            }
            else
            {
                    *sn=r+v; *cs=r-v;
            }
    }

    // Lower precision than ::fastsincos() but still decent
    inline void fastersincos( CalcT x, CalcT *sn, CalcT *cs )
    {
            //always wrap input angle to -PI..PI
            if              (x < -kPi) x += 2*kPi;
            else if (x >  kPi) x -= 2*kPi;
            //compute sine
            if (x < 0)      *sn = 1.27323954 * x + 0.405284735 * x * x;
```

(continues on next page)

(continued from previous page)

```
            else            *sn = 1.27323954 * x - 0.405284735 * x * x;
            //compute cosine: sin(x + PI/2) = cos(x)
            x += kPi_2;
            if (x > kPi ) x -= 2*kPi;
            if (x < 0)      *cs = 1.27323954 * x + 0.405284735 * x * x;
            else            *cs = 1.27323954 * x - 0.405284735 * x * x;
    }

    // Slower than ::fastersincos() but still faster than
    // sin(), and with the best accuracy of these routines.
    inline void fastsincos( CalcT x, CalcT *sn, CalcT *cs )
    {
            CalcT s, c;
            //always wrap input angle to -PI..PI
                     if (x < -kPi) x += 2*kPi;
            else if (x >  kPi) x -= 2*kPi;
            //compute sine
            if (x < 0)
            {
                    s = 1.27323954 * x + .405284735 * x * x;
                    if (s < 0)      s = .225 * (s * -s - s) + s;
                    else            s = .225 * (s *  s - s) + s;
            }
            else
            {
                    s = 1.27323954 * x - 0.405284735 * x * x;
                    if (s < 0)      s = .225 * (s * -s - s) + s;
                    else            s = .225 * (s *  s - s) + s;
            }
            *sn=s;
            //compute cosine: sin(x + PI/2) = cos(x)
            x += kPi_2;
            if (x > kPi ) x -= 2*kPi;
            if (x < 0)
            {
                    c = 1.27323954 * x + 0.405284735 * x * x;
                    if (c < 0)      c = .225 * (c * -c - c) + c;
                    else            c = .225 * (c *  c - c) + c;
            }
            else
            {
                    c = 1.27323954 * x - 0.405284735 * x * x;
                    if (c < 0)      c = .225 * (c * -c - c) + c;
                    else            c = .225 * (c *  c - c) + c;
            }
            *cs=c;
    }

    // Faster approximations to sqrt()
    //      From http://ilab.usc.edu/wiki/index.php/Fast_Square_Root
    //      The faster the routine, the more error in the approximation.

    // Log Base 2 Approximation
    // 5 times faster than sqrt()

    inline float fastsqrt1( float x )
    {
```

(continues on next page)

(continued from previous page)

```
            union { Int32 i; float x; } u;
            u.x = x;
            u.i = (Int32(1)<<29) + (u.i >> 1) - (Int32(1)<<22);
            return u.x;
    }

    inline double fastsqrt1( double x )
    {
            union { Int64 i; double x; } u;
            u.x = x;
            u.i = (Int64(1)<<61) + (u.i >> 1) - (Int64(1)<<51);
            return u.x;
    }

    // Log Base 2 Approximation with one extra Babylonian Step
    // 2 times faster than sqrt()

    inline float fastsqrt2( float x )
    {
            float v=fastsqrt1( x );
            v = 0.5f * (v + x/v); // One Babylonian step
            return v;
    }

    inline double fastsqrt2(const double x)
    {
            double v=fastsqrt1( x );
            v = 0.5f * (v + x/v); // One Babylonian step
            return v;
    }

    // Log Base 2 Approximation with two extra Babylonian Steps
    // 50% faster than sqrt()

    inline float fastsqrt3( float x )
    {
            float v=fastsqrt1( x );
            v =             v + x/v;
            v = 0.25f* v + x/v; // Two Babylonian steps
            return v;
    }

    inline double fastsqrt3(const double x)
    {
            double v=fastsqrt1( x );
            v =             v + x/v;
            v = 0.25 * v + x/v; // Two Babylonian steps
            return v;
    }

    //--------------------------------------------------------------------------
    //
    //      Complex
    //
    //--------------------------------------------------------------------------

#ifdef DSP_USE_STD_COMPLEX
```

(continues on next page)

(continued from previous page)

```

    template<typename T>
    inline std::complex<T> polar( const T &m, const T &a )
    {
            return std::polar( m, a );
    }

    template<typename T>
    inline T norm( const std::complex<T> &c )
    {
            return std::norm( c );
    }

    template<typename T>
    inline T abs( const std::complex<T> &c )
    {
            return std::abs(c);
    }

    template<typename T, typename To>
    inline std::complex<T> addmul( const std::complex<T> &c, T v, const std::complex<To> &c1 )
    {
            return std::complex<T>( c.real()+v*c1.real(), c.imag()+v*c1.imag() );
    }

    template<typename T>
    inline T arg( const std::complex<T> &c )
    {
            return std::arg( c );
    }

    template<typename T>
    inline std::complex<T> recip( const std::complex<T> &c )
    {
            T n=1.0/Dsp::norm(c);
            return std::complex<T>( n*c.real(), n*c.imag() );
    }
    template<typename T>
    inline std::complex<T> sqrt( const std::complex<T> &c )
    {
            return std::sqrt( c );
    }

    typedef std::complex<CalcT> Complex;

#else

    //--------------------------------------------------------------------------

    // "Its always good to have a few extra wheels in case one goes flat."

    template<typename T>
    struct ComplexT
    {
            ComplexT();
            ComplexT( T r_, T i_=0 );
```

(continues on next page)

(continued from previous page)

```

        template<typename To>
        ComplexT( const ComplexT<To> &c );

        T                   imag            ( void ) const;
        T                   real            ( void ) const;

        ComplexT &      neg                 ( void );
        ComplexT &      conj        ( void );

        template<typename To>
        ComplexT &      add                 ( const ComplexT<To> &c );
        template<typename To>
        ComplexT &      sub                 ( const ComplexT<To> &c );
        template<typename To>
        ComplexT &      mul                 ( const ComplexT<To> &c );
        template<typename To>
        ComplexT &      div                 ( const ComplexT<To> &c );
        template<typename To>
        ComplexT &      addmul      ( T v, const ComplexT<To> &c );

        ComplexT        operator-       ( void ) const;

        ComplexT        operator+       ( T v ) const;
        ComplexT        operator-       ( T v ) const;
        ComplexT        operator*       ( T v ) const;
        ComplexT        operator/       ( T v ) const;

        ComplexT &      operator+=      ( T v );
        ComplexT &      operator-=      ( T v );
        ComplexT &      operator*=      ( T v );
        ComplexT &      operator/=      ( T v );

        template<typename To>
        ComplexT        operator+       ( const ComplexT<To> &c ) const;
        template<typename To>
        ComplexT        operator-       ( const ComplexT<To> &c ) const;
        template<typename To>
        ComplexT        operator*       ( const ComplexT<To> &c ) const;
        template<typename To>
        ComplexT        operator/       ( const ComplexT<To> &c ) const;

        template<typename To>
        ComplexT &      operator+=      ( const ComplexT<To> &c );
        template<typename To>
        ComplexT &      operator-=      ( const ComplexT<To> &c );
        template<typename To>
        ComplexT &      operator*=      ( const ComplexT<To> &c );
        template<typename To>
        ComplexT &      operator/=      ( const ComplexT<To> &c );

    private:
        ComplexT &      add                 ( T v );
        ComplexT &      sub                 ( T v );
        ComplexT &      mul                 ( T c, T d );
        ComplexT &      mul                 ( T v );
        ComplexT &      div                 ( T v );
```

(continues on next page)

(continued from previous page)

```

        T r;
        T i;
};

//----------------------------------------------------------------------------

template<typename T>
inline ComplexT<T>::ComplexT()
{
}

template<typename T>
inline ComplexT<T>::ComplexT( T r_, T i_ )
{
        r=r_;
        i=i_;
}

template<typename T>
template<typename To>
inline ComplexT<T>::ComplexT( const ComplexT<To> &c )
{
        r=c.r;
        i=c.i;
}

template<typename T>
inline T ComplexT<T>::imag( void ) const
{
        return i;
}

template<typename T>
inline T ComplexT<T>::real( void ) const
{
        return r;
}

template<typename T>
inline ComplexT<T> &ComplexT<T>::neg( void )
{
        r=-r;
        i=-i;
        return *this;
}

template<typename T>
inline ComplexT<T> &ComplexT<T>::conj( void )
{
        i=-i;
        return *this;
}

template<typename T>
inline ComplexT<T> &ComplexT<T>::add( T v )
{
```

(continues on next page)

(continued from previous page)

```
		r+=v;
		return *this;
	}

	template<typename T>
	inline ComplexT<T> &ComplexT<T>::sub( T v )
	{
		r-=v;
		return *this;
	}

	template<typename T>
	inline ComplexT<T> &ComplexT<T>::mul( T c, T d )
	{
		T ac=r*c;
		T bd=i*d;
		// must do i first
		i=(r+i)*(c+d)-(ac+bd);
		r=ac-bd;
		return *this;
	}

	template<typename T>
	inline ComplexT<T> &ComplexT<T>::mul( T v )
	{
		r*=v;
		i*=v;
		return *this;
	}

	template<typename T>
	inline ComplexT<T> &ComplexT<T>::div( T v )
	{
		r/=v;
		i/=v;
		return *this;
	}

	template<typename T>
	template<typename To>
	inline ComplexT<T> &ComplexT<T>::add( const ComplexT<To> &c )
	{
		r+=c.r;
		i+=c.i;
		return *this;
	}

	template<typename T>
	template<typename To>
	inline ComplexT<T> &ComplexT<T>::sub( const ComplexT<To> &c )
	{
		r-=c.r;
		i-=c.i;
		return *this;
	}

	template<typename T>
```

(continues on next page)

(continued from previous page)

```
	template<typename To>
	inline ComplexT<T> &ComplexT<T>::mul( const ComplexT<To> &c )
	{
		return mul( c.r, c.i );
	}

	template<typename T>
	template<typename To>
	inline ComplexT<T> &ComplexT<T>::div( const ComplexT<To> &c )
	{
		T s=1.0/norm(c);
		return mul( c.r*s, -c.i*s );
	}

	template<typename T>
	inline ComplexT<T> ComplexT<T>::operator-( void ) const
	{
		return ComplexT<T>(*this).neg();
	}

	template<typename T>
	inline ComplexT<T> ComplexT<T>::operator+( T v ) const
	{
		return ComplexT<T>(*this).add( v );
	}

	template<typename T>
	inline ComplexT<T> ComplexT<T>::operator-( T v ) const
	{
		return ComplexT<T>(*this).sub( v );
	}

	template<typename T>
	inline ComplexT<T> ComplexT<T>::operator*( T v ) const
	{
		return ComplexT<T>(*this).mul( v );
	}

	template<typename T>
	inline ComplexT<T> ComplexT<T>::operator/( T v ) const
	{
		return ComplexT<T>(*this).div( v );
	}

	template<typename T>
	inline ComplexT<T> &ComplexT<T>::operator+=( T v )
	{
		return add( v );
	}

	template<typename T>
	inline ComplexT<T> &ComplexT<T>::operator-=( T v )
	{
		return sub( v );
	}

	template<typename T>
```

(continues on next page)

(continued from previous page)

```
    inline ComplexT<T> &ComplexT<T>::operator*=( T v )
    {
            return mul( v );
    }

    template<typename T>
    inline ComplexT<T> &ComplexT<T>::operator/=( T v )
    {
            return div( v );
    }

    template<typename T>
    template<typename To>
    inline ComplexT<T> ComplexT<T>::operator+( const ComplexT<To> &c ) const
    {
            return ComplexT<T>(*this).add(c);
    }

    template<typename T>
    template<typename To>
    inline ComplexT<T> ComplexT<T>::operator-( const ComplexT<To> &c ) const
    {
            return ComplexT<T>(*this).sub(c);
    }

    template<typename T>
    template<typename To>
    inline ComplexT<T> ComplexT<T>::operator*( const ComplexT<To> &c ) const
    {
            return ComplexT<T>(*this).mul(c);
    }

    template<typename T>
    template<typename To>
    inline ComplexT<T> ComplexT<T>::operator/( const ComplexT<To> &c ) const
    {
            return ComplexT<T>(*this).div(c);
    }

    template<typename T>
    template<typename To>
    inline ComplexT<T> &ComplexT<T>::operator+=( const ComplexT<To> &c )
    {
            return add( c );
    }

    template<typename T>
    template<typename To>
    inline ComplexT<T> &ComplexT<T>::operator-=( const ComplexT<To> &c )
    {
            return sub( c );
    }

    template<typename T>
    template<typename To>
    inline ComplexT<T> &ComplexT<T>::operator*=( const ComplexT<To> &c )
    {
```

(continues on next page)

(continued from previous page)

```
        return mul( c );
}

template<typename T>
template<typename To>
inline ComplexT<T> &ComplexT<T>::operator/=( const ComplexT<To> &c )
{
        return div( c );
}

//------------------------------------------------------------------------

template<typename T>
inline ComplexT<T> polar( const T &m, const T &a )
{
        return ComplexT<T>( m*cos(a), m*sin(a) );
}

template<typename T>
inline T norm( const ComplexT<T> &c )
{
        return c.real()*c.real()+c.imag()*c.imag();
}

template<typename T>
inline T abs( const ComplexT<T> &c )
{
        return ::sqrt( c.real()*c.real()+c.imag()*c.imag() );
}

template<typename T, typename To>
inline ComplexT<T> addmul( const ComplexT<T> &c, T v, const ComplexT<To> &c1 )
{
        return ComplexT<T>( c.real()+v*c1.real(), c.imag()+v*c1.imag() );
}

template<typename T>
inline T arg( const ComplexT<T> &c )
{
        return atan2( c.imag(), c.real() );
}

template<typename T>
inline ComplexT<T> recip( const ComplexT<T> &c )
{
        T n=1.0/norm(c);
        return ComplexT<T>( n*c.real(), -n*c.imag() );
}

template<typename T>
inline ComplexT<T> sqrt( const ComplexT<T> &c )
{
        return polar( ::sqrt(abs(c)), arg(c)*0.5 );
}

//------------------------------------------------------------------------

```

(continues on next page)

(continued from previous page)

```
    typedef ComplexT<CalcT> Complex;

#endif

    //--------------------------------------------------------------------------
    //
    //      Numerical Analysis
    //
    //--------------------------------------------------------------------------

    // Implementation of Brent's Method provided by
    // John D. Cook (http://www.johndcook.com/)

    // The return value of Minimize is the minimum of the function f.
    // The location where f takes its minimum is returned in the variable minLoc.
    // Notation and implementation based on Chapter 5 of Richard Brent's book
    // "Algorithms for Minimization Without Derivatives".

    template<class TFunction>
    CalcT BrentMinimize
    (
            TFunction& f,   // [in] objective function to minimize
            CalcT leftEnd,  // [in] smaller value of bracketing interval
            CalcT rightEnd, // [in] larger value of bracketing interval
            CalcT epsilon,  // [in] stopping tolerance
            CalcT& minLoc   // [out] location of minimum
    )
    {
            CalcT d, e, m, p, q, r, tol, t2, u, v, w, fu, fv, fw, fx;
            static const CalcT c = 0.5*(3.0 - ::sqrt(5.0));
            static const CalcT SQRT_DBL_EPSILON = ::sqrt(DBL_EPSILON);

            CalcT& a = leftEnd; CalcT& b = rightEnd; CalcT& x = minLoc;

            v = w = x = a + c*(b - a); d = e = 0.0;
            fv = fw = fx = f(x);
            int counter = 0;
    loop:
            counter++;
            m = 0.5*(a + b);
            tol = SQRT_DBL_EPSILON*::fabs(x) + epsilon; t2 = 2.0*tol;
            // Check stopping criteria
            if (::fabs(x - m) > t2 - 0.5*(b - a))
            {
                    p = q = r = 0.0;
                    if (::fabs(e) > tol)
                    {
                            // fit parabola
                            r = (x - w)*(fx - fv);
                            q = (x - v)*(fx - fw);
                            p = (x - v)*q - (x - w)*r;
                            q = 2.0*(q - r);
                            (q > 0.0) ? p = -p : q = -q;
                            r = e; e = d;
                    }
                    if (::fabs(p) < ::fabs(0.5*q*r) && p < q*(a - x) && p < q*(b - x))
                    {
```

(continues on next page)

(continued from previous page)

```
					// A parabolic interpolation step
					d = p/q;
					u = x + d;
					// f must not be evaluated too close to a or b
					if (u - a < t2 || b - u < t2)
						d = (x < m) ? tol : -tol;
				}
				else
				{
					// A golden section step
					e = (x < m) ? b : a;
					e -= x;
					d = c*e;
				}
				// f must not be evaluated too close to x
				if (::fabs(d) >= tol)
					u = x + d;
				else if (d > 0.0)
					u = x + tol;
				else
					u = x - tol;
				fu = f(u);
				// Update a, b, v, w, and x
				if (fu <= fx)
				{
					(u < x) ? b = x : a = x;
					v = w; fv = fw;
					w = x; fw = fx;
					x = u; fx = fu;
				}
				else
				{
					(u < x) ? a = u : b = u;
					if (fu <= fw || w == x)
					{
						v = w; fv = fw;
						w = u; fw = fu;
					}
					else if (fu <= fv || v == x || v == w)
					{
						v = u; fv = fu;
					}
				}
				goto loop;  // Yes, the dreaded goto statement. But the code
							// here is faithful to Brent's orginal␣
↪pseudocode.
			}
			return  fx;
	}

	//-------------------------------------------------------------------------
	//
	//	Infinite Impulse Response Filters
	//
	//-------------------------------------------------------------------------

	// IIR filter implementation using multiple second-order stages.
```

(continues on next page)

(continued from previous page)

```

    class CascadeFilter
    {
    public:
            // Process data in place using Direct Form I
            // skip is added after each frame.
            // Direct Form I is more suitable when the filter parameters
            // are changed often. However, it is slightly slower.
            template<typename T>
            void ProcessI( size_t frames, T *dest, int skip=0 );

            // Process data in place using Direct Form II
            // skip is added after each frame.
            // Direct Form II is slightly faster than Direct Form I,
            // but changing filter parameters on stream can result
            // in discontinuities in the output. It is best suited
            // for a filter whose parameters are set only once.
            template<typename T>
            void ProcessII( size_t frames, T *dest, int skip=0 );

            // Convenience function that just calls ProcessI.
            // Feel free to change the implementation.
            template<typename T>
            void Process( size_t frames, T *dest, int skip=0 );

            // Determine response at angular frequency (0<=w<=kPi)
            Complex Response( CalcT w );

            // Clear the history buffer.
            void Clear( void );

    protected:
            struct Hist;
            struct Stage;

            // for m_nchan==2
#ifdef DSP_SSE3_OPTIMIZED
            template<typename T>
            void ProcessISSEStageStereo( size_t frames, T *dest, Stage *stage, Hist␣
↪*h, int skip );

            template<typename T>
            void ProcessISSEStereo( size_t frames, T *dest, int skip );
#endif

    protected:
            void Reset              ( void );
            void Normalize  ( CalcT scale );
            void SetAStage  ( CalcT x1, CalcT x2 );
            void SetBStage  ( CalcT x0, CalcT x1, CalcT x2 );
            void SetStage   ( CalcT a1, CalcT a2, CalcT b0, CalcT b1, CalcT b2 );

    protected:
            struct Hist
            {
                    CalcT v[4];
            };
```

(continues on next page)

(continued from previous page)

```

		struct Stage
		{
			CalcT a[3];	// a[0] unused
			CalcT b[3];
			void Reset( void );
		};

		struct ResponseFunctor
		{
			CascadeFilter *m_filter;
			CalcT operator()( CalcT w );
			ResponseFunctor( CascadeFilter *filter );
		};

		int		m_nchan;
		int		m_nstage;
		Stage *	m_stagep;
		Hist *	m_histp;
	};

	//--------------------------------------------------------------------------

	template<typename T>
	void CascadeFilter::ProcessI( size_t frames, T *dest, int skip )
	{
#ifdef DSP_SSE3_OPTIMIZED
		if( m_nchan==2 )
			ProcessISSEStereo( frames, dest, skip );
		else
#endif
		while( frames-- )
		{
			Hist *h=m_histp;
			for( int j=m_nchan;j;j-- )
			{
				CalcT in=CalcT(*dest);
				Stage *s=m_stagep;
				for( int i=m_nstage;i;i--,h++,s++ )
				{
					CalcT out;
					out=s->b[0]*in		+ s->b[1]*h->v[0] + s->b[2]*h->v[1] +
						s->a[1]*h->v[2] + s->a[2]*h->v[3];
					h->v[1]=h->v[0]; h->v[0]=in;
					h->v[3]=h->v[2]; h->v[2]=out;
					in=out;
				}
				*dest++=T(in);
			}
			dest+=skip;
		}
	}

	// A good compiler already produces code that is optimized even for
	// the general case. The only way to make it go faster is to
	// to implement it in assembler or special instructions. Like this:
```

(continues on next page)

(continued from previous page)

```

#ifdef DSP_SSE3_OPTIMIZED
    // ALL SSE OPTIMIZATIONS ASSUME CalcT as double
    template<typename T>
    inline void CascadeFilter::ProcessISSEStageStereo(
            size_t frames, T *dest, Stage *s, Hist *h, int skip )
    {
            assert( m_nchan==2 );

#if 1
            CalcT b0=s->b[0];

            __m128d m0=_mm_loadu_pd( &s->a[1] );    // a1 , a2
            __m128d m1=_mm_loadu_pd( &s->b[1] );    // b1 , b2
            __m128d m2=_mm_loadu_pd( &h[0].v[0] );  // h->v[0] , h->v[1]
            __m128d m3=_mm_loadu_pd( &h[0].v[2] );  // h->v[2] , h->v[3]
            __m128d m4=_mm_loadu_pd( &h[1].v[0] );  // h->v[0] , h->v[1]
            __m128d m5=_mm_loadu_pd( &h[1].v[2] );  // h->v[2] , h->v[3]

            while( frames-- )
            {
                    CalcT in, b0in, out;

                    __m128d m6;
                    __m128d m7;

                    in=CalcT(*dest);
                    b0in=b0*in;

                    m6=_mm_mul_pd ( m1, m2 );       // b1*h->v[0] , b2*h->v[1]
                    m7=_mm_mul_pd ( m0, m3 );       // a1*h->v[2] , a2*h->v[3]
                    m6=_mm_add_pd ( m6, m7 );       // b1*h->v[0] + a1*h->v[2], b2*h->v[1] + a2*h->v[3]
                    m7=_mm_load_sd( &b0in );        // b0*in , 0
                    m6=_mm_add_sd ( m6, m7 );       // b1*h->v[0] + a1*h->v[2] + in*b0 , b2*h->v[1] + a2*h->v[3] + 0
                    m6=_mm_hadd_pd( m6, m7 );       // b1*h->v[0] + a1*h->v[2] + in*b0 + b2*h->v[1] + a2*h->v[3], in*b0
                       _mm_store_sd( &out, m6 );
                    m6=_mm_loadh_pd( m6, &in );     // out , in
                    m2=_mm_shuffle_pd( m6, m2, _MM_SHUFFLE2( 0, 1 ) ); // h->v[0]=in , h->v[1]=h->v[0]
                    m3=_mm_shuffle_pd( m6, m3, _MM_SHUFFLE2( 0, 0 ) ); // h->v[2]=out, h->v[3]=h->v[2]

                    *dest++=T(out);

                    in=CalcT(*dest);
                    b0in=b0*in;

                    m6=_mm_mul_pd ( m1, m4 );       // b1*h->v[0] , b2*h->v[1]
                    m7=_mm_mul_pd ( m0, m5 );       // a1*h->v[2] , a2*h->v[3]
                    m6=_mm_add_pd ( m6, m7 );       // b1*h->v[0] + a1*h->v[2], b2*h->v[1] + a2*h->v[3]
                    m7=_mm_load_sd( &b0in );        // b0*in , 0
                    m6=_mm_add_sd ( m6, m7 );       // b1*h->v[0] + a1*h->v[2] + in*b0 , b2*h->v[1] + a2*h->v[3] + 0
```

(continues on next page)

(continued from previous page)

```
                        m6=_mm_hadd_pd( m6, m7 );       // b1*h->v[0] + a1*h->v[2] + in*b0 + b2*h->v[1] + a2*h->v[3], in*b0
                            _mm_store_sd( &out, m6 );
                        m6=_mm_loadh_pd( m6, &in );     // out , in
                        m4=_mm_shuffle_pd( m6, m4, _MM_SHUFFLE2( 0, 1 ) ); // h->v[0]=in , h->v[1]=h->v[0]
                        m5=_mm_shuffle_pd( m6, m5, _MM_SHUFFLE2( 0, 0 ) ); // h->v[2]=out, h->v[3]=h->v[2]

                        *dest++=T(out);

                        dest+=skip;
                }

                // move history from registers back to state
                _mm_storeu_pd( &h[0].v[0], m2 );
                _mm_storeu_pd( &h[0].v[2], m3 );
                _mm_storeu_pd( &h[1].v[0], m4 );
                _mm_storeu_pd( &h[1].v[2], m5 );

#else
                // Template-specialized version from which the assembly was modeled
                CalcT a1=s->a[1];
                CalcT a2=s->a[2];
                CalcT b0=s->b[0];
                CalcT b1=s->b[1];
                CalcT b2=s->b[2];
                while( frames-- )
                {
                        CalcT in, out;

                        in=CalcT(*dest);
                        out=b0*in+b1*h[0].v[0]+b2*h[0].v[1] +a1*h[0].v[2]+a2*h[0].v[3];
                        h[0].v[1]=h[0].v[0]; h[0].v[0]=in;
                        h[0].v[3]=h[0].v[2]; h[0].v[2]=out;
                        in=out;
                        *dest++=T(in);

                        in=CalcT(*dest);
                        out=b0*in+b1*h[1].v[0]+b2*h[1].v[1] +a1*h[1].v[2]+a2*h[1].v[3];
                        h[1].v[1]=h[1].v[0]; h[1].v[0]=in;
                        h[1].v[3]=h[1].v[2]; h[1].v[2]=out;
                        in=out;
                        *dest++=T(in);

                        dest+=skip;
                }
#endif
        }

        // Note there could be a loss of accuracy here. Unlike the original version
        // of Process...() we are applying each stage to all of the input data.
        // Since the underlying type T could be float, the results from this function
        // may be different than the unoptimized version. However, it is much faster.
        template<typename T>
        void CascadeFilter::ProcessISSEStereo( size_t frames, T *dest, int skip )
        {
```

(continues on next page)

(continued from previous page)

```
            assert( m_nchan==2 );
            Stage *s=m_stagep;
            Hist *h=m_histp;
            for( int i=m_nstage;i;i--,h+=2,s++ )
            {
                    ProcessISSEStageStereo( frames, dest, s, h, skip );
            }
    }

#endif

    template<typename T>
    void CascadeFilter::ProcessII( size_t frames, T *dest, int skip )
    {
            while( frames-- )
            {
                    Hist *h=m_histp;
                    for( int j=m_nchan;j;j-- )
                    {
                            CalcT in=CalcT(*dest);
                            Stage *s=m_stagep;
                            for( int i=m_nstage;i;i--,h++,s++ )
                            {
                                    CalcT d2=h->v[2]=h->v[1];
                                    CalcT d1=h->v[1]=h->v[0];
                                    CalcT d0=h->v[0]=
                                            in+s->a[1]*d1 + s->a[2]*d2;
                                            in=s->b[0]*d0 + s->b[1]*d1 + s->b[2]*d2;
                            }
                            *dest++=T(in);
                    }
                    dest+=skip;
            }
    }

    template<typename T>
    inline void CascadeFilter::Process( size_t frames, T *dest, int skip )
    {
            ProcessI( frames, dest, skip );
    }

    inline Complex CascadeFilter::Response( CalcT w )
    {
            Complex ch( 1 );
            Complex cbot( 1 );
            Complex czn1=polar( 1., -w );
            Complex czn2=polar( 1., -2*w );

            Stage *s=m_stagep;
            for( int i=m_nstage;i;i-- )
            {
                    Complex ct( s->b[0] );
                    Complex cb( 1 );
                    ct=addmul( ct,  s->b[1], czn1 );
                    cb=addmul( cb, -s->a[1], czn1 );
                    ct=addmul( ct,  s->b[2], czn2 );
                    cb=addmul( cb, -s->a[2], czn2 );
```

(continues on next page)

(continued from previous page)

```
                        ch*=ct;
                        cbot*=cb;
                        s++;
                }

                return ch/cbot;
        }

        inline void CascadeFilter::Clear( void )
        {
                ::memset( m_histp, 0, m_nstage*m_nchan*sizeof(m_histp[0]) );
        }

        inline void CascadeFilter::Stage::Reset( void )
        {
                                a[1]=0; a[2]=0;
                b[0]=1; b[1]=0; b[2]=0;
        }

        inline void CascadeFilter::Reset( void )
        {
                Stage *s=m_stagep;
                for( int i=m_nstage;i;i--,s++ )
                        s->Reset();
        }

        // Apply scale factor to stage coefficients.
        inline void CascadeFilter::Normalize( CalcT scale )
        {
                // We are throwing the normalization into the first
                // stage. In theory it might be nice to spread it around
                // to preserve numerical accuracy.
                Stage *s=m_stagep;
                s->b[0]*=scale; s->b[1]*=scale; s->b[2]*=scale;
        }

        inline void CascadeFilter::SetAStage( CalcT x1, CalcT x2 )
        {
                Stage *s=m_stagep;
                for( int i=m_nstage;i;i-- )
                {
                        if( s->a[1]==0 && s->a[2]==0 )
                        {
                                s->a[1]=x1;
                                s->a[2]=x2;
                                s=0;
                                break;
                        }
                        if( s->a[2]==0 && x2==0 )
                        {
                                s->a[2]=-s->a[1]*x1;
                                s->a[1]+=x1;
                                s=0;
                                break;
                        }
                        s++;
                }
```

(continues on next page)

(continued from previous page)

```
            assert( s==0 );
    }

    inline void CascadeFilter::SetBStage( CalcT x0, CalcT x1, CalcT x2 )
    {
            Stage *s=m_stagep;
            for( int i=m_nstage;i;i-- )
            {
                    if( s->b[1]==0 && s->b[2]==0 )
                    {
                            s->b[0]=x0;
                            s->b[1]=x1;
                            s->b[2]=x2;
                            s=0;
                            break;
                    }
                    if( s->b[2]==0 && x2==0 )
                    {
                            // (b0 + z b1)(x0 + z x1) = (b0 x0 + (b1 x0+b0 x1) z + b1 x1 z^2)
                            s->b[2]=s->b[1]*x1;
                            s->b[1]=s->b[1]*x0+s->b[0]*x1;
                            s->b[0]*=x0;
                            s=0;
                            break;
                    }
                    s++;
            }
            assert( s==0 );
    }

    // Optimized version for Biquads
    inline void CascadeFilter::SetStage(
            CalcT a1, CalcT a2, CalcT b0, CalcT b1, CalcT b2 )
    {
            assert( m_nstage==1 );
            Stage *s=&m_stagep[0];
                                    s->a[1]=a1; s->a[2]=a2;
            s->b[0]=b0; s->b[1]=b1; s->b[2]=b2;
    }

    inline CalcT CascadeFilter::ResponseFunctor::operator()( CalcT w )
    {
            return -Dsp::abs(m_filter->Response( w ));
    }

    inline CascadeFilter::ResponseFunctor::ResponseFunctor( CascadeFilter *filter )
    {
            m_filter=filter;
    }

    //--------------------------------------------------------------------------

    template<int stages, int channels>
    class CascadeStages : public CascadeFilter
    {
    public:
```

(continues on next page)

(continued from previous page)

```
        CascadeStages();

private:
        Hist    m_hist  [stages*channels];
        Stage   m_stage [stages];
};

//--------------------------------------------------------------------------

template<int stages, int channels>
CascadeStages<stages, channels>::CascadeStages( void )
{
        m_nchan=channels;
        m_nstage=stages;
        m_stagep=m_stage;
        m_histp=m_hist;
        Clear();
}

//--------------------------------------------------------------------------
//
//      Biquad Second Order IIR Filters
//
//--------------------------------------------------------------------------

// Formulas from http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt
template<int channels>
class Biquad : public CascadeStages<1, channels>
{
protected:
        void Setup( const CalcT a[3], const CalcT b[3] );
};

//--------------------------------------------------------------------------

template<int channels>
inline void Biquad<channels>::Setup( const CalcT a[3], const CalcT b[3] )
{
        Reset();
        // transform Biquad coefficients
        CalcT ra0=1/a[0];
        SetAStage( -a[1]*ra0, -a[2]*ra0 );
        SetBStage(  b[0]*ra0,  b[1]*ra0, b[2]*ra0 );
}

//--------------------------------------------------------------------------

template<int channels>
class BiquadLowPass : public Biquad<channels>
{
public:
        void Setup                      ( CalcT normFreq, CalcT q );
        void SetupFast          ( CalcT normFreq, CalcT q );
protected:
        void SetupCommon        ( CalcT sn, CalcT cs, CalcT q );
};

```

(continues on next page)

(continued from previous page)

```
//--------------------------------------------------------------------------

template<int channels>
inline void BiquadLowPass<channels>::SetupCommon( CalcT sn, CalcT cs, CalcT q  )
{
        CalcT alph = sn / ( 2 * q );
        CalcT a0 =  1 / ( 1 + alph );
        CalcT b1 =  1 - cs;
        CalcT b0 = a0 * b1 * 0.5;
        CalcT a1 =  2 * cs;
        CalcT a2 = alph - 1;
        SetStage( a1*a0, a2*a0, b0, b1*a0, b0 );
}

template<int channels>
void BiquadLowPass<channels>::Setup( CalcT normFreq, CalcT q  )
{
        CalcT w0 = 2 * kPi * normFreq;
        CalcT cs = cos(w0);
        CalcT sn = sin(w0);
        SetupCommon( sn, cs, q );
}

template<int channels>
void BiquadLowPass<channels>::SetupFast( CalcT normFreq, CalcT q  )
{
        CalcT w0 = 2 * kPi * normFreq;
        CalcT sn, cs;
        fastsincos( w0, &sn, &cs );
        SetupCommon( sn, cs, q );
}

//--------------------------------------------------------------------------

template<int channels>
class BiquadHighPass : public Biquad<channels>
{
public:
        void Setup                      ( CalcT normFreq, CalcT q );
        void SetupFast          ( CalcT normFreq, CalcT q );
protected:
        void SetupCommon        ( CalcT sn, CalcT cs, CalcT q );
};

//--------------------------------------------------------------------------

template<int channels>
inline void BiquadHighPass<channels>::SetupCommon( CalcT sn, CalcT cs, CalcT q  )
{
        CalcT alph = sn / ( 2 * q );
        CalcT a0 = -1 / ( 1 + alph );
        CalcT b1 = -( 1 + cs );
        CalcT b0 = a0 * b1 * -0.5;
        CalcT a1 = -2 * cs;
        CalcT a2 =  1 - alph;
        SetStage( a1*a0, a2*a0, b0, b1*a0, b0 );
}
```

(continues on next page)

(continued from previous page)

```

template<int channels>
void BiquadHighPass<channels>::Setup( CalcT normFreq, CalcT q )
{
        CalcT w0 = 2 * kPi * normFreq;
        CalcT cs = cos(w0);
        CalcT sn = sin(w0);
        SetupCommon( sn, cs, q );
}

template<int channels>
void BiquadHighPass<channels>::SetupFast( CalcT normFreq, CalcT q )
{
        CalcT w0 = 2 * kPi * normFreq;
        CalcT sn, cs;
        fastsincos( w0, &sn, &cs );
        SetupCommon( sn, cs, q );
}

//----------------------------------------------------------------------------

// Constant skirt gain, peak gain=Q
template<int channels>
class BiquadBandPass1 : public Biquad<channels>
{
public:
        void Setup                      ( CalcT normFreq, CalcT q );
        void SetupFast          ( CalcT normFreq, CalcT q );
protected:
        void SetupCommon        ( CalcT sn, CalcT cs, CalcT q );
};

//----------------------------------------------------------------------------

template<int channels>
inline void BiquadBandPass1<channels>::SetupCommon( CalcT sn, CalcT cs, CalcT q  )
{
        CalcT alph = sn / ( 2 * q );
        CalcT a0 = -1 / ( 1 + alph );
        CalcT b0 = a0 * ( sn * -0.5 );
        CalcT a1 = -2 * cs;
        CalcT a2 =  1 - alph;
        SetStage( a1*a0, a2*a0, b0, 0, -b0 );
}

template<int channels>
void BiquadBandPass1<channels>::Setup( CalcT normFreq, CalcT q )
{
        CalcT w0 = 2 * kPi * normFreq;
        CalcT cs = cos(w0);
        CalcT sn = sin(w0);
        SetupCommon( sn, cs, q );
}

template<int channels>
void BiquadBandPass1<channels>::SetupFast( CalcT normFreq, CalcT q )
{
```

(continues on next page)

(continued from previous page)

```
                CalcT w0 = 2 * kPi * normFreq;
                CalcT sn, cs;
                fastsincos( w0, &sn, &cs );
                SetupCommon( sn, cs, q );
        }

        //------------------------------------------------------------------------------

        // Constant 0dB peak gain
        template<int channels>
        class BiquadBandPass2 : public Biquad<channels>
        {
        public:
                void Setup                              ( CalcT normFreq, CalcT q );
                void SetupFast                  ( CalcT normFreq, CalcT q );
        protected:
                void SetupCommon                ( CalcT sn, CalcT cs, CalcT q );
        };

        //------------------------------------------------------------------------------

        template<int channels>
        inline void BiquadBandPass2<channels>::SetupCommon( CalcT sn, CalcT cs, CalcT q  )
        {
                CalcT alph = sn / ( 2 * q );
                CalcT b0 = -alph;
                CalcT b2 =  alph;
                CalcT a0 = -1 / ( 1 + alph );
                CalcT a1 = -2 * cs;
                CalcT a2 =  1 - alph;
                SetStage( a1*a0, a2*a0, b0*a0, 0, b2*a0 );
        }

        template<int channels>
        void BiquadBandPass2<channels>::Setup( CalcT normFreq, CalcT q )
        {
                CalcT w0 = 2 * kPi * normFreq;
                CalcT cs = cos(w0);
                CalcT sn = sin(w0);
                SetupCommon( sn, cs, q );
        }

        template<int channels>
        void BiquadBandPass2<channels>::SetupFast( CalcT normFreq, CalcT q )
        {
                CalcT w0 = 2 * kPi * normFreq;
                CalcT sn, cs;
                fastsincos( w0, &sn, &cs );
                SetupCommon( sn, cs, q );
        }

        //------------------------------------------------------------------------------

        template<int channels>
        class BiquadBandStop : public Biquad<channels>
        {
        public:
```

(continues on next page)

(continued from previous page)

```
        void Setup                      ( CalcT normFreq, CalcT q );
        void SetupFast          ( CalcT normFreq, CalcT q );
protected:
        void SetupCommon        ( CalcT sn, CalcT cs, CalcT q );
};

//----------------------------------------------------------------------------

template<int channels>
inline void BiquadBandStop<channels>::SetupCommon( CalcT sn, CalcT cs, CalcT q  )
{
        CalcT alph = sn / ( 2 * q );
        CalcT a0 = 1 / ( 1 + alph );
        CalcT b1 = a0 * ( -2 * cs );
        CalcT a2 = alph - 1;
        SetStage( -b1, a2*a0, a0, b1, a0 );
}

template<int channels>
void BiquadBandStop<channels>::Setup( CalcT normFreq, CalcT q )
{
        CalcT w0 = 2 * kPi * normFreq;
        CalcT cs = cos(w0);
        CalcT sn = sin(w0);
        SetupCommon( sn, cs, q );
}

template<int channels>
void BiquadBandStop<channels>::SetupFast( CalcT normFreq, CalcT q )
{
        CalcT w0 = 2 * kPi * normFreq;
        CalcT sn, cs;
        fastsincos( w0, &sn, &cs );
        SetupCommon( sn, cs, q );
}

//----------------------------------------------------------------------------

template<int channels>
class BiquadAllPass: public Biquad<channels>
{
public:
        void Setup                      ( CalcT normFreq, CalcT q );
        void SetupFast          ( CalcT normFreq, CalcT q );
protected:
        void SetupCommon        ( CalcT sn, CalcT cs, CalcT q );
};

//----------------------------------------------------------------------------

template<int channels>
void BiquadAllPass<channels>::SetupCommon( CalcT sn, CalcT cs, CalcT q  )
{
        CalcT alph = sn / ( 2 * q );
        CalcT b2 =  1 + alph;
        CalcT a0 =  1 / b2;
        CalcT b0 =( 1 - alph ) * a0;
```

(continues on next page)

(continued from previous page)

```
            CalcT b1 = -2 * cs * a0;
            SetStage( -b1, -b0, b0, b1, b2*a0 );
    }

    template<int channels>
    void BiquadAllPass<channels>::Setup( CalcT normFreq, CalcT q )
    {
            CalcT w0 = 2 * kPi * normFreq;
            CalcT cs = cos(w0);
            CalcT sn = sin(w0);
            SetupCommon( sn, cs, q );
    }

    template<int channels>
    void BiquadAllPass<channels>::SetupFast( CalcT normFreq, CalcT q )
    {
            CalcT w0 = 2 * kPi * normFreq;
            CalcT sn, cs;
            fastsincos( w0, &sn, &cs );
            SetupCommon( sn, cs, q );
    }

    //--------------------------------------------------------------------------

    template<int channels>
    class BiquadPeakEq: public Biquad<channels>
    {
    public:
            void Setup                      ( CalcT normFreq, CalcT dB, CalcT bandWidth );
            void SetupFast          ( CalcT normFreq, CalcT dB, CalcT bandWidth );
    protected:
            void SetupCommon        ( CalcT sn, CalcT cs, CalcT alph, CalcT A );
    };

    //--------------------------------------------------------------------------

    template<int channels>
    inline void BiquadPeakEq<channels>::SetupCommon(
            CalcT sn, CalcT cs, CalcT alph, CalcT A )
    {
            CalcT t=alph*A;
            CalcT b0 =  1 - t;
            CalcT b2 =  1 + t;
            t=alph/A;
            CalcT a0 =  1 / ( 1 + t );
            CalcT a2 =  t - 1;
            CalcT b1 = a0 * ( -2 * cs );
            CalcT a1 = -b1;

            SetStage( a1, a2*a0, b0*a0, b1, b2*a0 );
    }

    template<int channels>
    void BiquadPeakEq<channels>::Setup( CalcT normFreq, CalcT dB, CalcT bandWidth )
    {
            CalcT A  = pow( 10, dB/40 );
```

(continues on next page)

(continued from previous page)

```
		CalcT w0 = 2 * kPi * normFreq;
		CalcT cs = cos(w0);
		CalcT sn = sin(w0);
		CalcT alph = sn * sinh( kLn2/2 * bandWidth * w0/sn );
		SetupCommon( sn, cs, alph, A );
	}

	template<int channels>
	void BiquadPeakEq<channels>::SetupFast( CalcT normFreq, CalcT dB, CalcT bandWidth )
	{
		CalcT A  = pow( 10, dB/40 );
		CalcT w0 = 2 * kPi * normFreq;
		CalcT sn, cs;
		fastsincos( w0, &sn, &cs );
		CalcT alph = sn * sinh( kLn2/2 * bandWidth * w0/sn );
		SetupCommon( sn, cs, alph, A );
	}

	//--------------------------------------------------------------------------

	template<int channels>
	class BiquadLowShelf : public Biquad<channels>
	{
	public:
		void Setup                      ( CalcT normFreq, CalcT dB, CalcT shelfSlope=1.0 );
		void SetupFast          ( CalcT normFreq, CalcT dB, CalcT shelfSlope=1.0 );
	protected:
		void SetupCommon        ( CalcT cs, CalcT A, CalcT sa );
	};

	//--------------------------------------------------------------------------

	template<int channels>
	inline void BiquadLowShelf<channels>::SetupCommon(
		CalcT cs, CalcT A, CalcT sa )
	{
		CalcT An        = A-1;
		CalcT Ap        = A+1;
		CalcT Ancs      = An*cs;
		CalcT Apcs      = Ap*cs;
		CalcT b0 =          A * (Ap - Ancs + sa );
		CalcT b2 =          A * (Ap - Ancs - sa );
		CalcT b1 = 2 * A * (An - Apcs);
		CalcT a2 =    sa - (Ap + Ancs);
		CalcT a0 =          1 / (Ap + Ancs + sa );
		CalcT a1 = 2 *      (An + Apcs);
		SetStage( a1*a0, a2*a0, b0*a0, b1*a0, b2*a0 );
	}

	template<int channels>
	void BiquadLowShelf<channels>::Setup( CalcT normFreq, CalcT dB, CalcT shelfSlope )
	{
		CalcT A  = pow( 10, dB/40 );
		CalcT w0 = 2 * kPi * normFreq;
```

(continues on next page)

(continued from previous page)

```
            CalcT cs = cos(w0);
            CalcT sn = sin(w0);
            CalcT al = sn / 2 * ::sqrt( (A + 1/A) * (1/shelfSlope - 1) + 2 );
            CalcT sa =              2 * ::sqrt( A ) * al;
            SetupCommon( cs, A, sa );
    }

    // This could be optimized further
    template<int channels>
    void BiquadLowShelf<channels>::SetupFast( CalcT normFreq, CalcT dB, CalcT␣
↪shelfSlope )
    {
            CalcT A  = pow( 10, dB/40 );
            CalcT w0 = 2 * kPi * normFreq;
            CalcT sn, cs;
            fastsincos( w0, &sn, &cs );
            CalcT al = sn / 2 * fastsqrt1( (A + 1/A) * (1/shelfSlope - 1) + 2 );
            CalcT sa =              2 * fastsqrt1( A ) * al;
            SetupCommon( cs, A, sa );
    }

    //--------------------------------------------------------------------------

    template<int channels>
    class BiquadHighShelf : public Biquad<channels>
    {
    public:
            void Setup                      ( CalcT normFreq, CalcT dB, CalcT␣
↪shelfSlope=1.0 );
            void SetupFast          ( CalcT normFreq, CalcT dB, CalcT shelfSlope=1.0␣
↪);
    protected:
            void SetupCommon        ( CalcT cs, CalcT A, CalcT sa );
    };

    //--------------------------------------------------------------------------

    template<int channels>
    void BiquadHighShelf<channels>::SetupCommon(
            CalcT cs, CalcT A, CalcT sa )
    {
            CalcT An        = A-1;
            CalcT Ap        = A+1;
            CalcT Ancs      = An*cs;
            CalcT Apcs      = Ap*cs;
            CalcT b0 =              A * (Ap + Ancs + sa );
            CalcT b1 = -2 * A * (An + Apcs);
            CalcT b2 =              A * (Ap + Ancs - sa );
            CalcT a0 =                      (Ap - Ancs + sa );
            CalcT a2 =                       Ancs + sa - Ap;
            CalcT a1 = -2   *       (An - Apcs);
            SetStage( a1/a0, a2/a0, b0/a0, b1/a0, b2/a0 );
    }

    template<int channels>
    void BiquadHighShelf<channels>::Setup( CalcT normFreq, CalcT dB, CalcT shelfSlope␣
↪)
```

(continues on next page)

(continued from previous page)

```
	{
		CalcT A  = pow( 10, dB/40 );
		CalcT w0 = 2 * kPi * normFreq;
		CalcT cs = cos(w0);
		CalcT sn = sin(w0);

		CalcT alph = sn / 2 * ::sqrt( (A + 1/A) * (1/shelfSlope - 1) + 2 );
		CalcT sa   =      2 * ::sqrt( A ) * alph;
		SetupCommon( cs, A, sa );
	}

	template<int channels>
	void BiquadHighShelf<channels>::SetupFast( CalcT normFreq, CalcT dB, CalcT shelfSlope )
	{
		CalcT A  = pow( 10, dB/40 );
		CalcT w0 = 2 * kPi * normFreq;
		CalcT sn, cs;
		fastsincos( w0, &sn, &cs );

		CalcT alph = sn / 2 * fastsqrt1( (A + 1/A) * (1/shelfSlope - 1) + 2 );
		CalcT sa   =      2 * fastsqrt1( A ) * alph;
		SetupCommon( cs, A, sa );
	}

	//--------------------------------------------------------------------------
	//
	//	General N-Pole IIR Filter
	//
	//--------------------------------------------------------------------------

	template<int stages, int channels>
	class PoleFilter : public CascadeStages<stages, channels>
	{
	public:
		PoleFilter();

		virtual int			CountPoles			( void )=0;
		virtual int			CountZeroes			( void )=0;

		virtual Complex GetPole				( int i )=0;
		virtual Complex GetZero				( int i )=0;

	protected:
		virtual Complex GetSPole			( int i, CalcT wc )=0;

	protected:
		// Determines the method of obtaining
		// unity gain coefficients in the passband.
		enum Hint
		{
			// No normalizating
			hintNone,
			// Use Brent's method to find the maximum
			hintBrent,
			// Use the response at a given frequency
			hintPassband
```

(continues on next page)

(continued from previous page)

```
                };

                Complex BilinearTransform       ( const Complex &c );
                Complex BandStopTransform       ( int i, const Complex &c );
                Complex BandPassTransform       ( int i, const Complex &c );
                Complex GetBandStopPole         ( int i );
                Complex GetBandStopZero         ( int i );
                Complex GetBandPassPole         ( int i );
                Complex GetBandPassZero         ( int i );
                void    Normalize                       ( void );
                void    Prepare                         ( void );

                virtual void    BrentHint       ( CalcT *w0, CalcT *w1 );
                virtual CalcT   PassbandHint( void );

        protected:
                Hint    m_hint;
                int             m_n;
                CalcT   m_wc;
                CalcT   m_wc2;
        };

        //--------------------------------------------------------------------------

        template<int stages, int channels>
        inline PoleFilter<stages, channels>::PoleFilter( void )
        {
                m_hint=hintNone;
        }

        template<int stages, int channels>
        inline Complex PoleFilter<stages, channels>::BilinearTransform( const Complex &c )
        {
                return (c+1.)/(-c+1.);
        }

        template<int stages, int channels>
        inline Complex PoleFilter<stages, channels>::BandStopTransform( int i, const Complex &c )
        {
                CalcT a=cos((m_wc+m_wc2)*.5) /
                                cos((m_wc-m_wc2)*.5);
                CalcT b=tan((m_wc-m_wc2)*.5);
            Complex c2(0);
            c2=addmul( c2, 4*(b*b+a*a-1), c );
            c2+=8*(b*b-a*a+1);
            c2*=c;
            c2+=4*(a*a+b*b-1);
                c2=Dsp::sqrt( c2 );
            c2*=((i&1)==0)?.5:-.5;
            c2+=a;
            c2=addmul( c2, -a, c );
            Complex c3( b+1 );
            c3=addmul( c3, b-1, c );
                return c2/c3;
        }

```

(continues on next page)

(continued from previous page)

```
    template<int stages, int channels>
    inline Complex PoleFilter<stages, channels>::BandPassTransform( int i, const␣
↪Complex &c )
    {
            CalcT a=  cos((m_wc+m_wc2)*0.5)/
                                  cos((m_wc-m_wc2)*0.5);
            CalcT b=1/tan((m_wc-m_wc2)*0.5);
            Complex c2(0);
        c2=addmul( c2, 4*(b*b*(a*a-1)+1), c );
        c2+=8*(b*b*(a*a-1)-1);
        c2*=c;
        c2+=4*(b*b*(a*a-1)+1);
            c2=Dsp::sqrt( c2 );
        if ((i & 1) == 0)
                    c2=-c2;
        c2=addmul( c2, 2*a*b, c );
        c2+=2*a*b;
        Complex c3(0);
        c3=addmul( c3, 2*(b-1), c );
        c3+=2*(1+b);
        return c2/c3;
    }

    template<int stages, int channels>
    Complex PoleFilter<stages, channels>::GetBandStopPole( int i )
    {
            Complex c=GetSPole( i/2, kPi_2 );
            c=BilinearTransform( c );
            c=BandStopTransform( i, c );
            return c;
    }

    template<int stages, int channels>
    Complex PoleFilter<stages, channels>::GetBandStopZero( int i )
    {
            return BandStopTransform( i, Complex( -1 ) );
    }

    template<int stages, int channels>
    Complex PoleFilter<stages, channels>::GetBandPassPole( int i )
    {
            Complex c=GetSPole( i/2, kPi_2 );
            c=BilinearTransform( c );
            c=BandPassTransform( i, c );
            return c;
    }

    template<int stages, int channels>
    Complex PoleFilter<stages, channels>::GetBandPassZero( int i )
    {
            return Complex( (i>=m_n)?1:-1 );
    }

    template<int stages, int channels>
    void PoleFilter<stages, channels>::Normalize( void )
    {
            switch( m_hint )
```

(continues on next page)

(continued from previous page)

```
			{
			default:
			case hintNone:
				break;

			case hintPassband:
				{
					CalcT w=PassbandHint();
					ResponseFunctor f(this);
					CalcT mag=-f(w);
					CascadeStages::Normalize( 1/mag );
				}
				break;

			case hintBrent:
				{
					ResponseFunctor f(this);
					CalcT w0, w1, wmin, mag;
					BrentHint( &w0, &w1 );
					mag=-BrentMinimize( f, w0, w1, 1e-4, wmin );
					CascadeStages::Normalize( 1/mag );
				}
				break;
			}
		}

	template<int stages, int channels>
	void PoleFilter<stages, channels>::Prepare( void )
	{
		if( m_wc2<1e-8 )
			m_wc2=1e-8;
		if( m_wc >kPi-1e-8 )
			m_wc =kPi-1e-8;

		Reset();

		Complex c;
		int poles=CountPoles();
		for( int i=0;i<poles;i++ )
		{
			c=GetPole( i );
			if( ::abs(c.imag())<1e-6 )
				c=Complex( c.real(), 0 );
			if( c.imag()==0 )
				SetAStage( c.real(), 0 );
			else if( c.imag()>0 )
				SetAStage( 2*c.real(), -Dsp::norm(c) );
		}

		int zeroes=CountZeroes();
		for( int i=0;i<zeroes;i++ )
		{
			c=GetZero( i );
			if( ::abs(c.imag())<1e-6 )
				c=Complex( c.real(), 0 );
			if( c.imag()==0 )
				SetBStage( -c.real(), 1, 0 );
```

(continues on next page)

(continued from previous page)

```
                    else if( c.imag()>0 )
                            SetBStage( Dsp::norm(c), -2*c.real(), 1 );
            }

            Normalize();
    }

    template<int stages, int channels>
    void PoleFilter<stages, channels>::BrentHint( CalcT *w0, CalcT *w1 )
    {
            // best that this never executes
            *w0=1e-4;
            *w1=kPi-1e-4;
    }

    template<int stages, int channels>
    CalcT PoleFilter<stages, channels>::PassbandHint( void )
    {
            // should never get here
            assert( 0 );
            return kPi_2;
    }

    //--------------------------------------------------------------------------
    //
    //      Butterworth Response IIR Filter
    //
    //--------------------------------------------------------------------------

    // Butterworth filter response characteristic.
    // Maximally flat magnitude response in the passband at the
    // expense of a more shallow rolloff in comparison to other types.
    template<int poles, int channels>
    class Butterworth : public PoleFilter<int((poles+1)/2), channels>
    {
    public:
            Butterworth();

            // cutoffFreq = freq / sampleRate
            void                    Setup                   ( CalcT cutoffFreq );

            virtual int             CountPoles              ( void );
            virtual int             CountZeroes             ( void );
            virtual Complex GetPole                 ( int i );

    protected:
                            Complex GetSPole                ( int i, CalcT wc );
    };

    //--------------------------------------------------------------------------

    template<int poles, int channels>
    Butterworth<poles, channels>::Butterworth( void )
    {
            m_hint=hintPassband;
    }

```

(continues on next page)

(continued from previous page)

```
	template<int poles, int channels>
	void Butterworth<poles, channels>::Setup( CalcT cutoffFreq )
	{
		m_n=poles;
		m_wc=2*kPi*cutoffFreq;
		Prepare();
	}

	template<int poles, int channels>
	int Butterworth<poles, channels>::CountPoles( void )
	{
		return poles;
	}

	template<int poles, int channels>
	int Butterworth<poles, channels>::CountZeroes( void )
	{
		return poles;
	}

	template<int poles, int channels>
	Complex Butterworth<poles, channels>::GetPole( int i )
	{
		return BilinearTransform( GetSPole( i, m_wc ) );
	}

	template<int poles, int channels>
	Complex Butterworth<poles, channels>::GetSPole( int i, CalcT wc )
	{
		return polar( tan(wc*0.5), kPi_2+(2*i+1)*kPi/(2*m_n) );
	}

	//--------------------------------------------------------------------------

	// Low Pass Butterworth filter
	// Stable up to 53 poles (frequency min=0.13% of Nyquist)
	template<int poles, int channels>
	class ButterLowPass : public Butterworth<poles, channels>
	{
	public:
		Complex GetZero			( int i );

	protected:
		CalcT	PassbandHint	( void );
	};

	//--------------------------------------------------------------------------

	template<int poles, int channels>
	Complex ButterLowPass<poles, channels>::GetZero( int i )
	{
		return Complex( -1 );
	}

	template<int poles, int channels>
	CalcT ButterLowPass<poles, channels>::PassbandHint( void )
	{
```

(continues on next page)

(continued from previous page)

```
		return 0;
	}

	//------------------------------------------------------------------------------

	// High Pass Butterworth filter
	// Maximally flat magnitude response in the passband.
	// Stable up to 110 poles (frequency max=97% of Nyquist)
	template<int poles, int channels>
	class ButterHighPass : public Butterworth<poles, channels>
	{
	public:
		Complex GetZero( int i );

	protected:
		CalcT	PassbandHint	( void );
	};

	//------------------------------------------------------------------------------

	template<int poles, int channels>
	Complex ButterHighPass<poles, channels>::GetZero( int i )
	{
		return Complex( 1 );
	}

	template<int poles, int channels>
	CalcT ButterHighPass<poles, channels>::PassbandHint( void )
	{
		return kPi;
	}

	//------------------------------------------------------------------------------

	// Band Pass Butterworth filter
	// Stable up to 80 pairs
	template<int pairs, int channels>
	class ButterBandPass : public Butterworth<pairs*2, channels>
	{
	public:
		// centerFreq = freq / sampleRate
		// normWidth  = freqWidth / sampleRate
		void			Setup			( CalcT centerFreq, CalcT
↪normWidth );

		virtual int		CountPoles		( void );
		virtual int		CountZeroes		( void );
		virtual Complex GetPole			( int i );
		virtual Complex GetZero			( int i );

	protected:
		CalcT	PassbandHint	( void );
	};

	//------------------------------------------------------------------------------

	template<int pairs, int channels>
```

(continues on next page)

(continued from previous page)

```
void ButterBandPass<pairs, channels>::Setup( CalcT centerFreq, CalcT normWidth )
{
        m_n=pairs;
        CalcT angularWidth=2*kPi*normWidth;
        m_wc2=2*kPi*centerFreq-(angularWidth/2);
        m_wc =m_wc2+angularWidth;
        Prepare();
}

template<int pairs, int channels>
int ButterBandPass<pairs, channels>::CountPoles( void )
{
        return pairs*2;
}

template<int pairs, int channels>
int ButterBandPass<pairs, channels>::CountZeroes( void )
{
        return pairs*2;
}

template<int pairs, int channels>
Complex ButterBandPass<pairs, channels>::GetPole( int i )
{
        return GetBandPassPole( i );
}

template<int pairs, int channels>
Complex ButterBandPass<pairs, channels>::GetZero( int i )
{
        return GetBandPassZero( i );
}

template<int poles, int channels>
CalcT ButterBandPass<poles, channels>::PassbandHint( void )
{
        return (m_wc+m_wc2)/2;
}

//--------------------------------------------------------------------------

// Band Stop Butterworth filter
// Stable up to 109 pairs
template<int pairs, int channels>
class ButterBandStop : public Butterworth<pairs*2, channels>
{
public:
        // centerFreq = freq / sampleRate
        // normWidth  = freqWidth / sampleRate
        void    Setup                   ( CalcT centerFreq, CalcT normWidth );

        virtual int             CountPoles              ( void );
        virtual int             CountZeroes             ( void );
        virtual Complex GetPole                 ( int i );
        virtual Complex GetZero                 ( int i );

protected:
```

(continues on next page)

(continued from previous page)

```
                CalcT   PassbandHint    ( void );
        };

        //--------------------------------------------------------------------------

        template<int pairs, int channels>
        void ButterBandStop<pairs, channels>::Setup( CalcT centerFreq, CalcT normWidth )
        {
                m_n=pairs;
                CalcT angularWidth=2*kPi*normWidth;
                m_wc2=2*kPi*centerFreq-(angularWidth/2);
                m_wc =m_wc2+angularWidth;
                Prepare();
        }

        template<int pairs, int channels>
        int ButterBandStop<pairs, channels>::CountPoles( void )
        {
                return pairs*2;
        }

        template<int pairs, int channels>
        int ButterBandStop<pairs, channels>::CountZeroes( void )
        {
                return pairs*2;
        }

        template<int pairs, int channels>
        Complex ButterBandStop<pairs, channels>::GetPole( int i )
        {
                return GetBandStopPole( i );
        }

        template<int pairs, int channels>
        Complex ButterBandStop<pairs, channels>::GetZero( int i )
        {
                return GetBandStopZero( i );
        }

        template<int poles, int channels>
        CalcT ButterBandStop<poles, channels>::PassbandHint( void )
        {
                if( (m_wc+m_wc2)/2<kPi_2 )
                        return kPi;
                else
                        return 0;
        }

        //--------------------------------------------------------------------------
        //
        //      Chebyshev Response IIR Filter
        //
        //--------------------------------------------------------------------------

        // Type I Chebyshev filter characteristic.
        // Minimum error between actual and ideal response at the expense of
        // a user-definable amount of ripple in the passband.
```

(continues on next page)

(continued from previous page)

```
template<int poles, int channels>
class Chebyshev1 : public PoleFilter<int((poles+1)/2), channels>
{
public:
                                        Chebyshev1();

        // cutoffFreq = freq / sampleRate
        virtual void    Setup                   ( CalcT cutoffFreq, CalcT rippleDb );

        virtual int             CountPoles              ( void );
        virtual int             CountZeroes             ( void );
        virtual Complex GetPole                 ( int i );
        virtual Complex GetZero                 ( int i );

protected:
        void                    SetupCommon             ( CalcT rippleDb );
        virtual Complex GetSPole                ( int i, CalcT wc );

protected:
        CalcT   m_sgn;
        CalcT   m_eps;
};

//------------------------------------------------------------------------------

template<int poles, int channels>
Chebyshev1<poles, channels>::Chebyshev1()
{
        m_hint=hintBrent;
}

template<int poles, int channels>
void Chebyshev1<poles, channels>::Setup( CalcT cutoffFreq, CalcT rippleDb )
{
        m_n=poles;
        m_wc=2*kPi*cutoffFreq;
        SetupCommon( rippleDb );
}

template<int poles, int channels>
void Chebyshev1<poles, channels>::SetupCommon( CalcT rippleDb )
{
        m_eps=::sqrt( 1/::exp( -rippleDb*0.1*kLn10 )-1 );
        Prepare();
        // This moves the bottom of the ripples to 0dB gain
        //CascadeStages::Normalize( pow( 10, rippleDb/20.0 ) );
}

template<int poles, int channels>
int     Chebyshev1<poles, channels>::CountPoles( void )
{
        return poles;
}

template<int poles, int channels>
int Chebyshev1<poles, channels>::CountZeroes( void )
```

(continues on next page)

(continued from previous page)

```
{
        return poles;
}

template<int poles, int channels>
Complex Chebyshev1<poles, channels>::GetPole( int i )
{
        return BilinearTransform( GetSPole( i, m_wc ) )*m_sgn;
}

template<int poles, int channels>
Complex Chebyshev1<poles, channels>::GetZero( int i )
{
        return Complex( -m_sgn );
}

template<int poles, int channels>
Complex Chebyshev1<poles, channels>::GetSPole( int i, CalcT wc )
{
        int n            = m_n;
        CalcT ni         = 1.0/n;
        CalcT alpha      = 1/m_eps+::sqrt(1+1/(m_eps*m_eps));
        CalcT pn         = pow( alpha,  ni );
        CalcT nn         = pow( alpha, -ni );
        CalcT a          = 0.5*( pn - nn );
        CalcT b          = 0.5*( pn + nn );
        CalcT theta = kPi_2 + (2*i+1) * kPi/(2*n);
        Complex c        = polar( tan( 0.5*(m_sgn==-1?(kPi-wc):wc) ), theta );
        return Complex( a*c.real(), b*c.imag() );
}

//--------------------------------------------------------------------------

// Low Pass Chebyshev Type I filter
template<int poles, int channels>
class Cheby1LowPass : public Chebyshev1<poles, channels>
{
public:
                        Cheby1LowPass();

        void    Setup                   ( CalcT cutoffFreq, CalcT rippleDb );

protected:
        CalcT   PassbandHint    ( void );
};

//--------------------------------------------------------------------------

template<int poles, int channels>
Cheby1LowPass<poles, channels>::Cheby1LowPass()
{
        m_sgn=1;
        m_hint=hintPassband;
}

template<int poles, int channels>
void Cheby1LowPass<poles, channels>::Setup( CalcT cutoffFreq, CalcT rippleDb )
```

(continues on next page)

(continued from previous page)

```
{
        Chebyshev1::Setup( cutoffFreq, rippleDb );
        // move peak of ripple down to 0dB
        if( !(poles&1) )
                CascadeStages::Normalize( pow( 10, -rippleDb/20.0 ) );
}

template<int poles, int channels>
CalcT Cheby1LowPass<poles, channels>::PassbandHint( void )
{
        return 0;
}

//--------------------------------------------------------------------------

// High Pass Chebyshev Type I filter
template<int poles, int channels>
class Cheby1HighPass : public Chebyshev1<poles, channels>
{
public:
        Cheby1HighPass();

        void    Setup                   ( CalcT cutoffFreq, CalcT rippleDb );

protected:
        CalcT   PassbandHint    ( void );
};

//--------------------------------------------------------------------------

template<int poles, int channels>
Cheby1HighPass<poles, channels>::Cheby1HighPass()
{
        m_sgn=-1;
        m_hint=hintPassband;
}

template<int poles, int channels>
void Cheby1HighPass<poles, channels>::Setup( CalcT cutoffFreq, CalcT rippleDb )
{
        Chebyshev1::Setup( cutoffFreq, rippleDb );
        // move peak of ripple down to 0dB
        if( !(poles&1) )
                CascadeStages::Normalize( pow( 10, -rippleDb/20.0 ) );
}

template<int poles, int channels>
CalcT Cheby1HighPass<poles, channels>::PassbandHint( void )
{
        return kPi;
}

//--------------------------------------------------------------------------

// Band Pass Chebyshev Type I filter
template<int pairs, int channels>
class Cheby1BandPass : public Chebyshev1<pairs*2, channels>
```

(continues on next page)

(continued from previous page)

```
	{
	public:
		Cheby1BandPass();

		void	Setup			( CalcT centerFreq, CalcT normWidth, CalcT rippleDb );

		int		CountPoles		( void );
		int		CountZeroes		( void );
		Complex GetPole			( int i );
		Complex GetZero			( int i );

	protected:
		void	BrentHint		( CalcT *w0, CalcT *w1 );
		//CalcT PassbandHint	( void );
	};

	//--------------------------------------------------------------------------

	template<int pairs, int channels>
	Cheby1BandPass<pairs, channels>::Cheby1BandPass()
	{
		m_sgn=1;
		m_hint=hintBrent;
	}

	template<int pairs, int channels>
	void Cheby1BandPass<pairs, channels>::Setup( CalcT centerFreq, CalcT normWidth, CalcT rippleDb )
	{
		m_n=pairs;
		CalcT angularWidth=2*kPi*normWidth;
		m_wc2=2*kPi*centerFreq-(angularWidth/2);
		m_wc =m_wc2+angularWidth;
		SetupCommon( rippleDb );
	}

	template<int pairs, int channels>
	int Cheby1BandPass<pairs, channels>::CountPoles( void )
	{
		return pairs*2;
	}

	template<int pairs, int channels>
	int Cheby1BandPass<pairs, channels>::CountZeroes( void )
	{
		return pairs*2;
	}

	template<int pairs, int channels>
	Complex Cheby1BandPass<pairs, channels>::GetPole( int i )
	{
		return GetBandPassPole( i );
	}

	template<int pairs, int channels>
	Complex Cheby1BandPass<pairs, channels>::GetZero( int i )
```

(continues on next page)

(continued from previous page)

```
{
        return GetBandPassZero( i );
}

template<int poles, int channels>
void Cheby1BandPass<poles, channels>::BrentHint( CalcT *w0, CalcT *w1 )
{
        CalcT d=1e-4*(m_wc-m_wc2)/2;
        *w0=m_wc2+d;
        *w1=m_wc-d;
}

/*
// Unfortunately, this doesn't work at the frequency extremes
// Maybe we can inverse pre-warp the center point to make sure
// it stays put after bilinear and bandpass transformation.
template<int poles, int channels>
CalcT Cheby1BandPass<poles, channels>::PassbandHint( void )
{
        return (m_wc+m_wc2)/2;
}
*/

//----------------------------------------------------------------------------

// Band Stop Chebyshev Type I filter
template<int pairs, int channels>
class Cheby1BandStop : public Chebyshev1<pairs*2, channels>
{
public:
        Cheby1BandStop();

        void    Setup                   ( CalcT centerFreq, CalcT normWidth, CalcT rippleDb );

        int             CountPoles              ( void );
        int             CountZeroes             ( void );
        Complex GetPole                 ( int i );
        Complex GetZero                 ( int i );

protected:
        void    BrentHint               ( CalcT *w0, CalcT *w1 );
        CalcT   PassbandHint    ( void );
};

//----------------------------------------------------------------------------

template<int pairs, int channels>
Cheby1BandStop<pairs, channels>::Cheby1BandStop()
{
        m_sgn=1;
        m_hint=hintPassband;
}

template<int pairs, int channels>
```

3.12.1 Comments

- **Date**: 2010-06-18 10:53:04
- **By**: ten.nozirev@nuSL

```
These codes are just what I am looking for.  Too bad they are incomplete as posted␣
↪here.  Could someone direct me to a complete version?
```

- **Date**: 2011-02-03 16:37:34
- **By**: moc.tenalpderyrgna@nrobkcah

```
I *think* this is the project homepage:
http://code.google.com/p/dspfilterscpp/
although I haven't downloaded anything to verify it is and that the code is complete.
```

- **Date**: 2012-05-06 01:26:08
- **By**: moc.liamg@oclaf.einniv

```
The project is here:
https://github.com/vinniefalco/DSPFilters.git
```

3.13 Butterworth Optimized C++ Class

- **Author or source:** neotec
- **Type:** 24db Resonant Lowpass
- **Created:** 2007-01-20 22:41:06

Listing 18: notes

```
This ist exactly the same as posted by "Zxform" (filters004.txt). The only difference␣
↪is,
that this version is an optimized one.

Parameters:
Cutoff [0.f -> Nyquist.f]
Resonance [0.f -> 1.f]

There are some minima and maxima defined, to make ist sound nice in all situations.␣
↪This
class is part of some of my VST Plugins, and works well and executes fast.
```

Listing 19: code

```
// FilterButterworth24db.h

#pragma once

class CFilterButterworth24db
{
public:
    CFilterButterworth24db(void);
    ~CFilterButterworth24db(void);
```

(continues on next page)

(continued from previous page)

```
    void SetSampleRate(float fs);
    void Set(float cutoff, float q);
    float Run(float input);

private:
    float t0, t1, t2, t3;
    float coef0, coef1, coef2, coef3;
    float history1, history2, history3, history4;
    float gain;
    float min_cutoff, max_cutoff;
};

// FilterButterworth24db.cpp

#include <math.h>

#define BUDDA_Q_SCALE 6.f

#include "FilterButterworth24db.h"

CFilterButterworth24db::CFilterButterworth24db(void)
{
    this->history1 = 0.f;
    this->history2 = 0.f;
    this->history3 = 0.f;
    this->history4 = 0.f;

    this->SetSampleRate(44100.f);
    this->Set(22050.f, 0.0);
}

CFilterButterworth24db::~CFilterButterworth24db(void)
{
}

void CFilterButterworth24db::SetSampleRate(float fs)
{
    float pi = 4.f * atanf(1.f);

    this->t0 = 4.f * fs * fs;
    this->t1 = 8.f * fs * fs;
    this->t2 = 2.f * fs;
    this->t3 = pi / fs;

    this->min_cutoff = fs * 0.01f;
    this->max_cutoff = fs * 0.45f;
}

void CFilterButterworth24db::Set(float cutoff, float q)
{
    if (cutoff < this->min_cutoff)
            cutoff = this->min_cutoff;
    else if(cutoff > this->max_cutoff)
            cutoff = this->max_cutoff;

    if(q < 0.f)
            q = 0.f;
```

(continues on next page)

(continued from previous page)

```
    else if(q > 1.f)
            q = 1.f;

    float wp = this->t2 * tanf(this->t3 * cutoff);
    float bd, bd_tmp, b1, b2;

    q *= BUDDA_Q_SCALE;
    q += 1.f;

    b1 = (0.765367f / q) / wp;
    b2 = 1.f / (wp * wp);

    bd_tmp = this->t0 * b2 + 1.f;

    bd = 1.f / (bd_tmp + this->t2 * b1);

    this->gain = bd * 0.5f;

    this->coef2 = (2.f - this->t1 * b2);

    this->coef0 = this->coef2 * bd;
    this->coef1 = (bd_tmp - this->t2 * b1) * bd;

    b1 = (1.847759f / q) / wp;

    bd = 1.f / (bd_tmp + this->t2 * b1);

    this->gain *= bd;
    this->coef2 *= bd;
    this->coef3 = (bd_tmp - this->t2 * b1) * bd;
}

float CFilterButterworth24db::Run(float input)
{
    float output = input * this->gain;
    float new_hist;

    output -= this->history1 * this->coef0;
    new_hist = output - this->history2 * this->coef1;

    output = new_hist + this->history1 * 2.f;
    output += this->history2;

    this->history2 = this->history1;
    this->history1 = new_hist;

    output -= this->history3 * this->coef2;
    new_hist = output - this->history4 * this->coef3;

    output = new_hist + this->history3 * 2.f;
    output += this->history4;

    this->history4 = this->history3;
    this->history3 = new_hist;

    return output;
}
```

3.13.1 Comments

- **Date**: 2007-01-22 18:38:23
- **By**: moc.oohay@bob

```
This sounds really nice, especially with resonance. Although it becomes unstable␣
↪below 4K (at 44100 s/r), which explains why the min_cutoff value has been set quite␣
↪high. Would using doubles help stabilise it?
Also, I can't figure out how to get a high pass out of this, can anybody help?
Cheers.
```

- **Date**: 2007-01-22 19:54:21
- **By**: neotec

```
I have checked the peak output of this filter and especially for low frequences ...␣
↪there is a simple fix, which makes it sound better with low frequences: change the␣
↪line in Set(...) that reads 'this->gain = bd * 0.5f;' to 'this->gain = bd;'
```

- **Date**: 2007-01-22 22:27:11
- **By**: moc.oohay@bob

```
Thanks for the quick reply. I've tried your change and it's made a slight tonal␣
↪difference here, but the tests were not particularly scientific. I've discovered␣
↪more detail in the problem, and it's one that has been commented on with other␣
↪filters: If I sweep the filter quickly up or down the low frequencies it blows out␣
↪really badly, even with zero Q. I'm new to filter math, so excuse my ignorance if␣
↪this is a common thing with Butterworth.
```

- **Date**: 2007-01-23 12:54:03
- **By**: neotec

```
Yep ... this filter reacts very extreme on fast cutoff changes. I've added a function␣
↪to my VST Synthesizer, which 'fades' the cutoff value from actual value to the␣
↪desired one in about 0.05 seconds. My modulation envelopes do have similar␣
↪restrictions concerning speed.
```

- **Date**: 2007-01-23 16:35:51
- **By**: neotec

```
If you want to know how this filter sounds, visit the kvraudio forum, and search␣
↪here: "KVR Forum » Instruments" for "Cetone VST Plugins".
```

- **Date**: 2007-01-29 21:57:52
- **By**: moc.erehwon@ydobon

```
I'm wondering about that tanh in the "Set."

Could replace with a pade approximation, maybe. What is the range of inputs going␣
↪into it?

In other words, how small and big does this get?...

this->t3 * cutoff
```

- **Date**: 2007-01-29 22:06:28

- **By**: moc.dniftnacuoyerehwemos@tsaot

```
Possible small optimization. It depends on how smart your compiler is, but sections␣
↪like this...

    output = new_hist + this->history3 * 2.f;
    output += this->history4;

can be changed to this to change the multiply to an addition:

output = this->history3;
output += output+new_Hist+this->history4;
```

- **Date**: 2007-01-30 03:05:05
- **By**: moc.dniftnacuoyerehwemos@tsaot

```
While I'm at it, one of these divisions can easily be switched to a multiply...

 b1 = (1.847759f / q) / wp;

b1=(1.847759f/(q*wp);
```

- **Date**: 2007-02-04 19:32:55
- **By**: moc.oohay@bob

```
Four times oversampling removes the problems with fast cut-off sweeps at low values.
This filter has the same shape as a normal biquad filter, with a more pronounced␣
↪resonance boost.
```

- **Date**: 2008-01-12 20:42:34
- **By**: gro.lortnocdnim@gro.psdcisum

```
Why would oversampling solve the problem? If you over-sample, the poles have to reach␣
↪even further into the relative frequencies, and stability would become more of a␣
↪problem AFAICT.
```

- **Date**: 2008-01-21 14:58:55
- **By**: moc.oohay@bob

```
It just seems to. If you 4X over-sample, then it gives it a 4X chance to recover from␣
↪each sweep change, presuming you're not changing the filter cut-off at 4X also.
```

- **Date**: 2008-01-25 14:46:31
- **By**: erehwon.ku.oc.snosrapsd@psdcisum

```
thing is with 4X oversampling on this is that you'll be reducing precision on omega␣
↪(wp here), and so should probably shift to double rather than float to help␣
↪accuracy.
```

- **Date**: 2013-04-24 02:48:07
- **By**: moc.liam@ttocs

```
Did anyone figure out how to get a high pass out of this?

Thanks!
```

3.14 C++ class implementation of RBJ Filters

- **Author or source:** moc.xinortceletrams@urugra
- **Created:** 2002-12-13 01:37:52
- **Linked files:** CFxRbjFilter.h.

Listing 20: notes

```
[WARNING: This code is not FPU undernormalization safe!]
```

3.15 C-Weighed Filter

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** digital implementation (after bilinear transform)
- **Created:** 2006-07-12 19:12:16

Listing 21: notes

```
unoptimized version!
```

Listing 22: code

```
First prewarp the frequency of both poles:

K1 = tan(0.5*Pi*20.6 / SampleRate) // for 20.6Hz
K2 = tan(0.5*Pi*12200 / SampleRate) // for 12200Hz

Then calculate the both biquads:

 b0 = 1
 b1 = 0
 b2 =-1
 a0 =  ((K1+1)*(K1+1)*(K2+1)*(K2+1));
 a1 =-4*(K1*K1*K2*K2+K1*K1*K2+K1*K2*K2-K1-K2-1)*t;
 a2 =-  ((K1-1)*(K1-1)*(K2-1)*(K2-1))*t;

and:

 b3 = 1
 b4 = 0
 b5 =-1
 a3 =  ((K1+1)*(K1+1)*(K2+1)*(K2+1));
 a4 =-4*(K1*K1*K2*K2+K1*K1*K2+K1*K2*K2-K1-K2-1)*t;
 a5 =-  ((K1-1)*(K1-1)*(K2-1)*(K2-1))*t;

Now use an equation for calculating the biquads like this:

Stage1 = b0*Input                 + State0;
State0 =           + a1/a0*Stage1 + State1;
State1 = b2*Input  + a2/a0*Stage1;

Output = b3*Stage1                + State2;
```

(continues on next page)

(continued from previous page)

```
State2 =            + a4/a3*Output + State2;
State3 = b5*Stage1 + a5/a3*Output;
```

3.15.1 Comments

- **Date**: 2006-07-12 21:07:28
- **By**: ed.luosfosruoivas@naitsirhC

```
You might still need to normalize the filter output. You can do this easily by␣
↪multipliing either the b0 and b2 or the b3 and b5 with a constant.
Typically the filter is normalized to have a gain of 0dB at 1kHz

Also oversampling of this filter might be useful.
```

3.16 Cascaded resonant lp/hp filter

- **Author or source:** ed.bew@raebybot
- **Type:** lp+hp
- **Created:** 2002-12-16 19:02:11

Listing 23: notes

```
// Cascaded resonant lowpass/hipass combi-filter
// The original source for this filter is from Paul Kellet from
// the archive. This is a cascaded version in Delphi where the
// output of the lowpass is fed into the highpass filter.
// Cutoff frequencies are in the range of 0<=x<1 which maps to
// 0..nyquist frequency

// input variables are:
// cut_lp: cutoff frequency of the lowpass (0..1)
// cut_hp: cutoff frequency of the hipass (0..1)
// res_lp: resonance of the lowpass (0..1)
// res_hp: resonance of the hipass (0..1)
```

Listing 24: code

```
var n1,n2,n3,n4:single; // filter delay, init these with 0!
    fb_lp,fb_hp:single; // storage for calculated feedback
const p4=1.0e-24; // Pentium 4 denormal problem elimination

function dofilter(inp,cut_lp,res_lp,cut_hp,res_hp:single):single;
begin
 fb_lp:=res_lp+res_lp/(1-cut_lp);
 fb_hp:=res_hp+res_hp/(1-cut_lp);
 n1:=n1+cut_lp*(inp-n1+fb_lp*(n1-n2))+p4;
 n2:=n2+cut_lp*(n1-n2);
 n3:=n3+cut_hp*(n2-n3+fb_hp*(n3-n4))+p4;
 n4:=n4+cut_hp*(n3-n4);
```

(continues on next page)

(continued from previous page)

```
 result:=i-n4;
end;
```

3.16.1 Comments

- **Date**: 2003-07-13 07:43:17
- **By**: moc.biesnnamreh@eciffo

```
I guess the last line should read
  result:=inp-n4;

Right?

Bye,

  Hermann
```

- **Date**: 2003-12-26 19:56:21
- **By**: moc.liamtoh@tsvreiruoc

```
excuse me which type is? 6db/oct or 12 or what?

thanks
```

- **Date**: 2004-02-02 15:43:00
- **By**: ed.xmg@suahtlanaitsirhc

```
result := n2-n4

:)
```

- **Date**: 2011-01-25 17:52:08
- **By**: ten.raenila@ssov

```
WOW this is old but handy. Anyway what to do about the divide-by-zero caused by the
↪feedback calc if the cutoff is set to 1.0?

Also, should the feedback for the hpf be:

fb_hp:=res_hp+res_hp/(1-cut_hp);

not:

fb_hp:=res_hp+res_hp/(1-cut_lp);

?

Thanks
NV
```

- **Date**: 2017-03-17 08:28:07
- **By**: moc.liamg@dnuosG

```
Nobody can see ?

There is two lowpass filters in series, no differences between them.
```

3.17 Cool Sounding Lowpass With Decibel Measured Resonance

- **Author or source:** ua.moc.ohay@renrew_bocaj_leinad
- **Type:** LP 2-pole resonant tweaked butterworth
- **Created:** 2004-09-01 17:56:44

Listing 25: notes

```
This algorithm is a modified version of the tweaked butterworth lowpass filter by␣
↪Patrice
Tarrabia posted on musicdsp.org's archives. It calculates the coefficients for a␣
↪second
order IIR filter. The resonance is specified in decibels above the DC gain. It can be␣
↪made
suitable to use as a SoundFont 2.0 filter by scaling the output so the overall gain
matches the specification (i.e. if resonance is 6dB then you should scale the output␣
↪by
-3dB). Note that you can replace the sqrt(2) values in the standard butterworth␣
↪highpass
algorithm with my "q =" line of code to get a highpass also. How it works: normally q␣
↪is
the constant sqrt(2), and this value controls resonance. At sqrt(2) resonance is 0dB,
smaller values increase resonance. By multiplying sqrt(2) by a power ratio we can␣
↪specify
the resonant gain at the cutoff frequency. The resonance power ratio is calculated␣
↪with a
standard formula to convert between decibels and power ratios (the powf statement...).

Good Luck,
Daniel Werner
http://experimentalscene.com/
```

Listing 26: code

```
float c, csq, resonance, q, a0, a1, a2, b1, b2;

c = 1.0f / (tanf(pi * (cutoff / samplerate)));
csq = c * c;
resonance = powf(10.0f, -(resonancedB * 0.1f));
q = sqrt(2.0f) * resonance;
a0 = 1.0f / (1.0f + (q * c) + (csq));
a1 = 2.0f * a0;
a2 = a0;
b1 = (2.0f * a0) * (1.0f - csq);
b2 = a0 * (1.0f - (q * c) + csq);
```

3.17.1 Comments

- **Date**: 2005-11-24 17:59:36
- **By**: acid_mutant[aat]yahoo[doot]com

```
For some reason when I tested this algorithm, even though the frequency response␣
↪looked OK in my graphs - i.e. it should resonate the output didn't seem to be very␣
↪resonant - it could be a phase issue, I'll keep checking.

(BTW: I use an impulse, then FFT, then display the power bands returned)
```

- **Date**: 2006-03-09 18:38:39
- **By**: moc.snad@snad

```
shouldn't it be

resonance = powf(10.0f, -(resonancedB * 0.05f));

instead of

resonance = powf(10.0f, -(resonancedB * 0.1f));

to get correct dB gain?

... since gain = 10^(dB/20) ...
```

- **Date**: 2007-01-06 04:29:10
- **By**: uh.etle.fni@yfoocs

```
Agree with the last post.
```

- **Date**: 2007-08-21 14:00:21
- **By**: moc.enecslatnemirepxe.ecnuob@ton.em.maps.renrewd

```
The algorithm was developed with a digital signal of 32-bit floating point pseudo-
↪random white noise running through it. The level of resonance was measured by␣
↪visually plotting the output of the FFT of the signal. I half agree with the second␣
↪last post, i.e. dB in acoustics is not the same as dB in digital audio. Correct me␣
↪if I am wrong, it is a long time since I thought about this.
```

- **Date**: 2010-12-10 23:53:27
- **By**: moc.oohay@tsvsoxox

```
there is something very wrong with this code as it is printed here, i don't expect␣
↪anyone is going to make any effort to correct it or verify it.
```

- **Date**: 2015-09-14 18:11:21
- **By**: moc.halb@halb

```
You need to use 0.5 in the power function to convert from the db to the linear.

You can apply a GAIN reduction by doing the same thing but make it smaller.

GAIN = powf(10.0f, -((resonance*0.25) * 0.05 ));

The use the GAIN to the input of the filter. If you use it to the output and change␣
↪resonance rapidly it will click.

For the gain you can change the 0.25 to 0.125 or 0.075 if you feel that it is to␣
↪quiet when you turn the resonance up.
```

- **Date**: 2015-09-14 18:14:36
- **By**: moc.halb@halb

```
Correction to me previous post, I meant 0.05 in the first line.

Convert resonance as dB to q for use in filter, use this formula.

  q = powf(10.0f, -(resonanceDb * 0.05) );

It will give you a value from 0 to 1. But for most butterworth you do not want zero␣
↪and you want it to go a little more than one to fully exploit the resonance, so␣
↪scale it like below.

For me this works to scale the q
  q = 0.1 + q * 1.5

For example, the 1.5 and 0.1 can be adjusted to suit your preference.
```

3.18 DC filter

- **Author or source:** hc.niweulb@lossor.ydna
- **Type:** 1-pole/1-zero DC filter
- **Created:** 2003-01-04 01:18:23

Listing 27: notes

```
This is based on code found in the document:
"Introduction to Digital Filters (DRAFT)"
Julius O. Smith III (jos@ccrma.stanford.edu)
(http://www-ccrma.stanford.edu/~jos/filters/)
---
```

(continues on next page)

(continued from previous page)

```
Some audio algorithms (asymmetric waveshaping, cascaded filters, ...) can produce DC
offset. This offset can accumulate and reduce the signal/noise ratio.

So, how to fix it? The example code from Julius O. Smith's document is:
...
y(n) = x(n) - x(n-1) + R * y(n-1)
// "R" between 0.9 .. 1
// n=current (n-1)=previous in/out value
...
"R" depends on sampling rate and the low frequency point. Do not set "R" to a fixed␣
↪value
(e.g. 0.99) if you don't know the sample rate. Instead set R to:
(-3dB @ 40Hz): R = 1-(250/samplerate)
(-3dB @ 30Hz): R = 1-(190/samplerate)
(-3dB @ 20Hz): R = 1-(126/samplerate)
```

3.18.1 Comments

- **Date**: 2003-01-04 01:58:04
- **By**: hc.niweulb@lossor.ydna

```
I just received a mail from a musicdsp reader:

'How to calculate "R" for a given (-3dB) low frequency point?'

R = 1 - (pi*2 * frequency /samplerate)

(pi=3.14159265358979)
```

- **Date**: 2003-05-10 09:11:42
- **By**: ten.labolgfrus@jbr

```
particularly if fixed-point arithmetic is used, this simple high-pass filter can␣
↪create it's own DC offset because of limit-cycles.  to cure that look at

http://www.dspguru.com/comp.dsp/tricks/alg/dc_block.htm

this trick uses the concept of "noise-shaping" to prevent DC in any limit-cycles.

r b-j
```

3.19 Delphi Class implementation of the RBJ filters

- **Author or source:** moc.liamtoh@retsbomyrgnayrev
- **Type:** Delphi class implementation of the RBJ filters
- **Created:** 2006-07-11 08:16:46

Listing 28: notes

```
I haven't tested this code thoroughly as it's pretty much a straight conversion from
Arguru c++ implementation.
```

Listing 29: code

```
  {
  RBJ Audio EQ Cookbook Filters
  A pascal conversion of arguru[AT]smartelectronix[DOT]com's
  c++ implementation.

  WARNING:This code is not FPU undernormalization safe.

  Filter Types
  0-LowPass
  1-HiPass
  2-BandPass CSG
  3-BandPass CZPG
  4-Notch
  5-AllPass
  6-Peaking
  7-LowShelf
  8-HiShelf
  }
unit uRbjEqFilters;

interface

uses math;

type
  TRbjEqFilter=class
  private
    b0a0,b1a0,b2a0,a1a0,a2a0:single;
    in1,in2,ou1,ou2:single;
    fSampleRate:single;
    fMaxBlockSize:integer;
    fFilterType:integer;
    fFreq,fQ,fDBGain:single;
    fQIsBandWidth:boolean;
    procedure SetQ(NewQ:single);
  public
    out1:array of single;
    constructor create(SampleRate:single;MaxBlockSize:integer);
    procedure CalcFilterCoeffs(pFilterType:integer;pFreq,pQ,pDBGain:single;
↪pQIsBandWidth:boolean);overload;
    procedure CalcFilterCoeffs;overload;
    function Process(input:single):single; overload;
    procedure Process(Input:psingle;sampleframes:integer); overload;
    property FilterType:integer read fFilterType write fFilterType;
    property Freq:single read fFreq write fFreq;
    property q:single read fQ write SetQ;
    property DBGain:single read fDBGain write fDBGain;
    property QIsBandWidth:boolean read fQIsBandWidth write fQIsBandWidth;
  end;

```

(continues on next page)

(continued from previous page)

```
implementation

constructor TRbjEqFilter.create(SampleRate:single;MaxBlockSize:integer);
begin
  fMaxBlockSize:=MaxBlockSize;
  setLength(out1,fMaxBlockSize);
  fSampleRate:=SampleRate;

  fFilterType:=0;
  fFreq:=500;
  fQ:=0.3;
  fDBGain:=0;
  fQIsBandWidth:=true;

  in1:=0;
  in2:=0;
  ou1:=0;
  ou2:=0;
end;

procedure TRbjEqFilter.SetQ(NewQ:single);
begin
  fQ:=(1-NewQ)*0.98;
end;

procedure TRbjEqFilter.CalcFilterCoeffs(pFilterType:integer;pFreq,pQ,pDBGain:single;pQIsBandWidth:boolean);
begin
  FilterType:=pFilterType;
  Freq:=pFreq;
  Q:=pQ;
  DBGain:=pDBGain;
  QIsBandWidth:=pQIsBandWidth;

  CalcFilterCoeffs;
end;

procedure TRbjEqFilter.CalcFilterCoeffs;
var
  alpha,a0,a1,a2,b0,b1,b2:single;
  A,beta,omega,tsin,tcos:single;
begin
  //peaking, LowShelf or HiShelf
  if fFilterType>=6 then
  begin
    A:=power(10.0,(DBGain/40.0));
    omega:=2*pi*fFreq/fSampleRate;
    tsin:=sin(omega);
    tcos:=cos(omega);

    if fQIsBandWidth then
      alpha:=tsin*sinh(log2(2.0)/2.0*fQ*omega/tsin)
    else
      alpha:=tsin/(2.0*fQ);

    beta:=sqrt(A)/fQ;

```

(continues on next page)

(continued from previous page)

```
    // peaking
    if fFilterType=6 then
    begin
      b0:=1.0+alpha*A;
      b1:=-2.0*tcos;
      b2:=1.0-alpha*A;
      a0:=1.0+alpha/A;
      a1:=-2.0*tcos;
      a2:=1.0-alpha/A;
    end else
    // lowshelf
    if fFilterType=7 then
    begin
      b0:=(A*((A+1.0)-(A-1.0)*tcos+beta*tsin));
      b1:=(2.0*A*((A-1.0)-(A+1.0)*tcos));
      b2:=(A*((A+1.0)-(A-1.0)*tcos-beta*tsin));
      a0:=((A+1.0)+(A-1.0)*tcos+beta*tsin);
      a1:=(-2.0*((A-1.0)+(A+1.0)*tcos));
      a2:=((A+1.0)+(A-1.0)*tcos-beta*tsin);
    end;
    // hishelf
    if fFilterType=8 then
    begin
      b0:=(A*((A+1.0)+(A-1.0)*tcos+beta*tsin));
      b1:=(-2.0*A*((A-1.0)+(A+1.0)*tcos));
      b2:=(A*((A+1.0)+(A-1.0)*tcos-beta*tsin));
      a0:=((A+1.0)-(A-1.0)*tcos+beta*tsin);
      a1:=(2.0*((A-1.0)-(A+1.0)*tcos));
      a2:=((A+1.0)-(A-1.0)*tcos-beta*tsin);
    end;
  end else  //other filter types
  begin
    omega:=2*pi*fFreq/fSampleRate;
    tsin:=sin(omega);
    tcos:=cos(omega);
    if fQIsBandWidth then
      alpha:=tsin*sinh(log2(2)/2*fQ*omega/tsin)
    else
      alpha:=tsin/(2*fQ);
    //lowpass
    if fFilterType=0 then
    begin
      b0:=(1-tcos)/2;
      b1:=1-tcos;
      b2:=(1-tcos)/2;
      a0:=1+alpha;
      a1:=-2*tcos;
      a2:=1-alpha;
    end else //hipass
    if fFilterType=1 then
    begin
      b0:=(1+tcos)/2;
      b1:=-(1+tcos);
      b2:=(1+tcos)/2;
      a0:=1+alpha;
      a1:=-2*tcos;
      a2:=1-alpha;
```

(continues on next page)

(continued from previous page)

```
    end else //bandpass CSG
    if fFilterType=2 then
    begin
      b0:=tsin/2;
      b1:=0;
      b2:=-tsin/2;
      a0:=1+alpha;
      a1:=-1*tcos;
      a2:=1-alpha;
    end else //bandpass CZPG
    if fFilterType=3 then
    begin
      b0:=alpha;
      b1:=0.0;
      b2:=-alpha;
      a0:=1.0+alpha;
      a1:=-2.0*tcos;
      a2:=1.0-alpha;
    end else  //notch
    if fFilterType=4 then
    begin
      b0:=1.0;
      b1:=-2.0*tcos;
      b2:=1.0;
      a0:=1.0+alpha;
      a1:=-2.0*tcos;
      a2:=1.0-alpha;
    end else   //allpass
    if fFilterType=5 then
    begin
      b0:=1.0-alpha;
      b1:=-2.0*tcos;
      b2:=1.0+alpha;
      a0:=1.0+alpha;
      a1:=-2.0*tcos;
      a2:=1.0-alpha;
    end;
  end;

  b0a0:=b0/a0;
  b1a0:=b1/a0;
  b2a0:=b2/a0;
  a1a0:=a1/a0;
  a2a0:=a2/a0;
end;


function TRbjEqFilter.Process(input:single):single;
var
  LastOut:single;
begin
  // filter
  LastOut:= b0a0*input + b1a0*in1 + b2a0*in2 - a1a0*ou1 - a2a0*ou2;

  // push in/out buffers
  in2:=in1;
  in1:=input;
```

(continues on next page)

(continued from previous page)

```
  ou2:=ou1;
  ou1:=LastOut;

  // return output
  result:=LastOut;
end;

{
the process method is overloaded.
use Process(input:single):single;
 for per sample processing
use Process(Input:psingle;sampleframes:integer);
 for block processing. The input is a pointer to
 the start of an array of single which contains
 the audio data.
 i.e.
 RBJFilter.Process(@WaveData[0],256);
}

procedure TRbjEqFilter.Process(Input:psingle;sampleframes:integer);
var
  i:integer;
  LastOut:single;
begin
  for i:=0 to SampleFrames-1 do
  begin
    // filter
    LastOut:= b0a0*(input^)+ b1a0*in1 + b2a0*in2 - a1a0*ou1 - a2a0*ou2;
    //LastOut:=input^;
    // push in/out buffers
    in2:=in1;
    in1:=input^;
    ou2:=ou1;
    ou1:=LastOut;

    Out1[i]:=LastOut;

    inc(input);
  end;
end;

end.
```

3.20 Digital RIAA equalization filter coefficients

- **Author or source:** Frederick Umminger
- **Type:** RIAA
- **Created:** 2002-10-14 16:33:34

Listing 30: notes

```
Use at your own risk. Confirm correctness before using. Don't assume I didn't goof
something up.
```

(continues on next page)

(continued from previous page)

```
-Frederick Umminger
```

Listing 31: 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
```

3.20.1 Comments

- **Date**: 2007-02-24 13:55:58
- **By**: moc.liamtoh@0691_ptj

```
Hmm... since I'm having lack in knowledge of utilizing this type of 'data' in␣
↪programming, could someone be kind and give a short code example of its usage (@␣
↪some samplerate), lets say, using Basic/VB language (though, C-C++/Pascal-Delphi/
↪Java goes as well)?


JT
```

- **Date**: 2007-03-01 13:20:51
- **By**: ku.oc.snosrapsdTUOEMEKAT@psdcisum

```
they are coefficients to plug into a std biquad. look through the filters section of␣
↪musicdsp you'll find a load of examples of biquads (essentially two quadratic␣
↪equations which are solved together to do the DSP stuff).

It's of the form
out = b0*in[0] + b1*in[-1] + b2*in[-2] - a1*out[-1] - a2*out[-2]

where in[0,-1,-2] are the current input and the previous 2; and out[-1,-2] are the␣
↪last two outputs.

Generally the previous output coefficients are subtracted, but sometimes the signs␣
↪are swapped, and they are added like the inputs.

some algorithms use a for ins and b for outs, others use them the other way around.␣
↪Generally (but not always) there are 3 input and 2 output coeffs, so you can work␣
↪out which is which.

HTH
DSP
```

- **Date**: 2007-03-02 03:31:08
- **By**: moc.erehwon@ydobon

```
I don't get it. How do you set the frequency, Kenneth?

What frequencies are being passed?
```

- **Date**: 2007-03-03 09:09:29
- **By**: moc.liamtoh@0691_ptj

```
Hmm...

Since no links allowed here, I have started a topic on this matter @ KVR

topic number: 170235
topic name: "Coefficients of the digital transfer function ... How to ?"

I tried the 44.1/48kHz version and it produced quite 'bad' results .. lots of rattle␣
↪in audio and the RIAA curve form is not as it should be (should be: 20Hz; ±19.27dB .
↪.. ~1kHz; ±0dB ... 20kHz; ±19.62dB). (couple of pictures linked in KVR topic).

Also, .. if this is the result in anyway, this is the 'production curve' used in␣
↪mastering process ... how can it be changed to 'opposite' ...

JT
```

- **Date**: 2007-03-04 22:52:32
- **By**: moc.liamtoh@0691_ptj

```
Thanks to all so far.

I found this quote from another forum:

QUOTE:
"All you should need to do to get the complementary curve is swap the a and b
vectors, and then multiply both vectors by 1/a(0) to normalize. That will
give the coefficients for the inverse filter.".
/QUOTE

w/ a note that it was taken from one of those OPs (Frederick Umminger's) postings ...
↪but the reference link was dead so I couldn't read the whole story. If OP or anyone
↪else can give some light in this matter of how to make that swap w/ normalization
↪(fully) so I could try w/ higher SR data. I did try and got values like -20.
↪1287341287123, etc..

I actually got the 44.1/48kHz curve managed w/ help from a post in another forum. But
↪there were nothing explained fully.

QUOTE:
"
; Filter coefficients (48kHz) for RIAA curve from Frederick
; Umminger; see
;
;  b = [  0.02675918611906  -0.04592084787595   0.01921229297239]
;  a = [  1.00000000000000  -0.73845850035973  -0.17951755477430]
;  error +/- 0.15dB

; inverted filter for phono playback (48kHz):
;
;  b = [ 0.2275882473429072 -0.1680644758323426 -0.0408560856583673 ]
;  a = [ 1.0000000000000000 -1.7160778983199925  0.7179700042784745 ]
;
; since a[1] is too large, it must be splitted into a11 and a12

static b0=0.2275882473429072, b1=-0.1680644758323426, b2=-0.0408560856583673
static a1=.85803894915999625, a2=-0.7179700042784745
"
/QUOTE

just lots of numbers ....

JT
```

- **Date**: 2007-03-15 00:41:09
- **By**: moc.liamtoh@0691_ptj

```
This seem to become a monologue but, ... I'm still having issues w/ those 88.2kHz and
↪96kHz filter coefficients when inverted.

Noticed that when those coefficients for 88.2kHz and 96kHz are inverted, in both
↪cases, a1 and a2 gets values which maybe are not good in equation

y[i] = b0x[i] + b1x[i-1] + b2x[i-2] - a1y[i-1] - a2y[i-1]
```

(continues on next page)

(continued from previous page)

```
because of, a1 gets a negative value and its decimal part is bigger than a2 is --> " -
↪-a1y[i-1] - a2y[i-2]" --> looks like y[i] starts growing after every sample␣
↪calculation. This is not an issue w/ data for 44.1kHz and 48kHz. When I change␣
↪those a1/a2 decimal parts so that the abs(a1)-a2 =< 1 becomes true then filter␣
↪works well (though not right recults). Also, while analyzing the VST plugin, using␣
↪C.W.Buddes VST PluginAnalyzer, Delphi tracer (Watch) shows y[i] become over 1.0␣
↪after ~830 sampleframes and after 8192 sampleframes, y[i] has value of 2.
↪488847401e+11 already (i.e. 248884740100). This shouldn't be a coding problem since␣
↪a friend of mine tested these w/ SynthMaker (no coding needed) and the results were␣
↪equal.

If this "-a1x[i-1]-a2x[i-2] > 1" is an issue, are there any methods to get it fixed w/
↪o loosing the accuracy OP got into those original coefficients?

jtp
```

- **Date**: 2007-03-15 22:41:37
- **By**: ed.luosfosruoivas@naitsirhC

```
Try to plot the poles and zeroes. If there are poles outside the unit circle, your␣
↪filter will be unstable!
To eliminate poles outside the unit circle, construct an allpass filter which has␣
↪zeroes at the same position as the unwanted poles. They are now canceling out␣
↪themself, so that you only have poles inside the unit circle. Your filter should be␣
↪stable now!

All you need to know now is how to transform the filter coefficients into poles and␣
↪zeroes and vice versa. If you're using delphi, you might want to have a look into␣
↪the DFilter class of the open source project 'Delphi ASIO & VST Packages'.
```

- **Date**: 2007-03-24 11:53:25
- **By**: moc.liamtoh@0691_ptj

```
Thanks for your suggestion Christian.
I didn't try this allpass method because of

- I managed to get this issue rounded through another way (I have now 3rd-4th order␣
↪filters working here as VST and standalone for all those four samplerates mentioned␣
↪here and I'm also considering to add ones for 174.6 kHz and 192 kHz as well)

- as I'm learning these filter matters and delphi programming, I would have needed␣
↪some good examples to do this


My final thoughts over those coefficients listed in F. Ummingers post:

As those coefficients needs to be inversed before getting the RIAA reproduction done,␣
↪I can't say 100% sure if any of those works properly then (maybe one set does).
When inversion is done as was suggested elsewhere:
- swap a/b vectors,
- multiply all with 1/a0 and
- optional: 'normalize' b's by dividing every b with sum of b's
, only coefficients for 44.1kHz and 48kHz seem to become stable but, which one is the␣
↪right one then since, those original coefficients are same for both? I suppose␣
↪those can't be equal coefficients because this is sample accurate filter in␣
↪question, or can those?. If not then, which one is the correct one ... you can find␣
↪it out by trying (least the resulting sound quality should tell this). Maybe Hannes␣
↪Rohde (quote in my 3rd post) went through this and found the right ones or just␣
↪used those given for 48kHz (SoundBlaster DSP is internally 48kHz).
```

(continues on next page)

(continued from previous page)

```
What's wrong with those others? It seems that both, 88.2kHz and 96kHz coefficients as
↪inversed, produces unstable filter which won't work (see my previous post)

jtp
```

- **Date**: 2007-04-22 12:03:52
- **By**: moc.liamtoh@0691_ptj

```
FYI, here are working filter coefficients for biquad implementation of RIAA EQ
↪Reproduction filters:

44.1kHz:
a = [ 1.0000000000 -1.7007240000  0.7029381524 ]
b = [ 1.0000000000 -0.7218922000 -0.1860520545 ]
error ~0.23dB

48kHz:
a = [ 1.0000000000 -1.7327655000  0.7345534436 ]
b = [ 1.0000000000 -0.7555521000 -0.1646257113 ]
error ~0.14dB

88.2kHz:
a = [ 1.0000000000 -1.8554648000  0.8559721393 ]
b = [ 1.0000000000 -0.8479577000 -0.1127631993 ]
error 0.008dB

and 96kHz:
a = [ 1.0000000000 -1.8666083000  0.8670382873 ]
b = [ 1.0000000000 -0.8535331000 -0.1104595113 ]
error ~0.006dB


NOTES:

# - By swapping the a1<->b1 and a2<->b2 you'll get the production filter.

# - All these given filter coefficients produces a bit gained filter (~+12.5dB or so)
↪so, if you like to adjust the 1 kHz = 0dB, it can be done quite accurately by
↪finding linear difference using software like Tobybear's FilterExplorer. Enter
↪coefficients into FilterExplorer, by moving mouse cursor over the plotted magnitude
↪curve in magnitude plot window, find/point the ~1kHz position and then check the
↪magnitude value (value inside the brackets) found in info field. Use this value as
↪divider for b coefficients.


jtp
jiiteepee@yahoo.se
```

- **Date**: 2009-08-07 18:43:49
- **By**: ude.nretsewhtron.ece@ztub

```
The amplitude response of Umminger's Fs = 48 kHz filter gives an excellent
↪approximation to the RIAA amplitude response curve but Umminger suggests the phase
↪response may be "totally whacked". Actually, the phase response is pretty good,
↪provided "phase response" is interpreted properly.
```

(continues on next page)

(continued from previous page)

```
Umminger's filters are carelessly presented. The 44.1 kHz version is not there at all,
↪ and the 88.2 kHz and 96 kHz cases have reproduction filter poles > 1 that should␣
↪be replaced by their reciprocals. For that reason I shall use the coefficients␣
↪given by jtp, though the general approach is Umminger's.
When computing phase of the digital filter, a linear phase term may be added to the␣
↪computation as convenient, as such a term corresponds to time shift. That is, two␣
↪phase responses are for our practical purposes equivalent if they differ only by a␣
↪phase linear in frequency f. The RIAA analog filter has an asymptotic phase of -90␣
↪deg and Umminger's asymptotic (Fs/2) phase is 0, so one may conjecture that a term␣
↪2Df/Fs ought to be added to the computation of the digital filter phase, where D = -
↪90. Actually, there is no reason to restrict D to multiples of 90.
In jtp's Fs=44.1 kHz case, an adjustment to the computed phase using D = -65.5␣
↪results in maximum computed phase error < 1.5 deg over the range 30 Hz - 10 kHz,␣
↪while the computation using D = -75.75 results in maximum computed phase error < 5.
↪2 deg over the range 30 Hz - 20 kHz. Of course the digital filter itself is␣
↪independent of D, which is used only to interpret the phase response. One is, in␣
↪effect, comparing the output of the digital filter with the output of the RIAA␣
↪analog filter delayed by D/180 sample intervals. The digital filter itself remains␣
↪as given by jtp.
In jtp's Fs=48 kHz case use D = -68 for a phase error < 1.2 deg  over the range 30 Hz␣
↪- 10 kHz, and D = -75 for a phase error < 3.8 deg over the range 30 Hz - 20 kHz.
In jtp's Fs=88.2 kHz case use D = -72 for a phase error < 0.31 deg  over the range 30␣
↪Hz - 10 kHz, and D = -72.8 for a phase error < 0.5 deg over the range 30 Hz - 20␣
↪kHz.
In jtp's Fs=96 kHz case use D = -72.4 for a phase error < 0.30 deg  over the range 30␣
↪Hz - 10 kHz, and D = -72.8 for a phase error < 0.375 deg over the range 30 Hz - 20␣
↪kHz.
In the Fs = 44.1 or 48 kHz cases, if a max phase error, over 30 Hz - 20 kHz, of about␣
↪0.5 deg is wanted, then one can double the sample rate in the usual way using a␣
↪linear phase FIR interpolating filter, then do equalization at sample rate 88.2 or␣
↪96 kHz, decimating the output by a factor 2. August 7, 2009
```

3.21 Direct Form II biquad

- **Author or source:** es.tuanopx@kileib.trebor
- **Created:** 2009-11-16 08:46:12

Listing 32: notes

```
The nominal implementation for biquads is the Direct Form I variant. But the Direct␣
↪Form
II is actually more suited for calculating the biquad since it needs only 2 memory
locations, and the multiplications can be pipelined better by the compiler. In release
build, I've noted a considerable speedup when compared to DF I. When processing␣
↪stereo,
the code below was ~ 2X faster. Until I develop a SIMD biquad that is faster, this␣
↪will
do.
```

Listing 33: code

```
// b0, b1, b2, a1, a2 calculated via r.b-j's cookbook
// formulae.
```

(continues on next page)

(continued from previous page)

```
// m1, m2 are the memory locations
// dn is the de-denormal coeff (=1.0e-20f)

void processBiquad(const float* in, float* out, unsigned length)
{
    for(unsigned i = 0; i < length; ++i)
    {
        register float w = in[i] - a1*m1 - a2*m2 + dn;
        out[i] = b1*m1 + b2*m2 + b0*w;
        m2 = m1; m1 = w;
    }
    dn = -dn;
}

void processBiquadStereo(const float* inL,
   const float* inR,
   float* outL,
   float* outR,
   unsigned length)
{
    for(unsigned i = 0; i < length; ++i)
    {
        register float wL = inL[i] - a1*m1L - a2*m2L + dn;
        register float wR = inR[i] - a1*m1R - a2*m2R + dn;
        outL[i] = b1*m1L + b2*m2L + b0*wL;
        m2L = m1L; m1L = wL;
        outR[i] = b1*m1R + b2*m2R + b0*wR;
        m2R = m1R; m1R = wR;
    }
    dn = -dn;
}
```

3.21.1 Comments

- **Date**: 2010-01-13 13:44:09
- **By**: moc.suomyn@ona

```
true, this structure is faster. but it is also (even) more sensitive to coefficients
↪changes, so it becomes unstable quite fast compaerd to the DF I form. I'd really
↪like to know if there's a way to change coefficients and at the same time time
↪changing the history of the filter for avoiding unstability.
```

- **Date**: 2012-01-31 20:43:12
- **By**: earlevel

```
Use direct form I (single accumulation point) when using fixed-point processors. For
↪floating point, use direct form II transposed, which has superior numerical
↪characteristics to direct form II (non-transposed).
```

3.22 Direct form II

- **Author or source:** Fuzzpilz

- **Type:** generic
- **Created:** 2004-06-28 10:42:44

Listing 34: notes

```
I've noticed there's no code for direct form II filters in general here, though␣
↪probably
many of the filter examples use it. I haven't looked at them all to verify that, but␣
↪there
certainly doesn't seem to be a snippet describing this.

This is a simple direct form II implementation of a k-pole, k-zero filter. It's a␣
↪little
faster than (a naive, real-time implementation of) direct form I, as well as more
numerically accurate.
```

Listing 35: code

```
Direct form I pseudocode:

y[n] = a[0]*x[n] + a[1]*x[n-1] + .. + a[k]*x[n-k]
                 - b[1]*y[n-1] - .. - b[k]*y[n-k];

Simple equivalent direct form II pseudocode:

y[n] = a[0]*x[n] + d[0];
d[0] = a[1]*x[n] - b[1]*y[n] + d[1];
d[1] = a[2]*x[n] - b[2]*y[n] + d[2];
.
.
d[k-2] = a[k-1]*x[n] - b[k-1]*y[n] + d[k-1];
d[k-1] = a[k]*x[n] - b[k]*y[n];

For example, a biquad:

out = a0*in + a1*h0 + a2*h1 - b1*h2 - b2*h3;
h1 = h0;
h0 = in;
h3 = h2;
h2 = out;

becomes

out = a0*in + d0;
d0 = a1*in - b1*out + d1;
d1 = a2*in - b2*out;
```

3.22.1 Comments

- **Date**: 2007-02-21 17:31:20
- **By**: uh.etle.fni@yfoocs

```
I think the per sample denormal number elimination on x87 FPUs is more difficult,␣
↪since you need to check for denormals at 3 places instead of one (if I'm right).
```

- **Date**: 2007-06-11 16:44:05
- **By**: moc.liamg@sgninnejtg

```
Are the constants (a and b) wrong here.  Don't they need to be switched?  If you look
↪at like wikipedia that's the case and it makes more since.  I'm trying to implement
↪a low pass filter at 25mhz passband edge.  I'm getting alot of fluctuation in my
↪output more that expect.  Any suggestions?

int main(int argc, char *argv[])
{
double b[3] = {1,2,1};
double a1[3] = {1,-1.9995181705254206,0.99952100328066507};
//double a1[3] = {1,-1.9252217796690612,0.95315661147483732};
double a2[3] = {1,-1.9985996261556458,0.99860245760957123};
double a3[3] = {1,-1.9977949691405856,0.99779779945453828};
double a4[3] = {1,-1.9971690447494761,0.99717187417666975};
double a5[3] = {1,-1.9967721889631873,0.9967750178281477};
double a6[2] = {1, -0.99831813425055116};
double d[3] = {0};
double y[5][3] = {0};
double out[2] = {0};
double x[3]={0}, x1,x2,in;
double i=0;
char wait;
while(i<10000000)
{
x1 = sin(2*10000*3.14159265*i);
x2 = sin(2*10000*3.14159265*i-3.14159265);
in = x1 * x2;


x[0] = in * 7.0818881108085789e-7;

y[0][0] = b[0]*x[0] + b[1]*x[1] + b[2]*x[2] - a1[1]*y[0][1] - a1[2]*y[0][1];
y[0][1] = y[0][0];
x[2] = x[1];
x[1] = x[0];
/////////////////////////////////////////////////////////////////////////////////
y[0][0] = y[0][0] * 7.0786348128153693e-7;
y[1][0] = b[0]*y[0][0] + b[1]*y[0][1] + b[2]*y[0][2] - a2[1]*y[1][1] - a2[2]*y[1][1];
y[0][2] = y[0][1];
y[0][1] = y[0][0];
y[1][1] = y[1][0];
/////////////////////////////////////////////////////////////////////////////////
y[1][0] = y[1][0] * 7.0757848807506174e-7;
y[2][0] = b[0]*y[1][0] + b[1]*y[1][1] + b[2]*y[1][2] - a3[1]*y[2][1] - a3[2]*y[2][1];
y[1][2] = y[1][1];
y[1][1] = y[1][0];
y[2][1] = y[2][0];
/////////////////////////////////////////////////////////////////////////////////
y[2][0] = y[2][0] * 7.0735679834155469e-7;
y[3][0] = b[0]*y[2][0] + b[1]*y[2][1] + b[2]*y[2][2] - a4[1]*y[3][1] - a4[2]*y[3][1];
y[2][2] = y[2][1];
y[2][1] = y[2][0];
y[3][1] = y[3][0];
/////////////////////////////////////////////////////////////////////////////////
y[3][0] = y[3][0] * 7.0721624006526327e-007;
```

(continues on next page)

(continued from previous page)

```
y[4][0] = b[0]*y[3][0] + b[1]*y[3][1] + b[2]*y[3][2] - a5[1]*y[4][1] - a5[2]*y[4][1];
y[3][2] = y[3][1];
y[3][1] = y[3][0];
y[4][1] = y[4][0];
///////////////////////////////////////////////////////////////////////////////////
/*y[4][0] = y[4][0]* 0.000840932874724457;
out[0] = 1*y[4][0] + 1*y[4][1] - a6[1]*out[1];
y[4][1] = y[4][0];
out[1] = out[0];*/

cout<<y[4][0]<<"\n";

i+=.1;
}
```

- **Date**: 2009-03-23 07:17:12
- **By**: es.tuanopx@kileib.trebor

```
Regarding denormals: Don't check for them. Prevent them by adding a small value (~1e-
↪20) to the filter memory pipeline.
```

- **Date**: 2009-03-23 18:15:40
- **By**: es.tuanopx@kileib.trebor

```
Processing a single biquad doesn't benefit much (if any) from doing a DF II␣
↪implementation. However, if you'd process stereo, the DF II variant is very␣
↪suitable for interleaving of non-dependent calculations, making it easier for the␣
↪compiler to generate effective code. Actually, the DF II stereo implementation␣
↪below is more than 2 times faster than the naive DF I stereo one:

struct stereo_biquad
{
    float b0,b1,b2,a1,a2 // From rb-j's cookbook with a0 normalized to 1.0
    float lm1,lm2,rm1,rm2; // Filter state
    float dn;              // de-denormal coeff (1.0e-20f)
};

void processStereoBiquadDF2(
    struct stereo_biquad& bq,
    const float* inL,
    const float* inR,
    float* outL,
    float* outR,
    unsigned length)
{
    for (unsigned i = 0; i < length; ++i)
    {
        register float w1 = inL[i] - bq.a1*bq.lm1 - bq.a2*bq.lm2 + bq.dn;
        register float w2 = inR[i] - bq.a1*bq.rm1 - bq.a2*bq.rm2 + bq.dn;
        outL[i] = bq.b1*bq.lm1 + bq.b2*bq.lm2 + bq.b0*w1;
        bq.lm2 = bq.lm1; bq.lm1 = w1;
        outR[i] = bq.b1*bq.rm1 + bq.b2*bq.rm2 + bq.b0*w2;
        bq.rm2 = bq.rm1; bq.rm1 = w2;
    }
    bq.dn = -bq.dn;
}
```

3.23 Fast Downsampling With Antialiasing

- **Author or source:** moc.liamg@tramum
- **Created:** 2005-12-22 20:34:58

Listing 36: notes

```
A quick and simple method of downsampling a signal by a factor of two with a useful␣
↪amount
of antialiasing. Each source sample is convolved with { 0.25, 0.5, 0.25 } before
downsampling.
```

Listing 37: code

```
int filter_state;

/* input_buf can be equal to output_buf */
void downsample( int *input_buf, int *output_buf, int output_count ) {
    int input_idx, input_end, output_idx, output_sam;
    input_idx = output_idx = 0;
    input_end = output_count * 2;
    while( input_idx < input_end ) {
            output_sam = filter_state + ( input_buf[ input_idx++ ] >> 1 );
            filter_state = input_buf[ input_idx++ ] >> 2;
            output_buf[ output_idx++ ] = output_sam + filter_state;
    }
}
```

3.23.1 Comments

- **Date**: 2006-01-06 11:22:36
- **By**: ku.oc.mapson.snosrapsd@psd

```
I see this is designed for integers; what are you thoughts on altering it to floats␣
↪and doing simple division rather than bit shifts?
```

- **Date**: 2006-01-07 01:35:56
- **By**: moc.liamg@tramum

```
It will work fine in floating point. I would probably use multiplication rather than␣
↪division though, as I would expect that to be faster (ie. >> 1 --> *0.5, >>2 --> *0.
↪25).
```

- **Date**: 2006-03-12 01:55:30
- **By**: ude.drofnats.amrcc@lfd

```
this triangular window is still not the greatest antialiaser... but it's probably␣
↪fine for something like an oversampled lowpass filter!
```

- **Date**: 2006-03-17 23:36:31
- **By**: moc.liamg@tramum

```
For my purposes(modelling a first-order-hold dac) it was fine. The counterpart to it␣
↪I suppose is this one - a classic exponential decay, which gives a lovely warm␣
↪sound. Each sample is convolved with { 0.5, 0.25, 0.125, ...etc }

int filter_state;

void downsample( int *input_buf, int *output_buf, int output_count ) {
    int input_idx, output_idx, input_ep1;
    output_idx = 0;
    input_idx = 0;
    input_ep1 = output_count * 2;
    while( input_idx < input_ep1 ) {
            filter_state = ( filter_state + input_buf[ input_idx ] ) >> 1;
            output_buf[ output_idx ] = filter_state;
            filter_state = ( filter_state + input_buf[ input_idx + 1 ] ) >> 1;
            input_idx += 2;
            output_idx += 1;
    }
}

I'm not a great fan of all these high-order filters, the mathematics are more than I␣
↪can cope with :)

Cheers,
Martin
```

- **Date**: 2008-11-05 14:12:10
- **By**: ed.bew@ehcsa-k

```
Hi @ all,

what is a good initialization value of filter_state?

Greetings

Karsten
```

- **Date**: 2009-05-02 23:38:31
- **By**: moc.liamg@tramum

```
filter_state is the previous input sample * 0.25, so zero is a good starting value␣
↪for a non-periodic waveform.
```

- **Date**: 2009-07-07 12:02:32
- **By**: moc.oohay@bob

```
I'm curious - as you're generating 1 sample for every 2, is it possible to then␣
↪upsample with zero padding to get a half band filter at the original sample rate?

Cheers
B
```

3.24 Formant filter

- **Author or source:** Alex
- **Created:** 2002-08-02 18:26:59

Listing 38: 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];
```

(continues on next page)

(continued from previous page)

```
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;
}
```

3.24.1 Comments

- **Date**: 2002-08-21 04:47:36
- **By**: moc.oohay@nosrednattehr

```
Where did the coefficients come from? Do they relate to frequencies somehow? Are they
↪male or female? Etc.
```

- **Date**: 2002-09-20 02:45:20
- **By**: es.umu.gni@nhs89le

```
And are the coeffiecients for 44k1hz?
/stefancrs
```

- **Date**: 2002-11-17 09:16:51
- **By**: ten.ooleem@ooleem

```
It seem to be ok at 44KHz although I get quite lot of distortion with this filter.
There are typos in the given code too, the correct version looks like this i think:
float formant_filter(float *in, int vowelnum)
{
  float res= (float) ( coeff[vowelnum][0]* (*in) +
  coeff[vowelnum][1] *memory[0] +
  coeff[vowelnum][2] *memory[1] +
  coeff[vowelnum][3] *memory[2] +
  coeff[vowelnum][4] *memory[3] +
  coeff[vowelnum][5] *memory[4] +
  coeff[vowelnum][6] *memory[5] +
  coeff[vowelnum][7] *memory[6] +
  coeff[vowelnum][8] *memory[7] +
  coeff[vowelnum][9] *memory[8] +
  coeff[vowelnum][10] *memory[9] );
...

(missing type and asterisk in the first calc line ;).

I tried morphing from one vowel to another and it works ok except in between 'A' and
↪'U' as I get a lot of distortion and sometime (depending on the signal) the filter
↪goes into auto-oscilation.

Sebastien Metrot
```

- **Date**: 2002-12-17 20:22:08

- **By**: gro.kale@ybsral

```
How did you get the coeffiecients?

Did I miss something?

/Larsby
```

- **Date**: 2003-01-22 15:22:02
- **By**: es.ecid@nellah.nafets

```
Yeah, morphing lineary between the coefficients works just fine. The distortion I␣
↪only get when not lowering the amplitude of the input. So I lower it :)

Larsby, you can approximate filter curves quite easily, check your dsp literature :)
```

- **Date**: 2003-07-07 08:45:53
- **By**: moc.xinortceletrams@xela

```
Correct, it is for sampling rate of 44kHz.
It supposed to be female (soprano), approximated with its five formants.

--Alex.
```

- **Date**: 2003-08-21 03:21:28
- **By**: moc.liamtoh@33reniur

```
Can you tell us how you calculated the coefficients?
```

- **Date**: 2003-10-04 18:42:31
- **By**: moc.liamtoh@sisehtnysorpitna

```
The distorting/sharp A vowel can be toned down easy by just changing the first coeff␣
↪from 8.11044e-06 to 3.11044e-06. Sounds much better that way.
```

- **Date**: 2005-05-04 22:40:18
- **By**: moc.liamg@grebronj

```
Hi, I get the last formant (U) to self-oscillate and distort out of control whatever␣
↪I feed it with. all the other ones sound fine...

any sugesstions?

Thanks,
Jonas
```

- **Date**: 2006-04-12 22:07:35
- **By**: if.iki@xemxet

```
I was playing around this filter, and after hours of debugging finally noticed that␣
↪converting those coeffecients to float just won't do it. The resulting filter is␣
↪not stable anymore. Doh...

I don't have any idea how to convert them, though.
```

- **Date**: 2008-10-29 00:35:27
- **By**: mysterious T

```
Fantastic, it's all I can say! Done the linear blending and open blending matrix (a-e,
↪ a-i, a-o, a-u, e-i, e-o...etc..etc..). Too much fun!

Thanks a lot, Alex!
```

- **Date**: 2010-12-14 13:16:19
- **By**: johnny

```
What about input and output range? When I feed the filter with audio data in -1 to 1␣
↪range, output doesn't stay in the same range. Maybe the input or output needs to be␣
↪scaled?
```

3.25 Frequency warped FIR lattice

- **Author or source:** ten.enegatum@liam
- **Type:** FIR using allpass chain
- **Created:** 2004-08-24 11:39:28

Listing 39: notes

```
Not at all optimized and pretty hungry in terms of arrays and overhead (function␣
↪requires
two arrays containing lattice filter's internal state and ouputs to another two arrays
with their next states).  In this implementation I think you'll have to juggle
taps1/newtaps in your processing loop, alternating between one set of arrays and the␣
↪other
for which to send to wfirlattice).

A frequency-warped lattice filter is just a lattice filter where every delay has been
replaced with an allpass filter.  By adjusting the allpass filters, the frequency␣
↪response
of the filter can be adjusted (e.g., design an FIR that approximates some filter. ␣
↪Play
with with warping coefficient to "sweep" the FIR up and down without changing any␣
↪other
coefficients).  Much more on warped filters can be found on Aki Harma's website (
http://www.acoustics.hut.fi/~aqi/ )
```

Listing 40: code

```
float wfirlattice(float input, float *taps1, float *taps2, float *reflcof, float␣
↪lambda, float *newtaps1, float *newtaps2, int P)
// input is filter input
// taps1,taps2 are previous filter states (init to 0)
// reflcof are reflection coefficients.  abs(reflcof) < 1 for stable filter
// lamba is warping (0 = no warping, 0.75 is close to bark scale at 44.1 kHz)
// newtaps1, newtaps2 are new filter states
// P is the order of the filter
{
```

(continues on next page)

(continued from previous page)

```
    float forward;
    float topline;

    forward = input;
    topline = forward;

    for (int i=0;i<P;i++)
    {
            newtaps2[i] = topline;
            newtaps1[i] = float(lambda)*(-topline + taps1[i]) + taps2[i];
            topline = newtaps1[i]+forward*(reflcof[i]);
            forward += newtaps1[i]*(reflcof[i]);
            taps1[i]=newtaps1[i];
            taps2[i]=newtaps2[i];
    }
    return forward;
}
```

3.25.1 Comments

- **Date**: 2004-08-24 17:26:59
- **By**: ten.xmg@zlipzzuf

```
Couldn't you easily do away with newtaps entirely? As in:

for(int i=0;i&lt;P;i++)
{
    taps1[i]=lambda*(taps1[i]-topline)+taps2[i];
    taps2[i]=topline;
    topline=taps1[i]+forward*reflcof[i];
    forward+=taps1[i]*reflcof[i];
}

I haven't had time to try this in a plugin yet, but if Maple is to be trusted at all,
↪that works.

(2WarpDelay is nice, by the way)
```

- **Date**: 2004-08-25 07:32:40
- **By**: ten.enegatum@liam

```
haha, thanks, that's awesome!  how embarassing ;)

(glad you like 2warpdelay!  the warped IIR lattice is up on harma's site too, though
↪you might save yourself time if you read the errata: http://www.acoustics.hut.fi/~
↪aqi/papers/oops.html :( )
```

- **Date**: 2008-11-12 01:15:07
- **By**: moc.toohay@ttevad

```
This looks really interesting.
How do I get the coeffs for it, and how do I invert it to get back to the original
↪signal?
```

(continues on next page)

(continued from previous page)

```
Thanks,
DaveT
```

3.26 Hilbert Filter Coefficient Calculation

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** Uncle Hilbert
- **Created:** 2005-04-17 19:05:37

Listing 41: notes

```
This is the delphi code to create the filter coefficients, which are needed to␣
↪phaseshift
a signal by 90°
This may be useful for an evelope detector...

By windowing the filter coefficients you can trade phase response flatness with␣
↪magnitude
response flatness.

I had problems checking its response by using a dirac impulse. White noise works fine.

Also this introduces a latency of N/2!
```

Listing 42: code

```
type TSingleArray = Array of Single;

procedure UncleHilbert(var FilterCoefficients: TSingleArray; N : Integer);
var i,j : Integer;
begin
 SetLength(FilterCoefficients,N);
 for i:=0 to (N div 4) do
  begin
   FilterCoefficients[(N div 2)+(2*i-1)]:=+2/(PI*(2*i-1));
   FilterCoefficients[(N div 2)-(2*i-1)]:=-2/(PI*(2*i-1));
  end;
end;
```

3.27 Hiqh quality /2 decimators

- **Author or source:** Paul Sernine
- **Type:** Decimators
- **Created:** 2006-07-28 17:59:03

Listing 43: notes

```
These are /2 decimators,
Just instanciate one of them and use the Calc method to obtain one sample while
↪inputing
two. There is 5,7 and 9 tap versions.
They are extracted/adapted from a tutorial code by Thierry Rochebois. The optimal
coefficients are excerpts of Traitement numérique du signal, 5eme edition, M
↪Bellanger,
Masson pp. 339-346.
```

Listing 44: code

```
//Filtres décimateurs
// T.Rochebois
// Based on
//Traitement numérique du signal, 5eme edition, M Bellanger, Masson pp. 339-346
class Decimateur5
{
  private:
  float R1,R2,R3,R4,R5;
  const float h0;
  const float h1;
  const float h3;
  const float h5;
  public:
  Decimateur5::Decimateur5():h0(346/692.0f),h1(208/692.0f),h3(-44/692.0f),h5(9/692.0f)
  {
    R1=R2=R3=R4=R5=0.0f;
  }
  float Calc(const float x0,const float x1)
  {
    float h5x0=h5*x0;
    float h3x0=h3*x0;
    float h1x0=h1*x0;
    float R6=R5+h5x0;
    R5=R4+h3x0;
    R4=R3+h1x0;
    R3=R2+h1x0+h0*x1;
    R2=R1+h3x0;
    R1=h5x0;
    return R6;
  }
};
class Decimateur7
{
  private:
  float R1,R2,R3,R4,R5,R6,R7;
  const float h0,h1,h3,h5,h7;
  public:
  Decimateur7::Decimateur7():h0(802/1604.0f),h1(490/1604.0f),h3(-116/1604.0f),h5(33/
↪1604.0f),h7(-6/1604.0f)
  {
    R1=R2=R3=R4=R5=R6=R7=0.0f;
  }
  float Calc(const float x0,const float x1)
  {
    float h7x0=h7*x0;
```

(continues on next page)

(continued from previous page)

```
    float h5x0=h5*x0;
    float h3x0=h3*x0;
    float h1x0=h1*x0;
    float R8=R7+h7x0;
    R7=R6+h5x0;
    R6=R5+h3x0;
    R5=R4+h1x0;
    R4=R3+h1x0+h0*x1;
    R3=R2+h3x0;
    R2=R1+h5x0;
    R1=h7x0;
    return R8;
  }
};
class Decimateur9
{
  private:
  float R1,R2,R3,R4,R5,R6,R7,R8,R9;
  const float h0,h1,h3,h5,h7,h9;
  public:
  Decimateur9::Decimateur9():h0(8192/16384.0f),h1(5042/16384.0f),h3(-1277/16384.0f),
↪h5(429/16384.0f),h7(-116/16384.0f),h9(18/16384.0f)
  {
    R1=R2=R3=R4=R5=R6=R7=R8=R9=0.0f;
  }
  float Calc(const float x0,const float x1)
  {
    float h9x0=h9*x0;
    float h7x0=h7*x0;
    float h5x0=h5*x0;
    float h3x0=h3*x0;
    float h1x0=h1*x0;
    float R10=R9+h9x0;
    R9=R8+h7x0;
    R8=R7+h5x0;
    R7=R6+h3x0;
    R6=R5+h1x0;
    R5=R4+h1x0+h0*x1;
    R4=R3+h3x0;
    R3=R2+h5x0;
    R2=R1+h7x0;
    R1=h9x0;
    return R10;
  }
};
```

3.28 Karlsen

- **Author or source:** Best Regards,Ove Karlsen
- **Type:** 24-dB (4-pole) lowpass
- **Created:** 2003-04-05 06:57:19

Listing 45: notes

```
There's really not much voodoo going on in the filter itself, it's a simple as␣
↪possible:

pole1 = (in  * frequency) + (pole1 * (1 - frequency));

Most of you can probably understand that math, it's very similar to how an analog
condenser works.
Although, I did have to do some JuJu to add resonance to it.
While studing the other filters, I found that the feedback phase is very important to␣
↪how
the overall
resonance level will be, and so I made a dynamic feedback path, and constant Q
approximation by manipulation
of the feedback phase.
A bonus with this filter, is that you can "overdrive" it... Try high input levels..
```

Listing 46: code

```
// Karlsen 24dB Filter by Ove Karlsen / Synergy-7 in the year 2003.
// b_f = frequency 0..1
// b_q = resonance 0..50
// b_in = input
// to do bandpass, subtract poles from eachother, highpass subtract with input.


   float b_inSH = b_in // before the while statement.

    while (b_oversample < 2) {                                              //2x␣
↪oversampling (@44.1khz)
            float prevfp;
            prevfp = b_fp;
            if (prevfp > 1) {prevfp = 1;}                                   // Q-
↪limiter

            b_fp = (b_fp * 0.418) + ((b_q * pole4) * 0.582);                //␣
↪dynamic feedback
            float intfp;
            intfp = (b_fp * 0.36) + (prevfp * 0.64);                        //␣
↪feedback phase
            b_in =  b_inSH - intfp;                                         //␣
↪inverted feedback

            pole1 = (b_in   * b_f) + (pole1 * (1 - b_f));                   // pole 1
            if (pole1 > 1) {pole1 = 1;} else if (pole1 < -1) {pole1 = -1;}  // pole 1␣
↪clipping
            pole2 = (pole1   * b_f) + (pole2 * (1 - b_f));                  // pole 2
            pole3 = (pole2   * b_f) + (pole3 * (1 - b_f));                  // pole 3
            pole4 = (pole3   * b_f) + (pole4 * (1 - b_f));                  // pole 4

            b_oversample++;
            }
    lowpassout = b_in;
```

3.28.1 Comments

- **Date**: 2003-08-08 18:35:52
- **By**: moc.7-ygrenys@evo

```
Hi.
Seems to be a slight typo in my code.
lowpassout = pole4; // ofcourse :)

Best Regards,
Ove Karlsen
```

- **Date**: 2003-09-20 15:31:19
- **By**: moc.tidosha@ttam

```
Hi Ove, we spoke once on the #AROS IRC channel... I'm trying to put this code into a␣
↪filter object, but I'm wandering what datatype the input and output should be?

I'm processing my audio data in packets of 8000 signed words (16 bits) at a time. can␣
↪I put one audio sample words into this function? Since it seems to require a␣
↪floating point input!

Thanks
```

- **Date**: 2004-05-17 18:49:43
- **By**: moc.tsv-nashi@edoc_evo

```
Hi Matt.

Yes, it does indeed need float inputs.

Best Regards,
Ove Karlsen.
```

- **Date**: 2005-03-20 11:36:07
- **By**: se.arret@htrehgraknu

```
Can somebody explain exactly howto make the band Pass and high pass, i tried as␣
↪explained and don't work exactly as expected

highpass = in  - pole4

make "some kind of highpass", but not as expected
cut frequency

and for band pass, how we substract the poles between them ?

pole4-pole3-pole2-pole1 ?
pole1-pole2-pole3-pole4 ?

Also, is there a way to get a Notch ?
```

- **Date**: 2005-03-22 14:14:21
- **By**: ed.luosfosruoivas@naitsirhC

```
Below you will find an object pascal version of the filter.

L=Lowpass
H=Highpass
N=Notch
B=Bandpass

Regards,

Christian

--
unit KarlsenUnit;

interface

type
  TKarlsen = class
  private
    fQ      : Single;
    fF1,fF  : Single;
    fFS     : Single;
    fTmp    : Double;
    fOS     : Byte;
    fPole   : Array[1..4] of Single;
    procedure SetFrequency(v:Single);
    procedure SetQ(v:Single);
  public
    constructor Create;
    destructor Destroy; override;
    procedure Process(const I : Single; var L,B,N,H: Single);
    property Frequency: Single read fF write SetFrequency;
    property SampleRate: Single read fFS write fFS;
    property Q: Single read fQ write SetQ;
    property OverSample: Byte read fOS write fOS;
  end;

implementation

uses sysutils;

const kDenorm = 1.0e-24;

constructor TKarlsen.Create;
begin
 inherited;
 fFS:=44100;
 Frequency:=1000;
 fOS:=2;
 Q:=1;
end;

destructor TKarlsen.Destroy;
begin
 inherited;
end;
```

(continues on next page)

(continued from previous page)

```
procedure TKarlsen.SetFrequency(v:Single);
begin
 if fFS<=0 then raise exception.create('Sample Rate Error!');
 if v<>fF then
  begin
   fF:=v;
   fF1:=fF/fFs; // fF1 range from 0..1
  end;
end;

procedure TKarlsen.SetQ(v:Single);
begin
 if v<>fQ then
  begin
   if v<0  then fQ:=0 else
   if v>50 then fQ:=50 else
   fQ:=v;
  end;
end;

procedure TKarlsen.Process(const I : Single; var L,B,N,H: Single);
var prevfp : Single;
    intfp  : Single;
    o      : Integer;
begin
 for o:=0 to fOS-1 do
  begin
   prevfp:=fTmp;
   if (prevfp > 1) then prevfp:=1; // Q-limiter
   fTmp:=(fTmp*0.418)+((fQ*fPole[4])*0.582); // dynamic feedback
   intfp:=(fTmp*0.36)+(prevfp*0.64); // feedback phase
   fPole[1]:= (((I+kDenorm)-intfp) * fF1) + (fPole[1] * (1 - fF1));
   if (fPole[1] > 1)
    then fPole[1]:= 1
    else if fPole[1] < -1
          then fPole[1]:= -1;
   fPole[2]:=(fPole[1]*fF1)+(fPole[2]*(1-fF1)); // pole 2
   fPole[3]:=(fPole[2]*fF1)+(fPole[3]*(1-fF1)); // pole 3
   fPole[4]:=(fPole[3]*fF1)+(fPole[4]*(1-fF1)); // pole 4
  end;
 L:=fPole[4];
 B:=fPole[4]-fPole[1];
 N:=I-fPole[1];
 H:=I-fPole[4]-fPole[1];
end;

end.
```

- **Date**: 2005-03-29 12:24:17
- **By**: se.sarret@htrehgraknu

```
Thanks Christian!!

Anyway, i tried something similar and seems that what you call Notch is reallya␣
↪Bandpass and the bandpass makes something really strange
```

(continues on next page)

(continued from previous page)

```
Anyway i'm having  other problems with this filter too. It seems to cut  too Loow for␣
↪low pass and too high for high pass. Also, resonance sets a peak far away from the␣
↪cut frequency.And last but not least, the slope isn't 24 db/oct, realkly is much␣
↪lesser, but not in a consistent way: sometimes is 6, sometimes 12, sometimes 20, etc

Any ideas ?
```

- **Date**: 2005-07-20 12:28:44
- **By**: moc.psd-nashi@liam

```
Your problem sounds a bit strange, maybe you should check your implementation.

Nice to see a pascal version too, Christian!

Although I really recommend one set a lower denormal threshold, maybe a 1/100, it␣
↪really affects the sound of the filter. The best is probably tweaking that value in␣
↪realtime to see what sounds best.
Also, doubles for the buffers.. :)

Very Best Regards,
Ove Karlsen
```

- **Date**: 2005-09-20 12:26:37
- **By**: ku.oc.snosrapsd@psdcisum

```
Christian, shouldn't your code end:
L:=fPole[4];
B:=fPole[4]-fPole[1];
//CWB posted
//N:=I-fPole[1];
//B:=I-fPole[4]-fPole[1];

//DSP posted
H:=I-fPole[4]; //Surely pole 4 would give a 24dB/Oct HP, rather than the 6dB version␣
↪posted
N:=I-fPole[4]-fPole[1]; //Inverse of BP

Any thoughts, anyone?

DSP
```

- **Date**: 2005-09-23 11:15:32
- **By**: moc.psd-nashi@liam

```
This filter was really written mostly to demonstrate the Q-limiter though, and also,␣
↪to write it in the most computationally effiecent way.
Here is a little more featured version.
---------------------------------------
// Karlsen, Second Order SVF type filter.

// b_in1, b_in2 stereo input
// fvar01 cutoff
// fvar02 slope
// fvar03 mode
// fvar04 res
```

(continues on next page)

(continued from previous page)

```
// fvar05 cutoff/res compensation

// inits, all doubles
//  b_noise = 19.1919191919191919191919191919191919191919;
//  filterbuffers = 0;

    b_noise = b_noise * b_noise;
    int i_noise = b_noise;
    b_noise = b_noise - i_noise;

    double b_lnoise = (b_noise - 0.5) * 2;
    double b_rnoise = ((1-b_noise) - 0.5) * 2;

    b_noise = b_noise + 19;

    b_lnoise = b_lnoise * 65536;
    b_rnoise = b_rnoise * 65536;
    if (b_lnoise > 1) {b_lnoise = 1;} else if (b_lnoise < -1) {b_lnoise = -1;}
    if (b_rnoise > 1) {b_rnoise = 1;} else if (b_rnoise < -1) {b_rnoise = -1;}

    b_lnoise = b_lnoise * 1e-24; // find optimal value
    b_rnoise = b_rnoise * 1e-24;

    b_in1 = b_in1 + (b_lnoise);  // denormal prevention (also doubling as dither and␣
↪analog noise).
    b_in2 = b_in2 + (b_rnoise);


    float b_slope = (1-fvar2) + 0.5;
    float b_cut = ((fvar1    * fvar1) + ((fvar1 / (b_slope)) * (1 - fvar1))) / ((1 *␣
↪fvar1) + ((1 / (b_slope)) * (1 - fvar1)));
    b_cut = b_cut*b_cut; // linearize this
    float b_res = fvar4 * 100;
    int i_kmode = fvar3 * 100;

    if (b_cut > 1) {b_cut = 1;}
    if (b_cut < 0) {b_cut = 0;}

    b_in1 = (b_in1 + b_lbuffb1);
    b_in2 = (b_in2 + b_lbuffb2);

    b_lbuf09 = ((b_in1    * b_cut) + ((b_lbuf09 / b_slope) * (1 - b_cut))) / ((1 * b_
↪cut) + ((1 / b_slope) * (1 - b_cut)));
    b_lbuf10 = ((b_in2    * b_cut) + ((b_lbuf10 / b_slope) * (1 - b_cut))) / ((1 * b_
↪cut) + ((1 / b_slope) * (1 - b_cut)));

    b_lbuf11 = ((b_lbuf09    * b_cut) + ((b_lbuf11 / b_slope) * (1 - b_cut))) / ((1 *␣
↪b_cut) + ((1 / b_slope) * (1 - b_cut)));
    b_lbuf12 = ((b_lbuf10    * b_cut) + ((b_lbuf12 / b_slope) * (1 - b_cut))) / ((1 *␣
↪b_cut) + ((1 / b_slope) * (1 - b_cut)));

    if (i_kmode == 0) { //lowpass
            b_in1 = b_lbuf11;
            b_in2 = b_lbuf12;
    }
    else if (i_kmode == 1) { // bandpass
            b_in1 = b_lbuf09 - b_lbuf11;
```

(continues on next page)

(continued from previous page)

```
            b_in2 = b_lbuf10 - b_lbuf12;
    }
    else if (i_kmode == 2) { // highpass
            b_in1 = b_in1 - b_lbuf11;
            b_in2 = b_in2 - b_lbuf12;
    }

    b_lbuffb1 = ((b_lbuf09 - b_lbuf11) * ((b_cut * fvar5) + 1)) * b_res;
    b_lbuffb2 = ((b_lbuf10 - b_lbuf12) * ((b_cut * fvar5) + 1)) * b_res;

    b_lbuffb1 = atan(b_lbuffb1);
    b_lbuffb2 = atan(b_lbuffb2);
---------------------------------------------
Works really well with control signals, where you keep the cutoff at a constant level.
Also, a bit more useful with audio, if you linearize the cutoff.

Best Regards,
Ove Karlsen.
```

- **Date**: 2005-10-13 14:00:39
- **By**: ku.oc.snosrapsd@psdcisum

```
I was looking at the b_cut assignment, and was going through looking
at optimising it and found this:

float b_cut = ((fvar1    * fvar1) + ((fvar1 / (b_slope)) * (1 - fvar1)))
/ ((1 * fvar1) + ((1 / (b_slope)) * (1 - fvar1)));

Rename for convenience and clarity
fvar1=co
b_slope=sl

=>  (co^2+(co(1-co)))
          --------
             sl
    ---------------
     (1*co)+(1-co)
             ----
              sl

multiply numerator & denominator by sl to even things up

=>  (sl*co^2+(co(1-co)))
    ------------------
      (sl*co)+(1-co)

expand brackets

=>   sl*co^2+co-co^2
    ---------------
      sl*co+1-co

refactor

=>   co(sl*co+1-co)
    ---------------
```

(continues on next page)

(continued from previous page)

```
      sl*co+1-co

(sl*co+1-co) cancels out, leaving..

=> co


if I've got anything wrong here, please pipe up..

Duncan
```

- **Date**: 2005-10-13 14:06:02
- **By**: ku.oc.snosrapsd@psdcisum

```
(actually, typing the assigment into Excel reveals the same as my proof..)
```

- **Date**: 2006-02-17 13:23:47
- **By**: moc.psd-nashi@liam

```
Final version, Stenseth, 17. february, 2006.

// Fast differential amplifier approximation

    double b_inr = b_in * b_filterdrive;
    if (b_inr < 0) {b_inr = -b_inr;}
    double b_inrns = b_inr;
    if (b_inr > 1) {b_inr = 1;}
    double b_dax = b_inr - ((b_inr * b_inr) * 0.5);
    b_dax = b_dax - b_inr;
    b_inr = b_inr + b_dax;

    b_inr = b_inr * 0.24;

    if (b_inr > 1) {b_inr = 1;}
    b_dax = b_inr - ((b_inr * 0.33333333) * (b_inr * b_inr));
    b_dax = b_dax - b_inr;
    b_inr = b_inr + b_dax;

    b_inr = b_inr / 0.24;

    double b_mul = b_inrns / b_inr; // beware of zero
    b_sbuf1 = ((b_sbuf1 - (b_sbuf1 * 0.4300)) + (b_mul * 0.4300));

    b_mul = b_sbuf1 + ((b_mul - b_sbuf1) * 0.6910);
    b_in = b_in / b_mul;

// This method sounds the best here..
// About denormals, it does not seem to be much of an issue here, probably because I␣
↪input the filters with oscillators, and not samples, or other, where the level may␣
↪drop below the denormal threshold for extended periods of time. However, if you do,␣
↪you probably want to quantize out the information below the threshold, in the␣
↪buffers, and raise/lower the inputlevel before/after the filter. Adding low levels␣
↪of noise may be effective aswell. This is described somewhere else on this site.

    double b_cutsc = pow(1024,b_cut) / 1024; // perfect tracking..
```

(continues on next page)

(continued from previous page)

```
    b_fbuf1 = ((b_fbuf1 - (b_fbuf1 * b_cutsc)) + (b_in * b_cutsc));
    b_in = b_fbuf1;
    b_fbuf2 = ((b_fbuf2 - (b_fbuf2 * b_cutsc)) + (b_in * b_cutsc));
    b_in = b_fbuf2;
    b_fbuf3 = ((b_fbuf3 - (b_fbuf3 * b_cutsc)) + (b_in * b_cutsc));
    b_in = b_fbuf3;
    b_fbuf4 = ((b_fbuf4 - (b_fbuf4 * b_cutsc)) + (b_in * b_cutsc));
    b_in = b_fbuf4;

Soundwise, it's somewhere between a transistor ladder, and a diode ladder..
Enjoy!

Ove Karlsen.
PS: I prefer IRL communication these days, so if you need to reach me, please dial my␣
↪cellphone, +047 928 50 803.
```

- **Date**: 2006-11-03 03:51:29
- **By**: read@bw

```
Another iteration, please delete all other posts than this.

Arif Ove Karlsen's 24dB Ladder Approximation, 3.nov 2007

As you may know, The traditional 4-pole Ladder found in vintage hardware synths,
had a particular sound. The nonlinearities inherent in the suboptimal components,␣
↪often
added a particular flavour to the sound.
Digital does mathematical calculations much better than any analog solution, and␣
↪therefore, when the filter was emulated by digital filter types, some of the␣
↪character got lost.
I believe this mainly boils down to the resonance limiting occuring in the analog␣
↪version.
Therefore I have written a very fast ladder approximation, not emulating any of what␣
↪may seem neccesary, such as pole saturaion, which in turn results in nonlinear␣
↪cutoff frequency, and loss of volume at lower cutoffs. However this can be␣
↪implemented, if wanted, by putting the neccesary saturation functions inside the␣
↪code. If you seek the true analog sound, you may want to do a full differential␣
↪amplifier emulation aswell.
But - I believe in the end, you would end up wanting a perfect filter, with just the␣
↪touch that makes it sound analog, resonance limiting.

So here it is, Karlsen Ladder, v4. A very resource effiecent ladder. Can furthermore␣
↪be optimized with asm.

rez = pole4 * rezamount; if (rez > 1) {rez = 1;}
input = input - rez;
pole1 = pole1 + ((-pole1 + input) * cutoffreq);
pole2 = pole2 + ((-pole2 + pole1) * cutoffreq);
pole3 = pole3 + ((-pole3 + pole2) * cutoffreq);
pole4 = pole4 + ((-pole4 + pole3) * cutoffreq);
output = pole4;

--

I can be reached by email @ 1a2r4i54f5o5v2ek1a1r5ls6en@3ho2tm6ail1.c5o6m!no!nums
```

- **Date**: 2013-06-21 14:42:33
- **By**: pleasee@otherpost

```
Please see
            http://musicdsp.org/showArchiveComment.php?ArchiveID=240

for the ultimate development of this filter.

Peace Be With You,
Ove Karlsen
```

3.29 Karlsen Fast Ladder

- **Author or source:** moc.liamtoh@neslrakevofira
- **Type:** 4 pole ladder emulation
- **Created:** 2007-01-08 10:49:56

Listing 47: notes

```
ATTN Admin: You should remove the old version named "Karlsen" on your website, and␣
↪rather
include this one instead.
```

Listing 48: code

```
// An updated version of "Karlsen 24dB Filter"
// This time, the fastest incarnation possible.
// The very best greetings, Arif Ove Karlsen.
// arifovekarlsen->hotmail.com

b_rscl = b_buf4; if (b_rscl > 1) {b_rscl = 1;}
b_in = (-b_rscl * b_rez) + b_in;
b_buf1 = ((-b_buf1 + b_in1) * b_cut) + b_buf1;
b_buf2 = ((-b_buf2 + b_buf1) * b_cut) + b_buf2;
b_buf3 = ((-b_buf3 + b_buf2) * b_cut) + b_buf3;
b_buf4 = ((-b_buf4 + b_buf3) * b_cut) + b_buf4;
b_lpout = b_buf4;
```

3.29.1 Comments

- **Date**: 2007-01-08 18:42:24
- **By**: moc.erehwon@ydobon

```
Where are the coefficients? How do I set the cutoff frequency?
```

- **Date**: 2007-01-09 09:49:18
- **By**: uh.etle.fni@yfoocs

```
The parameters are:
b_cut - cutoff freq
```

(continues on next page)

(continued from previous page)

```
b_rez - resonance
b_in1 - input

Cutoff is normalized frequency in rads (2*pi*cutoff/samplerate). Stability limit for␣
↪b_cut is around 0.7-0.8.

There's a typo, the input is sometimes b_in, sometimes b_in1. Anyways why do you use␣
↪a b_ prefix for all your variables? Wouldn't it be more easy to read like this:

resoclip = buf4; if (resoclip > 1) resoclip = 1;
in = in - (resoclip * res);
buf1 = ((in - buf1) * cut) + buf1;
buf2 = ((buf1 - buf2) * cut) + buf2;
buf3 = ((buf2 - buf3) * cut) + buf3;
buf4 = ((buf3 - buf4) * cut) + buf4;
lpout = buf4;

Also note that asymmetrical clipping gives you DC offset (at least that's what I get),
↪ so symmetrical clipping is better (and gives a much smoother sound).

-- peter schoffhauzer
```

- **Date**: 2007-06-26 16:23:57
- **By**: moc.psd8rts@fira

```
Tee b_prefix is simply a procedure I began using when I started programming C.␣
↪Influenced by the BEOS operating system. However it seemed to also make my code␣
↪more readable, atleast to me. So I started using various prefixes for various␣
↪things, making the variables easily reckognizable. Peter, everyone, I am now␣
↪reachable on www.str8dsp.com - Do also check out the plugin offers there!
```

- **Date**: 2007-07-17 20:21:47
- **By**: moc.psd8rts@koa

```
Here's even another filter, I will probably never get around to making any product␣
↪with this one so here it is, pseudo-vintage diode ladder.

                    Diode Ladder, (unbuffered)

                    // limit resonance, rzl, tweak smearing with fltw, 0.3230 seems␣
↪to be a good vintage sound.
                    in = in - rzl;
                    in = in + ((-in +kbuf1) * cutoff);
                    kbuf1 = in + ((-in + kbuf1) * fltw);
                    in = in + ((-in +kbuf2) * cutoff);
                    kbuf2 = in + ((-in + kbuf2) * fltw);
                    etc..
```

- **Date**: 2007-09-09 22:37:08
- **By**: moc.oiduatniopxf@ved

```
"Cutoff is normalized frequency in rads (2*pi*cutoff/samplerate):

This seems to be valid for very low ( < 200 Hz ) frequencies - higher sample rates␣
↪seem to be "Closer"
```

(continues on next page)

(continued from previous page)

```
thanks
```

- **Date**: 2010-07-17 09:25:41
- **By**: moc.liamerofegapkcehc@liamerofegapkcehc

```
I also did a 9th order gaussian filter (minimal phase), using only 5 orders, for my␣
↪limiter, which is released under the GPL LICENCE. http://www.paradoxuncreated.com
```

- **Date**: 2012-11-14 08:15:06
- **By**: Generalized perfect digital "ladder" filter, with the desired aspects of analog.

```
Hi, I have now generalized the ladder filter, into fast code, and with the desired␣
↪aspects of analog, but retaining digital perfectness.

Please see my blog: http://paradoxuncreated.com/Blog/wordpress/?p=1360

Peace Be With You.
```

- **Date**: 2013-06-21 14:44:18
- **By**: moc.golb@eesesaelp

```
I have also moved domains now, and consolidated the information on this ultimate␣
↪digital filter, with "analog sound", here:

http://ovekarlsen.com/Blog/abdullah-filter/

Peace Be With You!
```

- **Date**: 2016-02-14 01:31:32
- **By**: ove hy karlsen @ facebook.com

```
Karlsen Fast Ladder III - inspired by "transistors set to work as diode" type Roland␣
↪filters. The best fast and non-nonsensical approximation of popular analog filter␣
↪sound, as in for instance Roland SH-5, and the smaller TB-303.

//Coupled with oversampling and simple oscs you will probably get the best analog␣
↪approximation.

//          // for nice low sat, or sharper type low deemphasis saturation, one can␣
↪use a onepole shelf before the filter.
//          b_lf = b_lf + ((-b_lf + b_v) * b_lfcut); // b_lfcut 0..1
//          double b_lfhp = b_v - b_lf;
//          b_v = b_lf + (b_lf1hp * ((b_lfgain*0.5)+1));

            double b_rez = b_aflt4 - b_v; // no attenuation with rez, makes a stabler␣
↪filter.
            b_v = b_v - (b_rez*b_fres); // b_fres = resonance amount. 0..4 typical
↪"to selfoscillation", 0.6 covers a more saturated range.

            double b_vnc = b_v; // clip, and adding back some nonclipped, to get a␣
↪dynamic like analog.
            if (b_v > 1) {b_v = 1;} else if (b_v < -1) {b_v = -1;}
            b_v = b_vnc + ((-b_vnc + b_v) * 0.9840);
```

(continues on next page)

(continued from previous page)

```
            b_aflt1 = b_aflt1 + ((-b_aflt1 + b_v) * b_fenv); // straightforward 4␣
↪pole filter, (4 normalized feedback paths in series)
            b_aflt2 = b_aflt2 + ((-b_aflt2 + b_aflt1) * b_fenv);
            b_aflt3 = b_aflt3 + ((-b_aflt3 + b_aflt2) * b_fenv);
            b_aflt4 = b_aflt4 + ((-b_aflt4 + b_aflt3) * b_fenv);
            b_v = b_aflt4;

// Behave.
// Ove Hy Karlsen.
```

- **Date**: 2018-03-12 09:34:57
- **By**: moc.liamtoh@06rorrexatnys

```
Hey Ove

I am wondering about the last filter the  Fast ladder diode III. Where is the input␣
↪supposed to go?

Sorry, I am still learning and thanks for some great filters, btw :)

Thanks, Jakob
```

3.30 LP and HP filter

- **Author or source:** Patrice Tarrabia
- **Type:** biquad, tweaked butterworth
- **Created:** 2002-01-17 02:13:47

Listing 49: 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);
```

(continues on next page)

(continued from previous page)

```
        a2 = -2*a1;
        a3 = a1;
        b1 = 2.0 * ( c*c - 1.0) * a1;
        b2 = ( 1.0 - r * c + c * c) * a1;
```

3.30.1 Comments

- **Date**: 2002-03-14 15:24:20
- **By**: moc.liamtoh@lossor_ydna

```
Ok, the filter works, but how to use the resonance parameter (r)? The range from␣
↪sqrt(2)-lowest to 0.1 (highest res.) is Ok for a LP with Cutoff > 3 or 4 KHz, but␣
↪for lower cutoff frequencies and higher res you will get values much greater than 1!
↪ (And this means clipping like hell)

So, has anybody calculated better parameters (for r, b1, b2)?
```

- **Date**: 2003-04-03 10:23:36
- **By**: moc.liamtoh@trahniak

```
Below is my attempt to implement the above lowpass filter in c#. I'm just a beginner␣
↪at this so it's probably something that I've messed up. If anybody can offer a␣
↪suggestion of what I may be doing wrong please help. I'm getting a bunch of stable␣
↪staticky noise as my output of this filter currently.
```

- **Date**: 2003-04-03 10:25:15
- **By**: moc.liamtoh@trahniak

```
public class LowPassFilter
{
        /// <summary>
        /// rez amount, from sqrt(2) to ~ 0.1
        /// </summary>
        float r;
        /// <summary>
        /// cutoff frequency
        /// (from ~0 Hz to SampleRate/2 - though many
        /// synths seem to filter only up to SampleRate/4)
        ///</summary>
        float f;
        float c;

        float a1;
        float a2;
        float a3;
        float b1;
        float b2;

//      float in0 = 0;
        float in1 = 0;
        float in2 = 0;

//      float out0;
```

(continues on next page)

(continued from previous page)

```
        float out1 = 0;
        float out2 = 0;

        private int _SampleRate;

        public LowPassFilter(int sampleRate)
        {
                _SampleRate = sampleRate;

//              SetParams(_SampleRate / 2f, 0.1f);
                SetParams(_SampleRate / 8f, 1f);
        }


        public float Process(float input)
        {
                float output = a1 * input +
                                 a2 * in1 +
                                 a3 * in2 -
                                 b1 * out1 -
                                 b2 * out2;

                in2 = in1;
                in1 = input;

                out2 = out1;
                out1 = output;

                Console.WriteLine(input + ", " + output);

                return output;
        }
```

- **Date**: 2003-04-03 10:25:39
- **By**: moc.liamtoh@trahniak

```
        /// <summary>
        ///
        /// </summary>
        public float CutoffFrequency
        {
                set
                {
                        f = value;
                        c = (float) (1.0f / Math.Tan(Math.PI * f / _SampleRate));
                        SetParams();
                }
                get
                {
                        return f;
                }
        }

        /// <summary>
        ///
        /// </summary>
```

(continues on next page)

(continued from previous page)

```
        public float Resonance
        {
                set
                {
                        r = value;
                        SetParams();
                }
                get
                {
                        return r;
                }
        }

        public void SetParams(float cutoffFrequency, float resonance)
        {
                r = resonance;
                CutoffFrequency = cutoffFrequency;
        }

        /// <summary>
        /// TODO rename
        /// </summary>
        /// <param name="c"></param>
        /// <param name="resonance"></param>
        private void SetParams()
        {
                a1 = 1f / (1f + r*c + c*c);
                a2 = 2 * a1;
                a3 = a1;
                b1 = 2f * (1f - c*c) * a1;
                b2 = (1f - r*c + c*c) * a1;
        }
}
```

- **Date**: 2003-04-03 11:58:51
- **By**: moc.liamtoh@trahniak

```
Nevermind I think I solved my problem. I was missing parens around the coefficients␣
↪and the variables ...(a1 * input)...
```

- **Date**: 2003-04-22 17:30:14
- **By**: moc.liamtoh@trahniak

```
After implementing the lowpass algorithm I get a loud ringing noise on some␣
↪frequencies both high and low. Any ideas?
```

- **Date**: 2006-03-29 14:10:59
- **By**: ed.xmg@lhadl

```
hi,
since this is the best filter i found on the net, i really need bandpass and bandstop!
↪!! can anyone help me with the coefficents?
```

- **Date**: 2006-05-23 18:25:18
- **By**: uh.etle.fni@yfoocs

```
AFAIK there's no separate bandpass and bandstop version of Butterworth filters.␣
↪Instead, bandpass is usually done by cascading a HP and a LP filter, and bandstop␣
↪is the mixed output of a HP and a LP filter. However, there's bandpass biquad code␣
↪(for example RBJ biquad filters). Cheers Peter
```

- **Date**: 2006-05-28 20:15:48
- **By**: uh.etle.fni@yfoocs

```
You can save two divisions for lowpass using
  c = tan((0.5 - (f * inv_samplerate))*pi);
instead of
  c = 1.0 / tan(pi * f / sample_rate);
where inv_samplerate is 1.0/samplerate precalculated. (mul is faster than div)

However, the latter form can be approximated very well below 4kHz (at 44kHz␣
↪samplerate) with
  c = 1.0 / (pi * f * inv_sample_rate);
which is far better than both of the previous two equations, because it does not use␣
↪any transcendental functions. So, an optimized form is:

f0 = f * inv_sample_rate;
if (f0 < 0.1) c = 1.0 / (f0 * pi); // below 4.4k
else c = tan((0.5 - f0) * pi);

This needs only about ~60% CPU below 4.4kHz. Probably using lookup tables could make␣
↪it even faster...

Mapping resonance range 0..1 to 0..self-osc:
  float const sqrt_two = 1.41421356;
  r = sqrt_two - resonance * sqrt_two;

Setting resonance in the conventional q form (like in RBJ biquads):
  r = 1.0/q;

Cheers, Peter
```

- **Date**: 2006-05-28 20:43:28
- **By**: uh.etle@yfoocs

```
However I find that this algorythm has a slight tuning error regardless of using␣
↪approximation or not. 'inv_samplerate = 0.95 * samplerate' seems to give a more␣
↪accurate frequency tuning.
```

- **Date**: 2006-05-29 15:50:13
- **By**: uh.etle.fni@yfoocs

```
You can use the same trick for highpass:

precalc when setting up the filter:
  inv_samplerate = 1.0 / samplerate * 0.957;
(multipying by 0.957 seems to give the most precise tuning)

and then calculating c:

f0 = f * inv_samplerate;
```

(continues on next page)

(continued from previous page)

```
if (f0 < 0.05) c = (f0 * pi);
else c = tan(f0 * pi);

Now I used 0.05 instead of 0.1, thats 0.05 * 44100  = 2.2k instead of 4.4k. So, this␣
↪is a bit more precise than 0.1, becuase around 3-4k it had a slight error, however,␣
↪only noticeable on the analyzer when compared to the original version. This is␣
↪still about two third of the logarithmic frequency scale, so it's quite a bit of a␣
↪speed improvement. You can use either precision for both lowpass and highpass.

For calculating tan(), you can take some quick sin() approximation, and use:
  tan(x)=sin(x)/sin(half_pi-x)

There are many good pieces of code for that in the archive.

I tried to make some 1/x based approximations for 1.0/tan(x), here is one:

inline float tan_inv_approx(float x)
{
    float const two_div_pi = 2.0f/3.141592654f;
    if (x<0.5f) return 1.0f/x;
    else return 1.467f*(1.0f/x-two_div_pi);
}

This one is pretty fast, however it is a quite rough estimate; it has some 1-2␣
↪semitones frequency tuning error around 5-8 kHz and above 10kHz. Might be usable␣
↪for synths, however, or somewhere where scientific precision is not needed.

Cheers, Peter
```

- **Date**: 2006-05-30 21:12:13
- **By**: uh.etle.fni@yfoocs

```
Sorry, forget the * 0.957 tuning, this algorythm is precise without that, the mistake␣
↪was in my program. Everything else is valid, I hope.
```

- **Date**: 2008-03-11 13:31:52
- **By**: ur.kb@sexof

```
Optimization for Hipass:

c = tan(pi * f / sample_rate);

c = ( c + r ) * c;
a1 = 1.0 / ( 1.0 + c );
b1 = ( 1.0 - c );

out(n) = ( a1 * out(n-1) + in - in(n-1) ) * b1;
```

3.31 LPF 24dB/Oct

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** Chebyshev

- **Created:** 2006-07-28 17:58:33

Listing 50: code

```
First calculate the prewarped digital frequency:

K = tan(Pi * Frequency / Samplerate);

Now we calc some Coefficients:

sg = Sinh(PassbandRipple);
cg = Cosh(PassbandRipple);
cg *= cg;

Coeff[0] = 1 / (cg-0.85355339059327376220042218105097);
Coeff[1] = K * Coeff[0]*sg*1.847759065022573512256366378792;
Coeff[2] = 1 / (cg-0.14644660940672623779957781894758);
Coeff[3] = K * Coeff[2]*sg*0.76536686473017954345691996806;

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

Calculate the first biquad:

A0 =   (Coeff[1]+K+Coeff[0]);
A1 = 2*(Coeff[0]-K)*t;
A2 =   (Coeff[1]-K-Coeff[0])*t;
B0 = t*K;
B1 = 2*B0;
B2 = B0;

Calculate the second biquad:

A3 =   (Coeff[3]+K+Coeff[2]);
A4 = 2*(Coeff[2]-K)*t;
A5 =   (Coeff[3]-K-Coeff[2])*t;
B3 = t*K;
B4 = 2*B3;
B5 = B3;

Then calculate the output as follows:

Stage1 = B0*Input + State0;
State0 = B1*Input + A1/A0*Stage1 + State1;
State1 = B2*Input + A2/A0*Stage1;

Output = B3*Stage1 + State2;
State2 = B4*Stage1 + A4/A3*Output + State2;
State3 = B5*Stage1 + A5/A3*Output;
```

3.31.1 Comments

- **Date**: 2006-09-14 10:36:43
- **By**: musicdsp@Nospam dsparsons.co.uk

```
You've used two notations here (as admitted on KVR!)..
Updated calculation code reads:
```

(continues on next page)

(continued from previous page)

```
=-=-=-= Start =-=-=-=

Calculate the first biquad:

//A0 =   (Coeff[1]+K+Coeff[0]);
t  = 1/(Coeff[1]+K+Coeff[0]);
A1 = 2*(Coeff[0]-K)*t;
A2 =   (Coeff[1]-K-Coeff[0])*t;
B0 = t*K;
B1 = 2*B0;
B2 = B0;

Calculate the second biquad:

//A3 =   (Coeff[3]+K+Coeff[2]);
t  = 1/(Coeff[3]+K+Coeff[2]);
A4 = 2*(Coeff[2]-K)*t;
A5 =   (Coeff[3]-K-Coeff[2])*t;
B3 = t*K;
B4 = 2*B3;
B5 = B3;

Then calculate the output as follows:

Stage1 = B0*Input + State0;
State0 = B1*Input + A1*Stage1 + State1;
State1 = B2*Input + A2*Stage1;

Output = B3*Stage1 + State2;
State2 = B4*Stage1 + A4*Output + State2;
State3 = B5*Stage1 + A5*Output;

=-=-=-=  End  =-=-=-=

Hope that clears up any confusion for future readers :-)
```

- **Date**: 2008-06-24 13:57:19
- **By**: moc.liamg@tnemelCssoR

```
The variable State3 is assigned a value, but is never used anywhere. Is there a␣
↪reason for this?
```

- **Date**: 2008-10-17 00:40:33
- **By**: moc.liamg@321tiloen

```
Just ported this into Reaper's native JesuSonic.

There are errors in both of the codes above :D
Use this:

//start

A0  = 1/(Coeff[1]+K+Coeff[0]);
A1 = 2*(Coeff[0]-K)*A0;
```

(continues on next page)

(continued from previous page)

```
A2 = (Coeff[1]-K-Coeff[0])*A0;
B0 = A0*K;
B1 = 2*B0;
B2 = B0;

A3  = 1/(Coeff[3]+K+Coeff[2]);
A4 = 2*(Coeff[2]-K)*A3;
A5 = (Coeff[3]-K-Coeff[2])*A3;
B3 = A3*K;
B4 = 2*B3;
B5 = B3;

Stage1 = B0*Input + State0;
State0 = B1*Input + A1*Stage1 + State1;
State1 = B2*Input + A2*Stage1;
Output = B3*Stage1 + State2;
State2 = B4*Stage1 + A4*Output + State3;
State3 = B5*Stage1 + A5*Output;

//end

@RossClement[ AT ]gmail[ DOT ]com
'State3' should be added in this line
-> State2 = B4*Stage1 + A4*Output + State3;
```

3.32 Lowpass filter for parameter edge filtering

- **Author or source:** Olli Niemitalo
- **Created:** 2002-01-17 02:06:29
- **Linked files:** `filter001.gif`.

Listing 51: notes

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

Listing 52: 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;
```

(continues on next page)

(continued from previous page)

```

        /* 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;
        }
}
```

3.32.1 Comments

- **Date**: 2007-01-06 04:19:27
- **By**: uh.etle.fni@yfoocs

```
Wouldn't just one pole with a low cutoff suit this purpose? At least that's what I␣
↪usually do for smoothing parameter changes, and it works fine.
```

- **Date**: 2014-07-14 22:02:46
- **By**: moc.liamg@uttrep.imas

```
Works nicely, thanks. The gain calculations can be simplified quite a bit, to just␣
↪one gain parameter.

gain = 1.0 / (-1.0 / log(a) + 2.0 / log(a * b) - 1.0 / log(a * b * b))

Then,

again = gain,
bgain = -2.0 * gain, and
cgain = gain.
```

3.33 MDCT and IMDCT based on FFTW3

- **Author or source:** moc.liamg@gnahz.auhuhs
- **Type:** analysis and synthesis filterbank
- **Created:** 2009-07-31 06:37:37

Listing 53: notes

```
MDCT/IMDCT is the most widely used filterbank in digital audio coding, e.g. MP3, AAC,␣
↪WMA,
OGG Vorbis, ATRAC.

suppose input x and N=size(x,1)/2. the MDCT transform matrix is
    C=cos(pi/N*([0:2*N-1]'+.5+.5*N)*([0:N-1]+.5));
then MDCT spectrum for input x is
    y=C'*x;
A well known fast algorithm is based on FFT :
 (1) fold column-wisely the 2*N rows into N rows
 (2) complex arrange the N rows into N/2 rows
 (3) pre-twiddle, N/2-point complex fft, post-twiddle
 (4) reorder to form the MDCT spectrum
in fact, (2)-(4) is a fast DCT-IV algorithm.

Implementation of the algorithm can be found in faac, but a little bit mess to␣
↪extract for
standalone use, and I ran into that problem. So I wrote some c codes to implement
MDCT/IMDCT for any length that is a multiple of 4. Hopefully they will be useful to␣
↪people
here.

I benchmarked the codes using 3 FFT routines, FFT in faac, kiss_fft, and the awful␣
↪FFTW.
MDCT based on FFTW is the fastest, 2048-point MDCT single precision 10^5 times in 1.
↪54s,
about 50% of FFT in faac on my Petium IV 3G Hz.
```

(continues on next page)

(continued from previous page)

```
An author of the FFTW, Steven G. Johnson, has a hard-coded fixed size MDCT of 256␣
↪input
samples(http://jdj.mit.edu/~stevenj/mdct_128nr.c). My code is 13% slower than his.


Using the codes is very simple:
 (1) init (declare first "extern void* mdctf_init(int)")
       void* m_plan = mdctf_init(N);
 (2) run mdct/imdct as many times as you wish
       mdctf(freq, time, m_plan);
 (3) free
       mdctf_free(m_plan);

Of course you need the the fftw library. On Linux, gcc options are "-O2 -lfftw3f -lm".
This is single precision.


Enjoy :)
```

Listing 54: code

```
/*********************************************************
  MDCT/IMDCT of 4x length, Single Precision, based on FFTW
  shuhua dot zhang at gmail dot com
  Dept. of E.E., Tsinghua University
 *********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fftw3.h>


typedef struct {
    int                 N;              // Number of time data points
    float*              twiddle;        // Twiddle factor
    fftwf_complex*      fft_in;         // fft workspace, input
    fftwf_complex*      fft_out;        // fft workspace, output
    fftwf_plan          fft_plan;       // fft configuration
} mdctf_plan;


mdctf_plan* mdctf_init(int N);
void mdctf_free(mdctf_plan* m_plan);
void mdctf(float* mdct_line, float* time_signal, mdctf_plan* m_plan);
void imdctf(float* time_signal, float* mdct_line, mdctf_plan* m_plan);


mdctf_plan* mdctf_init(int N)
{
    mdctf_plan* m_plan;
    double alpha, omiga, scale;
    int    n;

    if( 0x00 != (N & 0x03))
    {
```

(continues on next page)

(continued from previous page)

```
        fprintf(stderr, " Expecting N a multiple of 4\n");
        return NULL;
    }

    m_plan = (mdctf_plan*) malloc(sizeof(mdctf_plan));

    m_plan->N = N;

    m_plan->twiddle = (float*) malloc(sizeof(float) * N >> 1);
    alpha = 2.f * M_PI / (8.f * N);
    omiga = 2.f * M_PI / N;
    scale = sqrt(sqrt(2.f / N));
    for(n = 0; n < (N >> 2); n++)
    {
        m_plan->twiddle[2*n+0] = (float) (scale * cos(omiga * n + alpha));
        m_plan->twiddle[2*n+1] = (float) (scale * sin(omiga * n + alpha));
    }

    m_plan->fft_in   = (fftwf_complex*) fftwf_malloc(sizeof(fftwf_complex) * N >> 2);
    m_plan->fft_out  = (fftwf_complex*) fftwf_malloc(sizeof(fftwf_complex) * N >> 2);
    m_plan->fft_plan = fftwf_plan_dft_1d(N >> 2,
                                         m_plan->fft_in,
                                         m_plan->fft_out,
                                         FFTW_FORWARD,
                                         FFTW_MEASURE);

    return m_plan;

}


void mdctf_free(mdctf_plan* m_plan)
{
    fftwf_destroy_plan(m_plan->fft_plan);
    fftwf_free(m_plan->fft_in);
    fftwf_free(m_plan->fft_out);
    free(m_plan->twiddle);
    free(m_plan);
}


void mdctf(float* mdct_line, float* time_signal, mdctf_plan* m_plan)
{
    float  *xr, *xi, r0, i0;
    float  *cos_tw, *sin_tw, c, s;
    int     N4, N2, N34, N54, n;

    N4  = (m_plan->N) >> 2;
    N2  = 2 * N4;
    N34 = 3 * N4;
    N54 = 5 * N4;

    cos_tw = m_plan->twiddle;
    sin_tw = cos_tw + 1;

    /* odd/even folding and pre-twiddle */
    xr = (float*) m_plan->fft_in;
```

(continues on next page)

(continued from previous page)

```
    xi = xr + 1;
    for(n = 0; n < N4; n += 2)
    {
        r0 = time_signal[N34-1-n] + time_signal[N34+n];
        i0 = time_signal[N4+n]    - time_signal[N4-1-n];

        c = cos_tw[n];
        s = sin_tw[n];

        xr[n] = r0 * c + i0 * s;
        xi[n] = i0 * c - r0 * s;
    }

    for(; n < N2; n += 2)
    {
        r0 = time_signal[N34-1-n] - time_signal[-N4+n];
        i0 = time_signal[N4+n]    + time_signal[N54-1-n];

        c = cos_tw[n];
        s = sin_tw[n];

        xr[n] = r0 * c + i0 * s;
        xi[n] = i0 * c - r0 * s;
    }

    /* complex FFT of N/4 long */
    fftwf_execute(m_plan->fft_plan);

    /* post-twiddle */
    xr = (float*) m_plan->fft_out;
    xi = xr + 1;
    for(n = 0; n < N2; n += 2)
    {
        r0 = xr[n];
        i0 = xi[n];

        c = cos_tw[n];
        s = sin_tw[n];

        mdct_line[n]      = - r0 * c - i0 * s;
        mdct_line[N2-1-n] = - r0 * s + i0 * c;
    }
}


void imdctf(float* time_signal, float* mdct_line, mdctf_plan* m_plan)
{
    float  *xr, *xi, r0, i0, r1, i1;
    float  *cos_tw, *sin_tw, c, s;
    int     N4, N2, N34, N54, n;

    N4  = (m_plan->N) >> 2;
    N2  = 2 * N4;
    N34 = 3 * N4;
    N54 = 5 * N4;

    cos_tw = m_plan->twiddle;
```

(continues on next page)

(continued from previous page)

```
    sin_tw = cos_tw + 1;

        /* pre-twiddle */
    xr = (float*) m_plan->fft_in;
    xi = xr + 1;
    for(n = 0; n < N2; n += 2)
    {
        r0 =  mdct_line[n];
        i0 =  mdct_line[N2-1-n];

        c = cos_tw[n];
        s = sin_tw[n];

            xr[n] = -2.f * (i0 * s + r0 * c);
        xi[n] = -2.f * (i0 * c - r0 * s);
    }

    /* complex FFT of N/4 long */
    fftwf_execute(m_plan->fft_plan);

    /* odd/even expanding and post-twiddle */
    xr = (float*) m_plan->fft_out;
    xi = xr + 1;
    for(n = 0; n < N4; n += 2)
    {
        r0 = xr[n];
        i0 = xi[n];

        c = cos_tw[n];
        s = sin_tw[n];

        r1 = r0 * c + i0 * s;
        i1 = r0 * s - i0 * c;

        time_signal[N34-1-n] =  r1;
        time_signal[N34+n]   =  r1;
        time_signal[N4+n]    =  i1;
        time_signal[N4-1-n]  = -i1;
    }

    for(; n < N2; n += 2)
    {
        r0 = xr[n];
        i0 = xi[n];

        c = cos_tw[n];
        s = sin_tw[n];

        r1 = r0 * c + i0 * s;
        i1 = r0 * s - i0 * c;

        time_signal[N34-1-n] =  r1;
        time_signal[-N4+n]   = -r1;
        time_signal[N4+n]    =  i1;
        time_signal[N54-1-n] =  i1;
    }
}
```

3.33.1 Comments

- **Date**: 2009-08-05 14:42:44
- **By**: none

```
Hi, your "freq, time" example in your comments feed into the main function as "mdct_
↪line, time_signal" float pointers.
Can you explain what these are?
Thanks
D
```

- **Date**: 2009-08-11 05:11:59
- **By**: moc.liamg@gnahz.auhuhs

```
Hi,

  Here I past a complete test bench for the MDCT/IMDCT routine. Suppose the MDCT/
↪IMDCT routines named "mdctf.c" and the following benchmark routine named
↪"ftestbench.c", the gcc compilation command will be
  gcc -o ftestbench -O2 ftestbench.c mdctf.c -lfftw3f -lm

Shuhua Zhang, Aug. 11, 2009


/* benchmark MDCT and IMDCT, floating point */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

extern void* mdctf_init(int);

int main(int argc, char* argv[])
{
    int N, r, i;
    float* time;
    float* freq;
    void* m_plan;
    clock_t t0, t1;

    if(3 != argc)
    {
        fprintf(stderr, " Usage: %s <MDCT_SIZE> <run_times> \n", argv[0]);
        return -1;
    }

    sscanf(argv[1], "%d", &N);
    sscanf(argv[2], "%d", &r);

    time = (float*)malloc(sizeof(float) * N);
    freq = (float*)malloc(sizeof(float) * (N >> 1));
    for(i = 0; i < N; i++)
        time[i] = 2.f * rand() / RAND_MAX - 1.f;

    /* MDCT/IMDCT floating point initialization */
    m_plan = mdctf_init(N);
```

(continues on next page)

(continued from previous page)

```
    if(NULL == m_plan)
    {
            free(freq);
            free(time);
            return -1;
    }

    /* benchmark MDCT floating point*/
    t0 = clock();
    for(i = 0; i < r; i++)
        mdctf(freq, time, m_plan);
    t1 = clock();
    fprintf(stdout, "MDCT of size %d, float, running %d times, uses %.2e s\n",
            N, r, (float) (t1 - t0) / CLOCKS_PER_SEC);

    /* benchmark IMDCT floating point*/
    t0 = clock();
    for(i = 0; i < r; i++)
        imdctf(time, freq, m_plan);
    t1 = clock();
    fprintf(stdout, "IMDCT of size %d, float, running %d times, uses %.2e s\n",
            N, r, (float) (t1 - t0) / CLOCKS_PER_SEC);

    /* free MDCT/IMDCT workspace */
    mdctf_free(m_plan);

    free(freq);
    free(time);

    return 0;

}
```

3.34 Moog Filter

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** Antti's version (nonlinearities)
- **Created:** 2005-04-27 08:54:50

Listing 55: notes

```
Here is a Delphi/Object Pascal translation of Antti's Moog Filter.

Antti wrote:

"At last DAFX I published a paper presenting a non-linear model of the Moog ladder.␣
↪For
that, see http://dafx04.na.infn.it/WebProc/Proc/P_061.pdf

I used quite different approach in that one. A half-sample delay ([0.5 0.5] FIR filter
basically) is inserted in the feedback loop. The remaining tuning and resonance error␣
↪are
corrected with polynomials. This approach depends on using at least 2X oversampling -␣
↪the
```

(continues on next page)

(continued from previous page)

```
response after nyquist/2 is abysmal but that's taken care of by the oversampling.

Victor Lazzarini has implemented my model in CSound:
http://www.csounds.com/udo/displayOpcode.php?opcode_id=32

In summary: You can use various methods, but you will need some numerically derived
correction to realize exact tuning and resonance control. If you can afford 2X
oversampling, use Victor's CSound code - the tuning has been tested to be very close
ideal.

Ps. Remember to use real oversampling instead of the "double sampling" the CSound code
uses."

I did not implemented real oversampling, but i inserted additional noise, which
simulates
the resistance noise and also avoids denormal problems...
```

Listing 56: code

```
http://www.savioursofsoul.de/Christian/MoogFilter.pas
```

3.34.1 Comments

- **Date**: 2005-04-27 18:08:06
- **By**: ed.luosfosruoivas@naitsirhC

```
You can also listen to it (Windows-VST) here: http://www.savioursofsoul.de/Christian/
VST/MoogVST.zip
```

- **Date**: 2005-05-06 21:57:17
- **By**: rlindner@gmx ..dot.. net

```
and here is the same thing written in C. It was written while translating the CSound
Code into code for the synthmaker code module as an intermediate step to enable
debugging thru gdb. The code was written to be easy adoptable for the synthmaker
code module (funny defines, static vars, single sample tick function,...) Has some
room for improvements, but nothing fancy for seasoned C programmers.




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


#define polyin  float
#define polyout float

#define BUFSIZE 64

float delta_func [BUFSIZE];
float out_buffer [BUFSIZE];
```

(continues on next page)

(continued from previous page)

```
void tick ( float in, float cf, float reso, float *out ) {


// start of sm code


// filter based on the text "Non linear digital implementation of the moog ladder␣
↪filter" by Antti Houvilainen
// adopted from Csound code at http://www.kunstmusik.com/udo/cache/moogladder.udo
polyin input;
polyin cutoff;
polyin resonance;

polyout sigout;


// remove this line in sm
input = in; cutoff = cf; resonance = reso;


// resonance [0..1]
// cutoff from 0 (0Hz) to 1 (nyquist)

float pi; pi = 3.1415926535;
float v2; v2 = 40000;   // twice the 'thermal voltage of a transistor'
float sr; sr = 22100;

float  cutoff_hz;
cutoff_hz = cutoff * sr;

static float az1;
static float az2;
static float az3;
static float az4;
static float az5;
static float ay1;
static float ay2;
static float ay3;
static float ay4;
static float amf;



float x;         // temp var: input for taylor approximations
float xabs;
float exp_out;
float tanh1_out, tanh2_out;
float kfc;
float kf;
float kfcr;
float kacr;
```

(continues on next page)

(continued from previous page)

```
float k2vg;

kfc  = cutoff_hz/sr; // sr is half the actual filter sampling rate
kf   = cutoff_hz/(sr*2);
// frequency & amplitude correction
kfcr = 1.8730*(kfc*kfc*kfc) + 0.4955*(kfc*kfc) - 0.6490*kfc + 0.9988;
kacr = -3.9364*(kfc*kfc)    + 1.8409*kfc       + 0.9968;

x  = -2.0 * pi * kfcr * kf;
exp_out  = expf(x);

k2vg = v2*(1-exp_out); // filter tuning


// cascade of 4 1st order sections
float x1 = (input - 4*resonance*amf*kacr) / v2;
float tanh1 = tanhf (x1);
float x2 = az1/v2;
float tanh2 = tanhf (x2);
ay1 = az1 + k2vg * ( tanh1 - tanh2);

// ay1  = az1 + k2vg * ( tanh( (input - 4*resonance*amf*kacr) / v2) - tanh(az1/v2) );
az1  = ay1;

ay2  = az2 + k2vg * ( tanh(ay1/v2) - tanh(az2/v2) );
az2  = ay2;

ay3  = az3 + k2vg * ( tanh(ay2/v2) - tanh(az3/v2) );
az3  = ay3;

ay4  = az4 + k2vg * ( tanh(ay3/v2) - tanh(az4/v2) );
az4  = ay4;

// 1/2-sample delay for phase compensation
amf  = (ay4+az5)*0.5;
az5  = ay4;



// oversampling (repeat same block)
ay1  = az1 + k2vg * ( tanh( (input - 4*resonance*amf*kacr) / v2) - tanh(az1/v2) );
az1  = ay1;

ay2  = az2 + k2vg * ( tanh(ay1/v2) - tanh(az2/v2) );
az2  = ay2;

ay3  = az3 + k2vg * ( tanh(ay2/v2) - tanh(az3/v2) );
az3  = ay3;

ay4  = az4 + k2vg * ( tanh(ay3/v2) - tanh(az4/v2) );
az4  = ay4;

// 1/2-sample delay for phase compensation
amf  = (ay4+az5)*0.5;
az5  = ay4;
```

(continues on next page)

(continued from previous page)

```
sigout = amf;



// end of sm code


*out   = sigout;

} // tick


int main ( int argc, char *argv[] )  {

    // set delta function
    memset ( delta_func, 0, sizeof(delta_func));
    delta_func[0] = 1.0;

    int i = 0;
    for ( i = 0; i < BUFSIZE; i++ ) {
            tick ( delta_func[i], 0.6, 0.7, out_buffer+i );
    }
    for ( i = 0; i < BUFSIZE; i++ ) {
            printf ("%f;", out_buffer[i] );
    }
    printf ( "\n" );


} // main
```

- **Date**: 2005-05-07 03:56:13
- **By**: eb.tenyks@didid

```
I think that a better speed optimization of Tanh2 would be to extract the sign bit␣
↪(using integer) instead of abs, and add it back to the final result, to avoid FABS,␣
↪FCOMP and the branching
```

- **Date**: 2005-05-08 19:03:40
- **By**: ed.luosfosruoivas@naitsirhC

```
After reading some more assembler documents for university, i had the same idea...
Now: Coding!
```

- **Date**: 2005-05-08 22:42:55
- **By**: eb.tenyks@didid

```
Btw1, is the idea to get rid of the "*0.5" in the "1/2 sample delay" block by using␣
↪*2 instead of *4 in the first filter?

Btw2, following the same simplification, you can also precompute "-2*fQ*fAcr" outside.
```

- **Date**: 2005-05-09 01:49:49
- **By**: eb.tenyks@didid

```
Forget the sign bit thing, actually your Tanh could already have been much faster at
↪the source:

 a:=f_abs(x);
 a:=a*(6+a*(3+a));
 if (x<0)
  then Result:=-a/(a+12)
  else Result:= a/(a+12);

can be written as the much simpler:

Result:=x*(6+Abs(x)*(3+Abs(x))/(Abs(x)+12)

..so in asm:

function Tanh2(x:Single):Single;
const c3 :Single=3;
      c6 :Single=6;
      c12:Single=12;
Asm
        FLD     x
        FLD     ST(0)
        FABS
        FLD     c3
        FADD    ST(0),ST(1)
        FMUL    ST(0),ST(1)
        FADD    c6
        FMUL    ST(0),ST(2)
        FXCH    ST(1)
        FADD    c12
        FDIVP   ST(1),ST(0)
        FSTP    ST(1)
End;


..but it won't be much faster than your code, because:
-of the slow FDIV
-it's still a function call. Since our dumb Delphi doesn't support assembler macro's,
↪you waste a lot in the function call. You can still try to inline a plain pascal
↪code, but since our dumb Delphi isn't good at compiling float code neither..

Solutions:
-a lookup table for the TanH
-you write the filter processing in asm as well, and you put the TanH code in a
↪separate file (without the header, and assuming in & out are ST(0)). You then $I to
↪insert that file when the function call is needed. Poorman's macro's in Delphi :)

Still, that's a lot of FDIV for a filter..
```

- **Date**: 2005-05-09 03:24:44
- **By**: eb.tenyks@didid

```
forget it, I was all wrong :)
gonna re-post a working version later

still I think that most of the CPU will always be wasted by the division.
```

- **Date**: 2005-05-09 03:48:33
- **By**: eb.tenyks@didid

```
Ignore the above, here it is working (for this code, assuming a premultiplied x):


function Tanh2(x:Single):Single;
const c3  :Single=3;
      c6  :Single=6;
      c12 :Single=12;
Asm
        FLD     x
        FLD     ST(0)
        FABS                 // a
        FLD     c3
        FADD    ST(0),ST(1)
        FMUL    ST(0),ST(1)
        FADD    c6           // b
        FMUL    ST(2),ST(0)  // x*b
        FMULP   ST(1),ST(0)  // a*b
        FADD    c12
        FDIVP   ST(1),ST(0)
End;
```

- **Date**: 2005-05-09 07:59:40
- **By**: ed.luosfosruoivas@naitsirhC

```
That code is nice and very fast! But it's not that accurate. But indeed very fast!␣
↪Thanks for the code. My approach was a lot slower.
```

- **Date**: 2005-05-09 15:05:28
- **By**: eb.tenyks@didid

```
But it should be as accurate as the one in your pascal code: it's the same␣
↪approximation as Tanh2_pas2. Of course we're still talking about a Tanh␣
↪approximation. You can still take it up to /24 for cheap.
Of course, don't try the first one I posted, it's completely wrong.

Btw it's also more accurate than pascal code, since it keeps values in the 80bit FPU␣
↪registers, while the Delphi compiler will put them back to the 32bit variables in-
↪between.

Btw2 I implemented the {$I macro.inc} trick, works pretty well. The whole thing␣
↪speeded up the code by almost 2. For more you could 3DNow 2 Tanh at once, it'd be␣
↪easy in that filter.
```

- **Date**: 2005-05-09 21:20:06
- **By**: ed.luosfosruoivas@naitsirhC

```
OK, it's indeed my tanh2_pas2 approximation, but i've plotted both functions once and␣
↪i found out, that the /24 version is much more accurate and that it is worth to␣
↪calculate the additional macc operation.

But of course the assembler version is much more accurate indeed.

```

(continues on next page)

(continued from previous page)

```
After assembler optimization, i could only speedup the whole thing to a factor of 1.5␣
↪the speed (measured with an impulse) and up to a factor of 1.7 measured with noise␣
↪and the initial version with the branch.

I will now do the 3DNow/SSE optimisation, let's see how it can be speeded up further␣
↪more...
```

- **Date**: 2005-05-11 04:19:47
- **By**: eb.tenyks@didid

```
something bugs me about this filter.. I was assuming that it was made for a standard -
↪1..1 normalized input. But looking at the 1/40000 drive gain, isn't it made for a -
↪32768..32767 input? Otherwise I don't see what the Tanh drive is doing, it's␣
↪basically linear for such low values, and I can't hear any difference with or␣
↪without it.
```

- **Date**: 2005-05-11 11:00:39
- **By**: ed.luosfosruoivas@naitsirhC

```
Usually the tanh component wasn't a desired feature in the analog filter design. They␣
↪trie to keep the input of a differential amplifier very low to retain the linearity.
I have uploaded some plots from my VST Plugin Analyser Pro:
http://www.savioursofsoul.de/Christian/VST/filter4.png (with -1..+1)
http://www.savioursofsoul.de/Christian/VST/filter5.png (with -32768..+32767)

http://www.savioursofsoul.de/Christian/VST/
filter1.png (other...)
http://www.savioursofsoul.de/Christian/VST/filter2.png (other...)
http://www.savioursofsoul.de/Christian/VST/filter3.png (other...)

Additionally i have updated the VST with a new slider for a gain multiplication␣
↪(http://www.savioursofsoul.de/Christian/VST/MoogVST.zip)
```

- **Date**: 2005-11-28 10:09:19
- **By**: moc.liamg@dwod_niatnif

```
There is an error in the C implementation above,
sr = 44100Hz, half the rate of the filter which is oversampled at a rate of 88200Hz.␣
↪So the 22100 needs to be changed.
Christians MoogFilter.pas implements it correctly.
```

3.35 Moog VCF

- **Author or source:** CSound source code, Stilson/Smith CCRMA paper.
- **Type:** 24db resonant lowpass
- **Created:** 2002-01-17 02:02:40

Listing 57: notes

```
Digital approximation of Moog VCF. Fairly easy to calculate coefficients, fairly easy to
process algorithm, good sound.
```

Listing 58: 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;
```

3.35.1 Comments

- **Date**: 2004-10-14 10:18:59
- **By**: ed.xmg@resiaknegah

```
I guess the input is supposed to be between -1..1
```

- **Date**: 2007-02-05 00:30:04
- **By**: moc.erehwon@ydobon

```
Still working on this one. Anyone notice it's got a few syntax problems? Makes me wonder if it's even been tried.

Missing parenthesis. Uses ^ twice. If I get it to work, I'll post usable code.
```

- **Date**: 2007-02-05 17:21:53
- **By**: moc.erehwon@tsaot

```
                        OK. Can't guarantee this is a 100% translation to real C++,␣
↪but it does work. Aside from possible mistakes, I mucked hard with the coefficient␣
↪translation, so that there is a slight difference in the numbers.

1. What kind of filter ends up with exp() in the coefficient calculation instead of␣
↪the usual sin(), cos(), tan() transcendentals? If someone can explain where the e␣
↪to the x comes from, I'd appreciate it.

2. Since I didn't understand the origin of the coefficients, I saw the whole section␣
↪as an excercise in algebra. There were some superfluous multiplications and␣
↪additions.

First I implemented the e to the x with pow(). That was stupid. I switched to exp().␣
↪Hated that too. Checked out the range of inputs and decided on a common␣
↪approximation for exp().

I think it's now one of the fastest filter coefficient calcs you'll see for a 4 pole.␣
↪Go ahead--put the cutoff on an envelope and the q on an LFO. Hell, FM the Q if you␣
↪want!

A pretty good filter. Watch the res (aka Q). Above 9, squeals like a pig. Not in a␣
↪good way.

Needs more work, i think, to stand up with the better LPs in the real VST world. But␣
↪an awfully cool starting place.

If I did something awful here in the translation, or if you have a question about how␣
↪to use it, best to ask me (mistertoast) over at KvRaudio.com in the dev section.

I'm toying with the idea of wedging this into TobyBear's filter designer as a flt␣
↪file. If you manage first, let me know. I'm mistertoast.
```

- **Date**: 2007-02-05 17:22:24
- **By**: moc.erehwon@tsaot

```
MoogFilter.h:

class MoogFilter
{
public:
MoogFilter();
void init();
void calc();
float process(float x);
~MoogFilter();
float getCutoff();
void setCutoff(float c);
float getRes();
void setRes(float r);
protected:
float cutoff;
float res;
float fs;
float y1,y2,y3,y4;
```

(continues on next page)

(continued from previous page)

```
float oldx;
float oldy1,oldy2,oldy3;
float x;
float r;
float p;
float k;
};
```

- **Date**: 2007-02-05 17:22:43
- **By**: moc.erehwon@tsaot

```
MoogFilter.cpp:

#include "MoogFilter.h"

MoogFilter::MoogFilter()
{
fs=44100.0;

init();
}

MoogFilter::~MoogFilter()
{
}

void MoogFilter::init()
{
// initialize values
y1=y2=y3=y4=oldx=oldy1=oldy2=oldy3=0;
calc();
};

void MoogFilter::calc()
{
float f = (cutoff+cutoff) / fs; //[0 - 1]
p=f*(1.8f-0.8f*f);
k=p+p-1.f;

float t=(1.f-p)*1.386249f;
float t2=12.f+t*t;
r = res*(t2+6.f*t)/(t2-6.f*t);
};

float MoogFilter::process(float input)
{
// process input
x = input - r*y4;

//Four cascaded onepole filters (bilinear transform)
y1= x*p +  oldx*p - k*y1;
y2=y1*p + oldy1*p - k*y2;
y3=y2*p + oldy2*p - k*y3;
y4=y3*p + oldy3*p - k*y4;

//Clipper band limited sigmoid
```

(continues on next page)

(continued from previous page)

```
y4-=(y4*y4*y4)/6.f;

oldx = x; oldy1 = y1; oldy2 = y2; oldy3 = y3;
return y4;
}

float MoogFilter::getCutoff()
{ return cutoff; }

void MoogFilter::setCutoff(float c)
{ cutoff=c; calc(); }

float MoogFilter::getRes()
{ return res; }

void MoogFilter::setRes(float r)
{ res=r; calc(); }
```

- **Date**: 2007-02-05 22:04:51
- **By**: moc.dniftnacuoyerehwemos@tsaot

```
I see where the exp() comes from. It just models the resonance. I think it needs more␣
↪work. At high frequencies it goes into self-oscillation much more quickly than at␣
↪low frequencies.
```

- **Date**: 2012-01-31 12:29:16
- **By**: moc.llun@ved

```
A much better tuning seems to be
k=2*sin(f*pi/2)-1;

(within a 0.1 cent up to 4kHz, at 44.1kHz sample rate)
```

- **Date**: 2014-03-26 21:16:08
- **By**: ten.eroctsid@alub

```
This filter works and sounds fine in my VST.
I've re-written the code using templates, which makes life easier when switching␣
↪between <float> and <double> implementation.



#pragma once

namespace DistoCore
{
  template<class T>
  class MoogFilter
  {
  public:
    MoogFilter();
    ~MoogFilter() {};

    T getSampleRate() const { return sampleRate; }
    void setSampleRate(T fs) { sampleRate = fs; calc(); }
```

(continues on next page)

(continued from previous page)

```
    T getResonance() const { return resonance; }
    void setResonance(T filterRezo) { resonance = filterRezo; calc(); }
    T getCutoff() const { return cutoff; }
    T getCutoffHz() const { return cutoff * sampleRate * 0.5; }
    void setCutoff(T filterCutoff) { cutoff = filterCutoff; calc(); }

    void init();
    void calc();
    T process(T input);
    // filter an input sample using normalized params
    T filter(T input, T cutoff, T resonance);

  protected:
    // cutoff and resonance [0 - 1]
    T cutoff;
    T resonance;
    T sampleRate;
    T fs;
    T y1,y2,y3,y4;
    T oldx;
    T oldy1,oldy2,oldy3;
    T x;
    T r;
    T p;
    T k;
  };

    /**
     * Construct Moog-filter.
     */
  template<class T>
  MoogFilter<T>::MoogFilter()
  : sampleRate(T(44100.0))
  , cutoff(T(1.0))
  , resonance(T(0.0))
  {
    init();
  }

    /**
     * Initialize filter buffers.
     */
  template<class T>
  void MoogFilter<T>::init()
  {
    // initialize values
    y1=y2=y3=y4=oldx=oldy1=oldy2=oldy3=T(0.0);
    calc();
  }

    /**
     * Calculate coefficients.
     */
  template<class T>
  void MoogFilter<T>::calc()
  {
    // TODO: replace with your constant
```

(continues on next page)

(continued from previous page)

```
    const double kPi = 3.1415926535897931;

    // empirical tuning
    p = cutoff * (T(1.8) - T(0.8) * cutoff);
    // k = p + p - T(1.0);
    // A much better tuning seems to be:
    k = T(2.0) * sin(cutoff * kPi * T(0.5)) - T(1.0);

    T t1 = (T(1.0) - p) * T(1.386249);
    T t2 = T(12.0) + t1 * t1;
    r = resonance * (t2 + T(6.0) * t1) / (t2 - T(6.0) * t1);
  };

    /**
     * Process single sample.
     */
  template<class T>
  T MoogFilter<T>::process(T input)
  {
    // process input
    x = input - r * y4;

    // four cascaded one-pole filters (bilinear transform)
    y1 =  x * p + oldx  * p - k * y1;
    y2 = y1 * p + oldy1 * p - k * y2;
    y3 = y2 * p + oldy2 * p - k * y3;
    y4 = y3 * p + oldy3 * p - k * y4;

    // clipper band limited sigmoid
    y4 -= (y4 * y4 * y4) / T(6.0);

    oldx = x; oldy1 = y1; oldy2 = y2; oldy3 = y3;

    return y4;
  }

    /**
     * Filter single sample using specified params.
     */
  template<class T>
  T MoogFilter<T>::filter(T input, T filterCutoff, T filterRezo)
  {
    // set params first
    cutoff = filterCutoff;
    resonance = filterRezo;
    calc();

    return process(input);
  }
}
```

- **Date**: 2014-05-27 19:30:12
- **By**: ten.enignepot@derf

```
Why samplerate is missing in calc function?
```

- **Date**: 2015-11-12 08:14:23

- **By**: moc.liamg@neslintreborkram

```
This code works great. I already had a filter in my program so sticking this in only␣
↪took about an hour. The sound is just what I hoped it would be. I started putting␣
↪some really gnarly low frequency  square waves through it and it returns that super␣
↪chewy sound you can get from a Moog. Thanks for this.
```

- **Date**: 2016-06-23 02:27:27
- **By**: moc.poon@poon

```
This filter works just fine and sweeps just fine almost all the way down, cutoff␣
↪easily goes below 40 Hz without issue.  I implemented it in Numerical Python␣
↪(Jupyter notebook) just to be able to see it working and fiddle with it (renders␣
↪sound to .wav in non realtime of course).

Cutoff sweep sounds good enough, a bit on the clean side compared to the analog Moog␣
↪filters I've heard.  It starts to ring around res=0.7 and self oscillates fine for␣
↪almost any res above 1.0.   I find this filter's ringing sounds a bit tinny, thin␣
↪and rather irritating though.  This seems to be a common problem in DSP filters. ␣
↪It needs something, a je ne sais quoi of some kind.

Any suggestions how to modify it?
```

3.36 Moog VCF, variation 1

- **Author or source:** CSound source code, Stilson/Smith CCRMA paper., Paul Kellett version
- **Type:** 24db resonant lowpass
- **Created:** 2002-01-17 02:03:52

Listing 59: 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.
```

Listing 60: 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]

```

(continues on next page)

(continued from previous page)

```
  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);
```

3.36.1 Comments

- **Date**: 2005-01-06 00:57:41
- **By**: ten.xmg@42nitram.leinad

```
I just tried the filter code and it seems like the highpass output is the same as the lowpass output, or at least another lowpass...

But i´m still testing the filter code...
```

- **Date**: 2005-01-06 01:30:37
- **By**: ten.xmg@42nitram.leinad

```
Sorry for the Confusion, it works....
I just had a typo in my code.

One thing i did to get the HP sound nicer was

HP output: (in - 3.0f * (b3 - b4))-b4

But I´m a newbie to DSP Filters...
```

- **Date**: 2005-08-19 00:17:24
- **By**: ten.epacsten@0002skcuswodniw

```
Hey, thanks for this code. I'm a bit confused as to the range to the frequency and resonance. Is it really 0.0-1.0? If so, how so I specify a certain frequency, such as... 400Hz? THANKS!
```

- **Date**: 2005-08-20 13:28:54
- **By**: ed.luosfosruoivas@naitsirhC

```
frequency * nyquist
or
frequency * samplerate

don't know the exact implementation.
```

- **Date**: 2007-02-05 18:28:48
- **By**: moc.erehwon@ydobon

```
>>Hey, thanks for this code. I'm a bit confused as to the range to the frequency and␣
↪resonance. Is it really 0.0-1.0? If so, how so I specify a certain frequency, such␣
↪as... 400Hz? THANKS!

I'd guess it would be:

frequency/(samplerate/2.f)
```

- **Date**: 2009-10-23 19:05:04
- **By**: moc.erehwon@yugsiht

```
The domain seems to be 0-nyquest (samplerate/2.0), but the range is 0-1

A better way to get smoother non-linear mapping of frequency would be this:
(give you a range of 20Hz to 20kHz)

freqinhz = 20.f * 1000.f ^ range;

then

frequency = freqinhz * (1.f/(samplerate/2.0f));
```

- **Date**: 2012-02-24 13:45:09
- **By**: ed.redienhcsslin@psdcisum

```
I like the sound of this one, unfortunately, it breaks quite fast, causing the␣
↪internal values b1-b4 to be "infinity". Any hints?
```

3.37 Moog VCF, variation 2

- **Author or source:** CSound source code, Stilson/Smith CCRMA paper., Timo Tossavainen (?) version
- **Type:** 24db resonant lowpass
- **Created:** 2002-01-17 02:04:57

Listing 61: 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]
```

Listing 62: 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;
}
```

3.37.1 Comments

- **Date**: 2003-03-29 21:50:47
- **By**: rf.oohay@elahwyksa

```
This one works pretty well, thanks !
```

- **Date**: 2003-11-08 06:14:19
- **By**: moc.liamtoh@serpudr

```
could somebody explain, what means this


input -= out4 * fb;
input *= 0.35013 * (f*f)*(f*f);

is "input-" and "input *"  the name of an variable ??
or is this an Csound specific parameter ??
I want to translate this piece to Assemblercode
Robert Dupres
```

- **Date**: 2003-11-11 11:05:42
- **By**: ten.bjc.fieltednabpop@cnahoj

```
input is name of a variable with type double.

input -= out4 * fb;

is just a shorter way for writing:

input = input - out4 * fb;

and the *= operator is works similar:

```

(continues on next page)

(continued from previous page)

```
input *= 0.35013 * (f*f)*(f*f);

is equal to

input = input * 0.35013 * (f*f)*(f*f);

/ Johan
```

- **Date**: 2004-07-12 22:11:20
- **By**: ude.drofnats.amrcc@lfd

```
I've found this filter is unstable at low frequencies, namely when changing quickly␣
↪from high to low frequencies...
```

- **Date**: 2004-07-17 13:39:23
- **By**: moc.kisuw@kmailliw

```
I'm trying to double-sample this filter, like the Variable-State one. But so far no␣
↪success, any tips?

Wk
```

- **Date**: 2004-08-25 08:51:08
- **By**: ten.enegatum@liam

```
What do you mean no success?  What happens?  Have you tried doing the usual␣
↪oversampling tricks (sinc/hermite/mix-with-zeros-and-filter), call the moogVCF␣
↪twice (with fc = fc*0.5) and then filter and decimate afterwards?

I'm been trying to find a good waveshaper to put in the feedback path but haven't␣
↪found a good sounding stable one yet.  I had one version of the filter that tracked␣
↪the envelope of out4 and used it to control the degree to which values below some␣
↪threshold (say 0.08) would get squashed towards zero.  That sounded ok (actually␣
↪quite good for very high inputs), but wasn't entirely stable and was glitching for␣
↪low frequencies.  Then I tried a *out4 = (1+d)* (*out4)/(1 + d* (*out4)) waveshaper,
↪ but that just aliased horribly and made the filter sound mushy and noisy.

Plain old polynomial (x = x-x*x*x) saturation sounds dull.  There must be something␣
↪better out there, though...  and I'd much prefer not to have to oversample to get␣
↪it, though I guess that might be unavoidable.
```

- **Date**: 2006-01-30 15:52:54
- **By**: moc.liamg@fIezees

```
Excuse me but just a basic question from a young
developper
in line  " input -= out4 * fb;  "
i don't understand when and how "out4" is initialised
is it the "out4" return by the previous execution?
which initialisation for the first execution?
```

- **Date**: 2006-01-31 17:15:24
- **By**: musicdsp@[remove this]dsparsons.co.uk

```
all the outs should be initialised to zero, so first time around, nothing is␣
↪subtracted. However, thereafter, the previous output is multiplied and subtracted␣
↪from the input.

HTH
```

- **Date**: 2009-11-10 16:02:55
- **By**: moc.liamg@gulcidrab

```
YAND (Yet Another Newbie Developer) here -

This filter sounds good, and with the addition of a 2nd harmonic waveshaper in the␣
↪feedback loop, it sounds VERY good.

I was hoping I could make it into a HP filter through the normal return in-out4 - but␣
↪that strategy doesn't work for this method.  I'm afraid I'm at a loss as to what to␣
↪try next - anyone have a suggestion?

--Coz
```

- **Date**: 2010-01-08 19:32:16
- **By**: http://www.myspace.com/paradoxuncreated

```
You have to subract each filter, from the input in the cascade.

Check also the Karlsen filters, which I made a few years ago, when going through this␣
↪stage in DSP.
```

- **Date**: 2012-03-02 12:05:25
- **By**: moc.llun@ved

```
The best sounding LP i've found here. Any suggestions how to extract HP/BP?

in - out4 doesn't work, as stated above, but "You have to subtract each filter, fron␣
↪the input in the cascade", what does this mean?

in - out4 - out3 - out2 - out1 doesn't work either
```

3.38 Notch filter

- **Author or source:** Olli Niemitalo
- **Type:** 2 poles 2 zeros IIR
- **Created:** 2002-01-17 02:11:07

Listing 63: 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.
```

Listing 64: 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;
}
```

3.38.1 Comments

- **Date**: 2006-09-27 09:53:49
- **By**: moc.liamtoh@18recnuor

```
i tried implementing it, failed,
and its wierd how it looks further in time
instead of backwards, you cant use it in
```

(continues on next page)

(continued from previous page)

```
a running rendered stream cause the end of
it stuffs up....
```

- **Date**: 2006-09-27 11:43:49
- **By**: musicdsp@Nospam dsparsons.co.uk

```
I think it's a type, and should be
=-=-=
reg0 = a0a2 * ((double)fromstream[streamofs]
+ fromstream[streamofs-2])
+ a1 * fromstream[streamofs-1]
- b1 * reg1
- b2 * reg2;
=-=-=
In which case there is some rangechecking to be done when streamofs<2

You could just use the coeffs, and stick them into whatever biquad code you have␣
↪hanging around :)

DSP
```

- **Date**: 2007-02-17 10:44:00
- **By**: moc.loa@561kluafla

```
Still doesn't work. Either way its looking one each side of a centre value, so it␣
↪doesn't change the maths. Not sure what the code should be. Ideas???
```

- **Date**: 2009-04-24 16:34:32
- **By**: moc.liamg@naecohsife

```
does this code work with float output?
I really need a notch filter, i will try the code.
```

- **Date**: 2011-05-12 11:22:57
- **By**: moc.oohay@skinyela

```
yes, there are some errors in this source.

Here is the correct version:


// (C) Sergey Aleynik.   aleyniks@yahoo.com

     // BW_Coeff is changing from 0.0 to 1.0 (excluded) and the more the narrow:
     // |   BW_Coeff    |  Real BandWidth (approxim.)   |
     // |    0.975      | 0.00907 * Sampling_Frequency |
     // |    0.95       | 0.01814 * Sampling_Frequency |
     // |    0.9        | 0.03628 * Sampling_Frequency |

void  Notch_Filter_Test(short int *Data_In,
                        short int *Data_Out,
                        long       nData_Length,
                        double     Sampling_Frequency,
                        double     CutOff_Frequency,
```

(continues on next page)

(continued from previous page)

```
                        double     BW_Coeff)
{
  // If wrong parameters:
  if((NULL == Data_In)||(Data_Out)||(nData_Length <= 0))   return;
  if((Sampling_Frequency < 0.0)||(CutOff_Frequency < 0.0)) return;
  if(CutOff_Frequency > (Sampling_Frequency/2))            return;
  if((BW_Coeff <= 0.0)||(BW_Coeff >= 1.0))                 return;

  static const double  pi = 3.141592654;

  // Filter coefficients:
  double z1x = cos(2*pi*CutOff_Frequency/Sampling_Frequency);
  double b0 = (1-BW_Coeff)*(1-BW_Coeff)/(2*(fabs(z1x)+1)) + BW_Coeff;
  double b2 = b0;
  double b1 = -2*z1x*b0;
  double a1 = -2*z1x*BW_Coeff;
  double a2 = BW_Coeff*BW_Coeff;

  // Filter internal vars:
  double y=0,  y1=0, y2=0;
  double x0=0, x1=0, x2=0;

  long i;
  for(i=0; i<nData_Length; i++)
  {
    y  = b0 * x0 + b1 * x1 + b2 * x2 - a1 * y1 - a2 * y2;
    y2 = y1;
    y1 = y;
    x2 = x1;
    x1 = x0;
    x0 = Data_In[i];

    if(      y >  32767) y =  32767;
    else if (y < -32768) y = -32768;

    Data_Out[i] = (short int)y;
  }
}
```

- **Date**: 2011-11-11 03:30:26
- **By**: rf.liamtoh@abe.yrduabreivilo

```
Just an idea on coef notch :
 a0 = 1;
 a1 = -2 * cs;
a2 = 1;
b0 = 1 + alpha;
b1 = -2 * cs;
b2 = 1 - alpha;
 notch: H(s) = (s^2 + 1) / (s^2 + s/Q + 1);
 omega = 2*PI*cf/sample_rate;
sn = sin(omega);
cs = cos(omega);
alpha = sn/(2*Q);
```

3.39 One pole LP and HP

- **Author or source:** Bram
- **Created:** 2002-08-26 23:33:27

Listing 65: 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
```

3.39.1 Comments

- **Date**: 2006-03-23 15:39:07
- **By**: moc.liamtoh@wta_sohpyks

```
coefficient: p = (2-cos(x)) - sqrt((2-cos(x))^2 - 1) with x = 2*pi*cutoff/samplerate

p is always -1 using the formula above. The square eliminates the squareroot and (2-
↪cos(x)) - (2-cos(x)) is 0.
```

- **Date**: 2006-03-24 09:37:19
- **By**: q@q

```
Look again. The -1 is inside the sqrt.
```

- **Date**: 2008-08-11 09:34:07
- **By**: batlord[.A.T.]o2[.D.O.T.]pl

```
skyphos:
sqrt((2-cos(x))^2 - 1) doesn't equal
sqrt((2-cos(x))^2) + sqrt(- 1)

so

-1 can be inside the sqrt, because (2-cos(x))^2 will be always >= one.
```

- **Date**: 2009-07-13 01:22:43
- **By**: No

```
HP is wrong!
Or at least it does not work here. It acts like a lofi low-shelf. However this works:

HP:
recursion: tmp = (1-p)*in + p*tmp with output = in-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
```

3.40 One pole filter, LP and HP

- **Author or source:** uh.etle.fni@yfoocs
- **Type:** Simple 1 pole LP and HP filter
- **Created:** 2006-10-08 14:53:38

Listing 66: notes

```
Slope: 6dB/Oct

Reference: www.dspguide.com
```

Listing 67: code

```
Process loop (lowpass):
out = a0*in - b1*tmp;
tmp = out;

Simple HP version: subtract lowpass output from the input (has strange behaviour
↪towards nyquist):
out = a0*in - b1*tmp;
tmp = out;
hp = in-out;

Coefficient calculation:
x = exp(-2.0*pi*freq/samplerate);
a0 = 1.0-x;
b1 = -x;
```

3.40.1 Comments

- **Date**: 2007-01-05 11:43:21
- **By**: moc.liamtoh@ojer_jd

```
Why don't you just say:

Process loop (lowpass):
out = a0*in + b1*tmp;
tmp = out;

Simple HP version: subtract lowpass output from the input (has strange behaviour
↪towards nyquist):
out = a0*in + b1*tmp;
tmp = out;
hp = in-out;

Coefficient calculation:
x = exp(-2.0*pi*freq/samplerate);
a0 = 1.0-x;
b1 = x;
```

- **Date**: 2007-01-06 04:12:56
- **By**: uh.etle.fni@yfoocs

```
There's a tradition among digital filter designers that the pole coefficients have a
negative sign. Of course the other one is also valid, and sometimes these notations
are mixed up.

If you're worried about the extra negation operation, then you could say

b1 = -x;
a0 = 1.0+b1;

so that there's no additional operation overhead.

-- peter schoffhauzer
```

- **Date**: 2007-01-06 16:26:27
- **By**: moc.erehwon@ydobon

```
Of course, you don't need tmp.

Process loop (lowpass):
out = a0*in + b1*out;
```

- **Date**: 2007-02-16 19:27:48
- **By**: uh.etle.fni@yfoocs

```
Indeed.
```

- **Date**: 2009-06-18 17:29:20
- **By**: moc.boohay@bob

```
Or...
out += a0 * (in - out);

:)
```

3.41 One pole, one zero LP/HP

- **Author or source:** ti.dniwni@tretsim
- **Created:** 2002-08-26 23:34:48

Listing 68: 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;
}

```

(continues on next page)

(continued from previous page)

```
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;
```

3.41.1 Comments

- **Date**: 2006-01-15 01:12:26
- **By**: ten.tramepyh@renietsretep

```
what is n? lol...sorry but i mean this seriously! ;)
```

- **Date**: 2006-01-16 14:17:35
- **By**: ku.oc.snosrapsd@psdcisum

```
n is the index of sample being considered.

out[] is an array of samples being output, and in[] is the input array. you would
↪construct a loop such that:
[Pseudocode]
loop n{0..numsamples-1}
  out[n] = in[n]*a0 + in[n-1]*a1 + out[n-1]*b1;
end loop;
[/Pseudocode]

You will need some cleverness so that [n-1] doesn't cause an index error when n=0,
↪but I'll leave that to you :)
```

- **Date**: 2006-01-18 09:28:30
- **By**: ten.tramepyh@renietsretep

```
whoops - sorry, of course n = number... stupid me ;)
interesting code, i will see if can adapt that to delphi, shouldn´t be a big deal :)

i assume i dont need to place either setHPF or LPF into the samples loop, just the
↪block itself?
```

- **Date**: 2006-01-23 10:57:26
- **By**: ku.oc.snosrapsd@psdcisum

```
absolutey - set the coefficients outside of the loop. There is the case of changes␣
↪being made whilst the loop is running, depends what platform/host you are writing␣
↪for.

I'm a delphi code as well. Feel free to use my posted address if you need to :) DSP
```

- **Date**: 2006-07-21 09:07:12
- **By**: moc.oohay@keelanej

```
Shouldn't that be float w = 2*PI*fSampling; ???

In which case we can simplify:

void SetLPF(float fCut, float fSampling)
{
a0 = fCut/(fSampling+fCut);
a1 = a0;
b1 = (fSampling-fCut)/(fSampling+fCut);
}

void SetHPF(float fCut, float fSampling)
{
a0 = fSampling/(fSampling+fCut);
a1 = -a0;
b1 = (fSampling-fCut)/(fSampling+fCut);
}

You can keep the norm = 1/(fSampling+fCut) if you like.
```

- **Date**: 2020-05-23
- **By**: JoergBitzer

```
The equation of the original contributor is correct. It is a first order Butterworth-
↪Filter
H(s') = 1/(1+s') and then denormalized s' = s/wcut and transformed by the bilinear␣
↪transform
s = 2f_s (z-1)/(z+1). Only the tan-prewarp is missing for fcut/wcut.
```

3.42 One zero, LP/HP

- **Author or source:** Bram
- **Created:** 2002-08-29 23:18:43

Listing 69: notes

```
LP is only 'valid' for cutoffs > samplerate/4
HP is only 'valid' for cutoffs < samplerate/4
```

Listing 70: code

```
theta = cutoff*2*pi / samplerate

```

(continues on next page)

(continued from previous page)

```
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
```

3.43 One-Liner IIR Filters (1st order)

- **Author or source:** moc.edocseira@sirhc
- **Type:** IIR 1-pole
- **Created:** 2009-01-18 10:53:44

Listing 71: notes

```
Here is a collection of one liner IIR filters.
Each filter has been transformed into a single C++ expression.

The filter parameter is f or g, and the state variable that needs to be kept around
between interations is s.

- Christian
```

Listing 72: code

```
    101 Leaky Integrator

        a0 = 1
        b1 = 1 - f

        out = s += in - f * s;


    102 Basic Lowpass (all-pole)

        A first order lowpass filter, by finite difference appoximation
(differentials --> differences).

            a0 = f
    b1 = 1 - f

            out = s += f * ( in - s );


    103 Lowpass with inverted control

    Same as above, except for different filter parameter is now inverted.
```

(continues on next page)

(continued from previous page)

```
        In this case, g equals the location of the pole.

            a0 = g - 1
    b1 = g

            out = s = in + g * ( s - in );


    104 Lowpass with zero at Nyquist

        A first order lowpass filter, by via the conformal map of the z-plane (0..
↪infinity --> 0..Nyquist).

        a0 = f
        a1 = f
        b1 = 1 - 2 * f

    s = temp + ( out = s + ( temp = f * ( in - s ) ) );


    105 Basic Highpass (DC-blocker)

        Input complement to basic lowpass, yields a finite difference highpass filter.

        a0 = 1 - f
        a1 = f - 1
        b1 = 1 - f

        out = in - ( s += f * ( in - s ) );


    106 Highpass with forced unity gain at Nyquist

        Input complement to filter 104, yields a conformal map highpass filter.

            a0 = 1 - f
            a1 = f - 1
            b1 = 1 - 2 * f

        out = in + temp - ( s += 2 * ( temp = f * ( in - s ) ) );


    107 Basic Allpass

        This corresponds to a first order allpass filter,
        where g is the location of the pole in the range -1..1.

        a0 = -g
        a1 = 1
        b1 = g

    s = in + g * ( out = s - g * in );
```

3.43.1 Comments

- **Date**: 2016-03-31 14:21:04

- **By**: moc.liamg@1ydarbfmot

```
Great help, although could you advise as to where the parameters a0, a1 and b1 are
↪used for the high pass filter 105?

Thanks
```

3.44 Output Limiter using Envelope Follower in C++

- **Author or source:** moc.oohay@nniveht
- **Created:** 2009-04-27 08:46:35

Listing 73: notes

```
Here's a Limiter class that will automatically compress a signal if it would cause
clipping.

You can control the attack and decay parameters of the limiter. The attack determines
↪how
quickly the limiter will respond to a sudden increase in output level. I have found
↪that
attack=10ms and decay=500ms works very well for my application.

This C++ example demonstrates the use of template parameters to allow the same piece
↪of
code to work with either floats or doubles (without needing to make a duplicate of the
code). As well as allowing the same code to work with interleaved audio data (any
↪number
of channels) or linear, via the "skip" parameter. Note that even in this case, the
compiler produces fully optimized output in the case where the template is
↪instantiated
for a compile-time constant value of skip.

In Limiter::Process() you can see the envelope class getting called for one sample,
↪this
shows how even calling a function for a single sample can get fully optimized out by
↪the
compiler if code is structured correctly.

While this is a fairly simple algorithm, I wanted to share the technique for using
template parameters to develop routines that can work with any size floating point
representation or multichannel audio data, while still remaining fully optimized.

These classes were based on ideas found in the musicdsp.org archives.
```

Listing 74: code

```
class EnvelopeFollower
{
public:
    EnvelopeFollower();

    void Setup( double attackMs, double releaseMs, int sampleRate );

```

(continues on next page)

(continued from previous page)

```
    template<class T, int skip>
    void Process( size_t count, const T *src );

    double envelope;

protected:
    double a;
    double r;
};

//----------

inline EnvelopeFollower::EnvelopeFollower()
{
    envelope=0;
}

inline void EnvelopeFollower::Setup( double attackMs, double releaseMs, int sampleRate )
{
    a = pow( 0.01, 1.0 / ( attackMs * sampleRate * 0.001 ) );
    r = pow( 0.01, 1.0 / ( releaseMs * sampleRate * 0.001 ) );
}

template<class T, int skip>
void EnvelopeFollower::Process( size_t count, const T *src )
{
    while( count-- )
    {
            double v=::fabs( *src );
            src+=skip;
            if( v>envelope )
                    envelope = a * ( envelope - v ) + v;
            else
                    envelope = r * ( envelope - v ) + v;
    }
}

//----------

struct Limiter
{
    void Setup( double attackMs, double releaseMs, int sampleRate );

    template<class T, int skip>
    void Process( size_t nSamples, T *dest );

private:
    EnvelopeFollower e;
};

//----------

inline void Limiter::Setup( double attackMs, double releaseMs, int sampleRate )
{
    e.Setup( attackMs, releaseMs, sampleRate );
}
```

(continues on next page)

(continued from previous page)

```

template<class T, int skip>
void Limiter::Process( size_t count, T *dest )
{
    while( count-- )
    {
            T v=*dest;
            // don't worry, this should get optimized
            e.Process<T, skip>( 1, &v );
            if( e.envelope>1 )
                    *dest=*dest/e.envelope;
            dest+=skip;
    }
}
```

3.45 Peak/Notch filter

- **Author or source:** ed.bew@raebybot
- **Type:** peak/notch
- **Created:** 2002-12-16 19:01:28

Listing 75: notes

```
// Peak/Notch filter
// I don't know anymore where this came from, just found it on
// my hard drive :-)
// Seems to be a peak/notch filter with adjustable slope
// steepness, though slope gets rather wide the lower the
// frequency is.
// "cut" and "steep" range is from 0..1
// Try to feed it with white noise, then the peak output does
// rather well eliminate all other frequencies except the given
// frequency in higher frequency ranges.
```

Listing 76: code

```
var f,r:single;
    outp,outp1,outp2:single; // init these with 0!
const p4=1.0e-24; // Pentium 4 denormal problem elimination

function PeakNotch(inp,cut,steep:single;ftype:integer):single;
begin
 r:=steep*0.99609375;
 f:=cos(pi*cut);
 a0:=(1-r)*sqrt(r*(r-4*(f*f)+2)+1);
 b1:=2*f*r;
 b2:=-(r*r);
 outp:=a0*inp+b1*outp1+b2*outp2+p4;
 outp2:=outp1;
 outp1:=outp;
 if ftype=0 then
  result:=outp //peak
```

(continues on next page)

(continued from previous page)

```
 else
  result:=inp-outp; //notch
end;
```

3.45.1 Comments

- **Date**: 2010-03-02 03:19:21
- **By**: slo77y (at) yahoo.de

```
this code sounds bitcrushed like hell translated to c++, any suggestions ?

    float pi = 3.141592654;
    float r = dQFactor*0.99609375;
    float f = cos(pi*iFreq);
    float a0 = (1-r) * sqrt ( r * ( r-4 * ( f * f ) + 2 ) + 1 );
    float b1 = 2 * f * r;
    float b2 = - ( r * r );
    float outp = 0.0, outp1 = 0.0, outp2 = 0.0;

    for (i = 0; i < iSamples; i++)
    {
            float inp = fInput[i];

            outp = a0 * inp + b1 * outp1 + b2 * outp2 + p4;
            outp2 = outp1;
            outp1 = outp;

            fOutput[i] = (inp-outp); //notch
    }
```

- **Date**: 2012-05-05 08:22:08
- **By**: ten.xoc@53namhsima

```
After about 3 hours wondering why I was getting back the original un-altered audio, I
↪finally got this version of a keeper filter, which I used with absurdly good
↪success on a power grid comb filter. When the power grid filter was fed with audio
↪from a lamp cord with one 1 Megohm resistor on each prong, all sorts of cool sounds
↪become audio when the output is amplified 40 dB. For wall cord audio, use 60.0 for
↪the cutoff.
---the function is below---
double keeper_1(double input, double cutoff,double rate,double *magnitude)
{
    const double steepness=1.0;
    const double p4=1.0e-24;
    static unsigned char first=1;
    static double nfreq=0.1;
    static double old_cutoff=0.0;
    static double the_magnitude=0;
    static double average=0.0;
    static int average_count=0;
    static double a=0.0;
    static double r=0.0;
    static double coeff=0.0;
    static double delay[3]={0.0,0.0,0.0};
```

(continues on next page)

(continued from previous page)

```
    static double delay1[3]={0.0,0.0,0.0};
    static double delay2[3]={0.0,0.0,0.0};
    static double delay3[3]={0.0,0.0,0.0};
    static double b[3]={0.0,0.0,0.0};
    if(first==1 || cutoff!=old_cutoff )
    {
            r=steepness * 0.99609375;
            nfreq=(cutoff/(double)rate) * 2.0 ;
            coeff= cos( M_PI * nfreq);
            a=(1.0 - r) * sqrt(r * (r - 4 * (coeff * coeff) + 2) +1);
            b[1]=2 * coeff * r;
            b[2]=-(r * r);

            first=0;
    }

    delay3[0] = a * input + b[1] * delay3[1] + b[2] * delay3[2] + p4;

    delay3[2]=delay3[1];
    delay3[1]=delay3[0];


    delay2[0] = a * delay3[0] + b[1] * delay2[1] + b[2] * delay2[2] + p4;

    delay2[2]=delay2[1];
    delay2[1]=delay2[0];



    delay1[0] = a * delay2[0] + b[1] * delay1[1] + b[2] * delay1[2] + p4;

    delay1[2]=delay1[1];
    delay1[1]=delay1[0];


    delay[0] = a * delay1[0] + b[1] * delay[1] + b[2] * delay[2] + p4;

    delay[2]=delay[1];
    delay[1]=delay[0];
    average+=delay[0];
    average_count++;
    if(average_count>dft_size-1)
    {
            double aver=average/(double)dft_size;
            the_magnitude=sqrt(aver * aver); /* we're only interested in the root␣
↪mean square */
            average=0.0;
            average_count=0;
    }
    magnitude[0]=the_magnitude;
    old_cutoff=cutoff;
    return delay[0];
}
```

3.46 Perfect LP4 filter

- **Author or source:** rf.eerf@aipotreza
- **Type:** LP
- **Created:** 2008-03-13 10:40:46

Listing 77: notes

```
hacked from the exemple of user script in FL Edison
```

Listing 78: code

```
TLP24DB = class
constructor create;
procedure process(inp,Frq,Res:single;SR:integer);
private
t, t2, x, f, k, p, r, y1, y2, y3, y4, oldx, oldy1, oldy2, oldy3: Single;
public outlp:single;
end;
----------------------------------------
implementation

constructor TLP24DB.create;
begin
  y1:=0;
  y2:=0;
  y3:=0;
  y4:=0;
  oldx:=0;
  oldy1:=0;
  oldy2:=0;
  oldy3:=0;
end;
procedure TLP24DB.process(inp: Single; Frq: Single; Res: Single; SR: Integer);
begin
  f := (Frq+Frq) / SR;
  p:=f*(1.8-0.8*f);
  k:=p+p-1.0;
  t:=(1.0-p)*1.386249;
  t2:=12.0+t*t;
  r := res*(t2+6.0*t)/(t2-6.0*t);
  x := inp - r*y4;
  y1:=x*p + oldx*p - k*y1;
  y2:=y1*p+oldy1*p - k*y2;
  y3:=y2*p+oldy2*p - k*y3;
  y4:=y3*p+oldy3*p - k*y4;
  y4 := y4 - ((y4*y4*y4)/6.0);
  oldx := x;
  oldy1 := y1+_kd;
  oldy2 := y2+_kd;;
  oldy3 := y3+_kd;;
  outlp := y4;
end;

// the result is in outlp
```

(continues on next page)

(continued from previous page)

```
// 1/ call MyTLP24DB.Process
// 2/then get the result from outlp.
// this filter have a fantastic sound w/a very special res
// _kd is the denormal killer value.
```

3.46.1 Comments

- **Date**: 2008-10-20 07:44:35
- **By**: moc.liamg@321tiloen

```
This is basically a Moog filter.
```

- **Date**: 2010-09-16 23:07:30
- **By**: moc.liamG@0356orbratiug

```
Same as http://www.musicdsp.org/showArchiveComment.php?ArchiveID=24
but seems to be in pascal.
```

3.47 Pink noise filter

- **Author or source:** Paul Kellett
- **Created:** 2002-02-11 17:40:39
- **Linked files:** `pink.txt`.

Listing 79: notes

```
(see linked file)
```

3.47.1 Comments

- **Date**: 2005-02-10 12:23:55
- **By**: ed.ap-ymot@ymoT

```
             Hi, first of all thanks a lot for this parameters.
I'm new to digital fitering, and need a 3dB highpass to correct a pink spectrum which␣
↪is used for measurement back to white for displaying the impulseresponse.
I checked some pages, but all demand a fixed ratio between d0 and d1 for a 6db␣
↪lowpass. But this ratio is not given on your filters, so I'm not able to transform␣
↪them into highpasses.
Any hints?

Tomy
```

- **Date**: 2005-02-10 19:58:16
- **By**: ed.ap-ymot@ymoT

```
Hi, first of all thanks a lot for this parameters. I'm new to digital fitering, and␣
↪need a 3dB highpass to correct a pink spectrum which is used for measurement back␣
↪to white for displaying the impulseresponse.I checked some pages, but all demand a␣
↪fixed ratio between d0 and d1 for a 6db lowpass. But this ratio is not given on␣
↪your filters, so I'm not able to transform them into highpasses.Any hints?Tomy
```

- **Date**: 2005-02-14 14:57:04
- **By**: ed.luosfosruoivas@naitsirhC

```
If computing power doesn't matter, than you may want do design the pink noise in the␣
↪frequency domain and transform it backt to timedomain via fft.
Christian
```

- **Date**: 2005-03-03 16:34:49
- **By**: rf.oohay@dejamdaddah

```
HI, could you please give me a code matlab to have a pink noise. I
tested a code where one did all into frequential mode then made an ifft.
Thank you
```

- **Date**: 2009-03-15 21:56:21
- **By**: moc.liamg@321tiloen

```
Here is a slightly less efficient implementation, which can be used to calculate␣
↪coefficients for different samplerates (in ranges).

Note: You may also want to check the sample-and-hold method.

//trc - test rate coeff, srate - samplerate
trc = 1;
sr = srate*trc;

//f0-f6 - freq array in hz
//
//---------------------
//samplerate <= 48100hz
f0 = 4752.456;
f1 = 4030.961;
f2 = 2784.711;
f3 = 1538.461;
f4 = 357.681;
f5 = 70;
f6 = 30;
//---------------------
//samplerate > 44800hz && samplerate <= 96000hz
f0 = 8227.219;
f1 = 8227.219;
f2 = 6388.570;
f3 = 3302.754;
f4 = 479.412;
f5 = 151.070;
f6 = 54.264;
//---------------------
//samplerate > 96000khz && samplerate < 192000khz
f0 = 9211.912;
```

(continues on next page)

(continued from previous page)

```
f1 = 8621.096;
f2 = 8555.228;
f3 = 8292.754;
f4 = 518.334;
f5 = 163.712;
f6 = 240.241;
//---------------------
//samplerate >= 192000hz
f0 = 10000;
f1 = 10000;
f2 = 10000;
f3 = 10000;
f4 = 544.948;
f5 = 142.088;
f6 = 211.616;

//----------------------
//calculate coefficients
k0 = exp(-2*$pi*f0/sr);
k1 = exp(-2*$pi*f1/sr);
k2 = exp(-2*$pi*f2/sr);
k3 = exp(-2*$pi*f3/sr);
k4 = exp(-2*$pi*f4/sr);
k5 = exp(-2*$pi*f5/sr);
k6 = exp(-2*$pi*f6/sr);

--- sample loop ---
//white - noise input
b0 = k0*white+k0*b0;
b1 = k1*white+k1*b1;
b2 = k2*white+k2*b2;
b3 = k3*white+k3*b3;
b4 = k4*white+k4*b4;
b5 = k5*white+k5*b5;
b6 = k6*white+k6*b6;
pink = (b0+b1+b2+b3+b4+b5+white-b6);

output = pink;
--- sample loop ---

Basically if you use the same coefficients, if comparing some outputs, you would
↪notice a degradation in the filter at higher sample rates - Thus the different
↪ranges. But the quality of your white noise (PRNG) may be important also.

These 'should' work...They do fairly well, at least mathematically for rendered
↪outputs.

Lubomir
```

3.48 Plot Filter (Analyze filter characteristics)

- **Author or source:** ku.oc.oohay@895rennacs
- **Type:** Test
- **Created:** 2007-11-26 14:05:40

Listing 80: notes

```
As a newbe, and one that has very, very little mathematical background, I wanted to␣
↪see
what all the filters posted here were doing to get a feeling of what was going on␣
↪here. So
with what I picked up from this site, I wrote a little filter test program. Hope it␣
↪is of
any use to you.
```

Listing 81: code

```
//
// plotFilter.cpp
//
// Simple test program to plot filter characteristics of a particular
// filter to stdout. Nice to see how the filter behaves under various
// conditions (cutoff/resonance/samplerate/etc.).
//
// Should work on any platform that supports C++ and should work on C
// as well with a little rework. It justs prints text, so no graphical
// stuff is used.
//
// Filter input and filter output should be between -1 and 1 (floating point)
//
// Output is a plotted graph (as text) with a logarithmic scale
// (so half a plotted bar is half of what the human ear can hear).
// If you dont like the vertical output, just print it and turn the paper :-)
//

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

#define myPI 3.1415926535897932384626433832795

#define FP double
#define DWORD unsigned long
#define CUTOFF       5000
#define SAMPLERATE  48000

// take enough samples to test the 20 herz frequency 2 times
#define TESTSAMPLES (SAMPLERATE/20) * 2

//
// Any filter can be tested, as long as it outputs
// between -1 and 1 (floating point). This filter
// can be replaced with any filter you would like
// to test.
//
class Filter {
    FP sdm1;        // sample data minus 1
    FP a0;          // multiply factor on current sample
    FP b1;          // multiply factor on sdm1
public:
    Filter (void) {
```

(continues on next page)

(continued from previous page)

```
                sdm1 = 0;
                // init on no filtering
                a0    = 1;
                b1    = 0;
        }
        void init(FP freq, FP samplerate) {
                FP x;
                x = exp(-2.0 * myPI * freq / samplerate);
                sdm1 = 0;
                a0    = 1.0 - x;
                b1    = -x;
        }
        FP getSample(FP sample) {
                FP out;
                out = (a0 * sample) - (b1 * sdm1);
                sdm1 = out;
                return out;
        }
};

int
main(void)
{
        DWORD  freq;
        DWORD  spos;
        double sIn;
        double sOut;
        double tIn;
        double tOut;
        double dB;
        DWORD  tmp;

        // define the test filter
        Filter filter;

        printf("                    9bB      6dB              3dB                           0dB\n");
        printf(" freq    dB          |        |               |                            |  \n");

        // test frequencies 20 - 20020 with 100 herz steps
        for (freq=20; freq<20020; freq+=100) {

                // (re)initialize the filter
                filter.init(CUTOFF, SAMPLERATE);

                // let the filter do it's thing here
                tIn = tOut = 0;
                for (spos=0; spos<TESTSAMPLES; spos++) {
                        sIn  = sin((2 * myPI * spos * freq) / SAMPLERATE);
                        sOut = filter.getSample(sIn);
                        if ((sOut>1) || (sOut<-1)) {
                                // If filter is no good, stop the test
                                printf("Error! Clipping!\n");
                                return(1);
                        }
                        if (sIn >0) tIn  += sIn;
```

(continues on next page)

(continued from previous page)

```
                    if (sIn <0) tIn  -= sIn;
                    if (sOut>0) tOut += sOut;
                    if (sOut<0) tOut -= sOut;
            }

            // analyse the results
            dB = 20*log(tIn/tOut);
            printf("%5d %5.1f ", freq, dB);
            tmp = (DWORD)(60.0/pow(2, (dB/3)));
            while (tmp) {
                    putchar('#');
                    tmp--;
            }
            putchar('\n');
    }
    return 0;
}
```

3.48.1 Comments

- **Date**: 2008-01-01 20:45:13
- **By**: niels m.

```
You need to change the log() to log10() to get the correct answer in dB's.

You can also replace the if(sIn >0) .. -= sOut; by:

tIn += sIn*sIn;
tOut += sOut*sOut;

this will measure signal power instead of amplitude. If you do this, you also need to␣
↪change 20*log10() to 10*log10().

Nice and useful tool for exploring filters. Thanks!
```

3.49 Plotting R B-J Equalisers in Excel

- **Author or source:** Web Surf
- **Created:** 2006-03-07 09:32:57
- **Linked files:** rbj_eq.xls.

Listing 82: notes

```
Interactive XL sheet that shows frequency response of the R B-K high pass/low pass,
Peaking and Shelf filters

Useful if --
--You want to validate your implementation against mine

--You want to convert given Biquad coefficients into Fo/Q/dBgain by visual curve␣
↪fitting.
```

(continues on next page)

(continued from previous page)

```
--You want the Coefficients to implement a particular fixed filter

-- Educational aid

Many thanks to R B-J for his personal support !!
Web Surf   WebsurffATgmailDOTcom
```

Listing 83: code

```
see attached file
```

3.49.1 Comments

- **Date**: 2006-04-14 07:07:00
- **By**: rf.liamtoh@57eninreS_luaP

```
It also works perfectly with the openoffice2.02 suite :-)
```

- **Date**: 2007-07-24 15:00:09
- **By**: jackmattson att gmial

```
Ok, so I'm about to make my first filter so I really have no idea about coefficients␣
↪yet but damn this is the coolest .xls I've ever seen! And I see a bunch every day.␣
↪:?
```

- **Date**: 2007-09-13 05:54:54
- **By**: WebsurffATgmailDOTcom

```
Thanks,

It's just an implementation of the R B-J cookbook formulae for parametric equalisers.␣
↪RB-J personally assisted me while I was writing this XLS. The real Kudos goes out␣
↪to him !!

I wrote this as in intermediate tool while I was writing some Guitar effect DSP␣
↪routines.

One prob it has : If the Q is too sharp, the peak filters have a very sharp tip. It␣
↪is possible then as you slide the F sliders, that the chosen F is not one of the␣
↪data points that I am plotting.

The net result is that the peak of a hi-Q filter seems to increase/decrease as you␣
↪move F. This is a shortcoming of the way I programmed my XLS and is not a problem␣
↪with the R B-J equations.

PS : I saw the same problem with other similar S/W on the net !!!


PS : Anyone know how to do curve fitting so that we enter in some points through␣
↪which the frequency response must pass, and it generates best set of F,G,Q ?
```

- **Date**: 2011-04-18 02:30:51

- **By**: WebsurffATgmailDOTcom

```
Date: Sun, 17 Apr 2011 12:34:09
Subject: RBJ Filter Plotter Spreadsheet
From: Robert Bonini <XXXXXXXXXX@gmail.com>
To: websurff @ gmail . com


Just wanted to complement you on the good work.  I came across your RBJ filter␣
↪plotter spreadsheet, and it's quite good.

The version I have is dated Feb.2006, so this may have been addressed already but I␣
↪did notice a small issue with the sample frequency input.

Anything other than 44.1 KHz would cause the response curves to peak at incorrect␣
↪values on the x axis.  The normalized frequency 'w' was
being referenced to 44100, and not to the Fs cell. I modified the formula to absolute␣
↪reference B56 and it solved that problem.

Thanks for your great work,
-robert
```

3.50 Polyphase Filters

- **Author or source:** C++ source code by Dave from Muon Software
- **Type:** polyphase filters, used for up and down-sampling
- **Created:** 2002-01-17 02:14:53
- **Linked files:** `BandLimit.cpp`.
- **Linked files:** `BandLimit.h`.

3.50.1 Comments

- **Date**: 2005-02-16 00:14:08
- **By**: ed.bew@ihsugat_aranoias

```
can someone give me a hint for a paper where this stuff is from?
```

- **Date**: 2005-03-29 20:39:59
- **By**: ABC

```
From there: http://www.cmsa.wmin.ac.uk/~artur/Poly.html

There is also this library, implementing the same filter, but optimised for down/
↪upsampling and ported to SSE and 3DNow!:
http://ldesoras.free.fr/prod.html#src_hiir
```

- **Date**: 2005-07-27 09:22:16
- **By**: ku.oc.nez@mahcleb.bor

```
There is an error in the 12th order, steep filter coefficients. Having checked the
↪output against that produced by HIIR (see previous comment), i have identified the
↪source of the error - the 4th b coefficient 0.06329609551399348, should be 0.
↪6329609551399348.
```

- **Date**: 2008-04-06 08:58:52
- **By**: bla

```
you also need to delete the pointers inside the array

CAllPassFilterCascade::~CAllPassFilterCascade()
{
    for (int i=0;i<numfilters;i++)
    {
            delete allpassfilter[i];
    }

    delete[] allpassfilter;
};
```

- **Date**: 2008-11-05 14:50:18
- **By**: moc.tob.3gall1pso1dua@0fn1

```
some questions..  is it normal for these halfband filters to cause significant gain
↪loss?  sonogram analysis shows a decrease in SNR if I have read the results
↪correctly.

if using these filters for oversampling and I do this:

upsample
halfband filter
*process*
halfband filter
decimate (discard samples)

then presumably the second halfband filter does the antialiasing at half the new
↪sampling rate?
```

- **Date**: 2009-02-26 21:39:21
- **By**: moc.toohay@bob

```
Hello, I'm getting the high pass from the function by subtracting the 'oldout'
↪variable.

output=(filter_a->process(input)-oldout)*0.5;

But this does not make an ideal QMF, I'm getting pass-band aliasing, so I guessing
↪the phase is off slightly between the low and high.
Is this the correct way of getting the high band?

Cheers,
Dave P
```

- **Date**: 2010-01-21 19:31:46
- **By**: bobby

```
Is the cutoff at 20kHz?  What sample rate are these coefficients for?  How would I␣
↪calculate suitable coefficients for arbitrary sample rates?
```

- **Date**: 2011-06-11 18:13:28
- **By**: moc.psdallahlav@naes

```
It is worth noting that if these filters are being used for upsampling/downsampling,␣
↪the "noble identity" can be used to reduce the CPU cost. The basic idea is that␣
↪operations that can be expressed in the form:

filter that uses z^-N for its states -> downsample by N

can be rearranged to use the form

downsample by N -> filter that uses z^-1 for its states

The same property holds true for upsampling. See

http://mue.music.miami.edu/thesis/jvandekieft/jvchapter3.htm

for more details.

For the above code, this would entail creating an alternative allpass process␣
↪function, that uses the z^-1 for its states, and then rearranging some of the␣
↪operations.
```

3.51 Polyphase Filters (Delphi)

- **Author or source:** moc.liamtoh@retsbomyrgnayrev
- **Type:** polyphase filters, used for up and down-sampling
- **Created:** 2006-07-05 20:13:50

Listing 84: notes

```
Pascal conversion of C++ code by Dave from Muon Software. Conversion by Shannon␣
↪Faulkner.
```

Listing 85: code

```
{

polyphase filters, used for up and down-sampling
original c++ code by Dave from Muon Software found
at MusicDSP.
rewritten in pascal by Shannon Faulkner, 4/7/06.

}

unit uPolyphase;

interface

```

(continues on next page)

(continued from previous page)

```
type
  TAllPass=class
  private
    a,x0,x1,x2,y0,y1,y2:single;
  public
    constructor create(coefficient:single);
    function Process(input:single):single;
  end;

  TAllPassFilterCascade=class
  private
    AllPassFilters:array of TAllPass;
    fOrder:integer;
  public
    constructor create(coefficients:psingle;Order:integer);
    function Process(input:single):single;
  end;

  THalfBandFilter=class
  private
    fOrder:integer;
    OldOut:single;
    aCoeffs,bCoeffs:array of single;
    FilterA,FilterB:TAllPassFilterCascade;
  public
    constructor create(order:integer;Steep:boolean);
    function process(input:single):single;
  end;


implementation
//-----------     AllPass Filter        -------------------
//-----------     AllPass Filter        ----------------------------------------------
//-----------     AllPass Filter        ----------------------------------------------
constructor TAllPass.create(coefficient:single);
begin
  a:=coefficient;

  x0:=0;
  x1:=0;
  x2:=0;

  y0:=0;
  y1:=0;
  y2:=0;
end;

function TAllPass.Process(input:single):single;
var output:single;
begin
  //shuffle inputs
  x2:=x1;
  x1:=x0;
  x0:=input;

  //shuffle outputs
  y2:=y1;
```

(continues on next page)

(continued from previous page)

```
  y1:=y0;

  //allpass filter 1
  output:=x2+((input-y2)*a);

  y0:=output;

  result:=output;
end;
//------------     AllPass Filter Cascade     -------------
//------------     AllPass Filter Cascade     -------------
//------------     AllPass Filter Cascade     -------------
constructor TAllPassFilterCascade.create(coefficients:psingle;Order:integer);
var i:integer;
begin
  fOrder:=Order;
  setlength(AllPassFilters,fOrder);
  for i:=0 to fOrder-1 do
  begin
    AllPassFilters[i]:=TAllPass.create(coefficients^);
    inc(coefficients);
  end;
end;

function TAllPassFilterCascade.Process(input:single):single;
var
  output:single;
  i:integer;
begin
  output:=input;
  for i:=0 to fOrder-1 do
  begin
    output:=allpassfilters[i].Process(output);
  end;
  result:=output;
end;
//--------------   Halfband Filter    -------------------------------------------
//--------------   Halfband Filter    ---------------------
//--------------   Halfband Filter    -------------------------------------------
constructor THalfBandFilter.create(order:integer;Steep:boolean);
begin
  fOrder:=order;
  setlength(aCoeffs,Order div 2);
  setlength(bCoeffs,Order div 2);

  if steep=true then
  begin
    if (order=12) then //rejection=104dB, transition band=0.01
    begin
      aCoeffs[0]:=0.036681502163648017;
      aCoeffs[1]:=0.2746317593794541;
      aCoeffs[2]:=0.56109896978791948;
      aCoeffs[3]:=0.769741833862266;
      aCoeffs[4]:=0.8922608180038789;
      aCoeffs[5]:=0.962094548378084;

      bCoeffs[0]:=0.13654762463195771;
```

(continues on next page)

(continued from previous page)

```
      bCoeffs[1]:=0.42313861743656667;
      bCoeffs[2]:=0.6775400499741616;
      bCoeffs[3]:=0.839889624849638;
      bCoeffs[4]:=0.9315419599631839;
      bCoeffs[5]:=0.9878163707328971;
    end
    else if (order=10) then //rejection=86dB, transition band=0.01
    begin
      aCoeffs[0]:=0.051457617441190984;
      aCoeffs[1]:=0.35978656070567017;
      aCoeffs[2]:=0.6725475931034693;
      aCoeffs[3]:=0.8590884928249939;
      aCoeffs[4]:=0.9540209867860787;

      bCoeffs[0]:=0.18621906251989334;
      bCoeffs[1]:=0.529951372847964;
      bCoeffs[2]:=0.7810257527489514;
      bCoeffs[3]:=0.9141815687605308;
      bCoeffs[4]:=0.985475023014907;
    end
    else if (order=8) then  //rejection=69dB, transition band=0.01
    begin
      aCoeffs[0]:=0.07711507983241622;
      aCoeffs[1]:=0.4820706250610472;
      aCoeffs[2]:=0.7968204713315797;
      aCoeffs[3]:=0.9412514277740471;

      bCoeffs[0]:=0.2659685265210946;
      bCoeffs[1]:=0.6651041532634957;
      bCoeffs[2]:=0.8841015085506159;
      bCoeffs[3]:=0.9820054141886075;
    end
    else if (order=6) then  //rejection=51dB, transition band=0.01
    begin
      aCoeffs[0]:=0.1271414136264853;
      aCoeffs[1]:=0.6528245886369117;
      aCoeffs[2]:=0.9176942834328115;

      bCoeffs[0]:=0.40056789819445626;
      bCoeffs[1]:=0.8204163891923343;
      bCoeffs[2]:=0.9763114515836773;
    end
    else if (order=4) then  //rejection=53dB,transition band=0.05
    begin
      aCoeffs[0]:=0.12073211751675449;
      aCoeffs[1]:=0.6632020224193995;

      bCoeffs[0]:=0.3903621872345006;
      bCoeffs[1]:=0.890786832653497;
    end
    else    //order=2, rejection=36dB, transition band=0.1
    begin
      aCoeffs[0]:=0.23647102099689224;
      bCoeffs[0]:=0.7145421497126001;
    end;
  end else  //softer slopes, more attenuation and less stopband ripple
  begin
```

(continues on next page)

(continued from previous page)

```
    if (order=12) then //rejection=104dB, transition band=0.01
    begin
      aCoeffs[0]:=0.01677466677723562;
      aCoeffs[1]:=0.13902148819717805;
      aCoeffs[2]:=0.3325011117394731;
      aCoeffs[3]:=0.53766105314488;
      aCoeffs[4]:=0.7214184024215805;
      aCoeffs[5]:=0.8821858402078155;

      bCoeffs[0]:=0.06501319274445962;
      bCoeffs[1]:=0.23094129990840923;
      bCoeffs[2]:=0.4364942348420355;

      //bug fix - coefficient changed,
      //rob[DOT]belcham[AT]zen[DOT]co[DOT]uk
      //bCoeffs[3]:=0.06329609551399348;  //original coefficient
      bCoeffs[3]:=0.6329609551399348;     //correct coefficient

      bCoeffs[4]:=0.80378086794111226;
      bCoeffs[5]:=0.9599687404800694;
    end
    else if (order=10) then //rejection=86dB, transition band=0.01
    begin
      aCoeffs[0]:=0.02366831419883467;
      aCoeffs[1]:=0.18989476227180174;
      aCoeffs[2]:=0.43157318062118555;
      aCoeffs[3]:=0.6632020224193995;
      aCoeffs[4]:=0.860015542499582;

      bCoeffs[0]:=0.09056555904993387;
      bCoeffs[1]:=0.3078575723749043;
      bCoeffs[2]:=0.5516782402507934;
      bCoeffs[3]:=0.7652146863779808;
      bCoeffs[4]:=0.95247728378667541;
    end
    else if (order=8) then  //rejection=69dB, transition band=0.01
    begin
      aCoeffs[0]:=0.03583278843106211;
      aCoeffs[1]:=0.2720401433964576;
      aCoeffs[2]:=0.5720571972357003;
      aCoeffs[3]:=0.827124761997324;

      bCoeffs[0]:=0.1340901419430669;
      bCoeffs[1]:=0.4243248712718685;
      bCoeffs[2]:=0.7062921421386394;
      bCoeffs[3]:=0.9415030941737551;
    end
    else if (order=6) then  //rejection=51dB, transition band=0.01
    begin
      aCoeffs[0]:=0.06029739095712437;
      aCoeffs[1]:=0.4125907203610563;
      aCoeffs[2]:=0.7727156537429234;

      bCoeffs[0]:=0.21597144456092948;
      bCoeffs[1]:=0.6043586264658363;
      bCoeffs[2]:=0.9238861386532906;
    end
```

(continues on next page)

(continued from previous page)

```
    else if (order=4) then  //rejection=53dB,transition band=0.05
    begin
      aCoeffs[0]:=0.07986642623635751;
      aCoeffs[1]:=0.5453536510711322;

      bCoeffs[0]:=0.28382934487410993;
      bCoeffs[1]:=0.8344118914807379;
    end
    else    //order=2, rejection=36dB, transition band=0.1
    begin
      aCoeffs[0]:=0.23647102099689224;
      bCoeffs[0]:=0.7145421497126001;
    end;
  end;

  FilterA:=TAllPassFilterCascade.create(@aCoeffs[0],fOrder div 2);
  FilterB:=TAllPassFilterCascade.create(@bCoeffs[0],fOrder div 2);

  oldout:=0;
end;

function THalfBandFilter.process(input:single):single;
begin
  result:=(FilterA.Process(input)+oldout)*0.5;
  oldout:=FilterB.Process(input);
end;

end.
```

3.52 Poor Man's FIWIZ

- **Author or source:** moc.oohay@ljbliam
- **Type:** Filter Design Utility
- **Created:** 2007-03-22 15:02:29

Listing 86: notes

```
FIWIZ is a neat little filter design utility that uses a genetic programming technique
called Differential Evolution. As useful as it is, it looks like it took about a week␣
↪to
write, and is thus very undeserving of the $70 license fee. So I decided to write my␣
↪own.
There's a freely available optimizer class that uses Differential Evolution and I␣
↪patched
it together with some filter-specific logic.

Use of the utility requires the ability to do simple coding in C, but you need only␣
↪revise
a single function, which basically describes your filter specification. There's a big
comment in the main source file that clarifies things a bit more.

Although it's not as easy to use as FIWIZ, it's arguably more powerful because your
specifications are limited only by what you can express in C, whereas FIWIZ is␣
↪completely
```

(continues on next page)

(continued from previous page)

```
GUI based.

Another thing: I'm afraid that due to the use of _kbhit and _getch, the code is a bit
Microsofty, but you can take those out and the code will still be basically usable.
```

Listing 87: code

```
// First File: DESolver.cpp

#include <stdio.h>
#include <memory.h>
#include <conio.h>
#include "DESolver.h"

#define Element(a,b,c)  a[b*nDim+c]
#define RowVector(a,b)  (&a[b*nDim])
#define CopyVector(a,b) memcpy((a),(b),nDim*sizeof(double))

DESolver::DESolver(int dim,int popSize) :
                                    nDim(dim), nPop(popSize),
                                    generations(0), strategy(stRand1Exp),
                                    scale(0.7), probability(0.5), bestEnergy(0.0),
                                    trialSolution(0), bestSolution(0),
                                    popEnergy(0), population(0)
{
    trialSolution = new double[nDim];
    bestSolution  = new double[nDim];
    popEnergy         = new double[nPop];
    population        = new double[nPop * nDim];

    return;
}

DESolver::~DESolver(void)
{
    if (trialSolution) delete trialSolution;
    if (bestSolution) delete bestSolution;
    if (popEnergy) delete popEnergy;
    if (population) delete population;

    trialSolution = bestSolution = popEnergy = population = 0;
    return;
}

void DESolver::Setup(double *min,double *max,
                                            int deStrategy,double diffScale,double
↪crossoverProb)
{
    int i;

    strategy        = deStrategy;
    scale           = diffScale;
    probability = crossoverProb;

    for (i=0; i < nPop; i++)
    {
```

(continues on next page)

(continued from previous page)

```
            for (int j=0; j < nDim; j++)
                    Element(population,i,j) = RandomUniform(min[j],max[j]);

            popEnergy[i] = 1.0E20;
    }

    for (i=0; i < nDim; i++)
            bestSolution[i] = 0.0;

    switch (strategy)
    {
            case stBest1Exp:
      calcTrialSolution = &DESolver::Best1Exp;
                    break;

            case stRand1Exp:
                    calcTrialSolution = &DESolver::Rand1Exp;
                    break;

            case stRandToBest1Exp:
                    calcTrialSolution = &DESolver::RandToBest1Exp;
                    break;

            case stBest2Exp:
                    calcTrialSolution = &DESolver::Best2Exp;
                    break;

            case stRand2Exp:
                    calcTrialSolution = &DESolver::Rand2Exp;
                    break;

            case stBest1Bin:
                    calcTrialSolution = &DESolver::Best1Bin;
                    break;

            case stRand1Bin:
                    calcTrialSolution = &DESolver::Rand1Bin;
                    break;

            case stRandToBest1Bin:
                    calcTrialSolution = &DESolver::RandToBest1Bin;
                    break;

            case stBest2Bin:
                    calcTrialSolution = &DESolver::Best2Bin;
                    break;

            case stRand2Bin:
                    calcTrialSolution = &DESolver::Rand2Bin;
                    break;
    }

    return;
}

bool DESolver::Solve(int maxGenerations)
{
```

(continues on next page)

(continued from previous page)

```
    int generation;
    int candidate;
    bool bAtSolution;
  int generationsPerLoop = 10;

    bestEnergy = 1.0E20;
    bAtSolution = false;

    for (generation=0;
        (generation < maxGenerations) && !bAtSolution && (0 == _kbhit());
        generation++)
  {
            for (candidate=0; candidate < nPop; candidate++)
            {
                    (this->*calcTrialSolution)(candidate);
                    trialEnergy = EnergyFunction(trialSolution,bAtSolution);

                    if (trialEnergy < popEnergy[candidate])
                    {
                            // New low for this candidate
                            popEnergy[candidate] = trialEnergy;
                            CopyVector(RowVector(population,candidate),trialSolution);

                            // Check if all-time low
                            if (trialEnergy < bestEnergy)
                            {
                                    bestEnergy = trialEnergy;
                                    CopyVector(bestSolution,trialSolution);
                            }
                    }
            }

    if ((generation % generationsPerLoop) == (generationsPerLoop - 1))
    {
      printf("Gens %u Cost %.15g\n", generation+1, Energy());
    }
  }

  if (0 != _kbhit())
  {
    _getch();
  }

    generations = generation;
    return(bAtSolution);
}

void DESolver::Best1Exp(int candidate)
{
    int r1, r2;
    int n;

    SelectSamples(candidate,&r1,&r2);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; (RandomUniform(0.0,1.0) < probability) && (i < nDim); i++)
```

(continues on next page)

(continued from previous page)

```
    {
            trialSolution[n] = bestSolution[n]
                                        + scale * (Element(population,r1,
↪n)
                                        - Element(population,r2,n));
            n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Rand1Exp(int candidate)
{
    int r1, r2, r3;
    int n;

    SelectSamples(candidate,&r1,&r2,&r3);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; (RandomUniform(0.0,1.0) < probability) && (i < nDim); i++)
    {
            trialSolution[n] = Element(population,r1,n)
                                        + scale * (Element(population,r2,
↪n)
                                        - Element(population,r3,n));
            n = (n + 1) % nDim;
    }

    return;
}

void DESolver::RandToBest1Exp(int candidate)
{
    int r1, r2;
    int n;

    SelectSamples(candidate,&r1,&r2);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; (RandomUniform(0.0,1.0) < probability) && (i < nDim); i++)
    {
            trialSolution[n] += scale * (bestSolution[n] - trialSolution[n])
                                        + scale * (Element(population,r1,
↪n)
                                        - Element(population,r2,n));
            n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Best2Exp(int candidate)
{
    int r1, r2, r3, r4;
    int n;
```

(continues on next page)

(continued from previous page)

```

    SelectSamples(candidate,&r1,&r2,&r3,&r4);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; (RandomUniform(0.0,1.0) < probability) && (i < nDim); i++)
    {
            trialSolution[n] = bestSolution[n] +
                                            scale * (Element(population,r1,n)
                                                                + Element(population,r2,n)
                                                                - Element(population,r3,n)
                                                                - Element(population,r4,n));
            n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Rand2Exp(int candidate)
{
    int r1, r2, r3, r4, r5;
    int n;

    SelectSamples(candidate,&r1,&r2,&r3,&r4,&r5);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; (RandomUniform(0.0,1.0) < probability) && (i < nDim); i++)
    {
            trialSolution[n] = Element(population,r1,n)
                                            + scale * (Element(population,r2,n)
                                                                + Element(population,r3,n)
                                                                - Element(population,r4,n)
                                                                - Element(population,r5,n));
            n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Best1Bin(int candidate)
{
    int r1, r2;
    int n;

    SelectSamples(candidate,&r1,&r2);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; i < nDim; i++)
```

(continues on next page)

(continued from previous page)

```
    {
            if ((RandomUniform(0.0,1.0) < probability) || (i == (nDim - 1)))
                    trialSolution[n] = bestSolution[n]
                                                    + scale *␣
↪(Element(population,r1,n)
                                                                          -␣
↪Element(population,r2,n));
            n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Rand1Bin(int candidate)
{
    int r1, r2, r3;
    int n;

    SelectSamples(candidate,&r1,&r2,&r3);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; i < nDim; i++)
    {
            if ((RandomUniform(0.0,1.0) < probability) || (i  == (nDim - 1)))
                    trialSolution[n] = Element(population,r1,n)
                                                    + scale *␣
↪(Element(population,r2,n)
                                                                          ␣
↪       - Element(population,r3,n));
            n = (n + 1) % nDim;
    }

    return;
}

void DESolver::RandToBest1Bin(int candidate)
{
    int r1, r2;
    int n;

    SelectSamples(candidate,&r1,&r2);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; i < nDim; i++)
    {
            if ((RandomUniform(0.0,1.0) < probability) || (i  == (nDim - 1)))
                    trialSolution[n] += scale * (bestSolution[n] - trialSolution[n])
                                                            + scale *␣
↪(Element(population,r1,n)
                                                                          ␣
↪       - Element(population,r2,n));
            n = (n + 1) % nDim;
    }

    return;
```

(continues on next page)

(continued from previous page)

```
}

void DESolver::Best2Bin(int candidate)
{
    int r1, r2, r3, r4;
    int n;

    SelectSamples(candidate,&r1,&r2,&r3,&r4);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; i < nDim; i++)
    {
            if ((RandomUniform(0.0,1.0) < probability) || (i  == (nDim - 1)))
                    trialSolution[n] = bestSolution[n]
                                                         + scale * (Element(population,r1,n)
                                                                                       + Element(population,r2,n)
                                                                                       - Element(population,r3,n)
                                                                                       - Element(population,r4,n));
            n = (n + 1) % nDim;
    }

    return;
}

void DESolver::Rand2Bin(int candidate)
{
    int r1, r2, r3, r4, r5;
    int n;

    SelectSamples(candidate,&r1,&r2,&r3,&r4,&r5);
    n = (int)RandomUniform(0.0,(double)nDim);

    CopyVector(trialSolution,RowVector(population,candidate));
    for (int i=0; i < nDim; i++)
    {
            if ((RandomUniform(0.0,1.0) < probability) || (i  == (nDim - 1)))
                    trialSolution[n] = Element(population,r1,n)
                                                         + scale * (Element(population,r2,n)
                                                                                       + Element(population,r3,n)
                                                                                       - Element(population,r4,n)
                                                                                       - Element(population,r5,n));
            n = (n + 1) % nDim;
    }

    return;
}

void DESolver::SelectSamples(int candidate,int *r1,int *r2,
```

(continues on next page)

(continued from previous page)

```
                                                                                    int *r3,int *r4,int *r5)
{
    if (r1)
    {
            do
            {
                    *r1 = (int)RandomUniform(0.0,(double)nPop);
            }
            while (*r1 == candidate);
    }

    if (r2)
    {
            do
            {
                    *r2 = (int)RandomUniform(0.0,(double)nPop);
            }
            while ((*r2 == candidate) || (*r2 == *r1));
    }

    if (r3)
    {
            do
            {
                    *r3 = (int)RandomUniform(0.0,(double)nPop);
            }
            while ((*r3 == candidate) || (*r3 == *r2) || (*r3 == *r1));
    }

    if (r4)
    {
            do
            {
                    *r4 = (int)RandomUniform(0.0,(double)nPop);
            }
            while ((*r4 == candidate) || (*r4 == *r3) || (*r4 == *r2) || (*r4 == *r1));
    }

    if (r5)
    {
            do
            {
                    *r5 = (int)RandomUniform(0.0,(double)nPop);
            }
            while ((*r5 == candidate) || (*r5 == *r4) || (*r5 == *r3)
                                                        || (*r5 == *r2) || (*r5 == *r1));
    }

    return;
}

/*------Constants for RandomUniform()---------------------------------------*/
#define SEED 3
#define IM1 2147483563
```

(continues on next page)

(continued from previous page)

```
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

double DESolver::RandomUniform(double minValue,double maxValue)
{
    long j;
    long k;
    static long idum;
    static long idum2=123456789;
    static long iy=0;
    static long iv[NTAB];
    double result;

    if (iy == 0)
            idum = SEED;

    if (idum <= 0)
    {
            if (-idum < 1)
                    idum = 1;
            else
                    idum = -idum;

            idum2 = idum;

            for (j=NTAB+7; j>=0; j--)
            {
                    k = idum / IQ1;
                    idum = IA1 * (idum - k*IQ1) - k*IR1;
                    if (idum < 0) idum += IM1;
                    if (j < NTAB) iv[j] = idum;
            }

            iy = iv[0];
    }

    k = idum / IQ1;
    idum = IA1 * (idum - k*IQ1) - k*IR1;

    if (idum < 0)
            idum += IM1;

    k = idum2 / IQ2;
    idum2 = IA2 * (idum2 - k*IQ2) - k*IR2;

    if (idum2 < 0)
```

(continues on next page)

(continued from previous page)

```
                idum2 += IM2;

    j = iy / NDIV;
    iy = iv[j] - idum2;
    iv[j] = idum;

    if (iy < 1)
            iy += IMM1;

    result = AM * iy;

    if (result > RNMX)
            result = RNMX;

    result = minValue + result * (maxValue - minValue);
    return(result);
}

// END FIRST FILE

// BEGIN SECOND FILE: DESolver.h
// Differential Evolution Solver Class
// Based on algorithms developed by Dr. Rainer Storn & Kenneth Price
// Written By: Lester E. Godwin
//             PushCorp, Inc.
//             Dallas, Texas
//             972-840-0208 x102
//             godwin@pushcorp.com
// Created: 6/8/98
// Last Modified: 6/8/98
// Revision: 1.0

#if !defined(_DESOLVER_H)
#define _DESOLVER_H

#define stBest1Exp                  0
#define stRand1Exp                  1
#define stRandToBest1Exp    2
#define stBest2Exp                  3
#define stRand2Exp                  4
#define stBest1Bin                  5
#define stRand1Bin                  6
#define stRandToBest1Bin    7
#define stBest2Bin                  8
#define stRand2Bin                  9

class DESolver;

typedef void (DESolver::*StrategyFunction)(int);

class DESolver
{
public:
    DESolver(int dim,int popSize);
    ~DESolver(void);

    // Setup() must be called before solve to set min, max, strategy etc.
```

(continues on next page)

(continued from previous page)

```
    void Setup(double min[],double max[],int deStrategy,
                                                        double diffScale,double crossoverProb);

    // Solve() returns true if EnergyFunction() returns true.
    // Otherwise it runs maxGenerations generations and returns false.
    virtual bool Solve(int maxGenerations);

    // EnergyFunction must be overridden for problem to solve
    // testSolution[] is nDim array for a candidate solution
    // setting bAtSolution = true indicates solution is found
    // and Solve() immediately returns true.
    virtual double EnergyFunction(double testSolution[],bool &bAtSolution) = 0;

    int Dimension(void) { return(nDim); }
    int Population(void) { return(nPop); }

    // Call these functions after Solve() to get results.
    double Energy(void) { return(bestEnergy); }
    double *Solution(void) { return(bestSolution); }

    int Generations(void) { return(generations); }

protected:
    void SelectSamples(int candidate,int *r1,int *r2=0,int *r3=0,
        int *r4=0,int *r5=0);
    double RandomUniform(double min,double max);

    int nDim;
    int nPop;
    int generations;

    int strategy;
    StrategyFunction calcTrialSolution;
    double scale;
    double probability;

    double trialEnergy;
    double bestEnergy;

    double *trialSolution;
    double *bestSolution;
    double *popEnergy;
    double *population;

private:
    void Best1Exp(int candidate);
    void Rand1Exp(int candidate);
    void RandToBest1Exp(int candidate);
    void Best2Exp(int candidate);
    void Rand2Exp(int candidate);
    void Best1Bin(int candidate);
    void Rand1Bin(int candidate);
    void RandToBest1Bin(int candidate);
    void Best2Bin(int candidate);
    void Rand2Bin(int candidate);
```

(continues on next page)

(continued from previous page)

```
};


// I added the following stuff 19 March 2007
// Brent Lehman

struct ASpectrum
{
  unsigned mNumValues;
  double* mReals;
  double* mImags;
};

bool ComputeSpectrum(double* evenZeros, unsigned numEvenZeros, double* oddZero,
                     double* evenPoles, unsigned numEvenPoles, double* oddPole,
                     double gain, ASpectrum* spectrum);

class FilterSolver : public DESolver
{
public:
  FilterSolver(int dim, int popSize, int spectrumSize,
               unsigned numZeros, unsigned numPoles, bool minimumPhase) :
    DESolver(dim, popSize)
  {
    mSpectrum.mNumValues = spectrumSize;
    mSpectrum.mReals = new double[spectrumSize];
    mSpectrum.mImags = new double[spectrumSize];
    mNumZeros = numZeros;
    mNumPoles = numPoles;
    mMinimumPhase = minimumPhase;
  }
  virtual ~FilterSolver()
  {
    delete[] mSpectrum.mReals;
    delete[] mSpectrum.mImags;
  }
    virtual double EnergyFunction(double testSolution[], bool& bAtSolution);
  virtual ASpectrum* Spectrum() {return &mSpectrum;}
private:
  unsigned  mNumZeros;
  unsigned  mNumPoles;
  bool      mMinimumPhase;
  ASpectrum mSpectrum;
};


#endif // _DESOLVER_H

// END SECOND FILE DESolver.h

// BEGIN FINAL FILE: FilterDesign.cpp
/*
 *
 *  Filter Design Utility
 *  Source
 *
 *  Brent Lehman
```

(continues on next page)

(continued from previous page)

```
 *  16 March 2007
 *
 *
 */


////////////////////////////////////////////////////////////////////
//                                                                //
//  The idea is that an optimization algorithm passes a bunch of  //
//  different filter specifications to the function               //
//  "EnergyFunction" below.  That function is supposed to         //
//  compute an "error" or "cost" value for each specification     //
//  it receives, which the algorithm uses to decide on other      //
//  filter specifications to try.  Over the course of several     //
//  thousand different specifications, the algorithm will         //
//  eventually converge on a single best one.  This one has the   //
//  lowest error value of all possible specifications.  Thus,     //
//  you effectively tell the optimization algorithm what it's     //
//  looking for through code that you put into EnergyFunction.    //
//                                                                //
//  Look for a note in the code like this one to see what part    //
//  you need to change for your own uses.                         //
//                                                                //
////////////////////////////////////////////////////////////////////


#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <conio.h>
#include <math.h>
#include <time.h>
#include "DESolver.h"


#define kIntIsOdd(x)  (((x) & 0x00000001) == 1)


double FilterSolver::EnergyFunction(double testSolution[], bool& bAtSolution)
{
  unsigned i;
  double   tempReal;
  double   tempImag;

  // You probably will want to keep this if statement and its contents
  if (mMinimumPhase)
  {
    // Make sure there are no zeros outside the unit circle
    unsigned lastEvenZero = (mNumZeros & 0xfffffffe) - 1;
    for (i = 0; i <= lastEvenZero; i+=2)
    {
      tempReal = testSolution[i];
      tempImag = testSolution[i+1];
      if ((tempReal*tempReal + tempImag*tempImag) > 1.0)
      {
        return 1.0e+300;
      }
```

(continues on next page)

(continued from previous page)

```
    }

    if (kIntIsOdd(mNumZeros))
    {
      tempReal = testSolution[mNumZeros - 1];
      if ((tempReal * tempReal) > 1.0)
      {
        return 1.0e+300;
      }
    }
  }

  // Make sure there are no poles on or outside the unit circle
  // You probably will want to keep this too
  unsigned lastEvenPole = mNumZeros + (mNumPoles & 0xfffffffe) - 2;
  for (i = mNumZeros; i <= lastEvenPole; i+=2)
  {
    tempReal = testSolution[i];
    tempImag = testSolution[i+1];
    if ((tempReal*tempReal + tempImag*tempImag) > 0.999999999)
    {
      return 1.0e+300;
    }
  }

  // If you keep the for loop above, keep this too
  if (kIntIsOdd(mNumPoles))
  {
    tempReal = testSolution[mNumZeros + mNumPoles - 1];
    if ((tempReal * tempReal) > 1.0)
    {
      return 1.0e+300;
    }
  }

  double* evenZeros = &(testSolution[0]);
  double* evenPoles = &(testSolution[mNumZeros]);
  double* oddZero   = NULL;
  double* oddPole   = NULL;
  double  gain = testSolution[mNumZeros + mNumPoles];

  if (kIntIsOdd(mNumZeros))
  {
    oddZero = &(testSolution[mNumZeros - 1]);
  }

  if (kIntIsOdd(mNumPoles))
  {
    oddPole = &(testSolution[mNumZeros + mNumPoles - 1]);
  }

  ComputeSpectrum(evenZeros, mNumZeros & 0xfffffffe, oddZero,
                  evenPoles, mNumPoles & 0xfffffffe, oddPole,
                  gain, &mSpectrum);

  unsigned numPoints = mSpectrum.mNumValues;

```

(continues on next page)

(continued from previous page)

```
//////////////////////////////////////////////////////////////////
//                                                              //
//   Use the impulse response, held in the variable             //
//   "mSpectrum", to compute a score for the solution that      //
//   has been passed into this function.  You probably don't    //
//   want to touch any of the code above this point, but        //
//   from here to the end of this function, it's all you!       //
//                                                              //
//////////////////////////////////////////////////////////////////

  #define kLnTwoToThe127 88.02969193111305
  #define kRecipLn10      0.4342944819032518

  // Compute square sum of errors for magnitude
  double magnitudeError = 0.0;
  double magnitude = 0.0;
  double logMagnitude = 0.0;
  tempReal = mSpectrum.mReals[0];
  tempImag = mSpectrum.mImags[0];
  magnitude = tempReal*tempReal + tempImag*tempImag;
  double baseMagnitude = 0.0;
  if (0.0 == magnitude)
  {
    baseMagnitude = -kLnTwoToThe127;
  }
  else
  {
    baseMagnitude = log(magnitude) * kRecipLn10;
    baseMagnitude *= 0.5;
  }

  for (i = 0; i < numPoints; i++)
  {
    tempReal = mSpectrum.mReals[i];
    tempImag = mSpectrum.mImags[i];
    magnitude = tempReal*tempReal + tempImag*tempImag;
    if (0.0 == magnitude)
    {
      logMagnitude = -kLnTwoToThe127;
    }
    else
    {
      logMagnitude = log(magnitude) * kRecipLn10;
      logMagnitude *= 0.5;  // Half the log because it's mag squared
    }

    logMagnitude -= baseMagnitude;
    magnitudeError += logMagnitude * logMagnitude;
  }

  // Compute errors for phase
  double phaseError = 0.0;
  double phase = 0.0;
  double componentError = 0.0;
  double degree = 1;//((mNumZeros + 1) & 0xfffffffe) - 1;
  double angleSpacing = -3.141592653589793 * 0.5 / numPoints * degree;
  double targetPhase = 0.0;
```

(continues on next page)

(continued from previous page)

```
  double oldPhase = 0.0;
  double phaseDifference = 0;
  double totalPhaseTraversal = 0.0;
  double traversalError = 0.0;
  for (i = 0; i < (numPoints - 5); i++)
  {
    tempReal = mSpectrum.mReals[i];
    tempImag = mSpectrum.mImags[i];
    oldPhase = phase;
    phase = atan2(tempImag, tempReal);
    phaseDifference = phase - oldPhase;
    if (phaseDifference > 3.141592653589793)
    {
      phaseDifference -= 3.141592653589793;
      phaseDifference -= 3.141592653589793;
    }
    else if (phaseDifference < -3.141592653589793)
    {
      phaseDifference += 3.141592653589793;
      phaseDifference += 3.141592653589793;
    }
    totalPhaseTraversal += phaseDifference;
    componentError = cosh(200.0*(phaseDifference - angleSpacing)) - 0.5;
    phaseError += componentError * componentError;
    targetPhase += angleSpacing;
    if (targetPhase < -3.141592653589793)
    {
      targetPhase += 3.141592653589793;
      targetPhase += 3.141592653589793;
    }
  }

  traversalError = totalPhaseTraversal - angleSpacing * numPoints;
  traversalError *= traversalError;

  double baseMagnitudeError = baseMagnitude * baseMagnitude;

  // Compute weighted sum of the two subtotals
  // Take square root
  return sqrt(baseMagnitudeError*1.0 + magnitudeError*100.0 +
              phaseError*400.0 + traversalError*4000000.0);
}


////////////////////////////////
int main(int argc, char** argv)
{
  srand((unsigned)time(NULL));

  unsigned numZeros;
  unsigned numPoles;
  bool     minimumPhase;

  if (argc < 4)
  {
    printf("Usage: FilterDesign.exe <minimumPhase?> <numZeros> <numPoles>\n");
    return 0;
```

(continues on next page)

(continued from previous page)

```
}
else
{
  if (0 == atoi(argv[1]))
  {
    minimumPhase = false;
  }
  else
  {
    minimumPhase = true;
  }

  numZeros = (unsigned)atoi(argv[2]);
  if (0 == numZeros)
  {
    numZeros = 1;
  }

  numPoles = (unsigned)atoi(argv[3]);
}

unsigned vectorLength   = numZeros + numPoles + 1;
unsigned populationSize = vectorLength * 10;
FilterSolver theSolver(vectorLength, populationSize, 200,
                       numZeros, numPoles, minimumPhase);

double* minimumSolution = new double[vectorLength];
unsigned i;
if (minimumPhase)
{
  for (i = 0; i < numZeros; i++)
  {
    minimumSolution[i] = -sqrt(0.5);
  }
}
else
{
  for (i = 0; i < numZeros; i++)
  {
    minimumSolution[i] = -10.0;
  }
}

for (; i < (vectorLength - 1); i++)
{
  minimumSolution[i] = -sqrt(0.5);
}

minimumSolution[vectorLength - 1] = 0.0;

double* maximumSolution = new double[vectorLength];
if (minimumPhase)
{
  for (i = 0; i < numZeros; i++)
  {
    maximumSolution[i] = sqrt(0.5);
  }
```

(continues on next page)

(continued from previous page)

```
  }
  else
  {
    for (i = 0; i < numZeros; i++)
    {
      maximumSolution[i] = 10.0;
    }
  }

  for (i = 0; i < (vectorLength - 1); i++)
  {
    maximumSolution[i] = sqrt(0.5);
  }

  maximumSolution[vectorLength - 1] = 2.0;

  theSolver.Setup(minimumSolution, maximumSolution, 0, 0.5, 0.75);
  theSolver.Solve(1000000);

  double* bestSolution = theSolver.Solution();
  printf("\nZeros:\n");
  unsigned numEvenZeros = numZeros & 0xfffffffe;
  for (i = 0; i < numEvenZeros; i+=2)
  {
    printf("%.10f +/- %.10fi\n", bestSolution[i], bestSolution[i+1]);
  }

  if (kIntIsOdd(numZeros))
  {
    printf("%.10f\n", bestSolution[numZeros-1]);
  }

  printf("Poles:\n");
  unsigned lastEvenPole = numZeros + (numPoles & 0xfffffffe) - 2;
  for (i = numZeros; i <= lastEvenPole; i+=2)
  {
    printf("%.10f +/- %.10fi\n", bestSolution[i], bestSolution[i+1]);
  }

  unsigned numRoots = numZeros + numPoles;
  if (kIntIsOdd(numPoles))
  {
    printf("%.10f\n", bestSolution[numRoots-1]);
  }

  double gain = bestSolution[numRoots];
  printf("Gain: %.10f\n", gain);

  _getch();
  unsigned j;
  ASpectrum* spectrum = theSolver.Spectrum();
  double logMagnitude;
  printf("Magnitude Response, millibels:\n");
  for (i = 0; i < 20; i++)
  {
    for (j = 0; j < 10; j++)
    {
```

(continues on next page)

(continued from previous page)

```
      logMagnitude = kRecipLn10 *
         log(spectrum->mReals[i*10 + j] * spectrum->mReals[i*10 + j] +
             spectrum->mImags[i*10 + j] * spectrum->mImags[i*10 + j]);
      if (logMagnitude < -9.999)
      {
        logMagnitude = -9.999;
      }
      printf("%+5.0f ", logMagnitude*1000);
    }
    printf("\n");
  }

  _getch();
  double phase;
  printf("Phase Response, milliradians:\n");
  for (i = 0; i < 20; i++)
  {
    for (j = 0; j < 10; j++)
    {
      phase = atan2(spectrum->mImags[i*10 + j], spectrum->mReals[i*10 + j]);
      printf("%+5.0f ", phase*1000);
    }
    printf("\n");
  }

  _getch();
  printf("Biquad Sections:\n");
  unsigned numBiquadSections =
    (numZeros > numPoles) ? ((numZeros + 1) >> 1) : ((numPoles + 1) >> 1);
  double x0, x1, x2;
  double y0, y1, y2;
  if (numZeros >=2)
  {
    x0 = (bestSolution[0]*bestSolution[0] + bestSolution[1]*bestSolution[1]) *
         gain;
    x1 = 2.0 * bestSolution[0] * gain;
    x2 = gain;
  }
  else if (1 == numZeros)
  {
    x0 = bestSolution[0] * gain;
    x1 = gain;
    x2 = 0.0;
  }
  else
  {
    x0 = gain;
    x1 = 0.0;
    x2 = 0.0;
  }

  if (numPoles >= 2)
  {
    y0 = (bestSolution[numZeros]*bestSolution[numZeros] +
          bestSolution[numZeros+1]*bestSolution[numZeros+1]);
    y1 = 2.0 * bestSolution[numZeros];
    y2 = 1.0;
```

(continues on next page)

(continued from previous page)

```
  }
  else if (1 == numPoles)
  {
    y0 = bestSolution[numZeros];
    y1 = 1.0;
    y2 = 0.0;
  }
  else
  {
    y0 = 1.0;
    y1 = 0.0;
    y2 = 0.0;
  }

  x0 /= y0;
  x1 /= y0;
  x2 /= y0;
  y1 /= y0;
  y2 /= y0;

  printf("y[n] = %.10fx[n]", x0);
  if (numZeros > 0)
  {
    printf(" + %.10fx[n-1]", x1);
  }
  if (numZeros > 1)
  {
    printf(" + %.10fx[n-2]", x2);
  }
  printf("\n");

  if (numPoles > 0)
  {
    printf("                   + %.10fy[n-1]", y1);
  }
  if (numPoles > 1)
  {
    printf(" + &.10fy[n-2]", y2);
  }
  if (numPoles > 0)
  {
    printf("\n");
  }

  int numRemainingZeros = numZeros - 2;
  int numRemainingPoles = numPoles - 2;
  for (i = 1; i < numBiquadSections; i++)
  {
    if (numRemainingZeros >= 2)
    {
      x0 = (bestSolution[i*2]   * bestSolution[i*2] +
            bestSolution[i*2+1] * bestSolution[i*2+1]);
      x1 = -2.0 * bestSolution[i*2];
      x2 = 1.0;
    }
    else if (numRemainingZeros >= 1)
    {
```

(continues on next page)

(continued from previous page)

```
        x0 = bestSolution[i*2];
        x1 = 1.0;
        x2 = 0.0;
      }
      else
      {
        x0 = 1.0;
        x1 = 0.0;
        x2 = 0.0;
      }

      if (numRemainingPoles >= 2)
      {
        y0 = (bestSolution[i*2+numZeros]   * bestSolution[i*2+numZeros] +
              bestSolution[i*2+numZeros+1] * bestSolution[i*2+numZeros+1]);
        y1 = -2.0 * bestSolution[i*2+numZeros];
        y2 = 1.0;
      }
      else if (numRemainingPoles >= 1)
      {
        y0 = bestSolution[i*2+numZeros];
        y1 = 1.0;
        y2 = 0.0;
      }
      else
      {
        y0 = 1.0;
        y1 = 0.0;
        y2 = 0.0;
      }

      x0 /= y0;
      x1 /= y0;
      x2 /= y0;
      y1 /= y0;
      y2 /= y0;

      printf("y[n] = %.10fx[n]", x0);
      if (numRemainingZeros > 0)
      {
        printf(" + %.10fx[n-1]", x1);
      }
      if (numRemainingZeros > 1)
      {
        printf(" + %.10fx[n-2]", x2);
      }
      printf("\n");

      if (numRemainingPoles > 0)
      {
        printf("                   + %.10fy[n-1]", -y1);
      }
      if (numRemainingPoles > 1)
      {
        printf(" + %.10fy[n-2]", -y2);
      }
      if (numRemainingPoles > 0)
```

(continues on next page)

(continued from previous page)

```
    {
      printf("\n");
    }

    numRemainingZeros -= 2;
    numRemainingPoles -= 2;
  }

  _getch();
  printf("Full Expansion:\n");
  double* xpolynomial = new double[numRoots + 1];
  memset(xpolynomial, 0, sizeof(double) * (numRoots + 1));
  xpolynomial[0] = 1.0;
  if (numZeros >= 2)
  {
    xpolynomial[0] = bestSolution[0] * bestSolution[0] +
                     bestSolution[1] * bestSolution[1];
    xpolynomial[1] = -2.0 * bestSolution[0];
    xpolynomial[2] = 1.0;
  }
  else if (numZeros == 1)
  {
    xpolynomial[0] = bestSolution[0];
    xpolynomial[1] = 1.0;
  }
  else
  {
    xpolynomial[0] = 1.0;
  }

  for (i  = 2, numRemainingZeros = numZeros; numRemainingZeros >= 2;
       i += 2, numRemainingZeros-=2)
  {
    x2 = 1.0;
    x1 = -2.0 * bestSolution[i];
    x0 = bestSolution[i]   * bestSolution[i] +
         bestSolution[i+1] * bestSolution[i+1];
    for (j = numRoots; j > 1; j--)
    {
      xpolynomial[j] = xpolynomial[j-2] + xpolynomial[j-1] * x1 +
                       xpolynomial[j] * x0;
    }
    xpolynomial[1]  = xpolynomial[0] * x1 + xpolynomial[1] * x0;
    xpolynomial[0] *= x0;
  }

  if (numRemainingZeros > 0)
  {
    x1 = 1.0;
    x0 = bestSolution[numZeros-1];
    for (j = numRoots; j > 0; j--)
    {
      xpolynomial[j] = xpolynomial[j-1] + xpolynomial[j] * x0;
    }
    xpolynomial[0] *= x0;
  }

```

(continues on next page)

(continued from previous page)

```
double* ypolynomial = new double[numRoots + 1];
memset(ypolynomial, 0, sizeof(double) * (numRoots + 1));
ypolynomial[0] = 1.0;
if (numPoles >= 2)
{
  ypolynomial[0] = bestSolution[numZeros]    * bestSolution[numZeros] +
                   bestSolution[numZeros+1] * bestSolution[numZeros+1];
  ypolynomial[1] = -2.0 * bestSolution[numZeros];
  ypolynomial[2] = 1.0;
}
else if (numPoles == 1)
{
  ypolynomial[0] = bestSolution[numZeros];
  ypolynomial[1] = 1.0;
}
else
{
  xpolynomial[0] = 1.0;
}

for (i  = 2, numRemainingPoles = numPoles; numRemainingPoles >= 2;
     i += 2, numRemainingPoles-=2)
{
  y2 = 1.0;
  y1 = -2.0 * bestSolution[numZeros+i];
  y0 = bestSolution[numZeros+i]   * bestSolution[numZeros+i] +
       bestSolution[numZeros+i+1] * bestSolution[numZeros+i+1];
  for (j = numRoots; j > 1; j--)
  {
    ypolynomial[j] = ypolynomial[j-2] + ypolynomial[j-1] * y1 +
                     ypolynomial[j] * y0;
  }
  ypolynomial[1]  = ypolynomial[0] * y1 + ypolynomial[1] * y0;
  ypolynomial[0] *= y0;
}

if (numRemainingPoles > 0)
{
  y1 = 1.0;
  y0 = bestSolution[numZeros+numPoles-1];
  for (j = numRoots; j > 0; j--)
  {
    ypolynomial[j] = ypolynomial[j-1] + ypolynomial[j] * y0;
  }
  ypolynomial[0] *= y0;
}

y0 = ypolynomial[0];
for (i = 0; i <= numRoots; i++)
{
  xpolynomial[i] /= y0;
  ypolynomial[i] /= y0;
}

printf("y[n] = %.10fx[n]", xpolynomial[0]*gain);
for (i = 1; i <= numZeros; i++)
{
```

(continues on next page)

(continued from previous page)

```
    printf(" + %.10fx[n-%d]", xpolynomial[i]*gain, i);
    if ((i % 3) == 2)
    {
      printf("\n");
    }
  }

  if ((i % 3) != 0)
  {
    printf("\n");
  }

  if (numPoles > 0)
  {
    printf("                 ");
  }

  for (i = 1; i <= numPoles; i++)
  {
    printf(" + %.10fy[n-%d]", -ypolynomial[i], i);
    if ((i % 3) == 2)
    {
      printf("\n");
    }
  }

  if ((i % 3) != 0)
  {
    printf("\n");
  }

  delete[] minimumSolution;
  delete[] maximumSolution;
  delete[] xpolynomial;
  delete[] ypolynomial;
}


bool ComputeSpectrum(double* evenZeros, unsigned numEvenZeros, double* oddZero,
                     double* evenPoles, unsigned numEvenPoles, double* oddPole,
                     double gain, ASpectrum* spectrum)
{
  unsigned i, j;

  // For equally spaced points on the unit circle
  unsigned numPoints = spectrum->mNumValues;
  double   spacingAngle = 3.141592653589793 / (numPoints - 1);
  double   pointArgument = 0.0;
  double   pointReal = 0.0;
  double   pointImag = 0.0;
  double   rootReal = 0.0;
  double   rootImag = 0.0;
  double   differenceReal = 0.0;
  double   differenceImag = 0.0;
  double   responseReal = 1.0;
  double   responseImag = 0.0;
  double   recipSquareMagnitude = 0.0;
```

(continues on next page)

(continued from previous page)

```
double   recipReal = 0.0;
double   recipImag = 0.0;
double   tempRealReal = 0.0;
double   tempRealImag = 0.0;
double   tempImagReal = 0.0;
double   tempImagImag = 0.0;

for (i = 0; i < numPoints; i++)
{
  responseReal = 1.0;
  responseImag = 0.0;

  // The imaginary component is negated because we're using 1/z, not z
  pointReal =  cos(pointArgument);
  pointImag = -sin(pointArgument);

  // For each even zero
  for (j = 0; j < numEvenZeros; j+=2)
  {
    rootReal = evenZeros[j];
    rootImag = evenZeros[j + 1];
    // Compute distance from that zero to that point
    differenceReal = pointReal - rootReal;
    differenceImag = pointImag - rootImag;
    // Multiply that distance by the accumulating product
    tempRealReal = responseReal * differenceReal;
    tempRealImag = responseReal * differenceImag;
    tempImagReal = responseImag * differenceReal;
    tempImagImag = responseImag * differenceImag;
    responseReal = tempRealReal - tempImagImag;
    responseImag = tempRealImag + tempImagReal;
    // Do the same with the conjugate root
    differenceImag = pointImag + rootImag;
    tempRealReal = responseReal * differenceReal;
    tempRealImag = responseReal * differenceImag;
    tempImagReal = responseImag * differenceReal;
    tempImagImag = responseImag * differenceImag;
    responseReal = tempRealReal - tempImagImag;
    responseImag = tempRealImag + tempImagReal;
    // The following way is little faster, if any
    // response *= (1/z - r) * (1/z - conj(r))
    //          *= r*conj(r) - (r + conj(r))/z + 1/(z*z)
    //          *= real(r)*real(r) + imag(r)*imag(r) - 2*real(r)/z + 1/(z*z)
    //          *= ... - 2*real(r)*conj(z) + conj(z)*conj(z)
    //          *= ... - 2*real(r)*real(z) + 2i*real(r)*imag(z) +
    //             real(z)*real(z) - 2i*real(z)*imag(z) + imag(z)*imag(z)
    //          *= real(r)*real(r) + imag(r)*imag(r) - 2*real(r)*real(z) +
    //             real(z)*real(z) + imag(z)*imag(z) +
    //              2i * imag(z) * (real(r) - real(z))
    //          *= (real(r) - real(z))^2  + imag(r)^2 + imag(z)^2 +
    //              2i * imag(z) * (real(r) - real(z))
    // This ends up being 8 multiplications, 6 additions
  }

  if (NULL != oddZero)
  {
    rootReal = *oddZero;
```

(continues on next page)

(continued from previous page)

```
      // Compute distance from that zero to that point
      differenceReal = pointReal - rootReal;
      differenceImag = pointImag;
      // Multiply that distance by the accumulating product
      tempRealReal = responseReal * differenceReal;
      tempRealImag = responseReal * differenceImag;
      tempImagReal = responseImag * differenceReal;
      tempImagImag = responseImag * differenceImag;
      responseReal = tempRealReal - tempImagImag;
      responseImag = tempRealImag + tempImagReal;
    }

    // For each pole
    for (j = 0; j < numEvenPoles; j+=2)
    {
      rootReal = evenPoles[j];
      rootImag = evenPoles[j + 1];
      // Compute distance from that pole to that point
      differenceReal = pointReal - rootReal;
      differenceImag = pointImag - rootImag;
      // Multiply the reciprocal of that distance by the accumulating product
      recipSquareMagnitude = 1.0 / (differenceReal * differenceReal +
                                    differenceImag * differenceImag);
      recipReal =  differenceReal * recipSquareMagnitude;
      recipImag = -differenceImag * recipSquareMagnitude;
      tempRealReal = responseReal * recipReal;
      tempRealImag = responseReal * recipImag;
      tempImagReal = responseImag * recipReal;
      tempImagImag = responseImag * recipImag;
      responseReal = tempRealReal - tempImagImag;
      responseImag = tempRealImag + tempImagReal;
      // Do the same with the conjugate root
      differenceImag = pointImag + rootImag;
      recipSquareMagnitude = 1.0 / (differenceReal * differenceReal +
                                    differenceImag * differenceImag);
      recipReal =  differenceReal * recipSquareMagnitude;
      recipImag = -differenceImag * recipSquareMagnitude;
      tempRealReal = responseReal * recipReal;
      tempRealImag = responseReal * recipImag;
      tempImagReal = responseImag * recipReal;
      tempImagImag = responseImag * recipImag;
      responseReal = tempRealReal - tempImagImag;
      responseImag = tempRealImag + tempImagReal;
    }

    if (NULL != oddPole)
    {
      rootReal = *oddPole;
      // Compute distance from that point to that zero
      differenceReal = pointReal - rootReal;
      differenceImag = pointImag;
      // Multiply the reciprocal of that distance by the accumulating product
      recipSquareMagnitude = 1.0 / (differenceReal * differenceReal +
                                    differenceImag * differenceImag);
      recipReal =  differenceReal * recipSquareMagnitude;
      recipImag = -differenceImag * recipSquareMagnitude;
      tempRealReal = responseReal * recipReal;
```

(continues on next page)

(continued from previous page)

```
      tempRealImag = responseReal * recipImag;
      tempImagReal = responseImag * recipReal;
      tempImagImag = responseImag * recipImag;
      responseReal = tempRealReal - tempImagImag;
      responseImag = tempRealImag + tempImagReal;
    }

    // Multiply by the gain
    responseReal *= gain;
    responseImag *= gain;

    spectrum->mReals[i] = responseReal;
    spectrum->mImags[i] = responseImag;

    pointArgument += spacingAngle;
  }

  return true;
}

// Half-band lowpass
/*
  #define kLnTwoToThe127 88.02969193111305
  #define kRecipLn10      0.4342944819032518

  // Compute square sum of errors for bottom half band
  unsigned numLoBandPoints = numPoints >> 1;
  double loBandError = 0.0;
  double magnitude = 0.0;
  double logMagnitude = 0.0;
  for (i = 0; i < numLoBandPoints; i++)
  {
    tempReal = mSpectrum.mReals[i];
    tempImag = mSpectrum.mImags[i];
    magnitude = tempReal*tempReal + tempImag*tempImag;
    if (0.0 == magnitude)
    {
      logMagnitude = -kLnTwoToThe127;
    }
    else
    {
      logMagnitude = log(magnitude) * kRecipLn10;
      logMagnitude *= 0.5;  // Half the log because it's mag squared
    }

    loBandError += logMagnitude * logMagnitude;
  }

  // Compute errors for top half of band
  double hiBandError = 0.0;
  for ( ; i < numPoints; i++)
  {
    tempReal = mSpectrum.mReals[i];
    tempImag = mSpectrum.mImags[i];
    magnitude = tempReal*tempReal + tempImag*tempImag;
    hiBandError += magnitude; // Already a squared value
  }
```

(continues on next page)

(continued from previous page)

```
  // Compute weighted sum of the two subtotals
  // Take square root
  return sqrt(loBandError + 5000.0 * hiBandError);
*/
```

3.53 Prewarping

- **Author or source:** robert bristow-johnson (better known as "rbj")
- **Type:** explanation
- **Created:** 2002-01-17 02:10:26

Listing 88: 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.
```

3.54 RBJ Audio-EQ-Cookbook

- **Author or source:** Robert Bristow-Johnson
- **Type:** EQ filter kookbook

- **Created:** 2005-05-04 20:31:18
- **Linked files:** `Audio-EQ-Cookbook.txt`.

Listing 89: notes

```
Equations for creating different equalization filters.
see linked file
```

3.54.1 Comments

- **Date**: 2006-08-30 22:14:22
- **By**: ude.odu@grebnesie.nitram

```
rbj writes with regard to shelving filters:

> _or_ S, a "shelf slope" parameter (for shelving EQ only).  When S = 1,
> the shelf slope is as steep as it can be and remain monotonically
> increasing or decreasing gain with frequency.  The shelf slope, in
> dB/octave, remains proportional to S for all other values for a
> fixed f0/Fs and dBgain.

The precise relation for both low and high shelf filters is

  S = -s * log_2(10)/40 * sin(w0)/w0 * (A^2+1)/(A^2-1)

where s is the true shelf midpoint slope in dB/oct and w0, A are defined in
the Cookbook just below the quoted paragraph. It's your responsibility to keep
the overshoots in check by using sensible s values. Also make sure that s has
the right sign -- negative for low boost or high cut, positive otherwise.

To find the relation I first differentiated the dB magnitude response of the
general transfer function in eq. 1 with regard to log frequency, inserted the
low shelf coefficient expressions, and evaluated at w0. Second, I equated this
derivative to s and solved for alpha. Third, I equated the result to rbj's
expression for alpha and solved for S yielding the above formula. Finally
I checked it with the high shelf filter.
```

- **Date**: 2006-08-31 17:08:27
- **By**: ude.odu@grebnesie.nitram

```
Sorry, a slight correction: rewrite the formula as

  S = s * log_2(10)/40 * sin(w0)/w0 * (A^2+1)/abs(A^2-1)

nad make s always positive.
```

- **Date**: 2013-10-05 18:06:20
- **By**: moc.liamg@56rekojbm

```
This is a very famous article. I saw many are asking what is the relationship between
↪"Q" and the resonance in low-pass and hi-pass filters.

By experimenting, I found that Q should always be >= 1/2. Value < 1/2 seems to alter␣
↪f0 "wherever it's happenin', man", cutting off frequencies not where it was planned.
↪ In fact Q = 1/2 is the value for which H(s) = 1 / (s^2 + s/Q + 1) gets two poles␣
↪real and coincident. In other words the filter becomes like two 1st order filters␣
↪in cascade, with no resonance at all.
```

(continues on next page)

(continued from previous page)

```
When Q tends to infinite the poles get close to the unit circle, the gain around the
↪cutoff frequency increases, creating resonance.
```

3.55 RBJ Audio-EQ-Cookbook

- **Author or source:** Robert Bristow-Johnson
- **Created:** 2005-05-04 20:33:31
- **Linked files:** `EQ-Coefficients.pdf`.

Listing 90: notes

```
see attached file
```

3.55.1 Comments

- **Date**: 2005-05-14 06:35:17
- **By**: eb.tenyks@didid

```
Hi

In your most recent version, you write:

--
    alpha = sin(w0)/(2*Q)       (case: Q)
          = sin(w0)*sinh( ln(2)/2 * BW * w0/sin(w0) )       (case: BW)
          = sin(w0)/2 * sqrt( (A + 1/A)*(1/S - 1) + 2 )        (case: S)
--


But the 'slope' case doesn't seem to work for me. It results in some kind of bad
↪resonance at higher samplerates.

Now I found this 'beta' in an older version of your paper (I think), describing:

--
    beta  = sqrt(A)/Q   (for shelving EQ filters only)
          = sqrt(A)*sqrt[ (A + 1/A)*(1/S - 1) + 2 ]        (if shelf slope is
↪specified)
          = sqrt[ (A^2 + 1)/S - (A-1)^2 ]
--

..and here the
sqrt(A)*sqrt[ (A + 1/A)*(1/S - 1) + 2 ]
formula works perfectly for me.

I must say I don't understand half of the theory, so it's probably my fault somewhere.
↪ But why the change in the newer version?
```

- **Date**: 2005-05-20 20:56:45
- **By**: moc.noitanigamioidua@jbr

```
>But why the change in the newer version?
-----------------------------------------------------------

i wanted to get rid of an extraneous intermediate variable and there was enough␣
↪similarity between alpha and beta that i changed the lowShelf and highShelf␣
↪coefficient equations to be in terms of alpha rather than beta.

i believe if you use the new version as shown, in terms of alpha (but remember the␣
↪coef equations are changed accordingly from the old version), it will come up with␣
↪the same coefficients given the same boost gain, Q (or S), and shelf frequency (and␣
↪same Fs).  lemme know if you still have trouble.

r b-j
```

3.56 Remez Exchange Algorithm (Parks/McClellan)

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** Linear Phase FIR Filter
- **Created:** 2006-05-06 08:40:18
- **Linked files:** `http://www.savioursofsoul.de/Christian/Remez.zip`.

Listing 91: notes

```
Here is an object pascal / delphi translation of the Remez Exchange Algorithm by
Parks/McClellan
There is at least one small bug in it (compared to the C++ version), which causes the
result to be slightly different to the C version.
```

Listing 92: code

```
http://www.savioursofsoul.de/Christian/Remez.zip
```

3.57 Remez Remez (Parks/McClellan)

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** FIR Remez (Parks/McClellan)
- **Created:** 2005-06-28 21:06:53

Listing 93: notes

```
Below you can find a Object Pascal / Delphi Translation of the Remez (Parks/
↪McClellan) FIR
Filter Design algorithm.
It behaves slightly different from the c++ version, but the results work very well.
```

Listing 94: code

```
http://www.savioursofsoul.de/Christian/remez.zip
```

3.58 Resonant IIR lowpass (12dB/oct)

- **Author or source:** Olli Niemitalo
- **Type:** Resonant IIR lowpass (12dB/oct)
- **Created:** 2002-01-17 02:05:38

Listing 95: notes

```
Hard to calculate coefficients, easy to process algorithm
```

Listing 96: 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;
}
```

3.58.1 Comments

- **Date**: 2002-05-05 08:59:19

- **By**: moc.ibtta@suocuar

```
This looks similar to the low-pass filter I used in FilterKing (http://home.attbi.com/
↪~spaztek4/) Can you cruft up a high-pass example for me?

Thanks,
__e
```

- **Date**: 2006-05-18 17:01:21
- **By**: faster init

```
thank you! works nicely...
here a simplified init version for faster changes of the filter properties for amps =␣
↪1.0

void init( double resofreq )
{
static const double FAC = pi * 2.0 /samplerate;
double q, w;

w = FAC * resofreq;
q = 1.0f - w /( ( 3.0 /( 1.0+w ) ) + w - 2.0 );

_r = q * q;
_c = r + 1.0f - 2.0f * cos(w) * q;
}
```

3.59 Resonant filter

- **Author or source:** Paul Kellett
- **Created:** 2002-01-17 02:07:02

Listing 97: 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.
```

Listing 98: 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;
```

3.59.1 Comments

- **Date**: 2006-01-18 10:59:55
- **By**: mr.just starting

```
very nice! how could i turn that into a HPF?
```

- **Date**: 2006-01-23 10:53:41
- **By**: ku.oc.mapson.snosrapsd@psd

```
The cheats way is to use HPF = sample - out;
If you do a plot, you'll find that it isn't as good as designing an HPF from scratch,␣
↪but it's good enuff for most ears.
This would also mean that you have a quick method for splitting a signal and␣
↪operating on the (in)discreet parts separately. :) DSP
```

- **Date**: 2006-09-12 14:42:25
- **By**: uh.etle.fni@yfoocs

```
This filter calculates bandpass and highpass outputs too during calculation, namely␣
↪bandpass is buf0 - buf1 and highpass is in - buf0. So, we can rewrite the algorithm:

// f and fb calculation
f = 2.0*sin(pi*freq/samplerate);
/* you can approximate this with f = 2.0*pi*freq/samplerate with tuning error towards␣
↪nyquist */
fb = q + q/(1.0 - f);

// loop
hp = in - buf0;
bp = buf0 - buf1;
buf0 = buf0 + f * (hp + fb * bp);
buf1 = buf1 + f * (buf0 - buf1);

out = buf1; // lowpass
out = bp; // bandpass
out = hp; // highpass

The slope of the highpass out is not constant, it varies between 6 and 12 dB/Octave␣
↪with different f and q settings. I'd be interested if anyone derived a proper␣
↪highpass output from this algorithm.

-- peter schoffhauzer
```

3.60 Resonant low pass filter

- **Author or source:** "Zxform"
- **Type:** 24dB lowpass
- **Created:** 2002-01-17 02:09:31
- **Linked files:** `filters004.txt`.

3.61 Reverb Filter Generator

- **Author or source:** Stephen McGovern
- **Type:** FIR
- **Created:** 2006-09-01 07:07:58

Listing 99: notes

```
This is a MATLAB function that makes a rough calculation of a room's impulse response.
The output can then be convolved with an audio clip to produce good and realistic␣
↪sounding
reverb.  I have written a paper discussing the theory used by this algorithm.  It is
available at http://stevem.us/rir.html.

NOTES:

   1) Large values of N will use large amounts of memory.
   2) The output is normalized to the largest value of the
      output.
```

Listing 100: code

```
function [h]=rir(fs, mic, n, r, rm, src);
%RIR   Room Impulse Response.
%   [h] = RIR(FS, MIC, N, R, RM, SRC) performs a room impulse
%         response calculation by means of the mirror image method.
%
%      FS  = sample rate.
%      MIC = row vector giving the x,y,z coordinates of
%            the microphone.
%      N   = The program will account for (2*N+1)3 virtual sources
%      R   = reflection coefficient for the walls, in general -1<R<1.
%      RM  = row vector giving the dimensions of the room.
%      SRC = row vector giving the x,y,z coordinates of
%            the sound source.
%
%   EXAMPLE:
%
%      >>fs=44100;
%      >>mic=[19 18 1.6];
%      >>n=12;
%      >>r=0.3;
%      >>rm=[20 19 21];
%      >>src=[5 2 1];
```

(continues on next page)

(continued from previous page)

```
%      >>h=rir(fs, mic, n, r, rm, src);
%
%   NOTES:
%
%   1) All distances are in meters.
%   2) The output is scaled such that the largest value of the
%      absolute value of the output vector is equal to one.
%   3) To implement this filter, you will need to do a fast
%      convolution.  The program FCONV.m will do this. It can be
%      found on the Mathworks File Exchange at
%      www.mathworks.com/matlabcentral/fileexchange/.  It can also
%      be found at www.2pi.us/code/fconv.m
%   4) A paper has been written on this model.  It is available at:
%      www.2pi.us/rir.html
%
%
%Version 3.4.1
%Copyright Â© 2003 Stephen G. McGovern

%Some of the following comments are references to equations the my paper.

nn=-n:1:n;                            % Index for the sequence
rms=nn+0.5-0.5*(-1).^nn;              % Part of equations 2,3,& 4
srcs=(-1).^(nn);                      % part of equations 2,3,& 4
xi=srcs*src(1)+rms*rm(1)-mic(1);      % Equation 2
yj=srcs*src(2)+rms*rm(2)-mic(2);      % Equation 3
zk=srcs*src(3)+rms*rm(3)-mic(3);      % Equation 4

[i,j,k]=meshgrid(xi,yj,zk);           % convert vectors to 3D matrices
d=sqrt(i.^2+j.^2+k.^2);               % Equation 5
time=round(fs*d/343)+1;               % Similar to Equation 6

[e,f,g]=meshgrid(nn, nn, nn);         % convert vectors to 3D matrices
c=r.^(abs(e)+abs(f)+abs(g));          % Equation 9
e=c./d;                               % Equivalent to Equation 10

h=full(sparse(time(:),1,e(:)));       % Equivalent to equation 11
h=h/max(abs(h));                      % Scale output
```

3.62 Simple Tilt equalizer

- **Author or source:** moc.liamg@321tiloen
- **Type:** Tilt
- **Created:** 2009-05-29 15:13:21

Listing 101: notes

```
There are a few ways to implement this. (crossover, shelves, morphing shelves [hs->lp,
ls->hp] ...etc)
This particular one tries to mimic the behavior of the "Niveau" filter from the
↪"Elysia:
mPressor" compressor.
```

(continues on next page)

(continued from previous page)

```
[The 'Tilt' filter]:
It uses a center frequency (F0) and then boosts one of the ranges above or below F0,
↪while
doing the opposite with the other range.

In the case of the "mPressor" - more extreme settings turn the filter into first order
low-pass or high-pass. This is achieved with the gain factor for one band going close
↪to
-1. (ex: +6db -> lp; -6db -> hp)

Lubomir I. Ivanov
```

Listing 102: code

```
//=======================
// tilt eq settings
//
// srate -> sample rate
// f0 -> 20-20khz
// gain -> -6 / +6 db
//=======================
amp = 6/log(2);
denorm = 10^-30;
pi = 22/7;
sr3 = 3*srate;

// conditition:
// gfactor is the proportional gain
//
gfactor = 5;
if (gain > 0) {
    g1 = -gfactor*gain;
    g2 = gain;
} else {
    g1 = -gain;
    g2 = gfactor*gain;
};

//two separate gains
lgain = exp(g1/amp)-1;
hgain = exp(g2/amp)-1;

//filter
omega = 2*pi*f0;
n = 1/(sr3 + omega);
a0 = 2*omega*n;
b1 = (sr3 - omega)*n;

//==================================
// sample loop
// in -> input sample
// out -> output sample
//==================================
lp_out = a0*in + b1*lp_out;
out = in + lgain*lp_out + hgain*(in - lp_out);
```

3.62.1 Comments

- **Date**: 2009-05-29 19:16:18
- **By**: moc.liamg@321tiloen

```
correction:

(ex: +6db -> hp; -6db -> lp)
```

- **Date**: 2017-04-01 09:05:48
- **By**: moc.liamg@59hsielgladsemaj

```
Where is the denorm value meant to be used in the code? The code works regardless by noise is created when the gain control being used, it may be a result of the denorm value not being used?
```

3.63 Simple biquad filter from apple.com

- **Author or source:** moc.liamg@321tiloen
- **Type:** LP
- **Created:** 2008-10-27 10:15:16

Listing 103: notes

```
Simple Biquad LP filter from the AU tutorial at apple.com
```

Listing 104: code

```
//cutoff_slider range 20-20000hz
//res_slider range -25/25db
//srate - sample rate

//init
mX1 = 0;
mX2 = 0;
mY1 = 0;
mY2 = 0;
pi = 22/7;

//coefficients
cutoff = cutoff_slider;
res = res_slider;

cutoff = 2 * cutoff_slider / srate;
res = pow(10, 0.05 * -res_slider);
k = 0.5 * res * sin(pi * cutoff);
c1 = 0.5 * (1 - k) / (1 + k);
c2 = (0.5 + c1) * cos(pi * cutoff);
c3 = (0.5 + c1 - c2) * 0.25;

mA0 = 2 * c3;
mA1 = 2 * 2 * c3;
```

(continues on next page)

(continued from previous page)

```
mA2 = 2 * c3;
mB1 = 2 * -c2;
mB2 = 2 * c1;

//loop
output = mA0*input + mA1*mX1 + mA2*mX2 - mB1*mY1 - mB2*mY2;

mX2 = mX1;
mX1 = input;
mY2 = mY1;
mY1 = output;
```

3.63.1 Comments

- **Date**: 2009-03-05 13:44:24
- **By**: moc.liamg@321tiloen

```
here are coefficients for the hp version.
the br,bp & peak are also easy to calculate:

k = 0.5*res*sin(pi*cutoff);
c1 = 0.5*(1-k)/(1+k);
c2 = (0.5+c1)*cos(pi*cutoff);
c3 = (0.5+c1+c2)*0.25;

a0 = 2*c3;
a1 = -4*c3;
a2 = 2*c3;
b1 = -2*c2;
b2 = 2*c1;

if you wish to create a cascade, use the this:

//----sample loop

//mem: buffer array
//N: number of biquads, n=4 -> 48dB/oct

//set input here

for (i=0;i<N;i++) {
output = a0 * input + a1 * mem[4*i+1] + a2 * mem[4*i+2] - b1 * mem[4*i+3] - b2 *
↪mem[4*i+4];
mem[4*i+2] = mem[4*i+1];
mem[4*i+1] = input;
mem[4*i+4] = mem[4*i+3];
mem[4*i+3] = output;
);

//----sample loop
```

- **Date**: 2009-04-20 11:44:26
- **By**: moc.liamg@321tiloen

```
i've missed a line:

===========================
mem[4*i+3] = output;

//>>> insert here
input = output;
//>>> insert here

);
//----sample loop
===========================

lubomir
```

3.64 Spuc's open source filters

- **Author or source:** moc.liamg@321tiloen
- **Type:** Elliptic, Butterworth, Chebyshev
- **Created:** 2008-10-27 10:14:41

Listing 105: notes

```
http://www.koders.com/info.aspx?c=ProjectInfo&pid=FQLFTV9LA27MF421YKXV224VWH

Spuc has good C++ versions of some classic filter models.

Download full package from:
http://spuc.sourceforge.net
```

3.65 State Variable Filter (Chamberlin version)

- **Author or source:** Hal Chamberlin, "Musical Applications of Microprocessors," 2nd Ed, Hayden Book Company 1985. pp 490-492.
- **Created:** 2003-04-14 18:33:53

Listing 106: code

```
//Input/Output
    I - input sample
    L - lowpass output sample
    B - bandpass output sample
    H - highpass output sample
    N - notch output sample
    F1 - Frequency control parameter
    Q1 - Q control parameter
    D1 - delay associated with bandpass output
    D2 - delay associated with low-pass output

// parameters:
```

(continues on next page)

(continued from previous page)

```
    Q1 = 1/Q
    // where Q1 goes from 2 to 0, ie Q goes from .5 to infinity

    // simple frequency tuning with error towards nyquist
    // F is the filter's center frequency, and Fs is the sampling rate
    F1 = 2*pi*F/Fs

    // ideal tuning:
    F1 = 2 * sin( pi * F / Fs )

// algorithm
    // loop
    L = D2 + F1 * D1
    H = I - L - Q1*D1
    B = F1 * H + D1
    N = H + L

    // store delays
    D1 = B
    D2 = L

    // outputs
    L,H,B,N
```

3.65.1 Comments

- **Date**: 2005-03-21 00:08:03
- **By**: ed.luosfosruoivas@naitsirhC

```
Object Pascal Implementation
----------------------------
-denormal fixed
-not optimized



unit SVFUnit;

interface

type
  TFrequencyTuningMethod= (ftmSimple, ftmIdeal);
  TSVF = class
  private
    fQ1,fQ  : Single;
    fF1,fF  : Single;
    fFS     : Single;
    fD1,fD2 : Single;
    fFTM    : TFrequencyTuningMethod;
    procedure SetFrequency(v:Single);
    procedure SetQ(v:Single);
  public
    constructor Create;
    destructor Destroy; override;
```

(continues on next page)

(continued from previous page)

```
    procedure Process(const I : Single; var L,B,N,H: Single);
    property Frequency: Single read fF write SetFrequency;
    property SampleRate: Single read fFS write fFS;
    property Q: Single read fQ write SetQ;
    property FrequencyTuningMethod: TFrequencyTuningMethod read fFTM write fFTM;
  end;

implementation

uses sysutils;

const kDenorm = 1.0e-24;

constructor TSVF.Create;
begin
 inherited;
 fQ1:=1;
 fF1:=1000;
 fFS:=44100;
 fFTM:=ftmIdeal;
end;

destructor TSVF.Destroy;
begin
 inherited;
end;

procedure TSVF.SetFrequency(v:Single);
begin
 if fFS<=0 then raise exception.create('Sample Rate Error!');
 if v<>fF then
  begin
   fF:=v;
   case fFTM of
    ftmSimple:
     begin
      // simple frequency tuning with error towards nyquist
      // F is the filter's center frequency, and Fs is the sampling rate
      fF1:=2*pi*fF/fFS;
     end;
    ftmIdeal:
     begin
      // ideal tuning:
     fF1:=2*sin(pi*fF/fFS);
     end;
   end;
  end;
end;

procedure TSVF.SetQ(v:Single);
begin
 if v<>fQ then
  begin
   if v>=0.5
    then fQ:=v
    else fQ:=0.5;
   fQ1:=1/fQ;
```

(continues on next page)

(continued from previous page)

```
  end;
end;

procedure TSVF.Process(const I : Single; var L,B,N,H: Single);
begin
 L:=fD2+fF1*fD1-kDenorm;
 H:=I-L-fQ1*fD1;
 B:=fF1*H+fD1;
 N:=H+L;
 // store delays
 fD1:=B;
 fD2:=kDenorm+L;
end;

end.
```

- **Date**: 2005-03-21 14:32:24
- **By**: ed.luosfosruoivas@naitsirhC

```
Ups, there are still denormal bugs in it...
(zu früh gefreut...)
```

3.66 State Variable Filter (Double Sampled, Stable)

- **Author or source:** Andrew Simper
- **Type:** 2 Pole Low, High, Band, Notch and Peaking
- **Created:** 2003-10-11 01:57:00

Listing 107: notes

```
Thanks to Laurent de Soras for the stability limit
and Steffan Diedrichsen for the correct notch output.
```

Listing 108: code

```
input  = input buffer;
output = output buffer;
fs     = sampling frequency;
fc     = cutoff frequency normally something like:
         440.0*pow(2.0, (midi_note - 69.0)/12.0);
res    = resonance 0 to 1;
drive  = internal distortion 0 to 0.1
freq   = 2.0*sin(PI*MIN(0.25, fc/(fs*2)));  // the fs*2 is because it's double sampled
damp   = MIN(2.0*(1.0 - pow(res, 0.25)), MIN(2.0, 2.0/freq - freq*0.5));
notch  = notch output
low    = low pass output
high   = high pass output
band   = band pass output
peak   = peaking output = low - high
--
double sampled svf loop:
for (i=0; i<numSamples; i++)
```

(continues on next page)

(continued from previous page)

```
{
  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;
}
```

3.66.1 Comments

- **Date**: 2004-11-19 13:30:07
- **By**: eb.tenyks@didid

```
Correct me if I'm wrong, but the double-sampling here looks like doubling the input, which is a bad resampling introducing aliasing, followed by an averaging of the 2 outputs, thus filtering that aliasing.
It works, but I think it (the averaging) has the side effect of smoothing up the high freqs in the source material, thus with this filter you can't really fully open it and have the original signal.
At least, it's what seems to happen practically in my tests.

Problem is that this SVF indeed has a crap stability near nyquist, but I can't think of any better way to make it work better, unless you use a better but much more costy upsampling/downsampling.

Anyone confirms?
```

- **Date**: 2004-11-26 09:45:28
- **By**: kd.utd.xaspmak@mj

```
Interesting that this question pops up right now. Lately I have been wondering about the same thing, not so much about the (possibly limited) frequency range, but about stability problems of the filter that I have had (even when using smoothed control signals). The non-linearity introduced by the "drive*band*band*band" factor does not seem to be covered by the stability measurements.
In particular I would like to know, how the filter graphs in http://vellocet.com/dsp/svf/svf-stability.html and http://www-2.cs.cmu.edu/~eli/tmp/svf/stability.png were obtained? Would you like to post the code that generated the stability graph to the musicdsp archive?
For the double-sampling scheme, wouldn't it make more sense to zero-stuff the input signal (that is interleave all input samples with zeros) instead of doubling the samples?
```

- **Date**: 2004-11-27 00:07:09
- **By**: kd.utd.xaspmak@mj

```
Oh, just noticed that Eli's SVF stability measurement code has already been made␣
↪available at http://www-2.cs.cmu.edu/~eli/tmp/svf/
However, I think it is up to him to decide whether he wants to include it in the␣
↪archive or not.
```

- **Date**: 2007-12-13 11:01:38
- **By**: moc.kisuw@kmailliw

```
I was having problems with this filter when DRIVE is set to MAX and Rezonance is set␣
↪to MIN. A quick way to fix it was to make DRIVE*REZO, so when there's no resonance,␣
↪there's no need for DRIVE anyway. That fixed the problem.
```

- **Date**: 2017-05-11 21:39:28
- **By**: moc.liamg@libojyr

```
Here is how I am handling the resampling.  I know from trying this zero padding is␣
↪nasty (terrible noise) without a good filter for downsampling back to base rate.

Below the input is linear interpolated input.  This is a slight improvement on what␣
↪Nigel Redmon suggests here: http://www.earlevel.com/main/2003/03/02/the-digital-
↪state-variable-filter/

Which is simply to tick the filter twice per sample with the same input.  This is␣
↪very similar to above code except that there should not be averaging of the two␣
↪outputs.  You just tick the filter twice with the same input and take the output. ␣
↪The state variables take care of the band limiting.  Remember the aliased terms are␣
↪multiples of the sample rate so they fall on 0 and nyquist frequencies, not really␣
↪having more severe artefacts than what you get from running the filter at base␣
↪sample rate.

For the low pass and bandpass outputs the filter itself performs the band-limiting␣
↪necessary for clean decimation.  Intuitively the high pass output is due to a phase␣
↪cancellation with the dual-integrator loop, so it should be about as clean as the␣
↪LP and BP outputs.  The dual integrator is band-limiting in nature...just some␣
↪thoughts.

//-- Here's the Code --//
//x[i] = input
//x1 = x[i-1] = last input
//Run 1 : Linear interpolate between x[n-1] and x[i]
lpf = lpf + f* bpf;
hpf = 0.5 * g * (x[i] + x1) - lpf - q*bpf;
bpf = f* hpf + bpf;

//Run 2
lpf = lpf + f* bpf;
hpf = g * x[i] - lpf - q*bpf;
bpf = f* hpf + bpf;

x1 = x[i];

// Coefficients on each state variable
// allows for any filter response function possible
// with a biquad filter structure
x[i] = lmix*lpf + hmix*hpf + bmix*bpf;
```

3.67 State variable

- **Author or source:** Effect Deisgn Part 1, Jon Dattorro, J. Audio Eng. Soc., Vol 45, No. 9, 1997 September
- **Type:** 12db resonant low, high or bandpass
- **Created:** 2002-01-17 02:01:50

Listing 109: notes

```
Digital approximation of Chamberlin two-pole low pass. Easy to calculate coefficients,
easy to process algorithm.
```

Listing 110: 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
```

3.67.1 Comments

- **Date**: 2006-01-11 15:06:56
- **By**: nope

```
Wow, great. Sounds good, thanks.
```

- **Date**: 2007-02-13 13:45:02
- **By**: es.aelp@maps.on

```
The variable "high" doesn't have to be initialised, does it? It looks to me like the
↪only variables that need to be kept around between iterations are "low" and "band".
```

- **Date**: 2007-02-13 15:28:30
- **By**: moc.erehwon@ydobon

```
Right. High and notch are calculated from low and band every iteration.
```

- **Date**: 2007-07-18 11:34:21

- **By**: og.on@alal

```
Anyone know what the difference is between q and scale?
```

- **Date**: 2007-07-29 17:17:16
- **By**: moc.liamtoh@rebbadrebbaj

```
"most res: q=1, less: q=0"

Someone correct me if I'm wrong, but isn't that backwards? q=0 is max res, q=1 is min␣
↪res.

q and scale are the same value. What the algorithm is doing is scaling the input the␣
↪higher the resonance is turned up to prevent clipping. One reason why I think 0␣
↪equals max resonance and 1 equals no resonance.

So as q approaches zero, the input is attenuated more and more. In other words, as␣
↪you turn up the resonance, the input is turned down.
```

- **Date**: 2007-11-16 12:58:03
- **By**: rettam.ton@seod

```
scale = sqrt(q);

and

//value (0;100) - for example
q = sqrt(1.0 - atan(sqrt(value)) * 2.0 / PI);
f = frqHz / sampleRate*4.;

uffffffff :)

Now enjoy!
```

- **Date**: 2008-11-29 20:04:47
- **By**: gro.ybbek@bk

```
One drawback of this is that the cutoff frequency can only go up to SR/4 instead of␣
↪SR/2 - but you can easily compensate it by using 2x oversampling, eg. simply␣
↪running this thing twice per sample (apply input interpolation or further output␣
↪filtering ad lib, but from my experience simple linear interpolation of the input␣
↪values (in and (in+lastin)/2) works well enough).
```

- **Date**: 2009-03-05 13:24:35
- **By**: moc.liamg@321tiloen

```
here is the filter with 2x oversampling + some x,y pad functionality to morph between␣
↪states:
like this fx (uses different filter)

http://img299.imageshack.us/img299/4690/statevarible.png

smoothing with interpolation is suggest for most parameters:

//sr: samplerate;
```

(continues on next page)

(continued from previous page)

```
//cutoff: 20 - 20k;
//qvalue: 0 - 100;
//x, y: 0 - 1

q = sqrt(1 - atan(sqrt(qvalue)) * 2 / pi);
scale = sqrt(q);
f = slider1 / sr * 2; // * 2 here instead of 4

//----------sample loop

//set 'input' here

//os x2
for (i=0; i<2; i++) {
low = low + f * band;
high = scale * input - low - q * band;
band = f * high + band;
notch = high + low;
);

//  x,y pad scheme
//
//  high -- notch
//  |           |
//  |           |
//  low ---- band
//
//
// use two pairs

//low, high
pair1 = low * y + high * (1-y);
//band, notch
pair2 = band * y + notch * (1-y);

//out
out = pair2 * x + pair1 * (1-x);

//----------sample loop
```

3.68 Stilson's Moog filter code

- **Author or source:** DFL
- **Type:** 4-pole LP, with fruity BP/HP
- **Created:** 2003-05-15 14:23:51

Listing 111: notes

```
Mind your p's and Q's...

This code was borrowed from Tim Stilson, and rewritten by me into a pd extern (moog~)
available here:
http://www-ccrma.stanford.edu/~dfl/pd/index.htm
```

(continues on next page)

(continued from previous page)

```
I ripped out the essential code and pasted it here...
```

Listing 112: code

```
WARNING: messy code follows ;)

// table to fixup Q in order to remain constant for various pole frequencies, from Tim Stilson's code @ CCRMA (also in CLM distribution)

static float gaintable[199] = { 0.999969, 0.990082, 0.980347, 0.970764, 0.961304, 0.951996, 0.94281, 0.933777, 0.924866, 0.916077, 0.90741, 0.898865, 0.89044
2, 0.882141 , 0.873962, 0.865906, 0.857941, 0.850067, 0.842346, 0.834686, 0.827148, 0.819733, 0.812378, 0.805145, 0.798004, 0.790955, 0.783997, 0.77713, 0.77
0355, 0.763672, 0.75708 , 0.75058, 0.744141, 0.737793, 0.731537, 0.725342, 0.719238, 0.713196, 0.707245, 0.701355, 0.695557, 0.689819, 0.684174, 0.678558, 0.
673035, 0.667572, 0.66217, 0.65686, 0.651581, 0.646393, 0.641235, 0.636169, 0.631134, 0.62619, 0.621277, 0.616425, 0.611633, 0.606903, 0.602234, 0.597626, 0.
593048, 0.588531, 0.584045, 0.579651, 0.575287 , 0.570953, 0.566681, 0.562469, 0.558289, 0.554169, 0.550079, 0.546051, 0.542053, 0.538116, 0.53421, 0.530334,
 0.52652, 0.522736, 0.518982, 0.515289, 0.511627, 0.507996 , 0.504425, 0.500885, 0.497375, 0.493896, 0.490448, 0.487061, 0.483704, 0.480377, 0.477081, 0.4738
16, 0.470581, 0.467377, 0.464203, 0.46109, 0.457977, 0.454926, 0.451874, 0.448883, 0.445892, 0.442932, 0.440033, 0.437134, 0.434265, 0.431427, 0.428619, 0.42
5842, 0.423096, 0.42038, 0.417664, 0.415009, 0.412354, 0.409729, 0.407135, 0.404572, 0.402008, 0.399506, 0.397003, 0.394501, 0.392059, 0.389618, 0.387207, 0.
384827, 0.382477, 0.380127, 0.377808, 0.375488, 0.37323, 0.370972, 0.368713, 0.366516, 0.364319, 0.362122, 0.359985, 0.357849, 0.355713, 0.353607, 0.351532,
0.349457, 0.347412, 0.345398, 0.343384, 0.34137, 0.339417, 0.337463, 0.33551, 0.333588, 0.331665, 0.329773, 0.327911, 0.32605, 0.324188, 0.322357, 0.320557,
0.318756, 0.316986, 0.315216, 0.313446, 0.311707, 0.309998, 0.308289, 0.30658, 0.304901, 0.303223, 0.301575, 0.299927, 0.298309, 0.296692, 0.295074, 0.293488
, 0.291931, 0.290375, 0.288818, 0.287262, 0.285736, 0.284241, 0.282715, 0.28125, 0.279755, 0.27829, 0.276825, 0.275391, 0.273956, 0.272552, 0.271118, 0.26974
5, 0.268341, 0.266968, 0.265594, 0.264252, 0.262909, 0.261566, 0.260223, 0.258911, 0.257599, 0.256317, 0.255035, 0.25375 };

static inline float saturate( float input ) { //clamp without branching
#define _limit 0.95
  float x1 = fabsf( input + _limit );
  float x2 = fabsf( input - _limit );
  return 0.5 * (x1 - x2);
}

static inline float crossfade( float amount, float a, float b ) {
  return (1-amount)*a + amount*b;
}

//code for setting Q
        float ix, ixfrac;
        int ixint;
        ix = x->p * 99;
        ixint = floor( ix );
        ixfrac = ix - ixint;
            Q = resonance * crossfade( ixfrac, gaintable[ ixint + 99 ], gaintable[ixint + 100 ] );
```

(continues on next page)

(continued from previous page)

```

//code for setting pole coefficient based on frequency
    float fc = 2 * frequency / x->srate;
    float x2 = fc*fc;
    float x3 = fc*x2;
    p = -0.69346 * x3 - 0.59515 * x2 + 3.2937 * fc - 1.0072; //cubic fit by DFL, not
↪100% accurate but better than nothing...
}


process loop:
  float state[4], output; //should be global scope / preserved between calls
  int i,pole;
  float temp, input;

  for ( i=0; i < numSamples; i++ ) {
          input = *(in++);
          output = 0.25 * ( input - output ); //negative feedback

          for( pole = 0; pole < 4; pole++) {
                  temp = state[pole];
                  output = saturate( output + p * (output - temp));
                  state[pole] = output;
                  output = saturate( output + temp );
          }
          lowpass = output;
          highpass = input - output;
          bandpass = 3 * x->state[2] - x->lowpass; //got this one from paul kellet
          *out++ = lowpass;

          output *= Q;  //scale the feedback
  }
```

3.68.1 Comments

- **Date**: 2004-06-07 08:01:13
- **By**: ku.oc.sdnuosdionamuh@nhoj

```
What is "x->p" in the code for setting Q?
```

- **Date**: 2004-06-09 07:19:43
- **By**: DFL

```
you should set the frequency first, to get the value of p.
Then use that value to get the normalized Q value.
```

- **Date**: 2004-06-09 12:22:00
- **By**: ku.oc.sdnuosdionamuh@nhoj

```
Ah! That p. Thanks.
```

- **Date**: 2009-01-06 13:35:18
- **By**: soeren.parton->soerenskleinewelt,de

```
Hi!

My Output gets stuck at about 1E-7 even when the input is way below. Is that a␣
↪quantisation problem? Looks as if it´s the saturation´s fault...

Cheers
Sören
```

- **Date**: 2010-10-27 15:51:42
- **By**: ten.reknirdaet@cisum

```
I have not tested, but it looks like gaintable and interpolation can be replaced␣
↪using approx:
 1 / (x * 1.48 + 0.85) - 0.1765

(range 0 -> 1)

Peace
/Martin
```

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

- **Author or source:** Wilfried Welti
- **Created:** 2002-02-10 12:38:01

Listing 113: 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
```

(continues on next page)

(continued from previous page)

```
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))
```

Listing 114: code

```
void convolution(value* in1, value* in2, value* out, value* buffer, int size)
{
  value* temp1 = buffer;
```

(continues on next page)

(continued from previous page)

```
  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];
}
```

3.69.1 Comments

- **Date**: 2003-11-05 11:53:05
- **By**: ed.luosfosruoivas@naitsirhC

```
Here is a delphi translation of the code:

// "value" may be a suitable type like int or float.
// Parameter "size" is the size of the input signals and must be a power of 2.
// out and buffer must point to arrays of size 2*n.

procedure signal_add(in1, in2, ou1 :PValue; Size:Integer);
var i            : Integer;
begin
 for i:=0 to Size-1 do
  begin
   ou1^[i] := in1^[i] + in2^[i];
  end;
end;

procedure signal_sub_from(in1, ou1 :PValue; Size:Integer);
var i       : Integer;
begin
 for i:=0 to Size-1 do
  begin
   ou1^[i] := ou1^[i] - in1^[i];
  end;
end;

procedure signal_add_to(in1, ou1: PValue; Size:Integer);
var i       : Integer;
    po, pi1 : PValue;
begin
 po:=ou1;
 pi1:=in1;
 for i:=0 to Size-1 do
  begin
   ou1^[i] := ou1^[i] + in1^[i];
   Inc(po);
   Inc(pi1);
  end;
end;

procedure convolution(in1, in2, ou1, buffer :PValue; Size:Integer);
var tmp1, tmp2 : PValue;
    i          : Integer;
begin
 tmp1:=Buffer;
 tmp2:=@(Buffer^[(Size div 2)]);

 // clear output.
 for i:=0 to size*2 do ou1^[i]:=0;

 // Break condition for recursion: 1x1 convolution is multiplication.
 if Size = 1 then
  begin
   ou1^[0] := in1^[0] * in2^[0];
```

(continues on next page)

(continued from previous page)

```
   exit;
  end;

 // first calculate (A1 + A2 * z^-n)*(B1 + B2 * z^-n)*z^n
 signal_add(in1, @(in1^[(Size div 2)]), tmp1, Size div 2);
 signal_add(in2, @(in1^[(Size div 2)]), tmp2, Size div 2);
 convolution(tmp1, tmp2, @(ou1^[(Size div 2)]), @(Buffer^[Size]), Size div 2);

 // then add A1*B1 and substract A1*B1*z^n
 convolution(in1, in2, tmp1, @(Buffer^[Size]), Size div 2);
 signal_add_to(ou1, tmp1, size);
 signal_sub_from(@(ou1^[(Size div 2)]), tmp1, size);

 // then add A2*B2 and substract A2*B2*z^-n
 convolution(@(in1^[(Size div 2)]), @(in2^[(Size div 2)]), tmp1, @(Buffer^[Size]),
↪Size div 2);
 signal_add_to(@(ou1^[Size]), tmp1, size);
 signal_sub_from(@(ou1^[Size]), tmp1, size);
end;
```

- **Date**: 2003-11-05 12:19:05
- **By**: ed.luosfosruoivas@naitsirhC

```
Sorry, i forgot the definitions:

type
  Values = Array[0..0] of Single;
  PValue = ^Values;
```

- **Date**: 2004-04-19 19:47:52
- **By**: ed.luosfosruoivas@naitsirhC

```
I have implemented a Suround-Plugin using this Source-Code.
Basicly a FIR-Filter with 512 Taps, bundled with some HRTF's for sourround panning

http://www.savioursofsoul.de/Christian/ITA-HRTF.EXE

(Delphi Sourcecode available on request)
```

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

- **Author or source:** Magnus Jonsson
- **Created:** 2002-09-07 23:23:50

Listing 115: notes

```
[see other code by Wilfried Welti too!]
```

Listing 116: code

```
void mul_brute(float *r, float *a, float *b, int w)
{
```

(continues on next page)

(continued from previous page)

```
    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];
        }
    }
}
```

3.71 Type : LPF 24dB/Oct

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** Bessel Lowpass
- **Created:** 2006-07-28 17:59:18

Listing 117: notes

```
The filter tends to be unsable for low frequencies in the way, that it seems to
↪flutter,
but it never explode. At least here it doesn't.
```

Listing 118: code

```
First calculate the prewarped digital frequency:

K  = tan(Pi * Frequency / Samplerate);
K2 = K*K; // speed improvement

Then calc the digital filter coefficients:

A0 =  ((((105*K + 105)*K + 45)*K + 10)*K + 1);
A1 = -( ((420*K + 210)*K2         - 20)*K - 4)*t;
A2 = -(  (630*K2          - 90)*K2         + 6)*t;
A3 = -( ((420*K - 210)*K2         + 20)*K - 4)*t;
A4 = -((((105*K - 105)*K + 45)*K - 10)*K + 1)*t;

B0 = 105*K2*K2;
B1 = 420*K2*K2;
B2 = 630*K2*K2;
B3 = 420*K2*K2;
B4 = 105*K2*K2;

Per sample calculate:

Output = B0*Input                + State0;
State0 = B1*Input + A1/A0*Output + State1;
State1 = B2*Input + A2/A0*Output + State2;
State2 = B3*Input + A3/A0*Output + State3;
State3 = B4*Input + A4/A0*Output;

For high speed substitude A1/A0 with A1' = A1/A0...
```

3.71.1 Comments

- **Date**: 2006-07-28 22:21:29
- **By**: ed.luosfosruoivas@naitsirhC

```
It turns out, that the filter is only unstable if the coefficient/state precision isn
↪'t high enough. Using double instead of single precision already makes it a lot␣
↪more stable.
```

- **Date**: 2006-10-16 00:42:11
- **By**: ed.luosfosruoivas@naitsirhC

```
Just found out, that I forgot to remove the temporary variable 't'. It was used in my␣
↪code for the speedup. You can simply ignore and delete it.
```

- **Date**: 2009-02-13 14:24:41
- **By**: kd.oohay@eeffocetarak

```
Changing the frequency also seems to affect the volume quite a lot. That can't be␣
↪right? Maybe you should re-post this (and remove the "t" this time)?  ;)
```

- **Date**: 2013-02-13 11:58:19

- **By**: ur.sgn@fez_jd

```
Your filter does not work! Arise overload in the calculation of coefficients A and B␣
↪in the cutoff frequency of approximately 10 khz. At low frequencies, the␣
↪coefficient of the gain increases proportional to the frequency cutoff, which␣
↪causes congestion. I also noticed that the filter on this basis, the package of␣
↪ASIO/DSP - DDspBesselFilter.pas does not work, although Butterworth␣
↪(DDspButterworthFilter.pas) and Chebyshev (DDspChebyshevFilter.pas) work perfectly!
```

3.72 Various Biquad filters

- **Author or source:** JAES, Vol. 31, No. 11, 1983 November
- **Created:** 2002-01-17 02:08:47
- **Linked files:** filters003.txt.

Listing 119: notes

```
(see linkfile)
Filters included are:
presence
shelvelowpass
2polebp
peaknotch
peaknotch2
```

3.72.1 Comments

- **Date**: 2008-10-19 23:02:44
- **By**: moc.liamg@321tiloen

```
I'm kinda stuck trying to figure out the 'pointer' 'structure pointer' loop in the␣
↪presence EQ.

Can someone explain:

...

  *a0 = a2plus1 + alphan*ma2plus1;
  *a1 = 4.0*a;
  *a2 = a2plus1 - alphan*ma2plus1;

  b0 = a2plus1 + alphad*ma2plus1;
  *b2 = a2plus1 - alphad*ma2plus1;

  recipb0 = 1.0/b0;
  *a0 *= recipb0;
  *a1 *= recipb0;
  *a2 *= recipb0;
  *b1 = *a1;
  *b2 *= recipb0;

....
```

(continues on next page)

(continued from previous page)

```
void setfilter_presence(f,freq,boost,bw)
filter *f;
double freq,boost,bw;
{
  presence(freq/(double)SR,boost,bw/(double)SR,
           &f->cx,&f->cx1,&f->cx2,&f->cy1,&f->cy2);
  f->cy1 = -f->cy1;
  f->cy2 = -f->cy2;
}


How can this be translated into something more easy to understand.

Input = ...
Output = ...
```

- **Date**: 2008-10-28 12:15:21
- **By**: moc.liamg@321tiloen

```
Managed to port the presence eq properly. And its sounds great!

Altho I did some changes to some of the code.

changed "d /= mag" to "d = mag"
"bw/srate" to "bw"

There results I got are stable within there parameters:

freq: 3100-18500hz
boost: 0-15db
bw: 0.07-0.40

Really good sound from this filter!
```

3.73 Windowed Sinc FIR Generator

- **Author or source:** Bob Maling
- **Type:** LP, HP, BP, BS
- **Created:** 2005-04-12 20:19:47
- **Linked files:** `wsfir.h`.

Listing 120: notes

```
This code generates FIR coefficients for lowpass, highpass, bandpass, and bandstop␣
↪filters
by windowing a sinc function.

The purpose of this code is to show how windowed sinc filter coefficients are␣
↪generated.
Also shown is how highpass, bandpass, and bandstop filters can be made from lowpass
```

(continues on next page)

(continued from previous page)

```
filters.

Included windows are Blackman, Hanning, and Hamming. Other windows can be added by
following the structure outlined in the opening comments of the header file.
```

3.73.1 Comments

- **Date**: 2005-04-15 00:02:33
- **By**: ed.luosfosruoivas@naitsirhC

```
// Object Pascal Port...

unit SincFIR;

(* Windowed Sinc FIR Generator
   Bob Maling (BobM.DSP@gmail.com)
   Contributed to musicdsp.org Source Code Archive
   Last Updated: April 12, 2005
   Translated to Object Pascal by Christian-W. Budde

   Usage:
   Lowpass:wsfirLP(H, WindowType, CutOff)
   Highpass:wsfirHP(H, WindowType, CutOff)
   Bandpass:wsfirBP(H, WindowType, LowCutOff, HighCutOff)
   Bandstop:wsfirBS(H, WindowType, LowCutOff, HighCutOff)

   where:
   H (TDoubleArray): empty filter coefficient table (SetLength(H,DesiredLength)!)
   WindowType (TWindowType): wtBlackman, wtHanning, wtHamming
   CutOff (double): cutoff (0 < CutOff < 0.5, CutOff = f/fs)
   --> for fs=48kHz and cutoff f=12kHz, CutOff = 12k/48k => 0.25

   LowCutOff (double):low cutoff (0 < CutOff < 0.5, CutOff = f/fs)
   HighCutOff (double):high cutoff (0 < CutOff < 0.5, CutOff = f/fs)

   Windows included here are Blackman, Blackman-Harris, Gaussian, Hanning
   and Hamming.*)

interface

uses Math;

type TDoubleArray = array of Double;
     TWindowType = (wtBlackman, wtHanning, wtHamming, wtBlackmanHarris,
                   wtGaussian); // Window type contstants

// Function prototypes
procedure wsfirLP(var H : TDoubleArray; const WindowType : TWindowType; const CutOff
↪: Double);
procedure wsfirHP(var H : TDoubleArray; const WindowType : TWindowType; const CutOff
↪: Double);
procedure wsfirBS(var H : TDoubleArray; const WindowType : TWindowType; const
↪LowCutOff, HighCutOff : Double);
procedure wsfirBP(var H : TDoubleArray; const WindowType : TWindowType; const
↪LowCutOff, HighCutOff : Double);
```

(continues on next page)

(continued from previous page)

```
procedure genSinc(var Sinc : TDoubleArray; const CutOff : Double);
procedure wGaussian(var W : TDoubleArray);
procedure wBlackmanHarris(var W : TDoubleArray);
procedure wBlackman(var W : TDoubleArray);
procedure wHanning(var W : TDoubleArray);
procedure wHamming(var W : TDoubleArray);

implementation

// Generate lowpass filter
// This is done by generating a sinc function and then windowing it
procedure wsfirLP(var H : TDoubleArray; const WindowType : TWindowType; const CutOff
↪: Double);
begin
 genSinc(H, CutOff);     // 1. Generate Sinc function
 case WindowType of  // 2. Generate Window function -> lowpass filter!
  wtBlackman: wBlackman(H);
  wtHanning: wHanning(H);
  wtHamming: wHamming(H);
  wtGaussian: wGaussian(H);
  wtBlackmanHarris: wBlackmanHarris(H);
 end;
end;

// Generate highpass filter
// This is done by generating a lowpass filter and then spectrally inverting it
procedure wsfirHP(var H : TDoubleArray; const WindowType : TWindowType; const CutOff
↪: Double);
var i : Integer;
begin
 wsfirLP(H, WindowType, CutOff); // 1. Generate lowpass filter

 // 2. Spectrally invert (negate all samples and add 1 to center sample) lowpass
↪filter
 // = delta[n-((N-1)/2)] - h[n], in the time domain
 for i:=0 to Length(H)-1
  do H[i]:=H[i]*-1.0;
 H[(Length(H)-1) div 2]:=H[(Length(H)-1) div 2]+1.0;
end;

// Generate bandstop filter
// This is done by generating a lowpass and highpass filter and adding them
procedure wsfirBS(var H : TDoubleArray; const WindowType : TWindowType; const
↪LowCutOff, HighCutOff : Double);
var i  : Integer;
    H2 : TDoubleArray;
begin
 SetLength(H2,Length(H));

 // 1. Generate lowpass filter at first (low) cutoff frequency
 wsfirLP(H, WindowType, LowCutOff);

 // 2. Generate highpass filter at second (high) cutoff frequency
 wsfirHP(H2, WindowType, HighCutOff);

 // 3. Add the 2 filters together
 for i:=0 to Length(H)-1
```

(continues on next page)

(continued from previous page)

```
  do H[i]:=H[i]+H2[i];

 SetLength(H2,0);
end;

// Generate bandpass filter
// This is done by generating a bandstop filter and spectrally inverting it
procedure wsfirBP(var H : TDoubleArray; const WindowType : TWindowType; const
↪LowCutOff, HighCutOff : Double);
var i : Integer;
begin
 wsfirBS(H, WindowType, LowCutOff, HighCutOff); // 1. Generate bandstop filter

 // 2. Spectrally invert (negate all samples and add 1 to center sample) lowpass
↪filter
 // = delta[n-((N-1)/2)] - h[n], in the time domain
 for i:=0 to Length(H)-1
  do H[i]:=H[i]*-1.0;
 H[(Length(H)-1) div 2]:=H[(Length(H)-1) div 2]+1.0;
end;

// Generate sinc function - used for making lowpass filter
procedure genSinc(var Sinc : TDoubleArray; const Cutoff : Double);
var i,j  : Integer;
    n,k  : Double;
begin
 j:=Length(Sinc)-1;
 k:=1/j;
 // Generate sinc delayed by (N-1)/2
 for i:=0 to j do
  if (i=j div 2)
   then Sinc[i]:=2.0*Cutoff
   else
    begin
     n:=i-j/2.0;
     Sinc[i]:=sin(2.0*PI*Cutoff*n)/(PI*n);
    end;
end;

// Generate window function (Gaussian)
procedure wGaussian(var W : TDoubleArray);
var i,j : Integer;
    k   : Double;
begin
 j:=Length(W)-1;
 k:=1/j;
 for i:=0 to j
  do W[i]:=W[i]*(exp(-5.0/(sqr(j))*(2*i-j)*(2*i-j)));
end;

// Generate window function (Blackman-Harris)
procedure wBlackmanHarris(var W : TDoubleArray);
var i,j : Integer;
    k   : Double;
begin
 j:=Length(W)-1;
 k:=1/j;
```

(continues on next page)

(continued from previous page)

```
 for i:=0 to j
  do W[i]:=W[i]*(0.35875-0.48829*cos(2*PI*(i+0.5)*k)+0.14128*cos(4*PI*(i+0.5)*k)-0.
↪01168*cos(6*PI*(i+0.5)*k));
end;

// Generate window function (Blackman)
procedure wBlackman(var W : TDoubleArray);
var i,j : Integer;
    k   : Double;
begin
 j:=Length(W)-1;
 k:=1/j;
 for i:=0 to j
  do W[i]:=W[i]*(0.42-(0.5*cos(2*PI*i*k))+(0.08*cos(4*PI*i*k)));
end;

// Generate window function (Hanning)
procedure wHanning(var W : TDoubleArray);
var i,j : Integer;
    k   : Double;
begin
 j:=Length(W)-1;
 k:=1/j;
 for i:=0 to j
  do W[i]:=W[i]*(0.5*(1.0-cos(2*PI*i*k)));
end;

// Generate window function (Hamming)
procedure wHamming(var W : TDoubleArray);
var i,j : Integer;
    k   : Double;
begin
 j:=Length(W)-1;
 k:=1/j;
 for i:=0 to j
  do W[i]:=W[i]*(0.54-(0.46*cos(2*PI*i*k)));
end;

end.
```

- **Date**: 2007-01-06 04:23:59
- **By**: uh.etle.fni@yfoocs

```
The Hanning window is often incorrecty referred to as 'Hanning', since it was named␣
↪after a guy called Julius von Hann. So it's more appropriate to call it 'Hann'␣
↪window.
```

- **Date**: 2007-09-02 20:33:31
- **By**: Dave in sinc land

```
I've seen a BASIC version of the same genSinc code.
Shouldn't a 'sin(2.0*Cutoff)' be used when the divide by zero check is 0?
...
  if (i=j div 2)
   then Sinc[i]:=2.0*Cutoff
```

(continues on next page)

(continued from previous page)

```
  else
   begin
    n:=i-j/2.0;
    Sinc[i]:=sin(2.0*PI*Cutoff*n)/(PI*n);
   end;
...
```

- **Date**: 2007-09-02 21:22:35
- **By**: Dave in sinc land

```
Scrap that, I've just FFT'd the response, and it appears to be correct as it was. Hey
↪it just looked wrong o.k.  ;)
```

- **Date**: 2008-09-08 00:57:52
- **By**: moc.liamg@olleocisuor

```
What do I have to do to apply this windowed sinc filter to a signal "Y" for example...
↪ what is the code for this?
Imagine signal "Y" is a stereo song which means that I have Y1 from channel 1 and Y2
↪from channel 2.. help me please as soon as posible because at least i want to know
↪how to aplly the filter to any signal...
```

- **Date**: 2012-06-10 07:57:12
- **By**: ten.xoc@53namhsima

```
Here is a FIR filter that works with up to about 80 coefficients on a Sony PSP
↪running at 280 Megahertz. The multipliers (8191.0 and 32767.0) may need to be
↪incremented by one, or not. You need to call 'fir_filter' with the sample to
↪process in 'sample' and the floating point coefficents in '*coefs' and how long the
↪entire filter is in 'len' and the returned sample is a 16 bit audio (signed short)
↪sample. When it is first run it will normalize the filter.

/* code starts here, remove spaces between the includes */
#include < math.h >
#include < string.h >
#include < stdio.h >
#include < stdlib.h >
              void normalizeCoefs(signed short *firshort, signed short *outfilter,
↪unsigned long len)
{
    // filter gain at uniform frequency intervals
    double *g=NULL;
    double *dtemp=NULL;
    double theta, s, c, sac, sas;
    double gMax = -100.0;
    double sc = 10.0/log(10.0);
    double t = M_PI / len;
    long i=0;
    long n=len-1;
    double normFactor =0;
    g=(double*)malloc((len+1)*sizeof(double));
    dtemp=(double*)malloc((len+1)*sizeof(double));
    for (i=0; i<len; i++)
    {
            dtemp[i]=firshort[i]/32768.0;
```

(continues on next page)

(continued from previous page)

```
    }
    for (i=0; i<len; i++)
    {
            theta = i*t;
            sac = 0.0;
            sas = 0.0;
            long k=0;
            for (k=0; k<len; k++)
            {
                    c =cos(k*theta);
                    s =sin(k*theta);
                    sac += c*(dtemp[k]);
                    sas += s*(dtemp[k]);
            }
            g[i] = sc*log(sac*sac + sas*sas);
            if(g[i]>gMax)
            {
                    gMax=g[i];
            }
    }
    // normalise to 0 dB maximum gain
    for (i=0; i<len; i++)
    {
            g[i] -= gMax;
    }
    // normalise coefficients
    normFactor =0;pow(10.0, -0.05*gMax);
    for (i=0; i<len; i++)
    {
            dtemp[i] *= normFactor;
    }
    n=len-1;
    for (i=0; i<len; i++)
    {
            outfilter[i]=dtemp[n--]*32767.0;
    }
    free(dtemp);
    free(g);
  }

signed short generalFIR(short input, short *coefs, unsigned long len,unsigned char␣
↪resetit)
{

    static signed short *delay_line=NULL;
    static unsigned int first=1;
    static signed long reacc=0;
    unsigned long filtcnt=0;
    static int consumer=1;
    static int producer=0;
    unsigned long tx=0;
    if(resetit==1)
    {
            first=1;
    }
    if(first==1)
    {
```

(continues on next page)

(continued from previous page)

```
            if(delay_line!=NULL)
                    free(delay_line);
            producer=0;
            consumer=1;
            delay_line=(signed short*)malloc(reallen*sizeof(signed short));
            first=0;
    }
    reacc=0;
    delay_line[producer++]=input;
    if(producer>len-1)
    {
            producer-=len;
    }
    int filtptr=consumer;
    for(tx=0;tx<len;tx++)
    {
            reacc+=(delay_line[filtptr++]*coefs[tx]);
            if(filtptr>len-1)
            {
                    filtptr-=len;
            }

    }
    consumer++;
    if(consumer>len-1)
    {
            consumer-=len;
    }
    reacc = reacc >> 15;
    if(reacc<-32768)
            reacc= -32768;
    if(reacc>32767)
            reacc=32767;
    return (signed short)reacc;
}


signed short fir_filter(signed short sample,double *coefs,unsigned long len)
{
    static unsigned char first=1;
    static signed short *newfir=NULL;
    if(first==1)
    {
            unsigned long i=0;
            if(newfir!=NULL)
            {
                    free(newfir);
                    newfir=NULL;
            }
            newfir=(signed short*)malloc((filterorder+1) * sizeof(signed short));
            if(newfir==NULL)
            {
                    return 0;
            }
            for(k=0;k<len;k++)
            {
                    newfir[k]=coefs[k] * 8191.0;
```

(continues on next page)

(continued from previous page)

```
            }
            normalizeCoefs(newfir,newfir,len);
            generalFIR(0,newfir,len,1);
            first=0;
    }
    return generalFIR(sample,newfir,len,0);
}
```

3.74 Zoelzer biquad filters

- **Author or source:** Udo Zoelzer: Digital Audio Signal Processing (John Wiley & Sons, ISBN 0 471 97226 6), Chris Townsend
- **Type:** biquad IIR
- **Created:** 2002-01-17 02:13:13

Listing 121: 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
```

Listing 122: 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

```

(continues on next page)

(continued from previous page)

```
      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
```

3.74.1 Comments

- **Date**: 2002-04-11 10:33:20
- **By**: moc.oohay@bdorezlangis

```
I get a different result for the low-shelf boost with parametric control.
Zolzer builds his lp shelf from a pair of poles and a pair of zeros at:
poles = Q(-1 +- j)
zeros = sqrt(V)Q(-1 +- j)
Where (in the book) Q=1/sqrt(2)
So,
       s^2 + 2sqrt(V)Qs + 2VQ^2
H(s) = ------------------------
           s^2 + 2Qs +2Q^2

If you analyse this in terms of:
H(s) = LPF(s) + 1, it sort of falls apart, as we've gained a zero in the LPF. (as does zolzers)

Then, if we bilinear transform that, we get:

a0= 1 + 2*sqrt(V)*Q*K + 2*V*Q^2*K^2
a1= 2 ( 2*V*Q^2*K^2 - 1 )
a2= 1 - 2*sqrt(V)*Q*K + 2*V*Q^2*K^2
b0= 1 + 2*Q*K + 2*Q^2*K^2
b1= 2 ( 2*Q^2*K^2 - 1)
b2= 1 - 2*Q*K + 2*Q^2*K^2

For:
```

(continues on next page)

(continued from previous page)

```
H(z) = a0z^2 + a1z +a2 / b0z^2 + b1z + b2

Which, i /think/ is right...

Dave.
```

- **Date**: 2002-04-13 08:14:38
- **By**: moc.oohay@bdorezlangis

```
Very sorry, I interpreted Zolzer's s-plane poles as z-plane poles. Too much digital
↪stuff.

After getting back to grips with s-plane maths :) and much graphing to test that it's
↪right, I still get slightly different results.

b0 = 1 + sqrt(V)*K/Q + V*K^2
b1 = 2*(V*K^2 - 1)
b2 = 1 - sqrt(V)*K/Q + V*K^2
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2
The way the filter works is to have two poles on a unit circle around the origin in
↪the s-plane, and two zeros that start at the poles at V0=1, and move outwards. The
↪above co-efficients represent that. Chris's original results put the poles in the
↪right place, but put the zeros at the location where the poles would be if they
↪were butterworth, and move out from there - yielding some rather strange results...

But I've graphed that extensively, and it works fine now :)

Dave.
```

- **Date**: 2005-01-17 07:21:21
- **By**: asynth(at)io(dot)com

```
Once you divide through by a0, the Zoelzer LPF gives coefficient values that are _
↪identical_ to the RBJ LPF.
This one is a cheaper formulation because there is only one transcendental function
↪call (tan) instead of two (sin, cos) for RBJ.
-- james mccartney
```

- **Date**: 2005-01-18 02:33:04
- **By**: asynth(at)io(dot)com

```
Actually, sin and cos are pretty cheap when done via taylor series, so I take that
↪last bit back.
-- james mccartney
```

- **Date**: 2005-05-04 14:23:01
- **By**: se.arret@htrehgraknu

```
Anybody know the formulation for Band Pass, High Pass and High shelf ?
```

- **Date**: 2005-05-04 16:57:08
- **By**: ed.luosfosruoivas@naitsirhC

```
Have a look at tobybear's Filter Explorer: http://www.tobybear.de/p_filterexp.html

Usually you can derivate a highpass from a lowpass and vice versa.
```

- **Date**: 2005-05-05 11:15:46
- **By**: se.arret@htrehgraknu

```
Thanks Christian, lots of things solved now !!

Unfortunately, Bandpass continues missing. I don't know if it's really posible to␣
↪obtain a Bandpass filter out of this ( my filters math knowlegde isn't so deep),␣
↪but i asked for it because would be nice to have the complete set of Zoeltzer␣
↪filters

I supose thta YOU can derive one from another as you stated, but this is not my case.␣
↪Anyway, lots of thanks for your help
```

- **Date**: 2005-05-20 21:04:10
- **By**: moc.noitanigamioidua@jbr

```
>Actually, sin and cos are pretty cheap when done via taylor series, so I take that␣
↪last bit back
-----------------------------------------------

also, James, the sin() and cos() are less of a problem for implementing in a fixed-
↪point context.  tan() is a bitch.

r b-j
```

- **Date**: 2006-06-27 17:32:53
- **By**: uh.etle.fni@yfoocs

```
Highpass version:

HPF:
b0 = 1 - K^2
b1 = -2*K^2
b2 = 1 - K^2
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2

Bandpass version:

BPF1 (peak gain = Q):
b0 = K
b1 = 0
b2 = -K
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2

BPF2 (peak gain = zero):

b0 = K/Q
```

(continues on next page)

(continued from previous page)

```
b1 = 0
b2 = -K/Q
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2

Couldn't figure out the notch coeffs yet...

-- peter schoffhauzer
```

- **Date**: 2006-06-28 00:07:25
- **By**: uh.etle.fni@yfoocs

```
Got the notch too finally ;)

Notch

b0 = 1 + K^2
b1 = 2*(K^2 - 1)
b2 = 1 + K^2
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2

The HPF seems to have an error in the previous post. The correct HPF version:

HPF:
b0 = 1 + K/Q
b1 = -2
b2 = 1 - K/Q
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2

Hopefully it works now. Anyone confirms?
The set is complete now. Happy coding :)

-- peter schoffhauzer
```

- **Date**: 2006-06-28 20:22:23
- **By**: uh.etle.fni@yfoocs

```
For sake of completeness ;)

Allpass:

b0 = 1 - K/Q + K^2
b1 = 2*(K^2 - 1)
b2 = 1
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2

-- peter schoffhauzer
```

- **Date**: 2006-06-30 00:59:04

- **By**: ed.luosfosruoivas@naitsirhC

```
What was wrong with the first version:

HPF:
b0 = 1 - K^2
b1 = -2 (!!)
b2 = 1 - K^2
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2

you only have to delete K^2. In the other version the cutoff frequency depends on the␣
↪Q!
```

- **Date**: 2006-06-30 01:50:50
- **By**: ed.luosfosruoivas@naitsirhC

```
Also the Allpass should be symmetrical:

b0 = 1 - K/Q + K^2
b1 = 2*(K^2 - 1)
b2 = 1 + K/Q + K^2  (!!)
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2

If you divide by a0 (to reduce a coefficient) b2 will get 1 of course.
```

- **Date**: 2006-06-30 11:33:24
- **By**: uh.etle.fni@yfoocs

```
Ah, thanks for the allpass correction! I used TobyBear Filter Explorer, where I see␣
↪only 5 coeffs instead of six, that was the source of confusion.

However, the highpass is still not perfect. In my 2nd version, the cutoff is not␣
↪dependent of Q, because the cutoff is determined by the pole positions, which are␣
↪set by a1 and a2. Instead, as the zero positions change according to Q, the cutoff␣
↪slope varies. So it has an interesting behaviour, for low Qs it has a 6dB/Oct slope,
↪ for infinite resonance, the slope becomes 12dB/Oct.

However, with your suggested HPF version, I got only a strange highshelf-like filter.␣
↪So here is my 3rd version, which I hope works fine:

HPF:
b0 = 1
b1 = -2
b2 = 1
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2

Quite simple isn't it? ;)

Cheers
Peter
```

- **Date**: 2006-06-30 12:25:36

- **By**: uh.etle.fni@yfoocs

```
james mccartney:

Tan can also be approximated using Taylor series (approx sin and cos with Taylor,
↪then tan(x)=sin(x)/cos(x)) well, there's a heavy division that you can't get rid of.
↪.. well, that's not true in all cases. The advantage of tan() is that you can use
↪that tan(x) ~= x when x is small. So you can get coefficients without any
↪transcendental functions for low and middle frequencies.

Peter
```

- **Date**: 2006-06-30 13:20:44
- **By**: rf.eerf@navi.neflow

```
I think it is possible to get a Taylor serie of the tan() function ? And it is
↪possible to do a polynomial division of the sin & cos series to get rid of the
↪division, you get the same thing...
```

- **Date**: 2006-07-01 00:20:28
- **By**: uh.etle.fni@yfoocs

```
Yes, there is a Taylor serie for tan(x), but near pi/2, it converges very slowly, so
↪high frequencies is a problem again.

Let's suppose you approximate sin(x) with x-x^3/6+x^5/120, and cos(x) with 1-x^2/2+x^
↪4/24.

So tan(x) would be

 x - x^3/6 + x^5/120
---------------------
 1 - x^2/2 + x^4/24

How do you do a polynomial division for that?
```

- **Date**: 2006-07-10 19:21:54
- **By**: ed.luosfosruoivas@naitsirhC

```
Here's also a highshelf filters for completeness

K:=tan(fW0*0.5);
t2:=;
t3:=K*fQ; t6:=(V);
t5:=sqrt(2*V)*K;
t1:=1/;

b0 =   (K*K + sqrt(2*V)*K + V);
b1 = 2*(K*K               - V);
b2 =   (K*K - sqrt(2*V)*K + V);
a0 =   (K*K + K*fQ + 1)
a1 =-2*(K*K        - 1);
a2 =-  (K*K - K*fQ + 1);
```

CHAPTER 4

Effects

4.1 2 Wave shaping things

- **Author or source:** Frederic Petrot
- **Created:** 2002-03-20 01:12:01

Listing 1: 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
```

Listing 2: code

```
y1 = (max>>1) * atan(k * x/max)

y2 = max * th(x/max)
```

4.1.1 Comments

- **Date**: 2006-06-26 01:12:05
- **By**: ten.labolgcbs@rohtlabi

```
Why are you calling decompiled script code?BALTHOR
```

- **Date**: 2013-01-18 02:20:59
- **By**: moc.liamtoh@niffumtohrepus

```
Yeah, atan & tanh, and really any sigmoid function, can create decent overdrive sound␣
↪if
oversampled and eq'ed properly. I've used them in some of my modelers w/ good results.
↪ For
a more realistic sound, two half-wave soft clippers in series will add duty-cycle␣
↪modulation
and a transfer curve similar to 3/2 tube power curve. Something like:
if(x < 0) y = -A * tanh(B * x); followed immediately by: if(y >= 0) y = -A * tanh(B *␣
↪y);
Don't forget to invert each output (-A * tanh). Coefficients A & B are left to the␣
↪designer.
I got this technique after reading a paper discussing a hardware implementation of␣
↪this type
of circuit used in Carvin amps, here: http://www.trueaudio.com/at_eetjlm.htm␣
↪(original link at
www.simulanalog.org)
```

4.2 Alien Wah

- **Author or source:** Nasca Octavian Paul (or.liame@acsanluap)
- **Created:** 2002-02-10 12:51:57
- **Linked files:** `alienwah.c`.

Listing 3: notes

```
I found this algoritm by "playing around" with complex numbers. Please email me your
opinions about it.

Paul.
```

4.3 Band Limited PWM Generator

- **Author or source:** rf.liamtoh@57eninres_luap
- **Type:** PWM generator
- **Created:** 2006-05-11 19:09:04

Listing 4: notes

```
This is a commented and deobfuscated version of my 1st April fish. It is based on a
tutorial code by Thierry Rochebois. I just translated and added comments.

Regards,

Paul Sernine.
```

Listing 5: code

```
// SelfPMpwm.cpp

// Antialised PWM oscillator

// Based on a tutorial code by Thierry Rochebois (98).
// Itself inspired by US patent 4249447 by Norio Tomisawa (81).
// Comments added/translated by P.Sernine (06).

// This program generates a 44100Hz-raw-PCM-mono-wavefile.
// It is based on Tomisawa's self-phase-modulated sinewave generators.
// Rochebois uses a common phase accumulator to feed two half-Tomisawa-
// oscillators. Each half-Tomisawa-oscillator generates a bandlimited
// sawtooth (band limitation depending on the feedback coeff B).
// These half oscillators are phase offseted according to the desired
// pulse width. They are finally combined to obtain the PW signal.
// Note: the anti-"hunting" filter is a critical feature of a good
// implementation of Tomisawa's method.
#include <math.h>
#include <stdio.h>
const float pi=3.14159265359f;
int main()
{
  float freq,dphi; //!< frequency (Hz) and phase increment(rad/sample)
  float dphif=0;   //!< filtered (anti click) phase increment
  float phi=-pi;   //!< phase
  float Y0=0,Y1=0; //!< feedback memories
  float PW=pi;     //!< pulse width ]0,2pi[
  float B=2.3f;    //!< feedback coef
  FILE *f=fopen("SelfPMpwm.pcm","wb");
  // séquence ('a'=mi=E)
  // you can edit this if you prefer another melody.
  static char seq[]="aiakahiafahadfaiakahiahafahadf"; //!< sequence
  int note=sizeof(seq)-2;  //!< note number in the sequence
  int octave=0;     //!< octave number
  float env,envf=0; //!< envelopped and filtered envelopped
  for(int ns=0;ns<8*(sizeof(seq)-1)*44100/6;ns++)
  {
//waveform control --------------------------------------------------
    //Frequency
    //freq=27.5f*powf(2.0f,8*ns/(8*30*44100.0f/6)); //sweep
    freq=27.5f*powf(2.0f,octave+(seq[note]-'a'-5)/12.0f);
    //freq*=(1.0f+0.01f*sinf(ns*0.0015f));        //vibrato
    dphi=freq*(pi/22050.0f);   // phase increment
    dphif+=0.1f*(dphi-dphif);
    //notes and enveloppe trigger
    if((ns%(44100/6))==0)
    {
      note++;
      if(note>=(sizeof(seq)-1))// sequence loop
      {
        note=0;
        octave++;
      }
      env=1; //env set
      //PW=pi*(0.4+0.5f*(rand()%1000)/1000.0f); //random PW
```

(continues on next page)

(continued from previous page)

```
    }
    env*=0.9998f;                 // exp enveloppe
    envf+=0.1f*(env-envf);        // de-clicked enveloppe
    B=1.0f;                       // feedback coefficient
    //try this for a nice bass sound:
    //B*=envf*envf;                 // feedback controlled by enveloppe
    B*=2.3f*(1-0.0001f*freq);     // feedback limitation
    if(B<0)
      B=0;

//waveform generation -----------------------------------------------
    //Common phase
    phi+=dphif;                    // phase increment
    if(phi>=pi)
      phi-=2*pi;                  // phase wrapping

    // "phase"    half Tomisawa generator 0
    // B*Y0 -> self phase modulation
    float out0=cosf(phi+B*Y0); // half-output 0
    Y0=0.5f*(out0+Y0);            // anti "hunting" filter

    // "phase+PW" half Tomisawa generator 1
    // B*Y1 -> self phase modulation
    // PW   -> phase offset
    float out1=cosf(phi+B*Y1+PW); // half-output 1
    Y1=0.5f*(out1+Y1);               // anti "hunting" filter

    // combination, enveloppe and output
    short s=short(15000.0f*(out0-out1)*envf);
    fwrite(&s,2,1,f);             // file output
  }
  fclose(f);
  return 0;
}
```

4.3.1 Comments

- **Date**: 2006-05-23 09:31:44
- **By**: —

```
Did anyone try this?

How is the antialiasing compared to applying phaserror between two oscs in zerocross,␣
↪one aliasing
the other not (but pitcherror).

Best Regards,
Arif Ove Karlsen.
```

- **Date**: 2010-02-04 08:37:45
- **By**: ude.notecnirp.inmula@esornep

```
The implementation certainly produces aliased waveforms -- they are glaring on a
↪spectrogram
at -60dB and faint at -30dB.  But it is a remarkably efficient algorithm. The
↪aliasing can be
mitigated somewhat by using a smaller feedback coefficient.
```

4.4 Bit quantization/reduction effect

- **Author or source:** Jon Watte
- **Type:** Bit-level noise-generating effect
- **Created:** 2002-04-12 13:53:03

Listing 6: 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.
```

Listing 7: code

```
short keep_bits_from_16( short input, int keepBits ) {
  return (input & (-1 << (16-keepBits)));
}
```

4.4.1 Comments

- **Date**: 2005-05-30 23:01:56
- **By**: moc.liamg@codlli

```
//I add some code to prevent offset.

// Code :
short keep_bits_from_16( short input, int keepBits ) {
    short prevent_offset = static_cast<unsigned short>(-1) >> keepBits+1;
    input &= (-1 << (16-keepBits)));
    return input + prevent_offset
}
```

4.5 Class for waveguide/delay effects

- **Author or source:** moc.xinortceletrams@urugra
- **Type:** IIR filter
- **Created:** 2002-04-23 06:46:34

Listing 8: 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)
```

Listing 9: 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);
```

(continues on next page)

(continued from previous page)

```
            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;
};
```

4.6 Compressor

- **Author or source:** ur.liam@cnihsalf
- **Type:** Hardknee compressor with RMS look-ahead envelope calculation and adjustable attack/decay
- **Created:** 2004-05-26 16:02:59

Listing 10: notes

```
RMS is a true way to estimate _musical_ signal energy,
our ears behaves in a same way.

to making all it work,
try this values (as is, routine accepts percents and milliseconds) for first time:

threshold = 50%
slope = 50%
RMS window width = 1 ms
lookahead = 3 ms
attack time = 0.1 ms
release time = 300 ms

This code can be significantly improved in speed by
changing RMS calculation loop to 'running summ'
(keeping the summ in 'window' -
adding next newest sample and subtracting oldest on each step)
```

Listing 11: code

```
void compress
    (
        float*  wav_in,     // signal
        int     n,          // N samples
        double  threshold,  // threshold (percents)
        double  slope,      // slope angle (percents)
        int     sr,         // sample rate (smp/sec)
        double  tla,        // lookahead  (ms)
        double  twnd,       // window time (ms)
        double  tatt,       // attack time  (ms)
        double  trel        // release time (ms)
    )
{
    typedef float   stereodata[2];
    stereodata*     wav = (stereodata*) wav_in; // our stereo signal
    threshold *= 0.01;          // threshold to unity (0...1)
    slope *= 0.01;              // slope to unity
    tla *= 1e-3;                // lookahead time to seconds
    twnd *= 1e-3;               // window time to seconds
    tatt *= 1e-3;               // attack time to seconds
    trel *= 1e-3;               // release time to seconds

    // attack and release "per sample decay"
    double  att = (tatt == 0.0) ? (0.0) : exp (-1.0 / (sr * tatt));
    double  rel = (trel == 0.0) ? (0.0) : exp (-1.0 / (sr * trel));

    // envelope
    double  env = 0.0;

    // sample offset to lookahead wnd start
    int     lhsmp = (int) (sr * tla);

    // samples count in lookahead window
    int     nrms = (int) (sr * twnd);

    // for each sample...
    for (int i = 0; i < n; ++i)
    {
        // now compute RMS
        double  summ = 0;

        // for each sample in window
        for (int j = 0; j < nrms; ++j)
        {
            int     lki = i + j + lhsmp;
            double  smp;

            // if we in bounds of signal?
            // if so, convert to mono
            if (lki < n)
                smp = 0.5 * wav[lki][0] + 0.5 * wav[lki][1];
            else
                smp = 0.0;      // if we out of bounds we just get zero in smp

            summ += smp * smp;  // square em..
        }
```

(continues on next page)

(continued from previous page)

```

        double  rms = sqrt (summ / nrms);    // root-mean-square

        // dynamic selection: attack or release?
        double  theta = rms > env ? att : rel;

        // smoothing with capacitor, envelope extraction...
        // here be aware of pIV denormal numbers glitch
        env = (1.0 - theta) * rms + theta * env;

        // the very easy hard knee 1:N compressor
        double  gain = 1.0;
        if (env > threshold)
            gain = gain - (env - threshold) * slope;

        // result - two hard kneed compressed channels...
        float  leftchannel = wav[i][0] * gain;
        float  rightchannel = wav[i][1] * gain;
    }
}
```

4.6.1 Comments

- **Date**: 2004-06-10 21:31:18
- **By**: moc.regnimmu@psd-cisum

```
My comments:

A rectangular window is not physical. It would make more physical sense, and be a lot␣
↪cheaper, to
use a 1-pole low pass filter to do the RMS averaging. A 1-pole filter means you can␣
↪lose the bounds
checks in the RMS calculation.

It does not make sense to convert to mono before squaring, you should square each␣
↪channel
separately and then add them together to get the total signal power.

You might also consider whether you even need any filtering other than the attack/
↪release filter.
You could modify the attack/release rates appropriately, place the sqrt after the␣
↪attack/release,
and lose the rms averager entirely.

I don't think your compressor actually approaches a linear slope in the decibel␣
↪domain. You need
a gain law more like

double gain = exp(max(0.0,log(env)-log(thresh))*slope);

Sincerely,
     Frederick Umminger
```

- **Date**: 2004-07-30 05:31:36

- **By**: moc.liamg@noteex

```
To sum up (and maybe augment) the RMS calculation method, this question and answer␣
↪may be of use...

**********

music-dsp@shoko.calarts.edu writes:
I am looking at gain processing algorithms. I haven't found much in the way of␣
↪reference material on
this, any pointers? In the level detection code, if one is doing peak detection, how␣
↪many samples
does one generally average over (if at all)? What kind of window size for RMS level␣
↪detection?
Is the RMS level detection generally the same algo. as peak, but with a bigger window?

The peak detector can be easily implemented as a one-pole low pass, you just have␣
↪modify it,
so that it tracks the peaks and gently falls down afterwards. RMS detection is done␣
↪squaring
the input signal, averaging with a lowpass and taking the root afterwards.

Hope this helps.

Kind regards

Steffan Diedrichsen

DSP developer

emagic GmbH

**********

I found the thread by searching old [music-dsp] forum posts. Hope it helps.
```

- **Date**: 2006-11-07 01:52:17
- **By**: gro.esabnaeco@euarg

```
How would you use a 1-pole lowpass filter to do RMS averaging? How do you pick a␣
↪coefficient to use?
```

- **Date**: 2006-11-09 01:19:41
- **By**: uh.etle.fni@yfoocs

```
Use x = exp(-1/d), where d is the time constant in samples. A 1 pole IIR filter has␣
↪an infinite
impulse response, so instead of window width, this coeff determines the time when the␣
↪impulse
response reaches 36.8% of the original value.

Coeffs:
a0 = 1.0-x;
b1 = -x;

Loop:
out = a0*in - b1*tmp;
```

(continues on next page)

(continued from previous page)

```
tmp = out;

-- peter schoffhauzer
```

- **Date**: 2008-11-20 08:30:28
- **By**: moc.361@oatuxt

```
I am looking at gain processing algorithms£°
There are too such sentences :
double att = (tatt == 0.0) ? (0.0) : exp (-1.0 / (sr * tatt));
double rel = (trel == 0.0) ? (0.0) : exp (-1.0 / (sr * trel));

can you tell me something about the exp (-1.0 / (sr * tatt))?

New day ~~
thanks
```

- **Date**: 2010-04-28 15:12:47
- **By**: moc.liamg@enin.kap

```
This is a useful reference, however the RMS calculations are pretty dodgy. Firstly␣
↪there is a
bug where is calculates the number of samples to use:

int sr, // sample rate (smp/sec)
...
double twnd, // window time (ms)
...
// samples count in lookahead window
int nrms = (int) (sr * twnd);

The units are mixed when calculating the number of samples in the RMS window. The␣
↪window time needs
to be converted to seconds before multiplying by the sample rate.

As others have mentioned the RMS calculation is also really expensive, and in my␣
↪tests I found it
was pretty innacurate unless you use a LOT of samples (you basically need a (sample␣
↪rate)/2 window
of samples in your RMS calculation to accurately measure the power of all␣
↪frequencies).

I ended up using the 1 pole low pass filter approach suggested here, and it is a good␣
↪cheap
approximation of power. I did, however, end up mulitplying it by root(2) (the RMS of␣
↪a sine wave,
which seemed like a reasonable normalisation factor) in order to get it between 0 and␣
↪1, which is
a more useful range.

Another slightly more accurate way to caculate the RMS without iterating over and␣
↪entire window
for each sample is to keep a running total of the squared sums of samples.

for( i = 0; i < NumSamples; ++i )
```

(continues on next page)

(continued from previous page)

```
{
  NewSample = Sample[i];
  OldSample = Sample[i - RMSWindowSize];

  SquaredSum = SquaredSum + NewSample * NewSample;
  SquaredSum = SquaredSum - OldSample * OldSample;

  RMS = sqrt( SquaredSum / RMSWindowSize );

  // etc...
}

Calculating the power in the signal is definately the awkward part of this DSP!
```

4.7 Decimator

- **Author or source:** ed.bew@raeybot
- **Type:** Bit-reducer and sample&hold unit
- **Created:** 2002-11-25 18:13:49

Listing 12: notes

```
This is a simple bit and sample rate reduction code, maybe some of you can use it. The
parameters are bits (1..32) and rate (0..1, 1 is the original samplerate).
Call the function like this:
y=decimate(x);

A VST plugin implementing this algorithm (with full Delphi source code included) can␣
↪be
downloaded from here:
http://tobybear.phreque.com/decimator.zip

Comments/suggestions/improvements are welcome, send them to: tobybear@web.de
```

Listing 13: code

```
// bits: 1..32
// rate: 0..1 (1 is original samplerate)

********** Pascal source **********
var m:longint;
    y,cnt,rate:single;

// call this at least once before calling
// decimate() the first time
procedure setparams(bits:integer;shrate:single);
begin
 m:=1 shl (bits-1);
 cnt:=1;
 rate:=shrate;
end;

```

(continues on next page)

(continued from previous page)

```
function decimate(i:single):single;
begin
 cnt:=cnt+rate;
 if (cnt>1) then
 begin
  cnt:=cnt-1;
  y:=round(i*m)/m;
 end;
 result:=y;
end;
```

Listing 14: code

```
int bits=16;
float rate=0.5;

long int m=1<<(bits-1);
float y=0,cnt=0;

float decimate(float i)
{
 cnt+=rate;
 if (cnt>=1)
 {
  cnt-=1;
  y=(long int)(i*m)/(float)m;
 }
 return y;
}
```

4.7.1 Comments

- **Date**: 2002-12-03 12:44:42
- **By**: moc.noicratse@ajelak

```
Nothing wrong with that, but you can also do fractional-bit-depth decimations,␣
↪allowing
smooth degradation from high bit depth to
low and back:
---------------------------------------

// something like this -- this is
// completely off the top of my head
// precalculate the quantization level
float bits; // effective bit depth
float quantum = powf( 2.0f, bits );

// per sample
y = floorf( x * quantum ) / quantum;

---------------------------------------
```

- **Date**: 2003-02-14 20:04:36
- **By**: es.yarps@fek.rd

```
it looks to me like the c-line

long int m=1<<(bits-1);

doesnt give the correct number of quantisation levels if the number of levels is
↪defined as
2^bits. if bits=2 for instance, the above code line returns a bit pattern of 10 (3)
↪and not
11 (2^2) like one would expect.

please, do correct me if im wrong.

/heatrof
```

4.8 Delay time calculation for reverberation

- **Author or source:** Andy Mucho
- **Created:** 2002-01-17 02:19:54

Listing 15: 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..
```

Listing 16: code

```c
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
```

(continues on next page)

(continued from previous page)

```
tell you right now..
*/
```

4.9 Dynamic convolution

- **Author or source:** Risto Holopainen
- **Type:** a naive implementation in C++
- **Created:** 2005-06-05 22:05:19

Listing 17: notes

```
This class illustrates the use of dynamic convolution with a set of IR:s consisting of
exponentially damped sinusoids with glissando. There's lots of things to improve for
efficiency.
```

Listing 18: code

```
#include <cmath>

class dynaconv
{
  public:
// sr=sample rate, cf=resonance frequency,
// dp=frq sweep or nonlinearity amount
            dynaconv(const int sr, float cf, float dp);
            double operator()(double);

    private:
// steps: number of amplitude regions, L: length of impulse response
            enum {steps=258, dv=steps-2, L=200};
            double x[L];
            double h[steps][L];
            int S[L];
            double conv(double *x, int d);
};



dynaconv::dynaconv(const int sr, float cfr, float dp)
{
    for(int i=0; i<L; i++)
            x[i] = S[i] = 0;

    double sc = 6.0/L;
    double frq = twopi*cfr/sr;

// IR's initialised here.
// h[0] holds the IR for samples with lowest amplitude.
    for(int k=0; k<steps; k++)
            {
            double sum = 0;
            double theta=0;
```

(continues on next page)

(continued from previous page)

```
            double w;
            for(int i=0; i<L; i++)
                    {
    // IR of exp. decaying sinusoid with glissando
                    h[k][i] = sin(theta)*exp(-sc*i);
                    w = (double)i/L;
                    theta += frq*(1 + dp*w*(k - 0.4*steps)/steps);
                    sum += fabs(h[k][i]);
                    }

            double norm = 1.0/sum;
            for(int i=0; i<L; i++)
                    h[k][i] *= norm;
            }
}

double dynaconv::operator()(double in)
{
    double A = fabs(in);
    double a, b, w, y;
    int sel = int(dv*A);

    for(int j=L-1; j>0; j--)
            {
            x[j] = x[j-1];
            S[j] = S[j-1];
            }
    x[0] = in;
    S[0] = sel;

    if(sel == 0)
            y = conv(x, 0);

    else if(sel > 0)
            {
            a = conv(x, 0);
            b = conv(x, 1);
            w = dv*A - sel;
            y = w*a + (1-w)*b;
            }

    return y;
}

double dynaconv::conv(double *x, int d)
{
    double y=0;
    for(int i=0; i<L; i++)
            y += x[i] * h[ S[i]+d ][i];

    return y;
}
```

4.9.1 Comments

- **Date**: 2005-06-06 08:26:12

- **By**: ed.luosfosruoivas@naitsirhC

```
You can speed things up by:

a) rewriting the "double dynaconv::conv(double *x, int d)" function using Assembler,␣
↪SSE and 3DNow
    routines.

b) instead of this

"else if(sel > 0)
{
a = conv(x, 0);
b = conv(x, 1);
w = dv*A - sel;
y = w*a + (1-w)*b;
}"

you can create a temp IR by fading the two impulse responses before convolution. Then␣
↪you'll only
    need ONE CPU-expensive-convolution.

c) this one only works with the upper half wave!

d) only nonlinear components can be modeled. For time-variant modeling (compressor/
↪limiter) you'll
    need more than this.

e) the algo is proteced by a patent. But it's easy to find more efficient ways, which␣
↪aren't
    protected by the patent.

With my implementation i can fold up to 4000 Samples (IR) in realtime on my machine.
```

- **Date**: 2005-07-20 13:52:49
- **By**: d.tniop@noitcerroc

```
Correction to d:

d) only time invariant nonlinear components can be modeled; and then adequate memory␣
↪must be used.
    Compressors/Limiters can be modelled, but the memory requirements will be␣
↪somewhat frightening.
    Time-variant systems, such as flangers, phasors, and sub-harmonic generators (i.e.
↪ anything
    with an internal clock) will need more than this.
```

4.10 ECE320 project: Reverberation w/ parameter control from PC

- **Author or source:** Brahim Hamadicharef (project by Hua Zheng and Shobhit Jain)
- **Created:** 2002-05-24 13:34:45
- **Linked files:** `rev.txt`.

Listing 19: 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)
```

4.11 Early echo's with image-mirror technique

- **Author or source:** Donald Schulz
- **Created:** 2002-02-11 17:35:48
- **Linked files:** early_echo.c.
- **Linked files:** early_echo_eng.c.

Listing 20: 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.
```

4.12 Fold back distortion

- **Author or source:** ed.bpu@eriflleh
- **Type:** distortion
- **Created:** 2005-05-28 19:11:06

Listing 21: notes

```
a simple fold-back distortion filter.
if the signal exceeds the given threshold-level, it mirrors at the positive/negative
threshold-border as long as the singal lies in the legal range (-threshold..
↪+threshold).
there is no range limit, so inputs doesn't need to be in -1..+1 scale.
threshold should be >0
depending on use (low thresholds) it makes sense to rescale the input to full␣
↪amplitude

performs approximately the following code
(just without the loop)

while (in>threshold || in<-threshold)
{
  // mirror at positive threshold
  if (in>threshold) in= threshold - (in-threshold);
  // mirror at negative threshold
  if (in<-threshold) in= -threshold + (-threshold-in);
}
```

Listing 22: code

```
float foldback(float in, float threshold)
{
  if (in>threshold || in<-threshold)
  {
    in= fabs(fabs(fmod(in - threshold, threshold*4)) - threshold*2) - threshold;
  }
  return in;
}
```

4.13 Guitar feedback

- **Author or source:** Sean Costello
- **Created:** 2002-02-10 20:01:07

Listing 23: 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):
```

Listing 24: 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.
```

4.13.1 Comments

- **Date**: 2016-06-10 19:59:36
- **By**: moc.liamg@1a7102egroj

```
'NO esta completo falta algo, no se, si es el filtro biquad

Public Function karplustrong_b1(PNumSamples As Long) As Integer()
Dim x() As Single
Dim y() As Single
Dim n As Long
Dim Num As Long
```

(continues on next page)

(continued from previous page)

```
Dim ArrResp() As Integer
Dim C As Single
Dim Fs As Long
Dim fP As Single
Dim amplitud As Long
Dim i As Long
Dim valor As Single
Dim Ind1 As Long
Dim Ind2 As Long
Dim suma As Single
Dim media As Single
Dim valorsigno As Single


    Fs = 44100
    fP = 400 '400hz pitch frequency

    'Num = (Fs / Fp)


    ReDim x(PNumSamples)
    ReDim y(PNumSamples)





    'generar numeros aleatorios rango 0 a 1
    'iniciar numeros aleatorios
    Randomize
    suma = 0
    For i = 0 To PNumSamples
        valor = Rnd * 1 - 0.5

        'valorsigno = Rnd * 1
        'If valorsigno > 0.5 Then valor = -valor

        x(i) = valor
        suma = suma + valor
    Next

    media = suma / PNumSamples
    'media = 1

    'calcular la media y dividir
    For i = 0 To PNumSamples
        valor = x(i)
        x(i) = valor / media
    Next



    Num = PNumSamples - 1
    C = 0.5

    For n = 0 To PNumSamples - 1
        'Ind1 = Abs(n - Num)
        'Ind2 = Abs(n - (Num + 1))
```

(continues on next page)

(continued from previous page)

```
        Ind2 = Abs((n))
        Ind2 = Abs((n + 1) Mod Num) / 2
        y(n) = x(n) + C * (y(Ind1) + y(Ind2))
    Next


    ReDim ArrResp(PNumSamples)
    amplitud = 1000
    For i = 0 To Num
        ArrResp(i) = RangoInteger(y(i) * amplitud)
    Next

    karplustrong_b1 = ArrResp
End Function
```

4.14 Lo-Fi Crusher

- **Author or source:** David Lowenfels
- **Type:** Quantizer / Decimator with smooth control
- **Created:** 2003-04-01 15:34:40

Listing 25: notes

```
Yet another bitcrusher algorithm. But this one has smooth parameter control.

Normfreq goes from 0 to 1.0; (freq/samplerate)
Input is assumed to be between 0 and 1.
Output gain is greater than unity when bits < 1.0;
```

Listing 26: code

```
function output = crusher( input, normfreq, bits );
    step = 1/2^(bits);
    phasor = 0;
    last = 0;

    for i = 1:length(input)
       phasor = phasor + normfreq;
       if (phasor >= 1.0)
          phasor = phasor - 1.0;
          last = step * floor( input(i)/step + 0.5 ); %quantize
       end
       output(i) = last; %sample and hold
    end
end
```

4.14.1 Comments

- **Date**: 2004-06-16 21:10:39
- **By**: moc.liamtoh@132197kk

```
what's the "bits" here? I tried to run the code, it seems it's a dead loop here, can␣
↪not
figure out why
```

- **Date**: 2005-10-26 23:25:13
- **By**: dfl

```
bits goes from 1 to 16
```

- **Date**: 2016-03-19 02:47:47
- **By**: moc.liamg@tnemniatretnEesneS2

```
I'm having trouble with the code as well. Is there something I'm missing?
```

4.15 Lookahead Limiter

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** Limiter
- **Created:** 2009-11-16 08:45:47

Listing 27: notes

```
I've been thinking about this for a long time and this is the best I came up with so␣
↪far.
I might be all wrong, but according to some simulations this looks quite nice (as I␣
↪want
it to be).

The below algorithm is written in prosa. It's up to you to transfer it into code.
```

Listing 28: code

```
Ingredients:
------------

1 circular buffers (size of the look ahead time)
2 circular buffers (half the size of the look ahead time)
4 parameters ('Lookahead Time [s]', 'Input Gain [dB]', 'Output Gain [dB]' and
↪'Release Time [s])
a handfull state variables

Recipe:
-------

0. Make sure all buffers are properly initialized and do not contain any dirt (pure␣
↪zeros are what
we need).

For each sample do the following procedure:

1. Store current sample in the lookahead time circular buffer, for later use (and␣
↪retrieve the value
```

(continues on next page)

(continued from previous page)

```
that falls out as the preliminary 'Output')

2. Find maximum within this circular buffer. This can also be implemented efficiently with an hold
algorithm.

3. Gain this maximum by the 'Input Gain [dB]' parameter

4. Calculate necessary gain reduction factor (=1, if no gain reduction takes place and <1 for any
signal above 0 dBFS)

5. Eventually subtract this value from 1 for a better numerical stability. (MUST BE UNDONE LATER!)

6. Add this gain reduction value to the first of the smaller circular buffers to calculate the short
time sum (add this value to a sum and subtract the value that felt out of the circular buffer).

7. normalize the sum by dividing it by the length of the circular buffer (-> / ('Lookahead Time'
[samples] / 2))

8. repeat step 6 & 7 with this sum in the second circular buffer. The reason for these steps is to
transform dirac impulses to a triangle (dirac -> rect -> triangle)

9. apply the release time (release time -> release slew rate 'factor' -> multiply by that factor) to
the 'Maximum Gain Reduction' state variable

10. check whether the currently calculated gain reduction is higher than the 'Maximum Gain Reduction'.
If so, replace!

11. eventually remove (1 - x) from step 5 here

12. calculate effective gain reduction by the above value gained by input and output gain.

13. Apply this gain reduction to the preliminary 'Output' from step 1

Repeat the above procedure (step 1-13) for all samples!
```

4.16 Most simple and smooth feedback delay

- **Author or source:** moc.liamtoh@sisehtnysorpitna
- **Type:** Feedback delay
- **Created:** 2003-09-03 12:58:52

Listing 29: notes

```
fDlyTime = delay time parameter (0-1)

i = input index
j = delay index
```

Listing 30: code

```
if( i >= SampleRate )
    i = 0;

j = i - (fDlyTime * SampleRate);

if( j < 0 )
    j += SampleRate;

Output = DlyBuffer[ i++ ] = Input + (DlyBuffer[ j ] * fFeedback);
```

4.16.1 Comments

- **Date**: 2003-09-03 13:25:47
- **By**: moc.liamtoh@sisehtnysorpitna

```
// This algo didn't seem to work on testing again, just change:
// -------------------------------------------------------------------
Output = DlyBuffer[ i++ ] = Input + (DlyBuffer[ j ] * fFeedback);
// -------------------------------------------------------------------
// to
// ---------------------------------------------------------------
Output = DlyBuffer[ i ] = Input + (DlyBuffer[ j ] * fFeedback);

i++;
// ---------------------------------------------------------------
// and it will work fine.
```

- **Date**: 2003-09-08 12:07:22
- **By**: moc.liamtoh@sisehtnysorpitna

```
// Here's a more clear source. both BufferSize and MaxDlyTime are amounts of samples.␣
↪BufferSize
// should best be 2*MaxDlyTime to have proper sound.
// -
if( i >= BufferSize )
    i = 0;

j = i - (fDlyTime * MaxDlyTime);

if( j < 0 )
    j += BufferSize;

Output = DlyBuffer[ i ] = Input + (DlyBuffer[ j ] * fFeedback);

i++;
```

4.17 Most simple static delay

- **Author or source:** moc.liamtoh@sisehtnysorpitna
- **Type:** Static delay
- **Created:** 2003-09-02 06:21:15

Listing 31: notes

```
This is the most simple static delay (just delays the input sound an amount of␣
↪samples).
Very useful for newbies also probably very easy to change in a feedback delay (for␣
↪comb
filters for example).

Note: fDlyTime is the delay time parameter (0 to 1)

i = input index
j = output index
```

Listing 32: code

```
if( i >= SampleRate )
    i = 0;

DlyBuffer[ i ] = Input;

j = i - (fDlyTime * SampleRate);

i++;

if( j < 0 )
    j = SampleRate + j;

Output = DlyBuffer[ j ];
```

4.17.1 Comments

- **Date**: 2003-09-02 09:17:47
- **By**: moc.liamtoh@sisehtnysorpitna

```
Another note: The delay time will be 0 if fDlyTime is 0 or 1.
```

- **Date**: 2003-09-08 12:21:20
- **By**: gro.ekaf@suomynona

```
// I think you should be careful with mixing floats and integers that way (error-
↪prone,
// slow float-to-int conversions, etc).

// This should also work (haven't checked, not best way of doing it):

// ... (initializing) ..
```

(continues on next page)

(continued from previous page)

```
float numSecondsDelay = 0.3f;
int numSamplesDelay_ = (int)(numSecondsDelay * sampleRate); // maybe want to round to an integer instead of truncating..

float *buffer_ = new float[2*numSamplesDelay];

for (int i = 0; i < numSamplesDelay_; ++i)
{
  buffer_[i] = 0.f;
}

int readPtr_ = 0;
int writePtr_ = numSamplesDelay_;

// ... (processing) ...

for (i = each sample)
{
  buffer_[writePtr_] = input[i];
  output[i] = buffer_[readPtr_];

  ++writePtr_;
  if (writePtr_ >= 2*numSamplesDelay_)
    writePtr_ = 0;

  ++readPtr_;
  if (readPtr_ >= 2*numSamplesDelay_)
    readPtr_ = 0;
}
```

- **Date**: 2004-08-24 19:31:45
- **By**: moc.liamg@noteex

```
I may be missing something, but I think it gets a little simpler than the previous two examples.
The difference in result is that actual delay will be 1 if d is 0 or 1.

i = input/output index
d = delay (in samples)

Code:

out = buffer[i];
buffer[i] = in;
i++;
if(i >= delay) i = 0;
```

- **Date**: 2005-12-18 02:35:00
- **By**: moc.liamg@noteex

```
or even in three lines...

out = buffer[i];
buffer[i++] = in;
if(i >= delay) i = 0;
```

- **Date**: 2006-12-30 16:29:07
- **By**: ku.oc.etativarg-jd@etativarg

```
The only problem with this implementation, is that it is not really an audio effect!␣
↪all this
will do is to delay the input signal by a given number of samples! ...why would you␣
↪ever want
to do that? ...this would only ever work if you had a DSP and speakers both connected␣
↪to the
audio source and run them at the same time, so the speakers would be playing the␣
↪original sorce
and the DSP containing the delayed source connected to another set of speakers! this␣
↪is not
really an audio efffect!

...Here is a pseudo code example of a delay effect that will mix both the original␣
↪sound with
the delayed sound:

Pseudo Code implementation for simple delay:
 - This implementation will put the current audio signal to the
   left channel and the delayed audio signal to the right channel.
   this is suitable for any stereo codec!

delay_function {

  left_channel   // for stereo left
  right_channel  // for stereo right
  mono           // mono representation of stereo input
  delay_time     // amount of time to delay input
  counter = 0    // counter

  //setup an array that is the same length as the maximum delay time:
  delay_array[max delay time]  // array containing delayed data

  // convert stereo to mono:
  (left_channel + right_channel) / 2

  // initalise time to delay signal - maybe input from user
  delay_time = x

  if (delay_time = 0){
    left_out = mono
    right_out = mono
  }
  else {
    // put current input data to left channel:
    left_out = mono
    // put oldest delayed input data to right channel:
    right_out = delay_array[index]

    // overwrite with newest input:
    delay_array[index] = mono;

    // is index at end of delay buffer? if not increment, else set to zero
    if (index < delay_time) index++
    else index = 0
```

(continues on next page)

(continued from previous page)

```
  }

}
```

4.18 Parallel combs delay calculation

- **Author or source:** Juhana Sadeharju (if.tenuf.cin@aihuok)
- **Created:** 2002-05-23 19:49:40

Listing 33: 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.
```

4.19 Phaser code

- **Author or source:** Ross Bencina
- **Created:** 2002-02-11 17:41:53
- **Linked files:** `phaser.cpp`.

Listing 34: notes

```
(see linked file)
```

4.19.1 Comments

- **Date**: 2005-06-01 22:48:17
- **By**: ed.luosfosruoivas@naitsirhC

```
// Delphi / Object Pascal Translation:
// -----------------------------------

unit Phaser;

// Still not efficient, but avoiding denormalisation.

```

(continues on next page)

(continued from previous page)

```
interface

type
  TAllPass=class(TObject)
  private
    fDelay : Single;
    fA1, fZM1 : Single;
    fSampleRate : Single;
    procedure SetDelay(v:Single);
  public
    constructor Create;
    destructor Destroy; override;
    function Process(const x:single):single;
    property SampleRate : Single read fSampleRate write fSampleRate;
    property Delay: Single read fDelay write SetDelay;
  end;

  TPhaser=class(TObject)
  private
    fZM1 : Single;
    fDepth : Single;
    fLFOInc : Single;
    fLFOPhase : Single;
    fFeedBack : Single;
    fRate : Single;
    fMinimum: Single;
    fMaximum: Single;
    fMin: Single;
    fMax: Single;
    fSampleRate : Single;
    fAllpassDelay: array[0..5] of TAllPass;
    procedure SetSampleRate(v:Single);
    procedure SetMinimum(v:Single);
    procedure SetMaximum(v:Single);
    procedure SetRate(v:Single);
    procedure Calculate;
  public
    constructor Create;
    destructor Destroy; override;
    function Process(const x:single):single;
    property SampleRate : Single read fSampleRate write SetSampleRate;
    property Depth: Single read fDepth write fDepth; //0..1
    property Feedback: Single read fFeedback write fFeedback; // 0..<1
    property Minimum: Single read fMin write SetMinimum;
    property Maximum: Single read fMax write SetMaximum;
    property Rate: Single read fRate write SetRate;
  end;

implementation

uses DDSPUtils;

const kDenorm=1E-25;

constructor TAllpass.Create;
begin
 inherited;
```

(continues on next page)

(continued from previous page)

```
 fA1:=0;
 fZM1:=0;
end;

destructor TAllpass.Destroy;
begin
 inherited;
end;

function TAllpass.Process(const x:single):single;
begin
 Result:=x*-fA1+fZM1;
 fZM1:=Result*fA1+x;
end;

procedure TAllpass.SetDelay(v:Single);
begin
 fDelay:=v;
 fA1:=(1-v)/(1+v);
end;

constructor TPhaser.Create;
var i : Integer;
begin
 inherited;
 fSampleRate:=44100;
 fFeedBack:=0.7;
 fLFOPhase:=0;
 fDepth:=1;
 fZM1:=0;
 Minimum:=440;
 Maximum:=1600;
 Rate:=5;
 for i:=0 to Length(fAllpassDelay)-1
  do fAllpassDelay[i]:=TAllpass.Create;
end;

destructor TPhaser.Destroy;
var i : Integer;
begin
 for i:=0 to Length(fAllpassDelay)-1
  do fAllpassDelay[i].Free;
 inherited;
end;

procedure TPhaser.SetRate(v:Single);
begin
 fLFOInc:=2*Pi*(v/SampleRate);
end;

procedure TPhaser.Calculate;
begin
 fMin:= fMinimum / (fSampleRate/2);
 fMax:= fMinimum / (fSampleRate/2);
end;

procedure TPhaser.SetMinimum(v:Single);
```

(continues on next page)

(continued from previous page)

```
begin
 fMinimum:=v;
 Calculate;
end;

procedure TPhaser.SetMaximum(v:Single);
begin
 fMaximum:=v;
 Calculate;
end;

function TPhaser.Process(const x:single):single;
var d: Single;
    i: Integer;
begin
 //calculate and update phaser sweep lfo...
 d := fMin + (fMax-fMin) * ((sin( fLFOPhase )+1)/2);
 fLFOPhase := fLFOPhase + fLFOInc;
 if fLFOPhase>=Pi*2
  then fLFOPhase:=fLFOPhase-Pi*2;

 //update filter coeffs
 for i:=0 to 5 do fAllpassDelay[i].Delay:=d;

 //calculate output
 Result:= fAllpassDelay[0].Process(
          fAllpassDelay[1].Process(
          fAllpassDelay[2].Process(
          fAllpassDelay[3].Process(
          fAllpassDelay[4].Process(
          fAllpassDelay[5].Process(kDenorm + x + fZM1 * fFeedBack ))))));
 fZM1:=tanh2a(Result);

 Result:=tanh2a(1.4*(x + Result * fDepth));
end;

procedure TPhaser.SetSampleRate(v:Single);
begin
 fSampleRate:=v;
end;

end.
```

- **Date**: 2005-06-01 22:51:25
- **By**: ed.luosfosruoivas@naitsirhC

```
Ups, forgot to remove my special, magic incredients "tanh2a(1.4*(". It's just
to make the sound even warmer.

The frequency range i used for Minimum and Maximum is 0..22000. But I believe there
is still an error in that formula. The input range doesn't matter (if you remove my␣
↪special
incredient), because it is a linear system.
```

- **Date**: 2005-06-05 21:40:35
- **By**: moc.yddaht@yddaht

```
// I thought I already posted this but here's my interpretation for Delphi and KOL.
// The reason I repost this, is that it is rather efficient and has no denormal
↪problems.

unit Phaser;
{

       Unit: Phaser
    purpose: Phaser is a six stage phase shifter, intended to reproduce the
             sound of a traditional analogue phaser effect.
     Author: Thaddy de Koning, based on a musicdsp.pdf C++ Phaser by
             Ross Bencina.http://www.musicdsp.org/musicdsp.pdf
  Copyright: This version (c) 2003, Thaddy de Koning
             Copyrighted Freeware

    Remarks: his implementation uses six first order all-pass filters in
             series, with delay time modulated by a sinusoidal.
             This implementation was created to be clear, not efficient.
             Obvious modifications include using a table lookup for the lfo,
             not updating the filter delay times every sample, and not
             tuning all of the filters to the same delay time.

             It sounds sensationally good!
}

interface

uses Kol, AudioUtils, SimpleAllpass;

type
  PPhaser = ^TPhaser;
  TPhaser = object(Tobj)
  private
    FSamplerate: single;
    FFeedback: single;
    FlfoPhase: single;
    FDepth: single;
    FOldOutput: single;
    FMinDelta: single;
    FMaxDelta: single;
    FLfoStep: single;
    FAllpDelays: array[0..5] of PAllpassdelay;
    FLowFrequency: single;
    FHighFrequency: single;
    procedure SetRate(TheRate: single); // cps
    procedure SetFeedback(TheFeedback: single); // 0 -> <1.
    procedure SetDepth(TheDepth: single);
    procedure SetHighFrequency(const Value: single);
    procedure SetLowFrequency(const Value: single); // 0 -> 1.
    procedure SetRange(LowFreq, HighFreq: single); // Hz
  public
    destructor Destroy; virtual;
    function Process(inSamp: single): single;
    property Rate: single write setrate;//In Cycles per second
    property Depth: single read Fdepth write setdepth;//0.. 1
    property Feedback: single read FFeedback write setfeedback; //0..< 1
    property Samplerate: single read Fsamplerate write Fsamplerate;
```

(continues on next page)

(continued from previous page)

```
    property LowFrequency: single read FLowFrequency write SetLowFrequency;
    property HighFrequency: single read FHighFrequency write SetHighFrequency;
  end;

function NewPhaser: PPhaser;

implementation



{ TPhaser }
function NewPhaser: PPhaser;
var
  i: integer;
begin
  New(Result, Create);
  with Result^ do
  begin
    Fsamplerate := 44100;
    FFeedback := 0.7;
    FlfoPhase := 0;
    Fdepth := 1;
    FOldOutput := 0;
    setrange(440,1720);
    setrate(0.5);
    for i := 0 to 5 do
      FAllpDelays[i] := NewAllpassDelay;
  end;
end;

destructor TPhaser.Destroy;
var
  i: integer;
begin
  for i := 5 downto 0 do FAllpDelays[i].Free;
  inherited;
end;

procedure TPhaser.SetDepth(TheDepth: single); // 0 -> 1.
begin
  Fdepth := TheDepth;
end;

procedure TPhaser.SetFeedback(TheFeedback: single);//0..1;
begin
  FFeedback := TheFeedback;
end;

procedure TPhaser.SetRange(LowFreq, HighFreq: single);
begin
  FMinDelta := LowFreq / (FsampleRate / 2);
  FMaxDelta := HighFreq / (FsampleRate / 2);
end;

procedure TPhaser.SetRate(TheRate: single);
begin
  FLfoStep := 2 * _PI * (Therate / FsampleRate);
```

(continues on next page)

(continued from previous page)

```
end;

const
  _1:single=1;
  _2:single=2;
function TPhaser.Process(inSamp: single): single;
var
  Delaytime, Output: single;
  i: integer;
begin

  //calculate and Process phaser sweep lfo...
  Delaytime := FMinDelta + (FMaxDelta - FMinDelta) * ((sin(FlfoPhase) + 1) / 2);
  FlfoPhase := FlfoPhase + FLfoStep;
  if (FlfoPhase >= _PI * 2) then
    FlfoPhase := FlfoPhase - _PI * 2;
  //Process filter coeffs
  for i := 0 to 5 do
    FAllpDelays[i].setdelay(Delaytime);
  //calculate output
  Output := FAllpDelays[0].Process(FAllpDelays[1].Process
    (FAllpDelays[2].Process(FAllpDelays[3].Process(FAllpDelays[4].Process
    (FAllpDelays[5].Process(inSamp + FOldOutput * FFeedback))))));
  FOldOutput := Output;
  Result := kDenorm + inSamp + Output * Fdepth;
end;

procedure TPhaser.SetHighFrequency(const Value: single);
begin
  FHighFrequency := Value;
  setrange(FlowFrequency, FHighFrequency);
end;

procedure TPhaser.SetLowFrequency(const Value: single);
begin
  FLowFrequency := Value;
  setrange(FlowFrequency, FHighFrequency);
end;

end.
```

- **Date**: 2005-06-05 21:44:47
- **By**: moc.yddaht@yddaht

```
// And here the allpass:
unit SimpleAllpass;
{
       Unit: SimpleAllpass
    purpose: Simple allpass delay for creating reverbs and phasing/flanging
     Author:
  Copyright:
    Remarks:
}
interface
uses kol, audioutils;
```

(continues on next page)

(continued from previous page)

```
type
  PAllpassDelay = ^TAllpassDelay;
  TAllpassdelay = object(Tobj)
  protected
    Fa1,
    Fzm1: single;
  public
    procedure SetDelay(delay: single);//sample delay time
    function Process(inSamp: single): single;
  end;

function NewAllpassDelay: PAllpassDelay;


implementation

function NewAllpassDelay: PAllpassDelay;
begin
  New(Result, Create);
  with Result^ do
  begin
    Fa1 := 0;
    Fzm1 := 0;
  end;
end;

function TallpassDelay.Process(Insamp: single): single;
begin
  Result := kDenorm+inSamp * -Fa1 + Fzm1;
  Fzm1 := Result * Fa1 + inSamp + kDenorm;
end;

procedure TAllpassDelay.setdelay(delay: single);// In sample time
begin
  Fa1 := (1 - delay) / (1 + delay);
end;

end.
```

- **Date**: 2005-06-06 08:15:37
- **By**: ed.luosfosruoivas@naitsirhC

```
// You'll get a good performance boost by combining the 6 allpasses to one and
↪rewriting
// that one to FPU code. Heavy speed increase AND you can make the number of allpasses
// variable as well.

// This would look similar to this:

function TMasterAllpass.Process(const x:single):single;
var a : array[0..1] of Single;
    b : Single;
    i : Integer;
begin
 a[0]:=x*fA1+fY[0];
 b:=a[0]*fA1;
```

(continues on next page)

(continued from previous page)

```
 fY[0]:=b-x;

 i:=0;
 while i<fStages do
  begin
   a[1]:=b-fY[i+1];
   b:=a[1]*fA1;
   fY[i+1]:=a[0]-b;
   a[0]:=b-fY[i+2];
   b:=a[0]*fA1;
   fY[i+2]:=a[1]-b;
   Inc(i,2);
  end;

 a[1]:=b-fY[5];
 b:=a[1]*fA1;
 fY[5]:=a[0]-b;
 Result:=a[1];
end;

Now all you have to do is crawling into the FPU registers...
```

- **Date**: 2005-06-07 11:31:05
- **By**: moc.yddaht@yddaht

```
Point taken ;)
Maybe we should combine all the stuff ;)
Btw:
It's lots of fun working from each others code, don't you think?
```

4.20 Polynominal Waveshaper

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** (discrete harmonics)
- **Created:** 2006-07-28 17:58:54

Listing 35: notes

```
The following code will describe how to excite discrete harmonics and only these
harmonics. A simple polynominal waveshaper for processing the data is included as well.
However the code don't claim to be optimized. Using a horner scheme with precalculated
coefficients should be your choice here.
Also remember to oversample the data (optimal in the order of the harmonics) to have them
alias free.
```

Listing 36: code

```
We assume the input is a sinewave (works for any input signal, but this makes everything more clear).
Then we have x = sin(a)
```

(continues on next page)

(continued from previous page)

```

the first harmonic is plain simple (using trigonometric identities):

cos(2*a)= cos^2(a) - sin^2(a) = 1 - 2 sin^2(a)

using the general trigonometric identities:

sin(x + y) = sin(x)*cos(y) + sin(y)*cos(x)
cos(x + y) = cos(x)*cos(y) - sin(y)*sin(x)

together with some math, you can easily calculate: sin(3x), cos(4x), sin(5x), and so on...


Here's how the resulting waveshaper may look like:

// o = output, i = input
o = fPhase[1]*     i                                                              * fGains[0]+
    fPhase[1]*(  2*i*i           -   1                                      ) * fGains[1]+
    fPhase[2]*(  4*i*i*i         -   3*i                                    ) * fGains[2]+
    fPhase[3]*(  8*i*i*i*i       -   8*i*i        +   1                     ) * fGains[3]-
    fPhase[4]*( 16*i*i*i*i*i     -  20*i*i*i      +   5 * i                 ) * fGains[4]+
    fPhase[5]*( 32*i*i*i*i*i*i   -  48*i*i*i*i    +  18 * i*i     -  1      ) * fGains[5]-
    fPhase[6]*( 64*i*i*i*i*i*i*i  - 112*i*i*i*i*i   +  56 * i*i*i   -  7 * i    ) * fGains[6]+
    fPhase[7]*(128*i*i*i*i*i*i*i*i - 256*i*i*i*i*i*i + 160 * i*i*i*i - 32 * i*i + 1 ) * fGains[7];

fPhase[..] is the sign array and fGains[..] is the gain factor array.

P.S.: I don't want to see a single comment about the fact that the code above is unoptimized. I know that!
```

4.20.1 Comments

- **Date**: 2006-07-28 22:16:36
- **By**: ed.luosfosruoivas@naitsirhC

```
Here's the more math like version:

// o = output, i = input
o = fPhase[1]* i * fGains[0]+
    fPhase[1]*(  2*i^2 - 1 ) * fGains[1]+
    fPhase[2]*(  4*i^3 - 3*i ) * fGains[2]+
    fPhase[3]*(  8*i^4 - 8*i^2 + 1 ) * fGains[3]-
    fPhase[4]*( 16*i^5 - 20*i^3 + 5*i ) * fGains[4]+
    fPhase[5]*( 32*i^6 - 48*i^4 + 18 * i^2 - 1 ) * fGains[5]-
    fPhase[6]*( 64*i^7 - 112*i^5 + 56 * i^3 - 7 * i ) * fGains[6]+
    fPhase[7]*(128*i^8 - 256*i^6 + 160 * i^4 - 32 * i^2 + 1 ) * fGains[7];
```

- **Date**: 2006-07-28 23:29:49
- **By**: ten.xmg@zlipzzuf

```
Actually, this *doesn't* work in the way described on any input, only on single␣
↪sinusoids of
amplitude 1. It's nonlinear - (a+b)^n is not the same thing as a^n+b^n, nor are (a*b)^
↪n and a*(b^n).
Even just a sum of two sinusoids or a single sinusoid of a different amplitude breaks␣
↪the
chosen-harmonics-only thing.
```

- **Date**: 2006-07-29 10:30:46
- **By**: ed.luosfosruoivas@naitsirhC

```
Do'oh. You're right. Once more I got fooled by the way of my measurement. That␣
↪explains a
lot of things... Thanks for the clarification!
```

- **Date**: 2006-07-29 20:34:11
- **By**: ed.luosfosruoivas@naitsirhC

```
Btw. the coeffitients follow the chebyshev polynomials. Just in case you wonder about␣
↪the
logic behind. Maybe we can call it chebyshev waveshaper from now on...
```

- **Date**: 2008-01-23 01:56:21
- **By**: ten.enilnotpo@kcirtscisyhp

```
I played with this idea for a while yesterday to no avail before discovering this␣
↪post tonight.
I thought I could excite any harmonic I wanted using select Chebyshev Polynomials.␣
↪But you are
totally right - it doesn't work that way. Any complex waveform that can be broken␣
↪down into a
Fourier series is a linear sum of terms. Squaring or cubing the waveform, and␣
↪therefor this sum,
leads to multiple cross terms which introduce additional frequencies. It does only␣
↪work with
normalized single sinusoids . . .which is too bad.

Right now, the only way I can see to do this sort of thing where you excite select␣
↪harmonics at will
is to run an FFT and then work from there in the frequency domain.

But my question is, if we are looking to simulate tube saturation, is the Chebyshev␣
↪method good
enough. What, after all, do tubes do? Does a tube amp actually add discrete harmonics␣
↪or is it
introducing all of those cross term frequencies as well?
```

- **Date**: 2009-05-19 22:26:03
- **By**: moc.liamg@neklov.neivalf

```
According to another post, the tube is simply a non linear function, for example a␣
↪tan(x).
Actually by saturating any signal you will get harmonics (any but a pure square which␣
↪cannot
be more saturated of couse...). As tan(x) is not linear, you should get harmonics...␣
↪that's all.
Now if you want to pass only the high frequencies,just split the signal into 2␣
↪frequencies using
a lowpass vs highpass = signal - lowpass and process the frequencies you want to.
```

- **Date**: 2010-09-16 19:17:13
- **By**: moc.liamg@libojyr

```
I would like to add that this method could result in an approximated harmonic exciter␣
↪using an
array of filters sufficiently narrow and steep to approximately single out individual␣
↪frequencies
composing the original signal.

As such, it would be processing intensive because the polynomial would need to be␣
↪calculated on
each band.

What you have presented is not completely bad.  You only need to take into␣
↪consideration that you're
getting frequency terms that are not necessarily harmonics.  Steve Harris has a␣
↪LADSPA plugin that
uses the chebychev polynomial as a waveshaper...and he calls it a harmonic exciter.

To user physicstrick:  Tube emulation is much more than waveshaping.  Bias conditions␣
↪change with
signal dynamics, and you essentially get signal-power modulated duty cycle.  I have␣
↪found some good
articles about this and also there is a commercial product that claims to solve the␣
↪discretized
system of ODE's in real time.  This model eats CPU like you would not imagine.

My simple "trick" is to include the nonlinear function in a 1rst order filter␣
↪calculation and
also to modulate the filter time constants with the filter state variable amplitude. ␣
↪This is not
quite right, but it produces an emulation that is more pleasing than plain␣
↪waveshaping.
```

- **Date**: 2013-01-18 02:31:27
- **By**: moc.liamtoh@niffumtohrepus

```
If I'm correct, you're describing the same technique achievable by use of Chebyschev␣
↪polynomials,
i.e. generating any integral harmonic of the original signal. I've experimented with␣
↪these,
synthesizing only the second, third, fourth, etc. harmonics, but could never get a␣
↪realistic
sound, probably because overdiven tubes/transistors don't work this way and there's␣
↪no intermodulation
distortion, only the pure harmonics.
```

4.21 Reverberation Algorithms in Matlab

- **Author or source:** Gautham J. Mysore (moc.oohay@mjmahtuag)
- **Created:** 2003-07-15 08:58:05
- **Linked files:** MATLABReverb.zip.

Listing 37: notes

```
These M-files implement a few reverberation algorithms (based on Schroeder's and␣
↪Moorer's
algorithms). Each of the M-files include a short description.

There are 5 M-files that implement reverberation. They are:

- schroeder1.m
- schroeder2.m
- schroeder3.m
- moorer.m
- stereoverb.m

The remaining 8 M-files implement filters, delay lines etc. Most of these are used in␣
↪the
above M-files. They can also be used as building blocks for other reverberation
algorithms.
```

4.21.1 Comments

- **Date**: 2003-08-15 04:04:56
- **By**: moc.loa@mnijwerb

```
StereoVerb is the name of an old car stereo "enhancer" from way back. I was just␣
↪trying to find
it's roots.
```

- **Date**: 2004-04-02 04:44:46
- **By**: moc.oohay@y_sunave

```
There is another allpass filter transfer function.
      -g+Z^(-m)
H(z)=------------
      1-gZ^(-m)

g is the attenuation
m is the number of delay (in sampel)

This allpass will give exponential decay impulse response, compare to your allpass␣
↪that give
half sinc decay impulse response.
```

4.22 Reverberation techniques

- **Author or source:** Sean Costello
- **Created:** 2002-02-10 20:00:11

Listing 38: 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.
```

4.23 Simple Compressor class (C++)

- **Author or source:** Citizen Chunk
- **Type:** stereo, feed-forward, peak compressor
- **Created:** 2005-05-28 19:11:42
- **Linked files:** `simpleSource.zip`.

Listing 39: notes

```
Everyone seems to want to make their own compressor plugin these days, but very few␣
↪know
where to start. After replying to so many questions on the KVR Dev Forum, I figured I
might as well just post some ready-to-use C++ source code.

This is a C++ implementation of a simple, stereo, peak compressor. It uses a feed-
↪forward
topology, detecting the sidechain level pre-gain reduction. The sidechain detects the
rectified peak level, with stereo linking to preserve imaging. The attack/release␣
↪uses the
EnvelopeDetector class (posted in the Analysis section).

Notes:
- Make sure to call initRuntime() before processing starts (i.e. call it in resume()).
- The process function takes a stereo input.
- VST params must be mapped to a practical range when setting compressor parameters.␣
↪(i.e.
don't try setAttack( 0.f ).)

(see linked files)
```

4.23.1 Comments

- **Date**: 2005-11-26 01:56:48
- **By**: moc.liamtoh@361_lt

```
This code works perfectly, and I have tried a number of sound and each worked␣
↪correctly. The
conversion is linear in logarithm domain.

The code has been written in such a professional style, can not believe it is FREE!!

Keep it up. Two super huge thumbs up.

Ting
```

4.24 Soft saturation

- **Author or source:** Bram de Jong
- **Type:** waveshaper
- **Created:** 2002-01-17 02:18:37

Listing 40: notes

```
This only works for positive values of x. a should be in the range 0..1
```

Listing 41: 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
```

4.24.1 Comments

- **Date**: 2005-09-09 21:17:55
- **By**: gro.esabnaeco@euarg

```
This is a most excellent waveshaper.

I have implemented it as an effect for the music tracker Psycle, and so far I am very␣
↪pleased with the
results. Thanks for sharing this knowledge, Bram!
```

- **Date**: 2006-03-12 02:35:16
- **By**: moc.erawtfosnetpot@nosniborb

```
I'm wondering about the >1 condition here.  If a is 0.8, values <1 approach 0.85 but␣
↪values >1
are clamped to 0.9  (there's a gap)

If you substitute x=1 to the equation for x>a you get: a+((1-a)/4) not (a+1)/2

Have I missed something or is there a reason for this?

(Go easy I'm new to all of this)
```

- **Date**: 2006-08-23 06:19:05
- **By**: moc.liamg@ubibik

```
Substituting x=1 into equation 2 (taking many steps)

f(x) = a + (x-a)/(1+ ((x-a)/(1-a))^2)
     = a + (1-a)/(1+ ((1-a)/(1-a))^2)
     = a + (1-a)/(1+ 1^2)
     = a + (1-a)/2
     = 2a/2 + (1-a)/2
     = (2a + 1 - a) /2
     = (a+1) / 2
```

- **Date**: 2006-08-24 11:03:04
- **By**: musicdsp@Nospam dsparsons.co.uk

```
You can normalise the output:
f(x)'=f(x)*(1/((a+1)/2))

This gives a nice variable shaper with smooth curve upto clipping at 0dBFS
```

4.25 Stereo Enhancer

- **Author or source:** vl.xobni@ksimruk
- **Created:** 2004-06-27 22:44:16

Listing 42: notes

```
Stereo Enhanca
```

Listing 43: code

```
// WideCoeff  0.0 .... 1.5

#define StereoEnhanca(SamplL,SamplR,MonoSign, \
  DeltaLeft,WideCoeff ) \
  MonoSign = (SamplL + SamplR)/2.0; \
  DeltaLeft = SamplL - MonoSign; \
  DeltaLeft = DeltaLeft * WideCoeff; \
  SamplL=SamplL + DeltaLeft; \
  SamplR=SamplR - DeltaLeft;
```

4.25.1 Comments

- **Date**: 2004-08-20 17:45:38
- **By**: moc.liamg@noteex

```
#define StereoEnhanca(SamplL,SamplR,MonoSign,
  DeltaLeft,DeltaRight,WideCoeff )
  MonoSign = (SamplL + SamplR)/2.0;

  DeltaLeft = SamplL - MonoSign;
  DeltaLeft *= WideCoeff;
  DeltaRight = SamplR - MonoSign;
  DeltaRight *= WideCoeff;

  SamplL += DeltaLeft;
  SamplR += DeltaRight;

I think this is more along the lines of what you were trying to accomplish. I doubt
this is
the correct way of implementing this type of thing however.
```

- **Date**: 2004-08-24 15:40:31
- **By**: moc.noomyab@grubmah_kram

```
I believe both pieces of code do the same thing. Since MonoSign is set equal to the␣
↪average
of the two signals, in the second case DeltaRight = -DeltaLeft.
```

- **Date**: 2005-01-07 10:20:20
- **By**: thaddy[@]thaddy.com

```
// Here's an implementation of the classic stereo enhancer in Delphi BASM
// Values below 0.1 have a narrowing effect
// Values abouve 0.1 widens
parameters:
Buffer = eax
Amount = edx
Samples = ecx

const
  Spread:single =  6.5536;

procedure Sound3d32f(Buffer:PSingle;Amount:Single;Samples:integer);
asm
  fld     Amount
  fmul    spread
  mov     ecx, edx        // move samples to ecx
  shr     ecx, 1          // divide by two, stereo = 2 samples
@Start:
  fld     [eax].dword     // left sample
  fld     [eax+4].dword   // right sample, whole calculation runs on the stack
  fld     st(0)           // copy
  fadd    st(0), st(2)
  fmul    half            // average =st(0), right sample = st(1), left = st(2),␣
↪amount=st(3)
  fld     st(0)           // copy average
  fsubr   st(0), st(3)    // left diffence
  fmul    st(0), st(4)    // amount
  fadd    st(0), st(1)    // add average
  fadd    st(0), st(3)    // add original
  fmul    half            // divide by two
  fstp    [eax].dword     // and store
  fld     st(0)
  fsubr   st(0), st(2)    // right difference
  fmul    st(0), st(4)    // amount
  faddp                   // add average
  faddp                   // add original
  fmul    half            // divide by 2
  fstp    [eax+4].dword;  // and store
  fxch                    // Dangling average?? remove it later, tdk
  ffree   st(1)
  add     eax, 8          // advance to next stereo pair
  loop    @Start
  ffree   st(0);          // Cleanup amount
end;
```

- **Date**: 2005-03-24 09:45:09
- **By**: moc.yddaht@yddaht

```
Note 'half' is defined as const half:single = 0.5;
This is an ommission in the above posting
```

- **Date**: 2005-04-21 22:04:02
- **By**: moc.liamtoh@gorpketg

```
This original code makes indeed no sense.

>#define StereoEnhanca(SamplL,SamplR,MonoSign, \
>DeltaLeft,WideCoeff ) \
>MonoSign = (SamplL + SamplR)/2.0; \
>DeltaLeft = SamplL - MonoSign; \
>DeltaLeft = DeltaLeft * WideCoeff; \
>SamplL=SamplL + DeltaLeft; \
>SamplR=SamplR - DeltaLeft;

Deltaleft hold no stereoinformation.
explained: Deltaleft=L-(L+R) = R!!!
So, in this example your stereo image would slide to the right more as you put␣
↪widecoeff higher.

A better implementation is the following code.
#define StereoEnhanca(SamplL,SamplR,MonoSign, \
stereo,WideCoeff ) \
MonoSign = (SamplL + SamplR)/2.0; \
stereo = SamplL - Sampl1L; \
stereo = DeltaLeft * WideCoeff; \
SamplL=SamplR + stereo; // R+Stereo = L
SamplR=SamplL - stereo; // L-Stereo = R

This way of stereoenhancement will lead to exaggerated reverberation effects (␣
↪snaredrums).
This is not the best way to do widening, but it is the easiest.

Gtekprog.

Evert Verduin
```

- **Date**: 2005-04-21 22:06:51
- **By**: moc.liamtoh@gorpketg

```
oops,

stereo = SamplL - Sampl1L;
needs ofcourse to be
stereo = SamplL - Sampl1R;

and

stereo = DeltaLeft * WideCoeff; \
needs to be
stereo = stereo * WideCoeff; \

Again the correct code:

#define StereoEnhanca(SamplL,SamplR,MonoSign, \
```

(continues on next page)

(continued from previous page)

```
stereo,WideCoeff ) \
MonoSign = (SamplL + SamplR)/2.0; \
stereo = SamplL - Sampl1R; \
stereo = stereo * WideCoeff; \
SamplL=SamplR + stereo; // R+Stereo = L
SamplR=SamplL - stereo; // L-Stereo = R

This will do.

Evert
```

- **Date**: 2009-04-17 13:16:42
- **By**: moc.liamg@nostohnotyalc

```
You mean to use MonoSign variable somewhere - as in:

   #define StereoEnhanca(SamplL,SamplR,MonoSign, \
stereo,WideCoeff ) \
MonoSign = (SamplL + SamplR)/2.0; \
stereo = SamplL - Sampl1R; \
stereo = stereo * WideCoeff; \

SamplL = MonoSign + stereo; // R+Stereo = L
SamplR = MonoSign - stereo; // L-Stereo = R

Or some variation?

Clayton
```

4.26 Stereo Field Rotation Via Transformation Matrix

- **Author or source:** Michael Gruhn
- **Type:** Stereo Field Rotation
- **Created:** 2008-03-17 09:40:10

Listing 44: notes

```
This work is hereby placed in the public domain for all purposes, including use in
commercial applications.

'angle' is the angle by which you want to rotate your stereo field.
```

Listing 45: code

```
// Calculate transformation matrix's coefficients
cos_coef = cos(angle);
sin_coef = sin(angle);

// Do this per sample
out_left  = in_left * cos_coef - in_right * sin_coef;
out_right = in_left * sin_coef + in_right * cos_coef;
```

4.26.1 Comments

- **Date**: 2008-10-07 19:56:47
- **By**: moc.liamtoh@iniluigj

```
This looks like the rotation formula for a point in space. Can you explain how does␣
↪it work
for a sound signal? Let's say that angle is 90 degrees, then you formula gives
out_left = -in_right
out_right = in_left
How would this be a 90 deg rotation of the sound?
```

- **Date**: 2008-10-15 12:16:02
- **By**: Foo

```
It IS the exact formula as rotation for a point in a 2D space (around its origin).␣
↪Now this is
applied to the stereo field. Imagine it as a left-right plot, so the values of the␣
↪left and right
channel get plotted (just like a goniometer: http://en.wikipedia.org/wiki/Goniometer_
↪(audio)).
So now you can see the stereo image (mono = straight line, stereo = circle, etc...).␣
↪Now when you
rotate THIS plot and then use the values of the rotated plot for the new left and␣
↪right sample
values, you rotated the stereo image.
So just get a goniometer and look at how the signal changes when you run it through␣
↪the algorithm,
it will be pretty obvious.

Hope this helps.
```

- **Date**: 2008-11-16 02:13:27
- **By**: Bar

```
Sorry this makes no sense at all. The rotation formula is predicated on the␣
↪assumption that
(x,y) are coordinates of two orthogonal dimensions. Now you can choose to visualize␣
↪stereo
signals anyway you like, including being on a Cartesian plan, or as polar coordinates,
↪ what have
you... But this visualization has no relationship to the physical location of the␣
↪sound. The left
and right channels are NOT orthogonal dimensions physically. What the formula does is␣
↪just some
weird panning. As the previous comment pointed out, just plug in some easy angles␣
↪like 90, 180 ...
and see if you can make any valid interpretations out of them. You can't.
```

- **Date**: 2008-11-18 21:22:49
- **By**: Foo

```
So you want mathematical prove? Even though I consider this childish, because it'd␣
↪take you <5
minutes to put this in Matlab or any other DSP prototyping bench and hear the␣
↪rotation effect for
```

(continues on next page)

(continued from previous page)

```
yourself. Anyway ...

For 180° the output should be totally inverted. So:
cos(180) = -1
sin(180) =  0
out_left = -in_left
out_right = -in_right

at 90° this means for a mono signal that the left channel will be a phase inverse of␣
↪the right
channel, so ... go directly to result, do not calculate:
out_left = -in_right
out_right = in_left

at 45° is just like hard panning to the right (with a 3dB volume attenuation), so for␣
↪a mono
signal the expected results would be one channel silence and the other would have the␣
↪signal,
so we calculate:
cos(45) = sqrt(2)/2
sin(45) = sqrt(2)/2
for mono signal we assume: mono = in_left = in_right ... so it follows:
out_left = mono * sqrt(2)/2 - mono * sqrt(2)/2 = 0
out_right = mono * sqrt(2)/2 + mono * sqrt(2)/2 = mono * sqrt(2) == mono * 3dB

and one more, 360° means same output as input, calculate for yourself.

Valid enough?
```

- **Date**: 2008-11-20 20:18:24
- **By**: Bar

```
Then I'm not sure what you mean by rotation. In my mind, I see two sound sources at␣
↪arbitrary
locations and I'm at the center of rotation. So the effect of a rotation would depend␣
↪on the
angle subtended by the three points to begin with, which doesn't even show up in the␣
↪formula.
Also please explains what does it mean by the two channels being orthogonal␣
↪dimensions, which
is what the formula is based on. (I assume you understand the mathematical basis of␣
↪how the
formula is derived.)

No, a phase inversion on both channels don't sound 180 deg rotated. It sounds exactly␣
↪the same
as before.
```

- **Date**: 2008-11-20 21:27:25
- **By**: Bar

```
Let's try another experiment. You're in the midpoint of the line joining the two␣
↪speakers and is
the center of rotation. Your signal happens to have all zeros for the left channel.␣
↪The formula
```

(continues on next page)

(continued from previous page)

```
simplifies to:
out_left = -in_right * sin
out_right = in_right * cos
As you rotate from 0 to 90, sin goes from 0 to 1, cos goes from 1 to 0. So the␣
↪formula predicts
that the left channel goes from silence to a phase inverted right, and the right␣
↪channel goes from
full sound to silence. Whereas physically the sound should move from my right to␣
↪directly in front of
me. Please explain.
```

- **Date**: 2008-11-20 21:36:54
- **By**: Bar

```
Yet another childish thought. If one can treat signals as if they were space␣
↪locations, then surely
translations will work just as well as rotations. So to move a sound source to␣
↪another location,
one just add constants to the signals?
```

- **Date**: 2008-11-23 18:10:37
- **By**: Foo

```
If the source would be dead center a 180° rotation would mean the source would be␣
↪behind you,
but since in stereo there is no front or behind (just left and right), behind gets␣
↪indicated by
phase reversal (I know it doesn't reflect the position, but you can't because there␣
↪is only left
and right).
Also the rotation is clockwise, so a positive angles shift the source to the right,␣
↪which means
for your example if you'd rotate from 0° to -90° you'd indeed get the signal one the␣
↪left channel
and the right blank. For a mono signal (both channels identical that is) and a␣
↪rotation range of
-45° to 45° is the same as panning (with a 0dB pan law).

But I'll admit I was totally wrong and this entry in the musicdsp is the most␣
↪faultiest that there
ever was and isn't going to be useful at all, to no one.
Anyway if this is not stereo field rotation, how would YOU call it? I'd happily␣
↪forward the new
terminology to the siteadmin, so the entries' description can be changed as soon as␣
↪possible to
whatever you think it is.

I'm just glad that I'm not the only one that is using wrong terminology, e.g. the␣
↪Waves S1-Imager's
"Rotation" does the same as the above posted code, as does Nick Whitehurst's c_
↪superstereo and
others ...

So tell me what it is called and I'll see if I can get the name changed, so everyone␣
↪can be happy.
```

(continues on next page)

(continued from previous page)

```
Though I doubt I can get Waves nor any audio engineers to also adapt the new, correct␣
↪terminology,
that you will proved, for this kind of effect.

BTW if you want to discuss this further please mail to: 1337foo42bar69@trashmail.net␣
↪because there
is no need to waste more comment space about this (I now think or at least hope that␣
↪it only is a ...)
terminology discussion, because there is nothing wrong with the code itself I posted,␣
↪or is there?
```

- **Date**: 2009-07-17 11:02:36
- **By**: null

```
Waves S1 Rotation, as you said, does exactly this. It is stereo field rotation, but␣
↪in the same
way could be considered panning.

Thanks a lot for the useful code, it will be put to good use. :)
```

- **Date**: 2014-03-28 12:52:38
- **By**: moc.liamg@nabihci.nasleinad

```
I just tried this out but as you rotate the field there are points where the field is␣
↪flattened
to become simply a mono signal rotated. visualize the output on a x/y vectorscope and␣
↪perform the
roation to see what I mean.

I made a correction.  The algorithm should be like this:

r = rotation_angle

out_left   = (in_left *  cos(r))  - (in_right * sin(r + pi));
out_right = (in_left * -sin(r)) - (in_right * cos(r + pi));
```

- **Date**: 2014-06-17 22:51:38
- **By**: moc.liamg@jdcisumff

```
Why are you adding pi to the sin and cos at the end?

What will adding 3.14 to the rotation do besides move the rotation angle further 3.14?
↪ IE if rotation
angle is 90, you're just adding 3.14 to it  equaling 93.14. Why only to the right␣
↪channel? Shouldn't
that cause problems?

Wouldn't that mean that the reason this wont sound flat is because the calculations␣
↪are 3.14 off?
```

4.27 Stereo Width Control (Obtained Via Transfromation Matrix)

- **Author or source:** Michael Gruhn

- **Type:** Stereo Widener
- **Created:** 2008-03-17 16:54:42

Listing 46: notes

```
(I was quite surprised that this wasn't already in the archive, so here it is.)

This work is hereby placed in the public domain for all purposes, including use in
commercial applications.

'width' is the stretch factor of the stereo field:
width < 1: decrease in stereo width
width = 1: no change
width > 1: increase in stereo width
width = 0: mono
```

Listing 47: code

```
// calculate scale coefficient
coef_S = width*0.5;

// then do this per sample
m = (in_left  + in_right)*0.5;
s = (in_right - in_left )*coef_S;

out_left  = m - s;
out_right = m + s;
```

4.27.1 Comments

- **Date**: 2008-04-06 11:32:43
- **By**: ku.oc.oohay@895rennacs

```
Nice peace of code. I would add the following code at the end of the source to␣
↪compensate for the
loss/gain of amplitude:

out_left  /= 0.5 + coef_S;
out_right /= 0.5 + coef_S;
```

- **Date**: 2008-04-06 17:40:29
- **By**: -

```
Scanner, no I wouldn't add that. First off it is unnecessary calculation you can␣
↪rescale the MS
matrix to your liking already! Plus your methode will cause a boost by 6dBs when you␣
↪set the width
to 0 = mono. So mono signals get boosted by 6dB which I'm sure isn't what you␣
↪intented.

Note: My original code is correct that is, when you'd look at an audio signal on a␣
↪goniometer it
would scale the audio signal at the S-axis and leaving everything else unaffected.
```

(continues on next page)

(continued from previous page)

```
But as I don't want people that add the additional calculation that scanner requested␣
↪(sorry not
trying to mock you), an volume adjusted version.

[code]
// calc coefs
tmp = 1/max(1 + width,2);
coef_M = 1 * tmp;
coef_S = width * tmp;

// then do this per sample
m = (in_left + in_right)*coef_M;
s = (in_right - in_left )*coef_S;

out_left = m - s;
out_right = m + s;
[/code]
```

- **Date**: 2008-04-06 21:42:16
- **By**: ku.oc.oohay@895rennacs

```
Hi Michael,

Thanks for the correction, I have build your solution in PureData and it is better␣
↪than my
suggestion was. B.t.w. there was already a posting on stereo enhancement on this site,
↪ you can
find it under the effects section.
```

- **Date**: 2008-04-07 22:47:14
- **By**: -

```
Scanner, no problem and yes I've seen the "Stereo Enhancer" entry, though (even␣
↪though it
seems to try to achieve the same as this here) it is (as far as I can see) broken.
```

4.28 Time compression-expansion using standard phase vocoder

- **Author or source:** Cournape
- **Type:** vocoder phase time stretching
- **Created:** 2002-12-01 21:15:54
- **Linked files:** `vocoder.m`.

Listing 48: notes

```
Standard phase vocoder. For imporved techniques ( faster ), see paper of Laroche :
"Improved phase vocoder time-scale modification of audio"
Laroche, J.; Dolson, M.
Speech and Audio Processing, IEEE Transactions on , Volume: 7 Issue: 3 , May 1999
Page(s): 323 -332
```

4.29 Transistor differential amplifier simulation

- **Author or source:** ed.luosfosruoivas@naitsirhC
- **Type:** Waveshaper
- **Created:** 2004-08-09 07:46:11

Listing 49: notes

```
Writting an exam about electronic components, i learned several equations about␣
↪simulating
that stuff. One simplified equation was the tanh(x) formula for the differential
amplifier. It is not exact, but since the amplifiers are driven with only small␣
↪amplitudes
the behaviour is most often  even advanced linear.
The fact, that the amp is differential, means, that the 2n order is eliminated, so the
sound is also similar to a tube.
For a very fast use, this code is in pure assembly language (not optimized with SSE-
↪Code
yet) and performs in VST-Plugins very fast.
The code was written in delphi and if you want to translate the assembly code, you␣
↪should
know, the the parameters passing is done via registers. So pinp=EAX pout=EDX sf=ECX.
```

Listing 50: code

```
procedure Transistor(pinp,pout : PSingle; sf:Integer; Faktor: Single);
asm
 fld Faktor
@Start:
 fld [eax].single
 fmul st(0),st(1)

 fldl2e
 fmul
 fld st(0)
 frndint
 fsub st(1),st
 fxch st(1)
 f2xm1
 fld1
 fadd
 fscale     { result := z * 2**i }
 fstp st(1)

 fld st(0)
 fmulp

 fld st(0)
 fld1
 faddp
 fld1
 fsubp st(2),st(0)
 fdivp

 fstp [edx].single
```

(continues on next page)

(continued from previous page)

```
 add eax,4
 add edx,4
 loop    @Start
 fstp st(0)
end;
```

4.30 UniVox Univibe Emulator

- **Author or source:** moc.liamg@libojyr
- **Type:** 4 Cascaded all-pass filters and optocoupler approximation
- **Created:** 2010-09-09 07:52:24

Listing 51: notes

```
This is a class and class member functions for a 'Vibe derived by means of bilinear
transform of the all-pass filter stages in a UniVibe.  Some unique things happen as
↪this
filter is modulated, so this has been somewhat involved computation of filter
coefficients, and is based on summation of 1rst-order filter stages as algebraically
decoupled during circuit analysis.  A second part is an approximated model of the
↪Vactrol
used to modulate the filters, including its time response to hopefully recapture the
modulation shape.  It is likely there is a more efficient way to re-create the LFO
↪shape,
and perhaps would be best with a lookup table.   Keeping the calculation in the code
↪makes
it possible for other people to modify and improve the algorithm.

Notice no wet/dry mix is implemented in this code block's "out" function.  Originally
↪this
was implemented in the calling routine, but if you use it as a stand-alone function
↪you
may want to add summation to the input signal as it is an important part of the
↪"chorus"
mode on the Vibe.  The code as is represents only the Vibrato (warble) mode.

This is a module found in the Rakarrack guitar effects program.  It is GPL, so please
↪give
credit due and keep it free.  You can find any of the omitted parts to see more
↪precisely
how it is implemented with JACK on Linux by looking at the original sources at
sourceforge.net/projects/rakarrack.
```

Listing 52: code

```
/*
  Copyright (C) 2008-2010 Ryan Billing
  Author: Ryan Billing


 This program is free software; you can redistribute it and/or modify
```

(continues on next page)

(continued from previous page)

```
 it under the terms of version 2 of the GNU General Public License
 as published by the Free Software Foundation.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License (version 2) for more details.

 You should have received a copy of the GNU General Public License
 (version2)  along with this program; if not, write to the Free Software
 Foundation,
 Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA

*/

class Vibe
{

public:

  Vibe (float * efxoutl_, float * efxoutr_);
  ~Vibe ();
//note some of these functions not pasted below to improve clarity
//and to save space
  void out (float * smpsl, float * smpsr);
  void setvolume(int value);
  void setpanning(int value);
  void setpreset (int npreset);
  void changepar (int npar, int value);
  int getpar (int npar);
  void cleanup ();

  float outvolume;
  float *efxoutl;
  float *efxoutr;


private:
  int Pwidth;
  int Pfb;
  int Plrcross;
  int Pdepth;
  int Ppanning;
  int Pvolume;
   //all the ints above are the parameters to modify with a proper function.

  float fwidth;
  float fdepth;
  float rpanning, lpanning;
  float flrcross, fcross;
  float fb;
  EffectLFO lfo; //EffectLFO is an object that calculates the next sample from the LFO each time it's called

  float Ra, Rb, b, dTC, dRCl, dRCr, lampTC, ilampTC, minTC, alphal, alphar, stepl, stepr, oldstepl, oldstepr;
  float fbr, fbl;
```

(continues on next page)

(continued from previous page)

```
  float dalphal, dalphar;
  float lstep,rstep;
  float cperiod;
  float gl, oldgl;
  float gr, oldgr;

  class fparams {
  public:
  float x1;
  float y1;
  //filter coefficients
  float n0;
  float n1;
  float d0;
  float d1;
  } vc[8], vcvo[8], ecvc[8], vevo[8], bootstrap[8];

  float vibefilter(float data, fparams *ftype, int stage);
  void init_vibes();
  void modulate(float ldrl, float ldrr);
  float bjt_shape(float data);

float R1;
float Rv;
float C2;
float C1[8];
float beta;  //transistor forward gain.
float gain, k;
float oldcvolt[8] ;
float en1[8], en0[8], ed1[8], ed0[8];
float cn1[8], cn0[8], cd1[8], cd0[8];
float ecn1[8], ecn0[8], ecd1[8], ecd0[8];
float on1[8], on0[8], od1[8], od0[8];

   class FPreset *Fpre;


};


Vibe::Vibe (float * efxoutl_, float * efxoutr_)
{
  efxoutl = efxoutl_;
  efxoutr = efxoutr_;

//Swing was measured on operating device of: 10K to 250k.
//400K is reported to sound better for the "low end" (high resistance)
//Because of time response, Rb needs to be driven further.
//End resistance will max out to around 10k for most LFO freqs.
//pushing low end a little lower for kicks and giggles
Ra = 500000.0f;  //Cds cell dark resistance.
Ra = logf(Ra);                //this is done for clarity
Rb = 600.0f;          //Cds cell full illumination
b = exp(Ra/logf(Rb)) - CNST_E;
dTC = 0.085f;
dRCl = dTC;
dRCr = dTC;   //Right & left channel dynamic time contsants
```

(continues on next page)

(continued from previous page)

```
minTC = logf(0.005f/dTC);
//cSAMPLE_RATE is 1/SAMPLE_RATE
alphal = 1.0f - cSAMPLE_RATE/(dRCl + cSAMPLE_RATE);
alphar = alphal;
dalphal = dalphar = alphal;
lampTC = cSAMPLE_RATE/(0.02 + cSAMPLE_RATE);  //guessing 10ms
ilampTC = 1.0f - lampTC;
lstep = 0.0f;
rstep = 0.0f;
Pdepth = 127;
Ppanning = 64;
lpanning = 1.0f;
rpanning = 1.0f;
fdepth = 1.0f;
oldgl = 0.0f;
oldgr = 0.0f;
gl = 0.0f;
gr = 0.0f;
for(int jj = 0; jj<8; jj++) oldcvolt[jj] = 0.0f;
cperiod = 1.0f/fPERIOD;

init_vibes();
cleanup();

}

Vibe::~Vibe ()
{
}


void
Vibe::cleanup ()
{
//Yeah, clean up some stuff

};

void
Vibe::out (float *smpsl, float *smpsr)
{

  int i,j;
  float lfol, lfor, xl, xr, fxl, fxr;
  float vbe,vin;
  float cvolt, ocvolt, evolt, input;
  float emitterfb = 0.0f;
  float outl, outr;

  input = cvolt = ocvolt = evolt = 0.0f;

  lfo.effectlfoout (&lfol, &lfor);

  lfol = fdepth + lfol*fwidth;
  lfor = fdepth + lfor*fwidth;

   if (lfol > 1.0f)
```

(continues on next page)

(continued from previous page)

```
    lfol = 1.0f;
  else if (lfol < 0.0f)
    lfol = 0.0f;
  if (lfor > 1.0f)
    lfor = 1.0f;
  else if (lfor < 0.0f)
    lfor = 0.0f;

    lfor = 2.0f - 2.0f/(lfor + 1.0f);    //
    lfol = 2.0f - 2.0f/(lfol + 1.0f); //emulate lamp turn on/off characteristic by typical curves

  for (i = 0; i < PERIOD; i++)
    {
    //Left Lamp
     gl = lfol*lampTC + oldgl*ilampTC;
     oldgl = gl;
    //Right Lamp
     gr = lfor*lampTC + oldgr*ilampTC;
     oldgr = gr;

    //Left Cds
    stepl = gl*alphal + dalphal*oldstepl;
    oldstepl = stepl;
    dRCl = dTC*expf(stepl*minTC);
    alphal = cSAMPLE_RATE/(dRCl + cSAMPLE_RATE);
    dalphal = 1.0f - cSAMPLE_RATE/(0.5f*dRCl + cSAMPLE_RATE);     //different attack & release character
    xl = CNST_E + stepl*b;
    fxl = expf(Ra/logf(xl));

    //Right Cds
    stepr = gr*alphar + dalphar*oldstepr;
    oldstepr = stepr;
    dRCr = dTC*expf(stepr*minTC);
    alphar = cSAMPLE_RATE/(dRCr + cSAMPLE_RATE);
    dalphar = 1.0f - cSAMPLE_RATE/(0.5f*dRCr + cSAMPLE_RATE);      //different attack & release character
    xr = CNST_E + stepr*b;
    fxr = expf(Ra/logf(xr));

    if(i%16 == 0)  modulate(fxl, fxr);

    //Left Channel

   input = bjt_shape(fbl + smpsl[i]);


    emitterfb = 25.0f/fxl;
    for(j=0;j<4;j++) //4 stages phasing
    {
   cvolt = vibefilter(input,ecvc,j) + vibefilter(input + emitterfb*oldcvolt[j],vc,j);
   ocvolt = vibefilter(cvolt,vcvo,j);
   oldcvolt[j] = ocvolt;
   evolt = vibefilter(input, vevo,j);

   input = bjt_shape(ocvolt + evolt);
```

(continues on next page)

(continued from previous page)

```
    }
    fbl = fb*ocvolt;
    outl = lpanning*input;

    //Right channel

    input = bjt_shape(fbr + smpsr[i]);

    emitterfb = 25.0f/fxr;
    for(j=4;j<8;j++) //4 stages phasing
    {
   cvolt = vibefilter(input,ecvc,j) + vibefilter(input + emitterfb*oldcvolt[j],vc,j);
   ocvolt = vibefilter(cvolt,vcvo,j);
   oldcvolt[j] = ocvolt;
   evolt = vibefilter(input, vevo,j);

   input = bjt_shape(ocvolt + evolt);
    }

    fbr = fb*ocvolt;
    outr = rpanning*input;

    efxoutl[i] = outl*fcross + outr*flrcross;
    efxoutr[i] = outr*fcross + outl*flrcross;

    };

};

float
Vibe::vibefilter(float data, fparams *ftype, int stage)
{
float y0 = 0.0f;
y0 = data*ftype[stage].n0 + ftype[stage].x1*ftype[stage].n1 - ftype[stage].
↪y1*ftype[stage].d1;
ftype[stage].y1 = y0 + DENORMAL_GUARD;
ftype[stage].x1 = data;
return y0;
};

float
Vibe::bjt_shape(float data)
{
float vbe, vout;
float vin = 7.5f*(1.0f + data);
if(vin<0.0f) vin = 0.0f;
if(vin>15.0f) vin = 15.0f;
vbe = 0.8f - 0.8f/(vin + 1.0f);  //really rough, simplistic bjt turn-on emulator
vout = vin - vbe;
vout = vout*0.1333333333f -0.90588f;  //some magic numbers to return gain to unity &␣
↪zero the DC
return vout;
}

void
Vibe::init_vibes()
{
```

(continues on next page)

(continued from previous page)

```
k = 2.0f*fSAMPLE_RATE;
float tmpgain = 1.0f;
 R1 = 4700.0f;
 Rv = 4700.0f;
 C2 = 1e-6f;
 beta = 150.0f;  //transistor forward gain.
 gain = -beta/(beta + 1.0f);

//Univibe cap values 0.015uF, 0.22uF, 470pF, and 0.0047uF
C1[0] = 0.015e-6f;
C1[1] = 0.22e-6f;
C1[2] = 470e-12f;
C1[3] = 0.0047e-6f;
C1[4] = 0.015e-6f;
C1[5] = 0.22e-6f;
C1[6] = 470e-12f;
C1[7] = 0.0047e-6f;

for(int i =0; i<8; i++)
{
//Vo/Ve driven from emitter
en1[i] = k*R1*C1[i];
en0[i] = 1.0f;
ed1[i] = k*(R1 + Rv)*C1[i];
ed0[i] = 1.0f + C1[i]/C2;

// Vc~=Ve/(Ic*Re*alpha^2) collector voltage from current input.
//Output here represents voltage at the collector

cn1[i] = k*gain*Rv*C1[i];
cn0[i] = gain*(1.0f + C1[i]/C2);
cd1[i] = k*(R1 + Rv)*C1[i];
cd0[i] = 1.0f + C1[i]/C2;

//Contribution from emitter load through passive filter network
ecn1[i] = k*gain*R1*(R1 + Rv)*C1[i]*C2/(Rv*(C2 + C1[i]));
ecn0[i] = 0.0f;
ecd1[i] = k*(R1 + Rv)*C1[i]*C2/(C2 + C1[i]);
ecd0[i] = 1.0f;

// %Represents Vo/Vc.  Output over collector voltage
on1[i] = k*Rv*C2;
on0[i] = 1.0f;
od1[i] = k*Rv*C2;
od0[i] = 1.0f + C2/C1[i];

//%Bilinear xform stuff
tmpgain =  1.0f/(cd1[i] + cd0[i]);
vc[i].n1 = tmpgain*(cn0[i] - cn1[i]);
vc[i].n0 = tmpgain*(cn1[i] + cn0[i]);
vc[i].d1 = tmpgain*(cd0[i] - cd1[i]);
vc[i].d0 = 1.0f;

tmpgain =  1.0f/(ecd1[i] + ecd0[i]);
ecvc[i].n1 = tmpgain*(ecn0[i] - ecn1[i]);
ecvc[i].n0 = tmpgain*(ecn1[i] + ecn0[i]);
ecvc[i].d1 = tmpgain*(ecd0[i] - ecd1[i]);
```

(continues on next page)

(continued from previous page)

```
ecvc[i].d0 = 1.0f;

tmpgain =  1.0f/(od1[i] + od0[i]);
vcvo[i].n1 = tmpgain*(on0[i] - on1[i]);
vcvo[i].n0 = tmpgain*(on1[i] + on0[i]);
vcvo[i].d1 = tmpgain*(od0[i] - od1[i]);
vcvo[i].d0 = 1.0f;

tmpgain =  1.0f/(ed1[i] + ed0[i]);
vevo[i].n1 = tmpgain*(en0[i] - en1[i]);
vevo[i].n0 = tmpgain*(en1[i] + en0[i]);
vevo[i].d1 = tmpgain*(ed0[i] - ed1[i]);
vevo[i].d0 = 1.0f;

// bootstrap[i].n1
// bootstrap[i].n0
// bootstrap[i].d1
}


};

void
Vibe::modulate(float ldrl, float ldrr)
{
float tmpgain;
float R1pRv;
float C2pC1;
Rv = 4700.0f + ldrl;
R1pRv = R1 + Rv;


for(int i =0; i<8; i++)
{
if(i==4) {
Rv = 4700.0f + ldrr;
R1pRv = R1 + Rv;
}

C2pC1 = C2 + C1[i];
//Vo/Ve driven from emitter
ed1[i] = k*(R1pRv)*C1[i];
//ed1[i] = R1pRv*kC1[i];

// Vc~=Ve/(Ic*Re*alpha^2) collector voltage from current input.
//Output here represents voltage at the collector
cn1[i] = k*gain*Rv*C1[i];
//cn1[i] = kgainCl[i]*Rv;
//cd1[i] = (R1pRv)*C1[i];
cd1[i]=ed1[i];

//Contribution from emitter load through passive filter network
ecn1[i] = k*gain*R1*cd1[i]*C2/(Rv*(C2pC1));
//ecn1[i] = iC2pC1[i]*kgainR1C2*cd1[i]/Rv;
ecd1[i] = k*cd1[i]*C2/(C2pC1);
//ecd1[i] = iC2pC1[i]*k*cd1[i]*C2/(C2pC1);

```

(continues on next page)

(continued from previous page)

```
// %Represents Vo/Vc.  Output over collector voltage
on1[i] = k*Rv*C2;
od1[i] = on1[i];

//%Bilinear xform stuff
tmpgain =  1.0f/(cd1[i] + cd0[i]);
vc[i].n1 = tmpgain*(cn0[i] - cn1[i]);
vc[i].n0 = tmpgain*(cn1[i] + cn0[i]);
vc[i].d1 = tmpgain*(cd0[i] - cd1[i]);

tmpgain =  1.0f/(ecd1[i] + ecd0[i]);
ecvc[i].n1 = tmpgain*(ecn0[i] - ecn1[i]);
ecvc[i].n0 = tmpgain*(ecn1[i] + ecn0[i]);
ecvc[i].d1 = tmpgain*(ecd0[i] - ecd1[i]);
ecvc[i].d0 = 1.0f;

tmpgain =  1.0f/(od1[i] + od0[i]);
vcvo[i].n1 = tmpgain*(on0[i] - on1[i]);
vcvo[i].n0 = tmpgain*(on1[i] + on0[i]);
vcvo[i].d1 = tmpgain*(od0[i] - od1[i]);

tmpgain =  1.0f/(ed1[i] + ed0[i]);
vevo[i].n1 = tmpgain*(en0[i] - en1[i]);
vevo[i].n0 = tmpgain*(en1[i] + en0[i]);
vevo[i].d1 = tmpgain*(ed0[i] - ed1[i]);

}


};
```

4.31 Variable-hardness clipping function

- **Author or source:** Laurent de Soras (moc.ecrofmho@tnerual)
- **Created:** 2004-04-07 09:36:46
- **Linked files:** `laurent.gif`.

Listing 53: 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"
```

Listing 54: code

```
f (x) = sign (x) * pow (atan (pow (abs (x), k)), (1 / k));
```

4.31.1 Comments

- **Date**: 2003-11-15 03:56:35

- **By**: moc.liamtoh@sisehtnysorpitna

```
// Use this function instead of atan and see performance increase drastically :)

inline double fastatan( double x )
{
    return (x / (1.0 + 0.28 * (x * x)));
}
```

- **Date**: 2004-07-16 09:36:33
- **By**: gro.psdcisum@maps

```
The greater k becomes the lesser is the change in the form of f(x, k). I recommend␣
↪using
f2(x, k2) = sign(x) * pow(atan(pow(abs(x), 1 / k2)), k2) ,  k2 in [0.01, 1]
where k2 is the "clipping softness" (k2 = 0.01 means "hardclipping", k2 = 1 means
"softclipping"). This gives better control over the clipping effect.
```

- **Date**: 2004-08-12 18:42:58
- **By**: gro.liamon@demrofniton

```
Don't know if i understood ok , but, how can i clip at diferent levels than -1.0/1.0␣
↪using this
func? tried several ways but none seems to work
```

- **Date**: 2004-08-14 04:02:00
- **By**: moc.liamg@noteex

```
The most straightforward way to adjust the level (x) at which the signal is clipped␣
↪would be to
multiply the signal by 1/x before the clipping function then multiply it again by x␣
↪afterwards.
```

- **Date**: 2004-10-09 23:27:57
- **By**: ed.xmg@releuhcsc

```
Atan is a nice softclipping function, but you can do without pow().

x: input value
a: clipping factor (0 = none, infinity = hard)
ainv: 1/a

y = ainv * atan( x * a );
```

- **Date**: 2006-05-28 20:32:49
- **By**: uh.etle.fni@yfoocs

```
Even better, you can normalize the output using:

shape = 1..infinity

precalc:
  inv_atan_shape=1.0/atan(shape);
process:
  output = inv_atan_shape * atan(input*shape);
```

(continues on next page)

(continued from previous page)

```
This gives a very soft transition from no distortion to hard clipping.
```

- **Date**: 2011-01-03 14:07:35
- **By**: moc.liamg@nalevart

```
Scoofy,

What do you mean with 'shape'?
Is it a new parameter?
```

- **Date**: 2013-01-18 02:42:09
- **By**: moc.liamtoh@niffumtohrepus

```
sign (x) * pow (atan (pow (abs (x), k)), (1 / k));

OUCH! That's a lot of pow, atan and floating point division - probably kill most CPU
↪'s :) My
experience has been that any sigmoid function will create decent distortion if␣
↪oversampled and
eq'ed properly. You can adjust the "hardness" of the clipping by simply changing a␣
↪couple
coefficients, or by increasing/decreasing the input gain: like so:

y = A * tanh(B * x)

Cascading a couple/few of these will give you bone-crushing, Megadeth/Slayer style␣
↪grind while
rolling back the gain gives a Fender Twin sound.

Two cascaded half-wave soft clippers gives duty-cycle modulation and a transfer curve␣
↪similar to
the 3/2 power curve of tubes. I came up w/ a model based on that solution after␣
↪reading reading
this: http://www.trueaudio.com/at_eetjlm.htm (orig. link at www.simulanalog.org)
```

- **Date**: 2013-06-14 11:42:26
- **By**: moc.liamtoh@niffumtohrepus

```
If anyone is interested, I have a working amp modeler and various c/c++ classes that␣
↪model
distortion circuits by numerical solutions to non-linear ODE's like those described by
Yeh, Smith, Macak, Pakarinen, et al. in their PhD disertations and DAFX papers.␣
↪Although
static waveshapers/filters can give decent approximations & cool sounds, they lack␣
↪the dynamic
properties of the actual circuits and have poor harmonics. I also have whitepapers on␣
↪my
implementations for those that think math is cool. Drop me a line for more info.
```

4.32 WaveShaper

- **Author or source:** Bram de Jong

- **Type:** waveshaper
- **Created:** 2002-01-17 02:17:49

Listing 55: 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.
```

Listing 56: code

```
f(x,a) = x*(abs(x) + a)/(x^2 + (a-1)*abs(x) + 1)
```

4.33 Waveshaper

- **Author or source:** Jon Watte
- **Type:** waveshaper
- **Created:** 2002-01-17 02:19:17

Listing 57: 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.
```

Listing 58: code

```
(input: a == "overdrive amount")

z = M_PI * a;
s = 1/sin(z)
b = 1/a

if (x > b)
  f(x) = 1
```

(continues on next page)

(continued from previous page)

```
else
  f(x) = sin(z*x)*s
```

4.34 Waveshaper

- **Author or source:** Partice Tarrabia and Bram de Jong
- **Created:** 2002-01-17 02:21:49

Listing 59: notes

```
amount should be in [-1..1[ Plot it and stand back in astonishment! ;)
```

Listing 60: code

```
x = input in [-1..1]
y = output
k = 2*amount/(1-amount);

f(x) = (1+k)*x/(1+k*abs(x))
```

4.34.1 Comments

- **Date**: 2002-06-27 07:15:59
- **By**: moc.noicratse@ajelak

```
I haven't compared this to the other waveshapers, but its behavior with input outside␣
↪the [-1..1]
range is interesting. With a relatively moderate shaping amounts which don't distort␣
↪in-range
signals severely, it damps extremely out-of-range signals fairly hard, e.g. x = 100,
k = 0.1 yields y = 5.26; as x goes to infinity, y approaches 5.5. This might come in␣
↪handy
to control nonlinear processes which would otherwise be prone to computational blowup.
```

4.35 Waveshaper (simple description)

- **Author or source:** Jon Watte
- **Type:** Polynomial; Distortion
- **Created:** 2002-08-15 00:45:22

Listing 61: 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
```

(continues on next page)

(continued from previous page)

```
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 :-)
```

Listing 62: code

```
float waveshape_distort( float in ) {
  return 1.5f * in - 0.5f * in *in * in;
}
```

4.35.1 Comments

- **Date**: 2005-06-30 09:41:07
- **By**: ed.luosfosruoivas@naitsirhC

```
Yes! It's one of the most simple waveshaper and you know the amount of oversampling!
Works very nice (and fast).
```

4.36 Waveshaper :: Gloubi-boulga

- **Author or source:** Laurent de Soras on IRC
- **Created:** 2002-03-17 15:40:13

Listing 63: notes

```
Multiply input by gain before processing
```

Listing 64: 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));
```

4.36.1 Comments

- **Date**: 2004-09-25 21:42:39
- **By**: ten.etelirt@gnihtliam

```
you can use a taylor series approximation for the exp , save time by realizing that
exp(-x) = 1/exp(x), use newton's method to calculate the sqrt with less precision...
and if you use SIMD instructions, you can calculate several values in parallel. dunno
what the savings would be like, but it would surely be faster.
```

- **Date**: 2005-05-25 22:32:21
- **By**: ed.luosfosruoivas@naitsirhC

```
// Maybe something like this:

function GloubiBoulga(x:Single):Single;
var a,b:Single;
begin
 x:=x*0.686306;
 a:=1+exp(sqrt(f_abs(x))*-0.75);
 b:=exp(x);
 Result:=(b-exp(-x*a))*b/(b*b+1);
end;

still expensive, but...
```

- **Date**: 2005-05-28 00:49:48
- **By**: ed.luosfosruoivas@naitsirhC

```
A Taylor series doesn't work very well, because the approximation effects the result
→very early due to
a) numerical critical additions & subtractions of approximations
b) approximating approximated "a" makes the result evene more worse.

The above version has already been improved, by removing 2 of 5 exp() functions.

You can also try to express the exp(x)+exp(-x) as cosh(x) with its approximation. So:

b:=exp(x);
Result:=(b-exp(-x*a))*b/(b*b+1);

would be:

Result:=(exp(x)-exp(-x*a))*20160/40320+x*x*(20160+ x*x*(1680+x*x*(56+x*x)));

but this is again more worse. Anyone else?
```

- **Date**: 2005-09-13 09:55:55
- **By**: llun.ved@regguHwodahS

```
Use table lookup with interpolation.
```

- **Date**: 2005-09-22 01:07:58
- **By**: ten.baltg@liced

```
IMHO, you can use
x - 0.15 * x^2 - 0.15 * x^3
instead of this scary formula.

I try to explain my position with this small graph:
```

(continues on next page)

(continued from previous page)

```
http://liteprint.com/download/replacment.png

This is only first step, if you want to get more correct result you can use␣
↪interpolation
method called method of minimal squares (this is translation from russian, maybe in␣
↪england
it has another name)
```

- **Date**: 2005-09-22 07:19:07
- **By**: ku.oc.snosrapsd@psdcisum

```
That's much better decil - thx for that!

DSP
```

- **Date**: 2005-09-22 11:05:07
- **By**: ten.baltg@liced

```
You are welcome :)

Now I've working under plugin with wapeshapping processing like this. I've put a link␣
↪to it
here, when I've done it.
```

- **Date**: 2005-09-24 01:15:38
- **By**: ten.baltg@liced

```
You can check my version:
http://liteprint.com/download/SweetyVST.zip

Please, send comments and suggestions to my email.

Dmitry.
```

- **Date**: 2005-10-27 09:57:44
- **By**: moc.liamtoh@12_namyaj

```
Which formula exactly did you use decil, for your plugin? How do you get different␣
↪harmonics
from this algo. thanx

jay
```

- **Date**: 2005-11-15 09:09:48
- **By**: ten.etelirt@liam

```
wow, blast from the past seeing this turn up on kvraudio.

christian - i'd have thought that an advantage of using a taylor series approximation␣
↪would be
that it limits the order of the polynomial (and the resulting bandwidth) somewhat.  it
↪'s been ages
since i tested, but i thought i got some reasonable sounding results using the taylor␣
↪series
```

(continues on next page)

(continued from previous page)

```
approximation.  maybe not.

decil - isn't that a completely unrelated polynomial (similar to the common and cheap␣
↪x - a x^3 ?).
i'd think you'd have to do something about the dc from the x^2 term, too (or do a␣
↪sign(x)*x^2).
anyway, your plugin sounds to be popular so i look forward to checking it out later␣
↪at home.
```

CHAPTER 5

Other

5.1 16-Point Fast Integer Sinc Interpolator.

- **Author or source:** moc.liamg@tramum
- **Created:** 2005-11-15 22:28:31

Listing 1: notes

```
This is designed for fast upsampling with good quality using only a 32-bit␣
↪accumulator.
Sound quality is very good. Conceptually it resamples the input signal 32768x and␣
↪performs
nearest-neighbour to get the requested sample rate. As a result downsampling will␣
↪result
in aliasing.

The provided Sinc table is Blackman-Harris windowed with a slight lowpass. The table
entries are 16-bit and are 16x linear-oversampled. It should be pretty easy to figure␣
↪out
how to make your own table for it.

Code provided is in Java. Converting to C/MMX etc. should be pretty trivial.

Remember the interpolator requires a number of samples before and after the sample to␣
↪be
interpolated, so you can't resample the whole of a passed input buffer in one go.

Have fun,
Martin
```

Listing 2: code

```
public class SincResampler {
    private final int FP_SHIFT = 15;
    private final int FP_ONE = 1 << FP_SHIFT;
    private final int FP_MASK = FP_ONE - 1;


    private final int POINT_SHIFT = 4; // 16 points

    private final int OVER_SHIFT = 4; // 16x oversampling

    private final short[] table = {

             0, -7,  27, -71,  142, -227,  299,  32439,   299,  -227,  142,  -71,  27,  -7,  0,  0,

             0,  0,  -5,  36, -142,  450, -1439, 32224,  2302,  -974,  455, -190,  64, -15,  2,  0,

             0,  6, -33, 128, -391, 1042, -2894, 31584,  4540, -1765,  786, -318, 105, -25,  3,  0,

             0, 10, -55, 204, -597, 1533, -4056, 30535,  6977, -2573, 1121, -449, 148, -36,  5,  0,

            -1, 13, -71, 261, -757, 1916, -4922, 29105,  9568, -3366, 1448, -578, 191, -47,  7,  0,

            -1, 15, -81, 300, -870, 2185, -5498, 27328, 12263, -4109, 1749, -698, 232, -58,  9,  0,

            -1, 15, -86, 322, -936, 2343, -5800, 25249, 15006, -4765, 2011, -802, 269, -68, 10,  0,

            -1, 15, -87, 328, -957, 2394, -5849, 22920, 17738, -5298, 2215, -885, 299, -77, 12,  0,

             0, 14, -83, 319, -938, 2347, -5671, 20396, 20396, -5671, 2347, -938, 319, -83, 14,  0,

             0, 12, -77, 299, -885, 2215, -5298, 17738, 22920, -5849, 2394, -957, 328, -87, 15, -1,

             0, 10, -68, 269, -802, 2011, -4765, 15006, 25249, -5800, 2343, -936, 322, -86, 15, -1,

             0,  9, -58, 232, -698, 1749, -4109, 12263, 27328, -5498, 2185, -870, 300, -81, 15, -1,

             0,  7, -47, 191, -578, 1448, -3366,  9568, 29105, -4922, 1916, -757, 261, -71, 13, -1,

             0,  5, -36, 148, -449, 1121, -2573,  6977, 30535, -4056, 1533, -597, 204, -55, 10,  0,

             0,  3, -25, 105, -318,  786, -1765,  4540, 31584, -2894, 1042, -391, 128, -33,  6,  0,
```

(continues on next page)

(continued from previous page)

```

            0,   2, -15,   64, -190,   455,   -974,   2302, 32224, -1439,   450, -142,   36,  -5,   0,   0,

            0,   0,   -7,   27,   -71,   142,   -227,    299, 32439,    299, -227,   142, -71,  27,  -7,   0

    };



    /*

    private final int POINT_SHIFT = 1; // 2 points

    private final int OVER_SHIFT = 0; // 1x oversampling

    private final short[] table = {

            32767,      0,

            0     , 32767

    };

    */



    private final int POINTS = 1 << POINT_SHIFT;

    private final int INTERP_SHIFT = FP_SHIFT - OVER_SHIFT;

    private final int INTERP_BITMASK = ( 1 << INTERP_SHIFT ) - 1;


    /*
            input - array of input samples
            inputPos - sample position ( must be at least POINTS/2 + 1, ie. 7 )
            inputFrac - fractional sample position ( 0 <= inputFrac < FP_ONE )
            step - number of input samples per output sample * FP_ONE
            lAmp - left output amplitude ( 1.0 = FP_ONE )
            lBuf - left output buffer
            rAmp - right output amplitude ( 1.0 = FP_ONE )
            rBuf - right output buffer
            pos - offset into output buffers
            count - number of output samples to produce
    */

    public void resample( short[] input, int inputPos, int inputFrac, int step,

                    int lAmp, int[] lBuf, int rAmp, int[] rBuf, int pos, int count ) {

            for( int p = 0; p < count; p++ ) {

                    int tabidx1 = ( inputFrac >> INTERP_SHIFT ) << POINT_SHIFT;

```

(continues on next page)

(continued from previous page)

```
                int tabidx2 = tabidx1 + POINTS;

                int bufidx = inputPos - POINTS/2 + 1;

                int a1 = 0, a2 = 0;

                for( int t = 0; t < POINTS; t++ ) {

                    a1 += table[ tabidx1++ ] * input[ bufidx ] >> 15;

                    a2 += table[ tabidx2++ ] * input[ bufidx ] >> 15;

                    bufidx++;

                }

                int out = a1 + ( ( a2 - a1 ) * ( inputFrac & INTERP_BITMASK ) >> INTERP_SHIFT );

                lBuf[ pos ] += out * lAmp >> FP_SHIFT;

                rBuf[ pos ] += out * rAmp >> FP_SHIFT;

                pos++;

                inputFrac += step;


                inputPos += inputFrac >> FP_SHIFT;

                inputFrac &= FP_MASK;

            }

        }

}
```

5.2 16-to-8-bit first-order dither

- **Author or source:** Jon Watte
- **Type:** First order error feedforward dithering code
- **Created:** 2002-04-12 13:52:36

Listing 3: 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.
```

(continues on next page)

(continued from previous page)

```
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.
```

Listing 4: 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;
}
```

5.3 3rd order Spline interpollation

- **Author or source:** Dave from Muon Software, originally from Josh Scholar
- **Created:** 2002-01-17 03:14:54

Listing 5: 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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.
```

Listing 6: 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 )))));
}
```

5.3.1 Comments

- **Date**: 2002-05-21 06:14:20
- **By**: moc.a@a

```
What is x ?
```

- **Date**: 2002-06-09 19:45:59
- **By**: yahoo.co.uk@sewar_ekim

```
The samples being interpolated represent the wave amplitude at a particular instant
↪of time, T - an impulse train. So each sample is the amplitude at T=0,1,2,3 etc.

The purpose of interpolation is to determine the amplitude, a, for an arbitrary t,
↪where t is any real number:

   p1       p0   a    n0       n1
   :        :    :    :        :
   0-------1---t---2-------3------> T
            :    :
```

(continues on next page)

(continued from previous page)

```
           :   :
           <-x->

x = t - T(p0)

-
myk
```

- **Date**: 2002-06-09 19:53:03
- **By**: yahoo.co.uk@sewar_ekim

```
Dang! My nice diagram had its spacing stolen, and it now makes no sense!

p1, p0, n0, n1 are supposed to line up with 0,1,2,3 respectively. a is supposed to␣
↪line up with the t. And finally, <-x-> spans between 1 and t.

-
myk
```

- **Date**: 2002-09-16 02:34:30
- **By**: lc.arret@assenacf

```
1.- What is 5f ?

2.- How I can test this procedure?.

Thank you
```

- **Date**: 2003-04-15 10:59:26
- **By**: moc.oohay@SIHT_EVOMER_ralohcshsoj

```
This is years later. but just in case anyone has the same problem as fcanessa...  In␣
↪C or C++ you can append an 'f' to a number to make it single precision, so .5f is␣
↪the same as .5
```

- **Date**: 2012-07-10 13:51:17
- **By**: ac.cisum-mutnauq@noidc

```
About that thing you've said "5 point seems to be the sweet spot". Well, it might␣
↪depends on the sampling rate.
```

5.4 5-point spline interpollation

- **Author or source:** Joshua Scholar,David Waugh
- **Type:** interpollation
- **Created:** 2002-01-17 03:12:34

Listing 7: 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)))));
};
```

5.4.1 Comments

- **Date**: 2003-05-27 12:20:46
- **By**: moc.oohay@SIHTEVOMERralohcshsoj

```
The code works much better if you oversample before interpolating. If you oversample␣
↪enough (maybe 4 to 6 times oversampling) then the results are audiophile quality.
```

- **Date**: 2010-08-26 20:55:45
- **By**: moc.oohay@xofirgomsnart

```
This looks old...but if anybody reads this:
What do you mean by oversample first?  That is practically what you are doing with␣
↪interpolation.  For example, if you want to oversample 6x, you would interpolate 5␣
↪evenly spaced points in between p2 and p3 using 5 points at base frequency centered␣
↪around p2.  The 5-point spline interpolation seems like a lower CPU algorithm than␣
↪a good sinc interpolation, and as a bonus it does not have much of a transient␣
↪response (only 5 samples worth).

My main target application for something like this is delay line interpolation where␣
↪there is a concern regarding high frequency notch depth...5th order interpolation␣
↪is certainly an improvement over linear interpolation :)
```

- **Date**: 2012-10-04 08:00:31
- **By**: Josh Scholar

```
By oversample I meant do a windowed sinc doubling oversample a couple times.

The point is that a 4 times oversample can be based on table values and only gives␣
↪you points exactly 1/4, 1/2 and 3/4 between the samples.
```

(continues on next page)

(continued from previous page)

```
Then the spline can be used to interpolate totally arbitrary points between those,␣
↪say speeding up and slowing down as needed, at very high quality.

If you don't oversample first, you'll get an audible amount of aliasing, though not␣
↪as much as a linear interpolation. Unless the source has a lot of roll off (which␣
↪is equivalent to it being oversampled anyway).
```

- **Date**: 2014-08-16 17:55:33
- **By**: moc.liamg@rellimennad.sirhc

```
Can any explain the derivation of this?
```

5.5 Allocating aligned memory

- **Author or source:** Benno Senoner
- **Type:** memory allocation
- **Created:** 2002-01-17 03:08:46

Listing 8: 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() !
```

Listing 9: 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) {

```

(continues on next page)

(continued from previous page)

```
  int *ptr2=(int *)ptr - 1;
  ptr -= *ptr2;
  free(ptr);
}
```

5.6 Antialiased Lines

- **Author or source:** moc.xinortceletrams@urugra
- **Type:** A slow, ugly, and unoptimized but short method to perform antialiased lines in a framebuffer
- **Created:** 2004-04-07 09:38:53

Listing 10: notes

```
Simple code to perform antialiased lines in a 32-bit RGBA (1 byte/component)
↪framebuffer.

pframebuffer <- unsigned char* to framebuffer bytes (important: Y flipped line order!
[like in the way Win32 CreateDIBSection works...])

client_height=framebuffer height in lines
client_width=framebuffer width in pixels (not in bytes)

This doesnt perform any clip checl so it fails if coordinates are set out of bounds.

sorry for the engrish
```

Listing 11: code

```
//
// By Arguru
//
void PutTransPixel(int const x,int const y,UCHAR const r,UCHAR const g,UCHAR const b,
↪UCHAR const a)
{
    unsigned char* ppix=pframebuffer+(x+(client_height-(y+1))*client_width)*4;
    ppix[0]=((a*b)+(255-a)*ppix[0])/256;
    ppix[1]=((a*g)+(255-a)*ppix[1])/256;
    ppix[2]=((a*r)+(255-a)*ppix[2])/256;
}

void LineAntialiased(int const x1,int const y1,int const x2,int const y2,UCHAR const
↪r,UCHAR const g,UCHAR const b)
{
    // some useful constants first
    double const dw=x2-x1;
    double const dh=y2-y1;
    double const slx=dh/dw;
    double const sly=dw/dh;

    // determine wichever raster scanning behaviour to use
    if(fabs(slx)<1.0)
    {
```

(continues on next page)

(continued from previous page)

```
            // x scan
            int tx1=x1;
            int tx2=x2;
            double raster=y1;

            if(x1>x2)
            {
                    tx1=x2;
                    tx2=x1;
                    raster=y2;
            }

            for(int x=tx1;x<=tx2;x++)
            {
                    int const ri=int(raster);

                    double const in_y0=1.0-(raster-ri);
                    double const in_y1=1.0-(ri+1-raster);

                    PutTransPixel(x,ri+0,r,g,b,in_y0*255.0);
                    PutTransPixel(x,ri+1,r,g,b,in_y1*255.0);

                    raster+=slx;
            }
    }
    else
    {
            // y scan
            int ty1=y1;
            int ty2=y2;
            double raster=x1;

            if(y1>y2)
            {
                    ty1=y2;
                    ty2=y1;
                    raster=x2;
            }

            for(int y=ty1;y<=ty2;y++)
            {
                    int const ri=int(raster);

                    double const in_x0=1.0-(raster-ri);
                    double const in_x1=1.0-(ri+1-raster);

                    PutTransPixel(ri+0,y,r,g,b,in_x0*255.0);
                    PutTransPixel(ri+1,y,r,g,b,in_x1*255.0);

                    raster+=sly;
            }
    }
}
```

5.6.1 Comments

- **Date**: 2004-02-11 12:53:12
- **By**: Gog

```
Sorry, but what does this have to do with music DSP ??
```

- **Date**: 2004-02-14 17:39:38
- **By**: moc.liamtoh@101_vap

```
well, for drawing envelopes, waveforms, etc on screen in your DSP app....
```

- **Date**: 2004-02-15 11:59:04
- **By**: Gog

```
But... there are TONS of graphic toolkits to do just that. No reason to "roll your own
↪". One f.i. is GDI+ (works darn well to be honest), or if you want non-M$ (and␣
↪better!) go with AGG at (http://www.antigrain.com). And there are even open-source␣
↪cross-platform toolkits (if you want to do Unix and Mac without coding).
Graphics and GUIs is a very time-consuming task to do from scratch, therefore I think␣
↪using libraries such as the above is the way to go, liberating energy to do the DSP␣
↪stuff... ;-)
```

- **Date**: 2004-03-11 02:56:19
- **By**: Rich

```
I don't want a toolkit, I want antialiased line drawing and nothing more. Everything␣
↪else is fine.
```

- **Date**: 2004-03-11 17:32:09
- **By**: Justin

```
Anyone know how to get the pointer to the framebuffer?  Perhaps there is a different␣
↪answer for different platforms?
```

- **Date**: 2004-09-12 07:39:28
- **By**: ed.stratnemesab@kcaahcs.leinad

```
you can also draw everything in a 2x (vertically and horizontally) higher resolution␣
↪and then reduce the size again by always taking the average of 4 pixels. that works␣
↪well.
```

- **Date**: 2008-11-20 18:07:21
- **By**: moc.liamg@okolila

```
I think it can be useful to those designing graphical synths.

But the Wu line algorithm is considerably more fast and works only with integers.

http://en.wikipedia.org/wiki/Xiaolin_Wu's_line_algorith
```

5.7 Automatic PDC system

- **Author or source:** Tebello Thejane
- **Type:** the type that actually works, completely
- **Created:** 2006-07-16 11:39:56
- **Linked files:** `pdc.pdf`.

Listing 12: notes

```
No, people, implementing PDC is actually not as difficult as you might think it is.

This paper presents a solution to the problem of latency inherent in audio effects
processors, and the two appendices give examples of the method being applied on␣
↪Cubase SX
(with an example which its native half-baked PDC fails to solve properly) as well as a
convoluted example in FL Studio (taking advantage of the flexible routing capabilities
introduced in version 6 of the software). All that's necessary to understand it is a␣
↪grasp
of basic algebra and an intermediate understanding of how music production software␣
↪works
(no need to understand the Laplace transform, linear processes, sigma and integral
notation... YAY!).

Please do send me any feedback (kudos, errata, flames, job offers, questions,␣
↪comments)
you might have - my email address is included in the paper - or simply use musicdsp.
↪org's
own commenting system.

Tebello Thejane.
```

Listing 13: code

```
(I have sent the PDF to Bram as he suggested)
```

5.7.1 Comments

- **Date**: 2006-07-18 08:12:00
- **By**: moc.liamg@saoxyz

```
Oops! RBJ's first name is Robert, not Richard! Man, that's a bad one...
```

- **Date**: 2006-07-21 10:24:53
- **By**: moc.liamg@saoxyz

```
Okay, I've sent a fixed version to Bram. It should be uploaded shortly. Bigger␣
↪diagrams, too, so there's less aliasing in Adode Acrobat Reader. Hopefully no more␣
↪embarisingly bad errors (like misspelling my own name, or something...).
```

- **Date**: 2006-10-09 15:27:24
- **By**: moc.liamg@saoxyz

```
                The revised version may be found here:
http(:)//www.vormdicht.nl /misc/PDC_paper-rev.pdf
Naturally, you need to remove the brackets from the address.
```

5.8 Base-2 exp

- **Author or source:** Laurent de Soras
- **Created:** 2002-01-17 03:06:08

Listing 14: 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.
```

Listing 15: 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);
}
```

5.8.1 Comments

- **Date**: 2002-04-10 02:48:33
- **By**: gro.ecruosrv@cimotabus

```
Here is the code to detect little endian processor:

         union
         {
```

(continues on next page)

(continued from previous page)

```
            short   val;
            char    ch[sizeof( short )];
         } un;
         un.val = 256; // 0x10;

         if (un.ch[1] == 1)
         {
            // then we're little
         }

I've tested the fast_exp2() on both little and big endian (intel, AMD, and motorola)␣
↪processors, and the comment is correct.

Here is the completed function that works on all endian systems:

inline double fast_exp2( const double val )
{
   // is the machine little endian?
   union
   {
      short   val;
      char    ch[sizeof( short )];
   } un;
   un.val = 256; // 0x10;
   // if un.ch[1] == 1 then we're little

   // return 2 to the power of val (exp base2)
   int    e;
   double ret;

   if (val >= 0)
   {
      e = int (val);
      ret = val - (e - 1);

      if (un.ch[1] == 1)
         ((*(1 + (int *) &ret)) &= ~(2047 << 20)) += (e + 1023) << 20;
      else
         ((*((int *) &ret)) &= ~(2047 << 20)) += (e + 1023) << 20;
   }
   else
   {
      e = int (val + 1023);
      ret = val - (e - 1024);

      if (un.ch[1] == 1)
         ((*(1 + (int *) &ret)) &= ~(2047 << 20)) += e << 20;
      else
         ((*((int *) &ret)) &= ~(2047 << 20)) += e << 20;
   }

   return ret;
}
```

5.9 Bit-Reversed Counting

- **Author or source:** moc.oohay@ljbliam
- **Created:** 2004-06-19 10:10:39

Listing 16: notes

```
Bit-reversed ordering comes up frequently in FFT implementations.  Here is a non-
↪branching
algorithm (given in C) that increments the variable "s" bit-reversedly from 0 to N-1,
where N is a power of 2.
```

Listing 17: code

```
int r = 0;        // counter
int s = 0;        // bit-reversal of r/2
int N = 256;      // N can be any power of 2
int N2 = N << 1;  // N<<1 == N*2

do {
  printf("%u ", s);
  r += 2;
  s ^= N - (N / (r&-r));
}
while (r < N2);
```

5.9.1 Comments

- **Date**: 2005-08-10 07:20:50
- **By**: moc.oohay@r_adihaw

```
This will give the bit reversal of N number of elements(where N is a power of 2). If␣
↪we want reversal of a particular number out of N,is there any optimised way other␣
↪than doing bit wise operations
```

- **Date**: 2006-03-30 23:17:55
- **By**: moc.oohay@ljbliam

```
There's a better way that doesn't require counting, branching, or division.  It's␣
↪probably the fastest way of doing bit reversal without a special instruction.  I␣
↪got this from Jörg's FXT book:

unsigned r; // value to be bit-reversed

// Assume r is 32 bits
r = ((r & 0x55555555) << 1) | ((r & 0xaaaaaaaa) >> 1);
r = ((r & 0x33333333) << 2) | ((r & 0xcccccccc) >> 2);
r = ((r & 0x0f0f0f0f) << 4) | ((r & 0xf0f0f0f0) >> 4);
r = ((r & 0x00ff00ff) << 8) | ((r & 0xff00ff00) >> 8);
r = ((r & 0x0000ffff) << 16) | ((r & 0xffff0000) >> 16);
```

- **Date**: 2010-09-20 15:22:09
- **By**: ed.bew@mfyknarf

```
The way mentioned in the comment might be faster but is fixed to 32 bits. If you do a␣
↪FFT with 1024 points you need 10 bits bit-reversal. Thus the originally mentioned␣
↪algorithm is more flexible because it works for any bit width. If you use it for␣
↪FFT (that's actually the only case you normally use bit-reversal) you either need␣
↪to calculate the bit-reversal for each array index, so counting upwards in bit-
↪reversal order is not such a bad way. I'm not sure whether the second algorithm is␣
↪really faster than the counter if you consider the whole array. (There are 5␣
↪instructions per line making 25 instructions in sum for each calculated index with␣
↪the second algorithm compared to 7 instructions in the counting algorithm)
```

5.10 Block/Loop Benchmarking

- **Author or source:** moc.xinortceletrams@urugra
- **Type:** Benchmarking Tool
- **Created:** 2003-06-24 07:30:43

Listing 18: notes

```
Requires CPU with RDTSC support
```

Listing 19: code

```
// Block-Process Benchmarking Code using rdtsc
// useful for measure DSP block stuff
// (based on Intel papers)
// 64-bit precission
// VeryUglyCode(tm) by Arguru

// globals
UINT time,time_low,time_high;

// call this just before enter your loop or whatever
void bpb_start()
{
    // read time stamp to EAX
    __asm rdtsc;
    __asm mov time_low,eax;
    __asm mov time_high,edx;
}

// call the following function just after your loop
// returns average cycles wasted per sample
UINT bpb_finish(UINT const num_samples)
{
    __asm rdtsc
    __asm sub eax,time_low;
    __asm sub edx,time_high;
    __asm div num_samples;
    __asm mov time,eax;
    return time;
}
```

5.10.1 Comments

- **Date**: 2004-05-16 18:20:13
- **By**: moc.sulp.52retsinnab@etep

```
If running windows on a mutliprocessor system, apparently it is worth calling:

SetThreadAffinityMask(GetCurrentThread(), 1);

to reduce artefacts.

(see http://msdn.microsoft.com/visualc/vctoolkit2003/default.aspx?pull=/library/en-us/
↪dv_vstechart/html/optimization.asp)
```

- **Date**: 2004-08-26 00:33:18
- **By**: rf.eerf@uerum.emualliug

```
__asm sub eax,time_low;
__asm sub edx,time_high;

should be

__asm sub eax,time_low
__asm SBB edx,time_high // substract with borrow
```

5.11 Branchless Clipping

- **Author or source:** ku.oc.snosrapsd@psdcisum
- **Type:** Clipping at 0dB, with none of the usual 'if..then..'
- **Created:** 2005-10-30 10:33:19

Listing 20: notes

```
I was working on something that I wanted to ensure that the signal never went above␣
↪0dB,
and a branchless solution occurred to me.

It works by playing with the structure of a single type, shifting the sign bit down to
make a new mulitplicand.

calling MaxZerodB(mydBSample) will ensure that it will never stray over 0dB.
By playing with signs or adding/removing offsets, this offers a complete branchless
limiting solution, no matter whether dB or not (after all, they're all just numbers...
↪).

eg:
Limit to <=0   : sample:=MaxZerodB(sample);
Limit to <=3   : sample:=MaxZerodB(sample-3)+3;
Limit to <=-4  : sample:=MaxZerodB(sample+4)-4;

Limit to >=0   : sample:=-MaxZerodB(-sample);
Limit to >=2   : sample:=-MaxZerodB(-sample+2)+2;
Limit to >=-1.5: sample:=-MaxZerodB(-sample-1.5)-1.5;
```

(continues on next page)

(continued from previous page)

```
Whether it actually saves any CPU cycles remains to be seen, but it was an interesting
diversion for half an hour :)

[Translating from pascal to other languages shouldn't be too hard, and for doubles,␣
↪you'll
need to fiddle it abit :)]
```

Listing 21: code

```
function MaxZerodB(dBin:single):single;
var tmp:longint;
begin
     //given that leftmost bit of a longint indicates the negative,
     //  if we shift that down to bit0, and multiply dBin by that
     //  it will return dBin, or zero :)
     tmp:=(longint((@dBin)^) and $80000000) shr 31;
     result:=dBin*tmp;
end;
```

5.11.1 Comments

- **Date**: 2005-11-29 18:33:09
- **By**: hotpop.com@blargg

```
Since most processors include a sign-preserving right shift, you can right shift by␣
↪31 to end up with either -1 (all bits set) or 0, then mask the original value with␣
↪it:

out = (in >> 31) & in;
```

- **Date**: 2005-12-01 20:33:28
- **By**: moc.liamg@tramum

```
I prefer this method, using a sign-preserving shift, as it can clip a signal to␣
↪arbitrary bounds:

over = upper_limit - samp
mask = over >> 31
over = over & mask
samp = samp + over
over = samp - lower_limit
mask = over >> 31
over = over & mask
samp = samp - over

Is it faster? Maybe on modern machines with 20-plus-stage pipelines and if the signal␣
↪is clipped often, as the branches are not predictable.
```

- **Date**: 2006-03-22 17:53:44
- **By**: ku.oc.snosrapsd@psdcisum

```
hmm.. Did some looking into the sign preserving thing. My laptop has an P3 which didn
↪'t preserve as mentioned, and my work PC (P4HT) didn't either. Maybe its an AMD or␣
↪motorola thing :)

unless it's how delphi interprets the shr.. what does a C++ compiler generate for '>>
↪' ?
```

- **Date**: 2006-03-28 14:42:44
- **By**: moc.liamg@tramum

```
C and C++ have sign-preserving shifts. If the value is negative, a right shift will␣
↪add ones onto the left hand side (thus -2 becomes -1 etc).

Java also has a non-sign-preserving right shift operator (>>>).

I tried googling for information on how Delphi handles shifts, but nothing turned up.␣
↪Looks like you might need to use in-line assembly :/
```

- **Date**: 2006-03-30 02:47:35
- **By**: ed.ko0@oreb

```
Here my SAR function for Delphi+FreePascal

FUNCTION SAR(Value,Shift:INTEGER):INTEGER; {$IFDEF CPU386}ASSEMBLER; REGISTER;{$ELSE}{
↪$IFDEF FPC}INLINE;{$ELSE}REGISTER;{$ENDIF}{$ENDIF}
{$IFDEF CPU386}
ASM
 MOV ECX,EDX
 SAR EAX,CL
END;
{$ELSE}
BEGIN
 RESULT:=(Value SHR Shift) OR (($FFFFFFFF+(1-((Value AND (1 SHL 31)) SHR 31) AND␣
↪ORD(Shift<>0))) SHL (32-Shift));
END;
{$ENDIF}
```

- **Date**: 2006-03-30 03:19:59
- **By**: ed.ko0@oreb

```
Ny branchless clipping functions (the first is faster than the second)

FUNCTION Clip(Value,Min,Max:SINGLE):SINGLE; ASSEMBLER; STDCALL;
CONST Constant0Dot5:SINGLE=0.5;
ASM
 FLD DWORD PTR Value
 FLD DWORD PTR Min
 FLD DWORD PTR Max

 FLD ST(2)
 FSUB ST(0),ST(2)
 FABS
 FADD ST(0),ST(2)
 FADD ST(0),ST(1)
```

(continues on next page)

(continued from previous page)

```
 FLD ST(3)
 FSUB ST(0),ST(2)
 FABS
 FSUBP ST(1),ST(0)
 FMUL DWORD PTR Constant0Dot5

 FFREE ST(4)
 FFREE ST(3)
 FFREE ST(2)
 FFREE ST(1)
END;

FUNCTION ClipDSP(Value:SINGLE):SINGLE; {$IFDEF CPU386} ASSEMBLER; REGISTER;
ASM
 MOV EAX,DWORD PTR Value
 AND EAX,$80000000

 AND DWORD PTR Value,$7FFFFFFF

 FLD DWORD PTR Value
 FLD1
 FSUBP ST(1),ST(0)
 FSTP DWORD PTR Value

 MOV EDX,DWORD PTR Value
 AND EDX,$80000000
 SHR EDX,31
 NEG EDX
 AND DWORD PTR Value,EDX

 FLD DWORD PTR Value
 FLD1
 FADDP ST(1),ST(0)
 FSTP DWORD PTR Value

 OR DWORD PTR Value,EAX

 FLD DWORD PTR Value
END;
{$ELSE}
VAR ValueCasted:LONGWORD ABSOLUTE Value;
    Sign:LONGWORD;
BEGIN
 Sign:=ValueCasted AND $80000000;
 ValueCasted:=ValueCasted AND $7FFFFFFF;
 Value:=Value-1;
 ValueCasted:=ValueCasted AND (-LONGWORD((ValueCasted AND $80000000) SHR 31));
 Value:=Value+1;
 ValueCasted:=ValueCasted OR Sign;
 RESULT:=Value;
END;
{$ENDIF}
```

5.12 Calculate notes (java)

- **Author or source:** gro.kale@ybsral
- **Type:** Java class for calculating notes with different in params
- **Created:** 2002-06-21 02:33:13
- **Linked files:** `Frequency.java`.

Listing 22: 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
```

5.13 Center separation in a stereo mixdown

- **Author or source:** Thiburce BELAVENTURE
- **Created:** 2004-02-11 14:00:08

Listing 23: notes

```
One year ago, i found a little trick to isolate or remove the center in a stereo␣
↪mixdown.

My method use the time-frequency representation (FFT). I use a min fuction between␣
↪left
and right channels (for each bin) to create the pseudo center. I apply a phase␣
↪correction,
and i substract this signal to the left and right signals.

Then, we can remix them after treatments (or without) to produce a stereo signal in
output.

This algorithm (I called it "TBIsolator") is not perfect, but the result is very nice,
better than the phase technic (L substract R...). I know that it is not mathematically
correct, but as an estimation of the center, the exact match is very hard to obtain.␣
↪So,
it is not so bad (just listen the result and see).

My implementation use a 4096 FFT size, with overlap-add method (factor 2). With a␣
↪lower
FFT size, the sound will be more dirty, and with a 16384 FFT size, the center will␣
↪have
too much high frequency (I don't explore why this thing appears).

I just post the TBIsolator code (see FFTReal in this site for implement the FFT␣
↪engine).

pIns and pOuts buffers use the representation of the FFTReal class (0 to N/2-1: real
parts, N/2 to N-1: imaginary parts).
```

(continues on next page)

(continued from previous page)

```
Have fun with the TBIsolator algorithm ! I hope you enjoy it and if you enhance it,
contact me (it's my baby...).

P.S.: the following function is not optimized.
```

Listing 24: code

```
/* ============================================================ */
/* nFFTSize must be a power of 2                                */
/* ============================================================ */
/* Usage examples:                                              */
/* - suppress the center:  fAmpL = 1.f, fAmpC = 0.f, fAmpR = 1.f */
/* - keep only the center: fAmpL = 0.f, fAmpC = 1.f, fAmpR = 0.f */
/* ============================================================ */

void processTBIsolator(float *pIns[2], float *pOuts[2], long nFFTSize, float fAmpL,
↪float fAmpC, float fAmpR)
{
    float fModL, fModR;
    float fRealL, fRealC, fRealR;
    float fImagL, fImagC, fImagR;
    double u;

    for ( long i = 0, j = nFFTSize / 2; i < nFFTSize / 2; i++ )
    {
            fModL = pIns[0][i] * pIns[0][i] + pIns[0][j] * pIns[0][j];
            fModR = pIns[1][i] * pIns[1][i] + pIns[1][j] * pIns[1][j];

            // min on complex numbers
            if ( fModL > fModR )
            {
                    fRealC = fRealR;
                    fImagC = fImagR;
            }
            else
            {
                    fRealC = fRealL;
                    fImagC = fImagL;
            }

            // phase correction...
            u = fabs(atan2(pIns[0][j], pIns[0][i]) - atan2(pIns[1][j], pIns[1][i])) /
↪3.141592653589;

            if ( u >= 1 ) u -= 1.;

            u = pow(1 - u*u*u, 24);

            fRealC *= (float) u;
            fImagC *= (float) u;

            // center extraction...
            fRealL = pIns[0][i] - fRealC;
            fImagL = pIns[0][j] - fImagC;

            fRealR = pIns[1][i] - fRealC;
```

(continues on next page)

(continued from previous page)

```
            fImagR = pIns[1][j] - fImagC;

            // You can do some treatments here...

            pOuts[0][i] = fRealL * fAmpL + fRealC * fAmpC;
            pOuts[0][j] = fImagL * fAmpL + fImagC * fAmpC;

            pOuts[1][i] = fRealR * fAmpR + fRealC * fAmpC;
            pOuts[1][j] = fImagR * fAmpR + fImagC * fAmpC;
    }
}
```

5.13.1 Comments

- **Date**: 2004-02-11 18:40:30
- **By**: moc.ecrubiht@cehcnamf

```
I am sorry, my source code is not totally correct.

1 - the for is:

for ( long i = 0, j = nFFTSize / 2; i < nFFTSize / 2; i++, j++ )

2 - the correct min is:

if ( fModL > fModR )
{
    fRealC = pIns[1][i];
    fImagC = pIns[1][j];
}
else
{
    fRealC = pIns[0][i];
    fImagC = pIns[0][j];
}

3 - in the phase correction:

if ( u >= 1 ) u -= 1.;

must be replaced by:

if ( u >= 1 ) u = 2 - u;

Thiburce 'TB' BELAVENTURE
```

5.14 Center separation in a stereo mixdown

- **Author or source:** Thiburce BELAVENTURE
- **Created:** 2004-02-14 15:14:09

Listing 25: notes

```
One year ago, i found a little trick to isolate or remove the center in a stereo
mixdown.

My method use the time-frequency representation (FFT). I use a min fuction between
left
and right channels (for each bin) to create the pseudo center. I apply a phase
correction,
and i substract this signal to the left and right signals.

Then, we can remix them after treatments (or without) to produce a stereo signal in
output.

This algorithm (I called it "TBIsolator") is not perfect, but the result is very nice,
better than the phase technic (L substract R...). I know that it is not mathematically
correct, but as an estimation of the center, the exact match is very hard to obtain.
So,
it is not so bad (just listen the result and see).

My implementation use a 4096 FFT size, with overlap-add method (factor 2). With a
lower
FFT size, the sound will be more dirty, and with a 16384 FFT size, the center will
have
too much high frequency (I don't explore why this thing appears).

I just post the TBIsolator code (see FFTReal in this site for implement the FFT
engine).

pIns and pOuts buffers use the representation of the FFTReal class (0 to N/2-1: real
parts, N/2 to N-1: imaginary parts).

Have fun with the TBIsolator algorithm ! I hope you enjoy it and if you enhance it,
contact me (it's my baby...).

P.S.: the following function is not optimized.
```

Listing 26: code

```
/* ============================================================= */
/* nFFTSize must be a power of 2                                 */
/* ============================================================= */
/* Usage examples:                                               */
/* - suppress the center:  fAmpL = 1.f, fAmpC = 0.f, fAmpR = 1.f */
/* - keep only the center: fAmpL = 0.f, fAmpC = 1.f, fAmpR = 0.f */
/* ============================================================= */

void processTBIsolator(float *pIns[2], float *pOuts[2], long nFFTSize, float fAmpL,
float fAmpC, float fAmpR)
{
    float fModL, fModR;
    float fRealL, fRealC, fRealR;
    float fImagL, fImagC, fImagR;
    double u;

    for ( long i = 0, j = nFFTSize / 2; i < nFFTSize / 2; i++ )
    {
```

(continues on next page)

(continued from previous page)

```
            fModL = pIns[0][i] * pIns[0][i] + pIns[0][j] * pIns[0][j];
            fModR = pIns[1][i] * pIns[1][i] + pIns[1][j] * pIns[1][j];

            // min on complex numbers
            if ( fModL > fModR )
            {
                    fRealC = fRealR;
                    fImagC = fImagR;
            }
            else
            {
                    fRealC = fRealL;
                    fImagC = fImagL;
            }

            // phase correction...
            u = fabs(atan2(pIns[0][j], pIns[0][i]) - atan2(pIns[1][j], pIns[1][i])) /
↪3.141592653589;

            if ( u >= 1 ) u -= 1.;

            u = pow(1 - u*u*u, 24);

            fRealC *= (float) u;
            fImagC *= (float) u;

            // center extraction...
            fRealL = pIns[0][i] - fRealC;
            fImagL = pIns[0][j] - fImagC;

            fRealR = pIns[1][i] - fRealC;
            fImagR = pIns[1][j] - fImagC;

            // You can do some treatments here...

            pOuts[0][i] = fRealL * fAmpL + fRealC * fAmpC;
            pOuts[0][j] = fImagL * fAmpL + fImagC * fAmpC;

            pOuts[1][i] = fRealR * fAmpR + fRealC * fAmpC;
            pOuts[1][j] = fImagR * fAmpR + fImagC * fAmpC;
    }
}
```

5.15 Cheap pseudo-sinusoidal lfo

- **Author or source:** moc.regnimmu@regnimmuf
- **Created:** 2004-04-07 09:39:28

Listing 27: notes

```
Although the code is written in standard C++, this algorithm is really better suited
↪for
dsps where one can take advantage of multiply-accumulate instructions and where the
```

(continues on next page)

(continued from previous page)

```
required phase accumulator can be easily implemented by masking a counter.

It provides a pretty cheap roughly sinusoidal waveform that is good enough for an lfo.
```

Listing 28: code

```
// x should be between -1.0 and 1.0
inline
double pseudo_sine(double x)
{
    // Compute 2*(x^2-1.0)^2-1.0
    x *= x;
    x -= 1.0;
    x *= x;
    // The following lines modify the range to lie between -1.0 and 1.0.
   // If a range of between 0.0 and 1.0 is acceptable or preferable
   // (as in a modulated delay line) then you can save some cycles.
    x *= 2.0;
    x -= 1.0;
}
```

5.15.1 Comments

- **Date**: 2004-03-31 09:08:57
- **By**: ed.bew@hakkeb

```
You forgot a

return x;
```

- **Date**: 2004-04-05 19:43:15
- **By**: moc.regnimmu@regnimmuf

```
Doh! You're right.

-Frederick
```

5.16 Clipping without branching

- **Author or source:** Laurent de Soras (moc.ecrofmho@tnerual)
- **Type:** Min, max and clip
- **Created:** 2004-04-07 09:35:57

Listing 29: 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
```

(continues on next page)

(continued from previous page)

```
temporary results). Thus, 1e-20 will be rounded to 0. The other (positive) side
↪effect is
the denormal number elimination.
```

Listing 30: 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);
}
```

5.16.1 Comments

- **Date**: 2002-04-15 04:05:45
- **By**: ten.xfer@spelk

```
AFAIK, the fabs() is using if()...
```

- **Date**: 2002-05-27 11:48:41
- **By**: moc.tecollev@ydna

```
fabs/fabsf do not use if and are quicker than:
if (x<0) x = -x;
Do the speed tests yourself if you don't believe me!
```

- **Date**: 2002-06-26 09:55:50
- **By**: moc.noicratse@ajelak

```
Depends on CPU and optimization options, but yes, Visual C++/x86/full optimization
↪uses intrinsic fabs, which is very cool.
```

- **Date**: 2003-10-21 15:38:40
- **By**: moc.semag-allirreug@regninned.trannel

```
And ofcourse you could always use one of those nifty bit-tricks for fabs :)

(Handy when you don't want to link with the math-library, like when coding a␣
↪softsynth for a 4Kb-executable demo :))
```

- **Date**: 2004-01-31 04:24:22
- **By**: moc.dh2a@ydna

```
according to my benchmarks (using the cpu clock cycle counter), fabs and the 'nifty␣
↪bit tricks' have identicle performance characterstics, EXCEPT that with the nifty␣
↪bit trick, sometimes it has a -horrible- penalty, which depends on the context...,␣
↪maybe it does not optimize consistently?  I use libmath fabs now.  (i'm using gcc-3.
↪3/linux on a P3)
```

- **Date**: 2004-04-12 05:07:58
- **By**: moc.noicratse@ajelak

```
Precision can be a major problem with these. In particular, if you have an algorithm␣
↪that blows up with negative input, don't guard via clip( in, 0.0, 1.0 ) - it will␣
↪occasionally go negative.
```

5.17 Constant-time exponent of 2 detector

- **Author or source:** Brent Lehman (moc.oohay@ljbliam)
- **Created:** 2002-02-10 12:53:15

Listing 31: 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  ;)
```

Listing 32: 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;
```

5.17.1 Comments

- **Date**: 2002-02-12 03:20:11

- **By**: moc.oohay@xrotcnuf

```
I think I prefer:

if (! (size & (size - 1))) return;

I'm not positive this is fewer instructions than the above, but I think it's easier
↪to see why it works (n and n-1 will share bits unless n is a power of two), and it
↪doesn't require two's-complement.

 - Tom 7
```

5.18 Conversion and normalization of 16-bit sample to a floating point number

- **Author or source:** George Yohng
- **Created:** 2007-05-02 13:34:21

Listing 33: code

```
float out;
signed short in;

// This code does the same as
//   out = ((float)in)*(1.0f/32768.0f);
//
// Depending on the architecture and conversion settings,
// it might be more optimal, though it is always
// advisable to check its speed against genuine
// algorithms.

((unsigned &)out)=0x43818000^in;
out-=259.0f;
```

5.18.1 Comments

- **Date**: 2007-05-15 06:09:54
- **By**: moc.mot@lx_iruy

```
Hi George Yohng

I tried it... but it's create the heavy noise!!
```

- **Date**: 2007-09-20 17:51:12
- **By**: George Yohng

```
Correction:
    ((unsigned &)out)=0x43818000^((unsigned short)in);
    out-=259.0f;

(needs to have a cast to 'unsigned short')
```

5.19 Conversions on a PowerPC

- **Author or source:** James McCartney
- **Type:** motorola ASM conversions
- **Created:** 2002-01-17 03:07:18

Listing 34: 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
```

5.20 Cubic interpollation

- **Author or source:** Olli Niemitalo
- **Type:** interpollation
- **Created:** 2002-01-17 03:05:33

- **Linked files:** `other001.gif`.

Listing 35: notes

```
(see linkfile)
finpos is the fractional, inpos the integer part.
```

Listing 36: 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;
```

5.21 Cure for malicious samples

- **Author or source:** moc.eh-u@sru
- **Type:** Filters Denormals, NaNs, Infinities
- **Created:** 2005-03-24 00:32:54

Listing 37: notes

```
A lot of incidents can happen during processing samples. A nasty one is␣
↪denormalization,
which makes cpus consume insanely many cycles for easiest instructions.

But even worse, if you have NaNs or Infinities inside recursive structures, maybe due␣
↪to
division by zero,  all subsequent samples that are multiplied with these values will␣
↪get
"infected" and become NaN or Infinity. Your sound makes BLIPPP and that was it,␣
↪silence
from the speakers.

Thus I've written a small function that sets all of these cases to 0.0f.

You'll notice that I treat a buffer of floats as unsigned integers. And I avaoid␣
↪branches
by using comparison results as 0 or 1.

When compiled with GCC, this function should not create any "hidden" branches, but you
should verify the assembly code anyway. Sometimes some parenthesis do the trick...

;)  Urs
```

Listing 38: code

```
#ifndef UInt32
#define UInt32 unsigned int
```

(continues on next page)

(continued from previous page)

```
#endif

void erase_All_NaNs_Infinities_And_Denormals( float* inSamples, int& inNumberOfSamples )
{
    UInt32* inArrayOfFloats = (UInt32*) inSamples;

    for ( int i = 0; i < inNumberOfSamples; i++ )
    {
            UInt32 sample = *inArrayOfFloats;
            UInt32 exponent = sample & 0x7F800000;

                    // exponent < 0x7F800000 is 0 if NaN or Infinity, otherwise 1
                    // exponent > 0 is 0 if denormalized, otherwise 1

            int aNaN = exponent < 0x7F800000;
            int aDen = exponent > 0;

            *inArrayOfFloats++ = sample * ( aNaN & aDen );

    }
}
```

5.21.1 Comments

- **Date**: 2005-05-14 17:18:12
- **By**: dont-email-me

```
#include <inttypes.h>
and use std::uint32_t
or typedef (not #define)

int const & inNumberOfSamples
```

- **Date**: 2005-10-14 18:36:07
- **By**: DevilishHabib

```
Isn't it bad to declare variables within for loop?
If someone has VC++ standard (no optimizer included, thanks Bill :-( ) , the cycles gained by removing denormals, will be eaten by declaring 4 variables per loop cycle, so watch out !
```

- **Date**: 2007-05-11 05:51:36
- **By**: if.iki@xemxet

```
DevilishHabib, that's rubbish. It doesn't matter where the declaration is as long as the code works. Declaring outside the loop is the same thing (you can verify this).

Urs, nice code but you don't get rid of branches just like that. Comparision is comparision no matter what. Your code is equal to "int aNaN = exponent < 0x7F800000 ? 1 : 0;" which is equal to "int aNan = 0; if (exponent < 0x7F800000) aNan = 1;" If we are talking about x86 asm here, there is no instruction that would do the conditional assignment needed. MMX/SSE has it, though.
```

- **Date**: 2014-10-18 18:36:44
- **By**: none

```
texmex, nope, the result of < or > does not create any branches on x86.
```

5.22 Denormal DOUBLE variables, macro

- **Author or source:** Jon Watte
- **Created:** 2002-03-17 15:44:31

Listing 39: notes

```
Use this macro if you want to find denormal numbers and you're using doubles...
```

Listing 40: 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;
}
```

5.22.1 Comments

- **Date**: 2005-05-14 17:19:48
- **By**: dont-email-me

```
put the #if inside the function itself
```

5.23 Denormal numbers

- **Author or source:** Compiled by Merlijn Blaauw
- **Created:** 2002-01-17 03:06:39
- **Linked files:** `other001.txt`.

Listing 41: notes

```
 this text describes some ways to avoid denormalisation. Denormalisation happens when
FPU's go mad processing very small numbers
```

5.23.1 Comments

- **Date**: 2004-01-31 05:20:38
- **By**: moc.dh2a@ydna

```
See also the entry about 'branchless min, max and clip' by Laurent Soras in this
↪section,

Using the following function,

float clip (float x, float a, float b)
{
   x1 = fabs (x-a);
   x2 = fabs (x-b);
   x = x1 + (a+b);
   x -= x2;
   x *= 0.5;
   return (x);
}

If you apply clipping from -1.0 to 1.0 will have a side effect of squashing denormal
↪numbers to zero due to loss of precision on the order of ~1.0.e-20.  The upside is
↪that it is branchless, but possibly more expensive than adding noise and certainly
↪more so than adding a DC offset.
```

5.24 Denormal numbers, the meta-text

- **Author or source:** Laurent de Soras
- **Created:** 2002-02-15 00:16:30
- **Linked files:** `denormal.pdf`.

Listing 42: 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)
```

5.25 Denormalization preventer

- **Author or source:** gol
- **Created:** 2005-03-31 16:57:07

Listing 43: notes

```
A fast tiny numbers sweeper using integer math. Only for 32bit floats. Den_Thres is␣
↪your
32bit (normalized) float threshold, something small enough but big enough to prevent
future denormalizations.

EAX=input buffer
EDX=length
(adapt to your compiler)
```

Listing 44: code

```
    MOV   ECX,EDX
    LEA   EDI,[EAX+4*ECX]
    NEG   ECX
    MOV   EDX,Den_Thres
    SHL   EDX,1
    XOR   ESI,ESI

    @Loop:
    MOV   EAX,[EDI+4*ECX]
    LEA   EBX,[EAX*2]
    CMP   EBX,EDX
    CMOVB EAX,ESI
    MOV   [EDI+4*ECX],EAX

    INC   ECX
    JNZ   @Loop
```

5.26 Denormalization preventer

- **Author or source:** eb.tenyks@didid
- **Created:** 2006-08-05 16:37:20

Listing 45: notes

```
Because I still see people adding noise or offset to their signal to avoid slow
denormalization, here's a piece of code to zero out (near) tiny numbers instead.

Why zeroing out is better? Because a fully silent signal is better than a little␣
↪offset,
or noise. A host or effect can detect silent signals and choose not to process them␣
↪in a
safe way.
Plus, adding an offset or noise reduces huge packets of denormalization, but still␣
↪leaves
some behind.
Also, truncating is what the DAZ (Denormals Are Zero) SSE flag does.

This code uses integer comparison, and a CMOV, so you need a Pentium Pro or higher.
There's no need for an SSE version, as if you have SSE code you're probably already␣
↪using
the DAZ flag instead (but I advise plugins not to mess with the SSE flags, as the␣
↪host is
```

(continues on next page)

(continued from previous page)

```
likely to have DAZ switched on already). This is for FPU code. Should work much faster
than crap FPU comparison.

Den_Thres is your threshold, it cannot be denormalized (would be pointless). The␣
↪function
is Delphi, if you want to adapt, just make sure EAX is the buffer and EDX is length
(Delphi register calling convention - it's not the same in C++).
```

Listing 46: code

```
const  Den_Thres:Single=1/$1000000;

procedure PrevFPUDen_Buffer(Buffer:Pointer;Length:Integer);
asm
    PUSH  ESI
    PUSH  EDI
    PUSH  EBX

    MOV   ECX,EDX
    LEA   EDI,[EAX+4*ECX]
    NEG   ECX
    MOV   EDX,Den_Thres
    SHL   EDX,1
    XOR   ESI,ESI

    @Loop:
    MOV   EAX,[EDI+4*ECX]
    LEA   EBX,[EAX*2]
    CMP   EBX,EDX
    CMOVB EAX,ESI
    MOV   [EDI+4*ECX],EAX

    INC   ECX
    JNZ   @Loop

    POP   EBX
    POP   EDI
    POP   ESI
End;
```

5.26.1 Comments

- **Date**: 2006-08-14 05:36:29
- **By**: uh.etle.fni@yfoocs

```
You can zero out denormals by adding and subtracting a small number.

void kill_denormal_by_quantization(float &val)
{
  static const float anti_denormal = 1e-18;
  val += anti_denormal;
  val -= anti_denormal;
}
```

(continues on next page)

(continued from previous page)

```
Reference: Laurent de Soras' great article on denormal numbers:
ldesoras.free.fr/doc/articles/denormal-en.pdf

I tend to add DC because it is faster than quantization. A slight DC offset (0.
↪000000000000000001) won't hurt. That's -360 decibels...
```

- **Date**: 2006-08-14 09:20:43
- **By**: gol

```
>>You can zero out denormals by adding and subtracting a small number

But with drawbacks as explained in his paper.

As for the speed, not sure which is the faster. Especially since the FPU speed is too␣
↪manufacturer-dependant (read: it's crap in pentiums), and mine is using integer.

>>A slight DC offset (0.000000000000000001) won't hurt

As I wrote, it really does.. hurt the sequencer, that can't detect pure silence and␣
↪optimize things accordingly. A host can detect near-silence, but it can't know␣
↪which offset value YOU chose, so may use a lower threshold.
```

- **Date**: 2006-08-14 09:33:35
- **By**: gol

```
Btw, I happen to see I had already posted this code, probably years ago, doh!

Anyway this version gives more explanation.

And here's more:
The LEA EBX,[EAX*2] is to get rid of the sign bit.
And the integer comparison of float values can be done providing both are the same␣
↪sign (I'm not quite sure it works on denormals, but we don't care, since they're␣
↪the ones we want to zero out, so our threshold won't be denormalized).
```

- **Date**: 2010-03-10 13:29:06
- **By**: moc.liamg@sisehtnysorpitna

```
You could also add input noise and assure output samples are reset to 0 if they're␣
↪below a certain treshold, slightly higher than your noise volume. That ensures␣
↪hosts can do proper tail detection and it's cheap.
```

5.27 Dither code

- **Author or source:** Paul Kellett
- **Type:** Dither with noise-shaping
- **Created:** 2002-01-17 03:13:20

Listing 47: 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;
```

Listing 48: 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
```

5.28 Dithering

- **Author or source:** Paul Kellett
- **Created:** 2002-02-11 17:41:21
- **Linked files:** `nsdither.txt`.

Listing 49: notes

```
(see linked file)
```

5.29 Double to Int

- **Author or source:** many people, implementation by Andy M00cho
- **Type:** pointer cast (round to zero, or 'trunctate')
- **Created:** 2002-01-17 03:04:41

Listing 50: 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
```

Listing 51: 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:

```

(continues on next page)

(continued from previous page)

```
__double2fixmagic    dd 000000000h,042380000h

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

5.29.1 Comments

- **Date**: 2007-01-28 20:13:49
- **By**: ude.odu@grebnesie.nitram

```
www.stereopsis.com/FPU.html credits one Sree Kotay for this code.
```

- **Date**: 2007-07-11 06:17:12
- **By**: kd.sgnik3@sumsar

```
On PC this may be faster/easier:

int ftoi(float x)
{
  int res;
  __asm
  {
    fld x
    fistp res
  }
  return res;
}

int dtoi(double x)
{
  return ftoi((float)x);
}
```

5.30 Envelope Follower

- **Author or source:** ers
- **Created:** 2003-03-12 04:08:16

Listing 52: code

```
#define V_ENVELOPE_FOLLOWER_NUM_POINTS      2000
class vEnvelopeFollower :
{
      public:
            vEnvelopeFollower();
            virtual ~vEnvelopeFollower();
            inline void Calculate(float *b)
            {
```

(continues on next page)

(continued from previous page)

```
                        envelopeVal -= *buff;
                        if (*b < 0)
                                envelopeVal += *buff = -*b;
                        else
                                envelopeVal += *buff = *b;
                        if (buff++ == bufferEnd)
                                buff = buffer;
                }
                void SetBufferSize(float value);
                void GetControlValue(){return envelopeVal / (float)bufferSize;}

        private:
                float   buffer[V_ENVELOPE_FOLLOWER_NUM_POINTS];
                float   *bufferEnd, *buff, envelopeVal;
                int     bufferSize;
          float val;
};

vEnvelopeFollower::vEnvelopeFollower()
{
        bufferEnd = buffer + V_ENVELOPE_FOLLOWER_NUM_POINTS-1;
        buff = buffer;
        val = 0;
        float *b = buffer;
        do
        {
                *b++ = 0;
        }while (b <= bufferEnd);
        bufferSize = V_ENVELOPE_FOLLOWER_NUM_POINTS;
        envelopeVal= 0;
}

vEnvelopeFollower::~vEnvelopeFollower()
{

}

void vEnvelopeFollower::SetBufferSize(float value)
{

        bufferEnd = buffer + (bufferSize = 100 + (int)(value * ((float)V_ENVELOPE_
↪FOLLOWER_NUM_POINTS-102)));
        buff = buffer;
        float val =  envelopeVal / bufferSize;
        do
        {
                *buff++ = val;
        }while (buff <= bufferEnd);
        buff = buffer;
}
```

5.30.1 Comments

- **Date**: 2007-01-17 13:46:04
- **By**: gro.akeeb@evets

```
Nice contribution, but I have a couple of questions...

Looks like there is a typo on GetControlValue... should return a float. Also, I am
↪not clear on the reason for it taking a pointer to a float.

Is there any noticeable speed improvement with the "if (*b < 0)" code, as opposed to
↪using fabs? I would hope that a decent compiler library would inline this (but haven
↪'t cracked open the disassembler to find out).
```

5.31 Exponential curve for

- **Author or source:** moc.liamg@321tiloen
- **Type:** Exponential curve
- **Created:** 2008-10-29 17:29:03

Listing 53: notes

```
When you design a frequency control for your filters you may need an exponential
↪curve to
give the lower frequencies more resolution.

F=20-20000hz
x=0-100%

Case (control middle point):
x=50%
F=1135hz

Ploted diagram with 5 points:
http://img151.imageshack.us/img151/9938/expplotur3.jpg
```

Listing 54: code

```
//tweak - 14.15005 to change middle point and F(max)
F = 19+floor(pow(4,x/14.15005))+x*20;
```

5.31.1 Comments

- **Date**: 2008-10-30 13:47:16
- **By**: moc.liamg@321tiloen

```
same function with the more friendly exp(x)

y = 19+floor(exp(x/10.2071))+x*20;

middle point (x=50) is still at 1135hz
```

- **Date**: 2008-10-31 01:14:13
- **By**: moc.liamg@321tiloen

```
Here is another function:
This one is much more expensive but should sound more linear.

//t - offset
//x - 0-100%
//y - 20-20000hz

t = 64.925;
y = floor(exp(x*log(1.059))*t - t/1.45);

Comparison between the two:
[IMG]http://img528.imageshack.us/img528/641/plot1nu1.jpg[/IMG]
```

- **Date**: 2008-11-01 14:58:20
- **By**: moc.liamg@321tiloen

```
Yet another one! :)
This is one should be the most linear one out of the 3. The 50% appears to be exactly
the same as Voxengo span midpoint.

//x - 0-100%
//y - 20-20k

y = floor(exp((16+x*1.20103)*log(1.059))*8.17742);

//x=0, y=20
//x=50, y=639
//x=100, y=20000
```

5.32 Exponential parameter mapping

- **Author or source:** Russell Borogove
- **Created:** 2002-03-17 15:42:33

Listing 55: notes

```
Use this if you want to do an exponential map of a parameter (mParam) to a range
(mMin -
mMax).
Output is in mData...
```

Listing 56: 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)
```

(continues on next page)

(continued from previous page)

```
{
  mData = mMax;
}
```

5.32.1 Comments

- **Date**: 2002-03-26 00:28:53
- **By**: moc.nsm@seivadrer

```
No point in using heavy functions when lighter-weight functions work just as well.␣
↪Use ln instead of log10f, and exp instead of pow(10,x). Log-linear is the same, no␣
↪matter which base you're using, and base e is way more efficient than base 10.
```

- **Date**: 2002-12-06 01:31:55
- **By**: moc.noicratse@ajelak

```
Thanks for the tip. A set of VST param wrapper classes which offers linear float,␣
↪exponential float, integer selection, and text selection controls, using this␣
↪technique for the exponential response, can be found in the VST source code archive␣
↪-- finally.
```

- **Date**: 2003-08-21 16:01:01
- **By**: moc.oohay@noi_tca

```
Just made my day!
pretty useful :) cheers Aktion
```

- **Date**: 2006-09-08 18:27:33
- **By**: agillesp@gmail

```
You can trade an (expensive) ln for a (cheaper) divide here because of the␣
↪logarithmic identity:

ln(x) - ln(y) == ln(x/y)
```

5.33 Fast Float Random Numbers

- **Author or source:** moc.liamg@seir.kinimod
- **Created:** 2009-10-29 13:39:29

Listing 57: notes

```
a small and fast implementation for random float numbers in the range [-1,1], usable␣
↪as
white noise oscillator.

compared to the naive usage of the rand() function it gives a speedup factor of 9-10.

compared to a direct implementation of the rand() function (visual studio␣
↪implementation)
```

(continues on next page)

(continued from previous page)

```
it still gives a speedup by a factor of 2-3.

apart from beeing faster it also provides more precision for the resulting floats␣
↪since
its base values use full 32bit precision.
```

Listing 58: code

```
// set the initial seed to whatever you like
static int RandSeed = 1;

// using rand() (16bit precision)
// takes about 110 seconds for 2 billion calls
float RandFloat1()
{
    return ((float)rand()/RAND_MAX) * 2.0f - 1.0f;
}

// direct implementation of rand() (16 bit precision)
// takes about 32 seconds for 2 billion calls
float RandFloat2()
{
    return ((float)(((RandSeed = RandSeed * 214013L + 2531011L) >> 16) & 0x7fff)/RAND_
↪MAX) * 2.0f - 1.0f;
}

// fast rand float, using full 32bit precision
// takes about 12 seconds for 2 billion calls
float Fast_RandFloat()
{
    RandSeed *= 16807;
    return (float)RandSeed * 4.6566129e-010f;
}
```

5.33.1 Comments

- **Date**: 2009-11-20 23:40:37
- **By**: judahmenter@gee mail.com

```
There is no doubt that implementation 3 is fast, but the problem I had with it is␣
↪that there's no obvious way to limit the amplitude of the generated signal.

So instead I tried implementation 2 and ran into a different problem. The code is␣
↪written such that it assumes that RAND_MAX is equal to 0x7FFF, which was not true␣
↪on my system (it was 0x7FFFFFFF). Fortunately, this was easy to fix. I simply␣
↪removed the >> 16 and worked fine for me. My final implementation was:

return (float)(RandSeed = RandSeed * 214013L + 2531011L) / 0x7FFFFFFF * 2.0f * amp -␣
↪amp;

where "amp" is the desired amplitude.
```

- **Date**: 2009-12-29 22:53:23
- **By**: earlevel [] earlevel [] com

```
It should be noted in the code that for method #3, you must initialize the seed to␣
↪non-zero before using it.
```

- **Date**: 2010-01-24 16:36:03
- **By**: moc.boohay@bob

```
I don't understand Judahmenter's comment about 3 not limiting the amplitude. As it␣
↪stands it returns a value -1 to 1, so just multiply by your 'amp' value.
This turns into a handy 0-1 random number if you take off the sign bit:
(float)(RandSeed & 0x7FFFFFFF) * 4.6566129e-010f;
```

5.34 Fast abs/neg/sign for 32bit floats

- **Author or source:** ed.bew@raebybot
- **Type:** floating point functions
- **Created:** 2002-12-18 20:27:04

Listing 59: notes

```
Haven't seen this elsewhere, probably because it is too obvious? Anyway, these␣
↪functions
are intended for 32-bit floating point numbers only and should work a bit faster than␣
↪the
regular ones.

fastabs() gives you the absolute value of a float
fastneg() gives you the negative number (faster than multiplying with -1)
fastsgn() gives back +1 for 0 or positive numbers, -1 for negative numbers

Comments are welcome (tobybear[AT]web[DOT]de)

Cheers

Toby (www.tobybear.de)
```

Listing 60: code

```
// C/C++ code:
float fastabs(float f)
{int i=((*(int*)&f)&0x7fffffff);return (*(float*)&i);}

float fastneg(float f)
{int i=((*(int*)&f)^0x80000000);return (*(float*)&i);}

int fastsgn(float f)
{return 1+(((*(int*)&f)>>31)<<1);}

//Delphi/Pascal code:
function fastabs(f:single):single;
begin i:=longint((@f)^) and $7FFFFFFF;result:=single((@i)^) end;

function fastneg(f:single):single;
```

(continues on next page)

(continued from previous page)

```
begin i:=longint((@f)^) xor $80000000;result:=single((@i)^) end;

function fastsgn(f:single):longint;
begin result:=1+((longint((@f)^) shr 31)shl 1) end;
```

5.34.1 Comments

- **Date**: 2003-01-05 05:01:59
- **By**: ed.bew@raebybot

```
Matthias (bekkah[AT]web[DOT]de) wrote me a mail with the following further
↪improvements for the C++ parts of the code:

// C++ code:
inline float fastabs(const float f)
{int i=((*(int*)&f)&0x7fffffff);return (*(float*)&i);}

inline float fastneg(const float f)
{int i=((*(int*)&f)^0x80000000);return (*(float*)&i);}

inline int fastsgn(const float f)
{return 1+(((*(int*)&f)>>31)<<1);}

Thanks!
```

- **Date**: 2003-01-11 15:53:56
- **By**: ur.liam@redocip

```
Too bad these 'tricks' need two additional FWAITs to work in a raw FPU code. Maybe
↪standard fabs and fneg are better? Although, that fastsgn() could be useful since
↪there's no FPU equivalent for it.

Cheers,
Aleksey.
```

- **Date**: 2003-01-11 15:55:35
- **By**: ur.liam@redocip

```
I meant 'fchs' in place of 'fneg'.
```

- **Date**: 2003-05-05 20:49:34
- **By**: ten.llavnnarb@divad

```
I don't know if this is any faster, but atleast you can avoid some typecasting.

function fastabs(f: Single): Single; var i: Integer absolute f;
begin i := i and $7fffffff; Result := f; end;
```

- **Date**: 2003-07-29 04:55:55
- **By**: moc.oidua-m@sirhc

```
Note that a reasonable compiler should be able to perform these optimizations for you.
↪ I seem to recall that GCC in particular has the capability to replace calls to␣
↪[f]abs() with instructions optimized for the platform.
```

- **Date**: 2005-05-25 20:22:59
- **By**: moc.noicratse@ajelak

```
On MS compilers for x86, just do:
#pragma intrinsic(fabs)

...and then use fabs() for doubles, fabsf() for floats. The compiler will generate␣
↪the FABS instruction, which is generally 1 cycle on modern x86 FPUs. (Internally,␣
↪the FPU just masks the bit.)
```

5.35 Fast binary log approximations

- **Author or source:** gro.lortnocdnim@gro.psdcisum
- **Type:** C code
- **Created:** 2002-04-04 17:00:05

Listing 61: 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
```

Listing 62: 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 ) {
```

(continues on next page)

(continued from previous page)

```
  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;
}
```

5.35.1 Comments

- **Date**: 2002-12-18 20:08:06
- **By**: ed.bew@raebybot

```
Here is some code to do this in Delphi/Pascal:

function approxLn2(f:single):single;
begin
 result:=(((longint((@f)^) and $7f800000) shr 23)-$7f)+(longint((@f)^) and $007fffff)/
↪$800000;
end;

function floorOfLn2(f:single):longint;
begin
 result:=(((longint((@f)^) and $7f800000) shr 23)-$7f);
end;

Cheers,

Tobybear
www.tobybear.de
```

5.36 Fast cube root, square root, and reciprocal for x86/SSE CPUs.

- **Author or source:** moc.noicratse@ajelak
- **Created:** 2005-05-29 18:36:40

Listing 63: notes

```
All of these methods use SSE instructions or bit twiddling tricks to get a rough
approximation to cube root, square root, or reciprocal, which is then refined with␣
↪one or
more Newton-Raphson approximation steps.

Each is named to indicate its approximate level of accuracy and a comment describes␣
↪its
performance relative to the conventional versions.
```

Listing 64: code

```
// These functions approximate reciprocal, square root, and
// cube root to varying degrees of precision substantially
// faster than the straightforward methods 1/x, sqrtf(x),
// and powf( x, 1.0f/3.0f ). All require SSE-enabled CPU & OS.
// Unlike the powf() solution, the cube roots also correctly
// handle negative inputs.

#define REALLY_INLINE __forceinline

// ~34 clocks on Pentium M vs. ~360 for powf
REALLY_INLINE float cuberoot_sse_8bits( float x )
{
    float z;
    static const float three = 3.0f;
    _asm
    {
            mov             eax, x                          // x as bits
            movss   xmm2, x                         // x2: x
            movss   xmm1, three                     // x1: 3
            // Magic floating point cube root done with integer math.
            // The exponent is divided by three in such a way that
            // remainder bits get shoved into the top of the normalized
            // mantissa.
            mov             ecx, eax                        // copy of x
            and             eax, 0x7FFFFFFF         // exponent & mantissa of x in␣
↪biased-127
            sub     eax, 0x3F800000         // exponent & mantissa of x in 2's comp
            sar     eax, 10                         //
            imul    eax, 341                        // 341/1024 ~= .333
            add             eax, 0x3F800000         // back to biased-127
            and     eax, 0x7FFFFFFF         // remask
            and             ecx, 0x80000000         // original sign and mantissa
            or      eax, ecx                        // masked new exponent, old sign␣
↪and mantissa
            mov             z, eax                          //

            // use SSE to refine the first approximation
            movss   xmm0, z                         ;// x0: z
            movss   xmm3, xmm0                      ;// x3: z
            mulss   xmm3, xmm0                      ;// x3: z*z
            movss   xmm4, xmm3                      ;// x4: z*z
            mulss   xmm3, xmm1                      ;// x3: 3*z*z
            rcpss   xmm3, xmm3                      ;// x3: ~ 1/(3*z*z)
            mulss   xmm4, xmm0                      ;// x4: z*z*z
```

(continues on next page)

(continued from previous page)

```
            subss   xmm4, xmm2                      ;// x4: z*z*z-x
            mulss   xmm4, xmm3                      ;// x4: (z*z*z-x) / (3*z*z)
            subss   xmm0, xmm4                      ;// x0: z' accurate to within about 0.3%
            movss   z, xmm0
    }

    return z;
}

// ~60 clocks on Pentium M vs. ~360 for powf
REALLY_INLINE float cuberoot_sse_16bits( float x )
{
    float z;
    static const float three = 3.0f;
    _asm
    {
            mov             eax, x                          // x as bits
            movss   xmm2, x                         // x2: x
            movss   xmm1, three                     // x1: 3
            // Magic floating point cube root done with integer math.
            // The exponent is divided by three in such a way that
            // remainder bits get shoved into the top of the normalized
            // mantissa.
            mov             ecx, eax                        // copy of x
            and             eax, 0x7FFFFFFF         // exponent & mantissa of x in biased-127
            sub     eax, 0x3F800000         // exponent & mantissa of x in 2's comp
            sar     eax, 10                         //
            imul    eax, 341                        // 341/1024 ~= .333
            add             eax, 0x3F800000         // back to biased-127
            and     eax, 0x7FFFFFFF         // remask
            and             ecx, 0x80000000         // original sign and mantissa
            or      eax, ecx                        // masked new exponent, old sign and mantissa
            mov             z, eax                          //

            // use SSE to refine the first approximation
            movss   xmm0, z                         ;// x0: z
            movss   xmm3, xmm0                      ;// x3: z
            mulss   xmm3, xmm0                      ;// x3: z*z
            movss   xmm4, xmm3                      ;// x4: z*z
            mulss   xmm3, xmm1                      ;// x3: 3*z*z
            rcpss   xmm3, xmm3                      ;// x3: ~ 1/(3*z*z)
            mulss   xmm4, xmm0                      ;// x4: z*z*z
            subss   xmm4, xmm2                      ;// x4: z*z*z-x
            mulss   xmm4, xmm3                      ;// x4: (z*z*z-x) / (3*z*z)
            subss   xmm0, xmm4                      ;// x0: z' accurate to within about 0.3%

            movss   xmm3, xmm0                      ;// x3: z
            mulss   xmm3, xmm0                      ;// x3: z*z
            movss   xmm4, xmm3                      ;// x4: z*z
            mulss   xmm3, xmm1                      ;// x3: 3*z*z
            rcpss   xmm3, xmm3                      ;// x3: ~ 1/(3*z*z)
            mulss   xmm4, xmm0                      ;// x4: z*z*z
            subss   xmm4, xmm2                      ;// x4: z*z*z-x
```

(continues on next page)

(continued from previous page)

```
            mulss   xmm4, xmm3                        ;// x4: (z*z*z-x) / (3*z*z)
            subss   xmm0, xmm4                        ;// x0: z'' accurate to within about 0.001%

            movss   z, xmm0
    }

    return z;
}

// ~77 clocks on Pentium M vs. ~360 for powf
REALLY_INLINE float cuberoot_sse_22bits( float x )
{
    float z;
    static const float three = 3.0f;
    _asm
    {
            mov             eax, x                            // x as bits
            movss   xmm2, x                           // x2: x
            movss   xmm1, three                       // x1: 3
            // Magic floating point cube root done with integer math.
            // The exponent is divided by three in such a way that
            // remainder bits get shoved into the top of the normalized
            // mantissa.
            mov             ecx, eax                          // copy of x
            and             eax, 0x7FFFFFFF           // exponent & mantissa of x in biased-127
            sub     eax, 0x3F800000           // exponent & mantissa of x in 2's comp
            sar     eax, 10                           //
            imul    eax, 341                          // 341/1024 ~= .333
            add             eax, 0x3F800000           // back to biased-127
            and     eax, 0x7FFFFFFF           // remask
            and             ecx, 0x80000000           // original sign and mantissa
            or      eax, ecx                          // masked new exponent, old sign and mantissa
            mov             z, eax                            //

            // use SSE to refine the first approximation
            movss   xmm0, z                           // x0: z
            movss   xmm3, xmm0                        // x3: z
            mulss   xmm3, xmm0                        // x3: z*z
            movss   xmm4, xmm3                        // x4: z*z
            mulss   xmm3, xmm1                        // x3: 3*z*z
            rcpss   xmm3, xmm3                        // x3: ~ 1/(3*z*z)
            mulss   xmm4, xmm0                        // x4: z*z*z
            subss   xmm4, xmm2                        // x4: z*z*z-x
            mulss   xmm4, xmm3                        // x4: (z*z*z-x) / (3*z*z)
            subss   xmm0, xmm4                        // x0: z' accurate to within about 0.3%

            movss   xmm3, xmm0                        // x3: z
            mulss   xmm3, xmm0                        // x3: z*z
            movss   xmm4, xmm3                        // x4: z*z
            mulss   xmm3, xmm1                        // x3: 3*z*z
            rcpss   xmm3, xmm3                        // x3: ~ 1/(3*z*z)
            mulss   xmm4, xmm0                        // x4: z*z*z
            subss   xmm4, xmm2                        // x4: z*z*z-x
```

(continues on next page)

(continued from previous page)

```
            mulss   xmm4, xmm3                      // x4: (z*z*z-x) / (3*z*z)
            subss   xmm0, xmm4                      // x0: z'' accurate to within about 0.001%

            movss   xmm3, xmm0                      // x3: z
            mulss   xmm3, xmm0                      // x3: z*z
            movss   xmm4, xmm3                      // x4: z*z
            mulss   xmm3, xmm1                      // x3: 3*z*z
            rcpss   xmm3, xmm3                      // x3: ~ 1/(3*z*z)
            mulss   xmm4, xmm0                      // x4: z*z*z
            subss   xmm4, xmm2                      // x4: z*z*z-x
            mulss   xmm4, xmm3                      // x4: (z*z*z-x) / (3*z*z)
            subss   xmm0, xmm4                      // x0: z''' accurate to within about 0.000012%

            movss   z, xmm0
    }

    return z;
}

// ~6 clocks on Pentium M vs. ~24 for single precision sqrtf
REALLY_INLINE float squareroot_sse_11bits( float x )
{
    float z;
    _asm
    {
            rsqrtss xmm0, x
            rcpss   xmm0, xmm0
            movss   z, xmm0                 // z ~= sqrt(x) to 0.038%
    }
    return z;
}

// ~19 clocks on Pentium M vs. ~24 for single precision sqrtf
REALLY_INLINE float squareroot_sse_22bits( float x )
{
    static float half = 0.5f;
    float z;
    _asm
    {
            movss   xmm1, x                 // x1: x
            rsqrtss xmm2, xmm1              // x2: ~1/sqrt(x) = 1/z
            rcpss   xmm0, xmm2              // x0: z == ~sqrt(x) to 0.05%

            movss   xmm4, xmm0              // x4: z
            movss   xmm3, half
            mulss   xmm4, xmm4              // x4: z*z
            mulss   xmm2, xmm3              // x2: 1 / 2z
            subss   xmm4, xmm1              // x4: z*z-x
            mulss   xmm4, xmm2              // x4: (z*z-x)/2z
            subss   xmm0, xmm4              // x0: z' to 0.000015%

            movss   z, xmm0
    }
    return z;
}
```

(continues on next page)

(continued from previous page)

```
// ~12 clocks on Pentium M vs. ~16 for single precision divide
REALLY_INLINE float reciprocal_sse_22bits( float x )
{
    float z;
    _asm
    {
            rcpss   xmm0, x                 // x0: z ~= 1/x
            movss   xmm2, x                 // x2: x
            movss   xmm1, xmm0              // x1: z ~= 1/x
            addss   xmm0, xmm0              // x0: 2z
            mulss   xmm1, xmm1              // x1: z^2
            mulss   xmm1, xmm2              // x1: xz^2
            subss   xmm0, xmm1              // x0: z' ~= 1/x to 0.000012%

            movss   z, xmm0
    }
    return z;
}
```

5.37 Fast exp() approximations

- **Author or source:** uh.etle.fni@yfoocs
- **Type:** Taylor series approximation
- **Created:** 2006-05-26 04:54:44

Listing 65: notes

```
I needed a fast exp() approximation in the -3.14..3.14 range, so I made some
approximations based on the tanh() code posted in the archive by Fuzzpilz. Should be
pretty straightforward, but someone may find this useful.

The increasing numbers in the name of the function means increasing precision. Maximum
error in the -1..1 range:
fastexp3: 0.05      (1.8%)
fastexp4: 0.01      (0.36%)
fastexp5: 0.0016152 (0.59%)
fastexp6: 0.0002263 (0.0083%)
fastexp7: 0.0000279 (0.001%)
fastexp8: 0.0000031 (0.00011%)
fastexp9: 0.0000003 (0.000011%)

Maximum error in the -3.14..3.14 range:
fastexp3: 8.8742 (38.4%)
fastexp4: 4.8237 (20.8%)
fastexp5: 2.28   (9.8%)
fastexp6: 0.9488 (4.1%)
fastexp7: 0.3516 (1.5%)
fastexp8: 0.1172 (0.5%)
fastexp9: 0.0355 (0.15%)

These were done using the Taylor series, for example I got fastexp4 by using:
exp(x) = 1 + x + x^2/2 + x^3/6 + x^4/24 + ...
```

(continues on next page)

(continued from previous page)

```
= (24 + 24x + x^2*12 + x^3*4 + x^4) / 24
(using Horner-scheme:)
= (24 + x * (24 + x * (12 + x * (4 + x)))) * 0.041666666f
```

Listing 66: code

```
inline float fastexp3(float x) {
    return (6+x*(6+x*(3+x)))*0.16666666f;
}

inline float fastexp4(float x) {
    return (24+x*(24+x*(12+x*(4+x))))*0.041666666f;
}

inline float fastexp5(float x) {
    return (120+x*(120+x*(60+x*(20+x*(5+x)))))*0.0083333333f;
}

inline float fastexp6(float x) {
    return 720+x*(720+x*(360+x*(120+x*(30+x*(6+x))))))*0.0013888888f;
}

inline float fastexp7(float x) {
    return (5040+x*(5040+x*(2520+x*(840+x*(210+x*(42+x*(7+x)))))))*0.00019841269f;
}

inline float fastexp8(float x) {
    return (40320+x*(40320+x*(20160+x*(6720+x*(1680+x*(336+x*(56+x*(8+x))))))))*2.
↪4801587301e-5;
}

inline float fastexp9(float x) {
  return␣
↪(362880+x*(362880+x*(181440+x*(60480+x*(15120+x*(3024+x*(504+x*(72+x*(9+x)))))))))*2.
↪75573192e-6;
}
```

5.37.1 Comments

- **Date**: 2006-07-03 00:40:15
- **By**: uh.etle.fni@yfoocs

```
These series converge fast only near zero. But there is an identity:

 exp(x) = exp(a) * exp(x-a)

So, if you want a relatively fast polynomial approximation for exp(x) for 0 to ~7.5,␣
↪you can use:

// max error in the 0 .. 7.5 range: ~0.45%
inline float fastexp(float const &x)
{
  if (x<2.5)
      return 2.7182818f * fastexp5(x-1.f);
```

(continues on next page)

(continued from previous page)

```
  else if (x<5)
      return 33.115452f * fastexp5(x-3.5f);
  else
      return 403.42879f * fastexp5(x-6.f);
}

where 2.7182.. = exp(1), 33.1154.. = exp(3.5) and 403.428.. = exp(6). I chose these␣
↪values because fastexp5 has a maximum error of 0.45% between -1 - 1.5 (using␣
↪fastexp6, the maximum error is 0.09%).

Using the identity

  pow(a,x) = exp(x * log(a))

you can use any base, for example to get 2^x:

// max error in the 0-10.58 range: ~0.45%
inline float fastpow2(float const &x)
{
  float const log_two = 0.6931472f;
  return fastexp(x * log_two);
}

These functions are about 3x faster than exp().

-- Peter Schoffhauzer
```

5.38 Fast exp2 approximation

- **Author or source:** Laurent de Soras (moc.ecrofmho@tnerual)
- **Created:** 2002-07-29 18:42:23

Listing 67: 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
```

Listing 68: 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)
```

(continues on next page)

(continued from previous page)

```
{
   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);
}
```

5.39 Fast log2

- **Author or source:** Laurent de Soras
- **Created:** 2002-02-10 12:31:20

Listing 69: 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);
}
```

5.39.1 Comments

- **Date**: 2002-12-18 20:13:06
- **By**: ed.bew@raebybot

```
And here is some native Delphi/Pascal code that
does the same thing:

function fast_log2(val:single):single;
var log2,x:longint;
begin
```

(continues on next page)

(continued from previous page)

```
 x:=longint((@val)^);
 log2:=((x shr 23) and 255)-128;
 x:=x and (not(255 shl 23));
 x:=x+127 shl 23;
 result:=single((@x)^)+log2;
end;

Cheers

Toby

www.tobybear.de
```

- **Date**: 2003-02-07 23:54:32
- **By**: ed.sulydroc@yrneh

```
instead of using this pointer casting expressions one can also use a enum like this:

enum FloatInt
{
float f;
```

- **Date**: 2003-02-07 23:55:27
- **By**: ed.sulydroc@yrneh

```
instead of using this pointer casting expressions one can also use a enum like this:

enum FloatInt
{
float f;
int l;
} p;

and then access the data with:

p.f = x;
p.l >>= 23;

Greetings, Henry
```

- **Date**: 2003-02-07 23:56:11
- **By**: ed.sulydroc@yrneh

```
Sorry :

didnt mean enum, ment UNION !!!
```

- **Date**: 2005-10-18 10:03:47
- **By**: Laurent de Soras

```
More precision can be obtained by adding the following line just before the return() :

val = map_lin_2_exp (val, 1.0f / 2);

```

(continues on next page)

(continued from previous page)

```
Below is the function (everything is constant, so most operations should be done at␣
↪compile time) :

inline float map_lin_2_exp (float val, float k)
{
   const float a = (k - 1) / (k + 1);
   const float b = (4 - 2*k) / (k + 1);     // 1 - 3*a
   const float c = 2*a;
   val = (a * val + b) * val + c;

   return (val);
}

You can do the mapping you want for the range [1;2] -> [1;2] to approximate the␣
↪function log(x)/log(2).
```

- **Date**: 2005-10-18 10:05:48
- **By**: Laurent de Soras

```
Sorry I meant log(x)/log(2) + 1
```

5.40 Fast power and root estimates for 32bit floats

- **Author or source:** ed.bew@raebybot
- **Type:** floating point functions
- **Created:** 2002-12-18 21:07:27

Listing 70: notes

```
Original code by Stefan Stenzel (also in this archive, see "pow(x,4) approximation") -
extended for more flexibility.

fastpow(f,n) gives a rather *rough* estimate of a float number f to the power of an
integer number n (y=f^n). It is fast but result can be quite a bit off, since we␣
↪directly
mess with the floating point exponent.-> use it only for getting rough estimates of␣
↪the
values and where precision is not that important.

fastroot(f,n) gives the n-th root of f. Same thing concerning precision applies here.

Cheers

Toby (www.tobybear.de)
```

Listing 71: code

```
//C/C++ source code:
float fastpower(float f,int n)
{
 long *lp,l;
 lp=(long*)(&f);
```

(continues on next page)

(continued from previous page)

```
 l=*lp;l-=0x3F800000l;l<<=(n-1);l+=0x3F800000l;
 *lp=l;
 return f;
}

float fastroot(float f,int n)
{
 long *lp,l;
 lp=(long*)(&f);
 l=*lp;l-=0x3F800000l;l>>=(n-1);l+=0x3F800000l;
 *lp=l;
 return f;
}

//Delphi/Pascal source code:
function fastpower(i:single;n:integer):single;
var l:longint;
begin
 l:=longint((@i)^);
 l:=l-$3F800000;l:=l shl (n-1);l:=l+$3F800000;
 result:=single((@l)^);
end;

function fastroot(i:single;n:integer):single;
var l:longint;
begin
 l:=longint((@i)^);
 l:=l-$3F800000;l:=l shr (n-1);l:=l+$3F800000;
 result:=single((@l)^);
end;
```

5.41 Fast rounding functions in pascal

- **Author or source:** moc.liamtoh@abuob
- **Type:** round/ceil/floor/trunc
- **Created:** 2008-03-09 13:09:44

Listing 72: notes

```
Original documentation with cpp samples:
http://ldesoras.free.fr/prod.html#doc_rounding
```

Listing 73: code

```
Pascal translation of the functions (accurates ones) :

function ffloor(f:double):integer;
var
  i:integer;
  t : double;
begin
t := -0.5 ;
```

(continues on next page)

(continued from previous page)

```
  asm
    fld   f
    fadd  st,st(0)
    fadd  t
    fistp i
    sar   i, 1
end;
result:= i
end;

function fceil(f:double):integer;
var
  i:integer;
  t : double;
begin
t := -0.5 ;
  asm
    fld   f
    fadd  st,st(0)
    fsubr t
    fistp i
    sar   i, 1
end;
result:= -i
end;

function ftrunc(f:double):integer;
var
  i:integer;
  t : double;
begin
t := -0.5 ;
  asm
    fld   f
    fadd  st,st(0)
    fabs
    fadd t
    fistp i
    sar   i, 1
end;
if f<0 then i := -i;
result:= i
end;

function fround(f:double):integer;
var
  i:integer;
  t : double;
begin
t := 0.5 ;
  asm
    fld   f
    fadd  st,st(0)
    fadd t
    fistp i
    sar   i, 1
end;
```

(continues on next page)

(continued from previous page)

```
result:= i
end;
```

5.41.1 Comments

- **Date**: 2008-04-23 11:46:58
- **By**: eb.mapstenykson@didid

```
the fround doesn't make much sense in Pascal, as in Pascal (well, Delphi & I'm pretty
↪sure FreePascal too), the default rounding is already a fast rounding. The default
↪being FPU rounding to nearest mode, the compiler doesn't change it back & forth. &
↪since it's inlined (well, compiler magic), it's very fast.
```

5.42 Fast sign for 32 bit floats

- **Author or source:** Peter Schoffhauzer
- **Created:** 2007-05-14 10:15:43

Listing 74: notes

```
Fast functions which give the sign of a 32 bit floating point number by checking the
↪sign
bit. There are two versions, one which gives the value as a float, and the other
↪gives an
int.

The _nozero versions are faster, but they give incorrect 1 or -1 for zero (depending
↪on
the sign bit set in the number). The int version should be faster than the Tobybear
↪one in
the archive, since this one doesn't have an addition, just bit operations.
```

Listing 75: code

```
/*---------------------------------------------------------------------------
   fast sign, returns float
---------------------------------------------------------------------------*/

// returns 1.0f for positive floats, -1.0f for negative floats
// for zero it does not work (may gives 1.0f or -1.0f), but it's faster
inline float fast_sign_nozero(float f) {
    float r = 1.0f;
    (int&)r |= ((int&)f & 0x80000000); // mask sign bit in f, set it in r if necessary
    return r;
}

// returns 1.0f for positive floats, -1.0f for negative floats, 0.0f for zero
inline float fast_sign(float f) {
    if (((int&)f & 0x7FFFFFFF)==0) return 0.f; // test exponent & mantissa bits: is
↪input zero?
    else {
```

(continues on next page)

(continued from previous page)

```
        float r = 1.0f;
        (int&)r |= ((int&)f & 0x80000000); // mask sign bit in f, set it in r if necessary
        return r;
    }
}

/*---------------------------------------------------------------------------
   fast sign, returns int
---------------------------------------------------------------------------*/

// returns 1 for positive floats, -1 for negative floats
// for 0.0f input it does not work (may give 1 or -1), but it's faster
inline int fast_sign_int_nozero(float f) {
      return (signed((int&)f & 0x80000000) >> 31 ) | 1;
}

// returns 1 for positive floats, -1 for negative floats, 0 for 0.0f
inline int fast_sign_int(float f) {
    if (((int&)f & 0x7FFFFFFF)==0) return 0; // test exponent & mantissa bits: is input zero?
    return (signed((int&)f & 0x80000000) >> 31 ) | 1;
}
```

5.42.1 Comments

- **Date**: 2007-05-15 16:49:02
- **By**: uh.etle.fni@yfoocs

```
Now consider when you want to multiply a number by the sign of another:

  if (a>0.f) b = b;
    else b = -b;

This involves 1) a comparison, 2) a branch, 3) an inversion (multiply or bit flip) in one branch. Another method for calculating the same:

  b *= fast_sign_nozero_(a);

This involves 1) a bitwise and, 2) a bitwise or 3) and a multiply. Using only bit operations, the branch and/or multiply can be totally eliminated:

// equivalent to dest *= sgn(source)
inline void mul_sign(float &dest, float &source) {
    (int&)dest &= 0x7FFFFFFF; // clear sign bit
    (int&)dest |= ((int&)dest ^ (int&)source) & 0x80000000f; // set sign bit if necessary
}

This function has only three bitwise operations, which should be very fast. Usage:

  mul_sign(b,a); // b = b*sign(a)

The speed increase with all these functions greatly depends on the predictability of the branches. If the branch is highly predictable (a lot of positive numbers, then a lot of negative numbers, without mixing them), then an if/else solution is pretty fast. If the branch is unpredictable (random numbers, or audio similar to white noise), then bit operations should perform significantly better on today's most CPUs with multi-level pipelines.
```

(continues on next page)

(continued from previous page)

```
-- Peter Schoffhauzer
```

- **Date**: 2007-07-01 19:14:04
- **By**: uh.etle.fni@yfoocs

```
Sorry, there is a bug in the above code. Correctly:

// equivalent to dest *= sgn(source)
inline void mul_sign_nozero(float &dest, float const &source) {
    int sign_mask = ((int&)dest ^ (int&)source) & 0x80000000; // XOR and mask
    (int&)dest &= 0x7FFFFFFF; // clear sign bit
    (int&)dest |= sign_mask; // set sign bit if necessary
}
```

5.43 Float to int

- **Author or source:** Ross Bencina
- **Created:** 2002-01-17 03:15:14

Listing 76: notes

```
intel only
```

Listing 77: code

```
int truncate(float flt)
{
  int i;
  static const double half = 0.5f;
  _asm
  {
     fld flt
     fsub half
     fistp i
  }
  return i
}
```

5.43.1 Comments

- **Date**: 2002-06-11 20:12:11
- **By**: moc.xinortceletrams@ahaj

```
Note: Round nearest doesn't work, because the Intel FPU uses Even-Odd rounding in␣
↪order to conform to the IEEE floating-point standard: when the FPU is set to use␣
↪the round-nearest rule, values whose fractional part is exactly 0.5
round toward the nearest *even* integer.  Thus, 1.5 rounds to 2, 2.5 rounds to 2, 3.5␣
↪rounds to 4. This is quite disastrous for the FLOOR/CEIL
functions if you use the strategy you describe.
```

- **Date**: 2003-05-28 21:25:24
- **By**: moc.oohay@tuanogus

```
This version below seems to be more accurate on my Win32/P4.  Doesn't deal with the
↪Intel FPU issue.  A faster solution than c-style casts though.  But you're not
↪always going to get the most accurate conversion.  Like the previous comment; 2.5
↪will convert to 2, but 2.501 will convert to 3.

int truncate(float flt)
{
    int i;
    _asm
    {
        fld flt
        fistp i
    }
    return i
}
```

- **Date**: 2004-04-07 01:20:49
- **By**: moc.erehwon@amsuk

```
if you use msvc, just use the /QIfist compile-switch
```

5.44 Float to int (more intel asm)

- **Author or source:** Laurent de Soras (via flipcode)
- **Created:** 2004-06-14 14:51:47

Listing 78: notes

```
[Found this on flipcode, seemed worth posting here too, hopefully Laurent will
↪approve :)
-- Ross]

Here is the code I often use. It is not a _ftol replacement, but it provides useful
functions. The current processor rounding mode should be "to nearest" ; it is the
↪default
setting for most of the compilers.

The [fadd st, st (0) / sar i,1] trick fixes the "round to nearest even number"
↪behaviour.
Thus, round_int (N+0.5) always returns N+1 and floor_int function is appropriate to
convert floating point numbers into fixed point numbers.

---------------------

Laurent de Soras
Audio DSP engineer & software designer
Ohm Force - Digital audio software
http://www.ohmforce.com
```

Listing 79: code

```
inline int round_int (double x)
{
   int      i;
   static const float round_to_nearest = 0.5f;
   __asm
   {
      fld      x
      fadd     st, st (0)
      fadd     round_to_nearest
      fistp    i
      sar      i, 1
   }

   return (i);
}

inline int floor_int (double x)
{
   int      i;
   static const float round_toward_m_i = -0.5f;
   __asm
    {
      fld      x
      fadd     st, st (0)
      fadd     round_toward_m_i
      fistp    i
      sar      i, 1
   }

   return (i);
}

inline int ceil_int (double x)
{
   int      i;
   static const float round_toward_p_i = -0.5f;
   __asm
   {
      fld      x
      fadd     st, st (0)
      fsubr    round_toward_p_i
      fistp    i
      sar      i, 1
   }

   return (-i);
}
```

5.44.1 Comments

- **Date**: 2004-06-15 07:29:25
- **By**: daniel.schaack[-dot-]basementarts.de

```
cool trick, but using round to zero FPU mode is still the best methode (2 lines):

__asm
{
 fld x
 fistp y
}
```

- **Date**: 2004-06-18 01:28:41
- **By**: moc.krod@dje

```
If anyone is using NASM, here is how I implemented the first function, if anyone is
↪interested.  I did this after sitting down for a while with the intel manuals.  I
↪'ve not done any x86-FPU stuff before, so this may not be the *best* code.  The
↪other functions are easily implemented by modifying this one.

bits 32
global dsp_round

HALF dq 0.5

align 4
dsp_round:
    fld qword[HALF]
    fld qword[esp+4]

    fadd st0
    fadd st1

    fistp qword[eax]
    mov eax, [eax]
    sar eax, 1
    ret
```

- **Date**: 2004-06-18 13:48:09
- **By**: moc.krod@dje

```
Whoops.  I've really gone and embarassed myself this time.  The code I posted is
↪actually broken -- I don't know what I was thinking.  I'll post the fixed version a
↪bit later today.  My appologies.
```

- **Date**: 2004-08-10 10:29:44
- **By**: pj.krowtenctu@remmah

```
Will this method also work for other processor types like AMD and CELERON?
```

- **Date**: 2007-02-19 11:13:19
- **By**: uh.etle.fni@yfoocs

```
It works with AMD and Celeron too (and as I know, probably with all x87 FPUs)
```

5.45 Float to integer conversion

- **Author or source:** Peter Schoffhauzer
- **Created:** 2007-02-19 10:03:07

Listing 80: notes

```
The fld x/fistp i instructions can be used for fast conversion of floating point␣
↪numbers
to integers. By default, the number is rounded to the nearest integer. However,␣
↪sometimes
that's not what we need.

Bits 12 and 11 if the FPU control word determine the way the FPU rounds numbers. The␣
↪four
rounding states are:

00 = round to nearest (this is the default)
01 = round down (towards -infinity)
10 = round up(towards +infinity)
11 = truncate up(towards zero)

The status word is loaded/stored using the fldcw/fstcw instructions. After setting the
rounding mode as desired, the float2int() function will use that rounding mode until␣
↪the
control mode is reset. The ceil() and floor() functions set the rounding mode for␣
↪every
instruction, which is very slow. Therefore, it is a lot faster to set the rounding␣
↪mode
(down or up) before processing a block, and use float2int() instead.

References: SIMPLY FPU by Raymond Filiatreault
http://www.website.masmforum.com/tutorials/fptute/index.html

MSVC (and probably other compilers too) has functions defined in <float.h> for␣
↪changing
the FPU control word: _control87/_controlfp. The equivalent instructions are:

_controlfp(_RC_CHOP, _MCW_RC); // set rounding mode to truncate
_controlfp(_RC_UP    _MCW_RC); // set rounding mode to up (ceil)
_controlfp(_RC_DOWN, _MCW_RC); // set rounding mode to down (floor)
_controlfp(_RC_NEAR, _MCW_RC); // set rounding mode to near (default)

Note that the FPU rounding mode affects other calculations too, so the same code may␣
↪give
different results.

Alternatively, single precision floating point numbers can be truncated or rounded to
integers by using SSE instructions cvtss2si, cvttss2si, cvtps2pi or cvttps2pi, where␣
↪SSE
instructions are available (which means probably on all modern CPUs). These are not
discussed here in detail, but I provided the function truncateSSE which always␣
↪truncates a
single precision floating point number to integer, regardless of the current rounding
mode.
```

(continues on next page)

(continued from previous page)

```
(Also I think the SSE rounding mode can differ from the rounding mode set in the FPU
control word, but I haven't tested it so far.)
```

Listing 81: code

```
#ifndef __FPU_ROUNDING_H_
#define __FPU_ROUNDING_H_

static short control_word;
static short control_word2;

// round a float to int using the actual rounding mode
inline int float2int(float x) {
    int i;
    __asm {
            fld x
            fistp i
    }
    return i;
}

// set rounding mode to nearest
inline void set_round2near() {
    __asm   {
            fstcw   control_word            // store fpu control word
            mov     dx, word ptr [control_word]
            and             dx,0xf9ff                       // round towards nearest (default)
            mov             control_word2, dx
            fldcw   control_word2           // load modfied control word
    }
}

// set rounding mode to round down
inline void set_round_down() {
    __asm  {
            fstcw   control_word            // store fpu control word
            mov     dx, word ptr [control_word]
            or              dx,0x0400                       // round towards -inf
            and             dx,0xf7ff
            mov             control_word2, dx
            fldcw   control_word2           // load modfied control word
    }
}

// set rounding mode to round up
inline void set_round_up() {
    __asm  {
            fstcw   control_word            // store fpu control word
            mov     dx, word ptr [control_word]
            or              dx,0x0800                       // round towards +inf
            and             dx,0xfbff
            mov             control_word2, dx
            fldcw   control_word2           // load modfied control word
    }
}
```

(continues on next page)

(continued from previous page)

```

// set rounding mode to truncate
inline void set_truncate() {
    __asm  {
                fstcw   control_word             // store fpu control word
                mov     dx, word ptr [control_word]
                or              dx,0x0c00                       // rounding: truncate
                mov             control_word2, dx
                fldcw   control_word2           // load modfied control word
    }
}

// restore original rounding mode
inline void restore_cw() {
    __asm fldcw     control_word
}

// truncate to integer using SSE
inline long truncateSSE(float x) {
    __asm cvttss2si eax,x
}

#endif
```

5.45.1 Comments

- **Date**: 2007-12-30 02:01:18
- **By**: moc.liamtoh@kcuncorj

```
I've seen a lot of talk about using the function lrintf() when converting from float
↪to int, regarding bypassing some 'slow' opcodes in what standard compilers put in
↪for something like:

int myint = (int) myfloat;

in otherwords the following code would be faster:
int myint = lrintf(myfloat);

this lrintf function is available on GNU C/C++
```

5.46 Float-to-int, coverting an array of floats

- **Author or source:** Stefan Stenzel
- **Created:** 2002-01-17 03:10:32

Listing 82: notes

```
intel only
```

Listing 83: 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
```

(continues on next page)

(continued from previous page)

```
    cmp     esi,ebx      ; end reached ?
    jb      saturate
    pop     eax
    mov     [ebx],eax    ; restore end flag
    }
}
```

5.46.1 Comments

- **Date**: 2003-01-22 20:20:20
- **By**: moc.xinortceletrams@sungam

```
_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

I think the last line here reads outside the buffer.
```

5.47 Gaussian dithering

- **Author or source:** Aleksey Vaneev (ur.liam@redocip)
- **Type:** Dithering
- **Created:** 2002-09-29 23:02:52

Listing 84: 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.
```

Listing 85: 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.
```

5.48 Gaussian random numbers

- **Author or source:** ed.bew@raebybot
- **Type:** random number generation
- **Created:** 2002-12-16 19:01:01

Listing 86: notes

```
// Gaussian random numbers
// This algorithm (adapted from "Natur als fraktale Grafik" by
// Reinhard Scholl) implements a generation method for gaussian
// distributed random numbers with mean=0 and variance=1
// (standard gaussian distribution) mapped to the range of
// -1 to +1 with the maximum at 0.
// For only positive results you might abs() the return value.
// The q variable defines the precision, with q=15 the smallest
// distance between two numbers will be 1/(2^q div 3)=1/10922
// which usually gives good results.

// Note: the random() function used is the standard random
// function from Delphi/Pascal that produces *linear*
// distributed numbers from 0 to parameter-1, the equivalent
// C function is probably rand().
```

Listing 87: code

```
const q=15;
      c1=(1 shl q)-1;
      c2=(c1 div 3)+1;
      c3=1/c1;

function GRandom:single;
begin
 result:=(2*(random(c2)+random(c2)+random(c2))-3*(c2-1))*c3;
end;
```

5.49 Hermite Interpolator (x86 ASM)

- **Author or source:** moc.oiduacbr@kileib.trebor
- **Type:** Hermite interpolator in x86 assembly (for MS VC++)
- **Created:** 2004-04-07 09:38:32

Listing 88: notes

```
An "assemblified" variant of Laurent de Soras hermite interpolator. I tried to do
calculations as parallell as I could muster, but there is almost certainly room for
improvements. Right now, it works about 5.3 times (!) faster, not bad to start with...

Parameter explanation:
frac_pos: fractional value [0.0f - 1.0f] to interpolator
pntr: pointer to float array where:
    pntr[0] = previous sample (idx = -1)
```

(continues on next page)

(continued from previous page)

```
    pntr[1] = current sample (idx = 0)
    pntr[2] = next sample (idx = +1)
    pntr[3] = after next sample (idx = +2)

The interpolation takes place between pntr[1] and pntr[2].

Regards,
/Robert Bielik
RBC Audio
```

Listing 89: code

```
const float c_half = 0.5f;

__declspec(naked) float __hermite(float frac_pos, const float* pntr)
{
    __asm
    {
            push    ecx;
            mov             ecx, dword ptr[esp + 12]; /////////////////////////////////////////////////////////////////////////////////////////////////
            add             ecx, 0x04;                      //      ST(0)           ST(1)           ST(2)           ST(3)           ST(4)           ST(5)           ST(6)           ST(7)
            fld             dword ptr [ecx+4];      //      x1
            fsub    dword ptr [ecx-4];      //      x1-xm1
            fld             dword ptr [ecx];        //      x0              x1-xm1
            fsub    dword ptr [ecx+4];      //      v               x1-xm1
            fld             dword ptr [ecx+8];      //      x2              v               x1-xm1
            fsub    dword ptr [ecx];        //      x2-x0           v               x1-xm1
            fxch    st(2);                          //      x1-m1           v               x2-x0
            fmul    c_half;                         //      c               v               x2-x0
            fxch    st(2);                          //      x2-x0           v               c
            fmul    c_half;                         //      0.5*(x2-x0)     v               c
            fxch    st(2);                          //      c               v               0.5*(x2-x0)
            fst             st(3);                          //      c               v               0.5*(x2-x0)     c
            fadd    st(0), st(1);           //      w               v               0.5*(x2-x0)     c
            fxch    st(2);                          //      0.5*(x2-x0)     v               w               c
            faddp   st(1), st(0);           //  v+.5(x2-x0) w               c
            fadd    st(0), st(1);           //      a               w               c
            fadd    st(1), st(0);           //      a               b_neg           c
            fmul    dword ptr [esp+8];      //      a*frac          b_neg           c
            fsubp   st(1), st(0);           //      a*f-b           c
```

(continues on next page)

(continued from previous page)

```
            fmul    dword ptr [esp+8];      //      (a*f-b)*f       c
            faddp   st(1), st(0);           //      res-x0/f
            fmul    dword ptr [esp+8];      //      res-x0
            fadd    dword ptr [ecx];        //      res
            pop             ecx;
            ret;
    }
}
```

5.49.1 Comments

- **Date**: 2003-11-27 04:53:16
- **By**: dmi@smartelectronix

```
This code produces a nasty buzzing sound.  I think there might be a bug somwhere but␣
↪I haven't found it yet.
```

- **Date**: 2003-11-29 16:41:41
- **By**: gro.lortnocdnim@psdcisum-sulph

```
I agree; the output, when plotted, looks like it has overlaid rectangular shapes on␣
↪it.
```

- **Date**: 2003-12-02 21:18:57
- **By**: moc.oiduacbr@kileib.trebor

```
AHH! True! A bug has sneaked in! Change the row that says:
fsubp st(1), st(0); // a*f-b c
to:
fsubrp st(1), st(0); // a*f-b c

and it should be much better. Although I noticed that a good optimization by the␣
↪compiler generates nearly as fast a code, but only nearly. This is still about 10%␣
↪faster.
```

- **Date**: 2004-09-07 02:24:34
- **By**: moc.enecslatnemirepxe@renrewd

```
I have tested the four hermite interpolation algorithms posted to musicdsp.org plus␣
↪the assemblified version of

Laurent de Soras' code by Robert Bielik and found that on a Pentium 4 with full␣
↪optimization (targeting the

Pentium 4 and above, but not using code that won't work on older processors) with MS␣
↪VC++ 7 that the second

function is the fastest.


Function    Percent Total    Time    Return Value

```

(continues on next page)

(continued from previous page)

```
hermite1:   18.90%,          375ms,  0.52500004f
hermite2:   16.53%,          328ms,  0.52500004f
hermite3:   17.34%,          344ms,  0.52500004f
hermite4:   19.66%,          390ms,  0.52500004f
hermite5:   27.57%,          547ms,  0.52500004f

- Daniel Werner
http://experimentalscene.com/
```

5.50 Hermite interpollation

- **Author or source:** various
- **Created:** 2002-04-09 16:55:35

Listing 90: notes

```
These are all different ways to do the same thing : hermite interpollation. Try'm all␣
↪and
benchmark.
```

Listing 91: 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;
```

(continues on next page)

(continued from previous page)

```
        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);
}
```

5.50.1 Comments

- **Date**: 2002-05-25 13:37:32
- **By**: theguylle

```
               great sources but what is Hermite ?
if you don't describe what is your code made for, you will made a great sources but I␣
↪don't know why?

cheers Paul
```

- **Date**: 2002-08-15 00:48:25
- **By**: gro.psdcisum@marb

```
hermite is interpollation.
have a look around the archive, you'll see that the word 'hermite' is in more than␣
↪one item ;-)

  -bram
```

- **Date**: 2003-07-29 10:46:40
- **By**: rb.moc.rapenas@fwodlanor

```
Please, would like to know of hermite code it exists in delphi.

thankful

Ronaldo
Cascavel/Paraná/Brasil
```

- **Date**: 2003-10-10 08:36:06
- **By**: moc.rotces-dabMAPSON@inirgam.m

```
Please,
add, at least, the meaning of each parameter (I mean x, y0,  y1,y2,  y3)....
m.
```

- **Date**: 2003-11-28 10:48:17
- **By**: musicdsp@[remove this]dsparsons.co.uk

```
Ronaldo, it doesn't take much to translate these to Delphi - for float, either use
↪single or double to your preference!

Looking at the codes, it seems quite clear that the parameters follow a pattern of:
↪Sample Position between middle two samples, then the sample before current, current
↪sample, curernt sample +1, current sample +2.

HTH
DSP
```

- **Date**: 2004-03-28 20:51:59
- **By**: moc.liamtoh@sisehtnysorpitna

```
What are all these variables standing for? Not very clear :|
```

- **Date**: 2004-04-20 00:25:29
- **By**: George

```
parameters are allright.

xm1 ---> x[n-1]
x0  ---> x[n]
x1  ---> x[n+1]
x2  ---> x[n+2]

fractional position stands for a fraction between 0 and 1 to interpolate
```

- **Date**: 2004-10-17 01:16:50
- **By**: snehpyhehttuoekatdnatahwwonkuoy.liamg@rood-nosiop-saionarap

```
Couldn't #2 be sped up a hair by commenting out

float c0 = y1;

and then replacing c0 with y1 in the return line?  Or do the compilers do that kind
↪of thing automatically when they optimize?
```

- **Date**: 2007-07-13 01:52:27
- **By**: moc.oi@htnysa

```
"Couldn't #2 be sped up a hair"
It gets optimized out.
```

5.51 Linear interpolation

- **Author or source:** uh.etle.fni@yfoocs
- **Type:** Linear interpolators for oversampled audio
- **Created:** 2007-02-19 10:02:41

Listing 92: notes

```
Simple, fast linear interpolators for upsampling a signal by a factor of 2,4,8,16 or␣
↪32.
Not very usable on their own since they introduce aliasing (but still better than zero
order hold). These are best used with already oversampled signals.

-- Peter Schoffhauzer
```

Listing 93: code

```
#ifndef __LIN_INTERPOLATOR_H_
#define __LIN_INTERPOLATOR_H_

/************************************************************************
*   Linear interpolator class                                                                    ␣
↪                                *
************************************************************************/

class interpolator_linear
{
public:
    interpolator_linear() {
            reset_hist();
    }

    // reset history
    void reset_hist() {
            d1 = 0.f;
    }

    // 2x interpolator
    // out: pointer to float[2]
    inline void process2x(float const in, float *out) {
            out[0] = d1 + 0.5f*(in-d1);     // interpolate
            out[1] = in;
            d1 = in; // store delay
    }

    // 4x interpolator
    // out: pointer to float[4]
    inline void process4x(float const in, float *out) {
            float y = in-d1;
            out[0] = d1 + 0.25f*y;   // interpolate
            out[1] = d1 + 0.5f*y;
            out[2] = d1 + 0.75f*y;
            out[3] = in;
            d1 = in; // store delay
    }
```

(continues on next page)

(continued from previous page)

```

    // 8x interpolator
    // out: pointer to float[8]
    inline void process8x(float const in, float *out) {
            float y = in-d1;
            out[0] = d1 + 0.125f*y; // interpolate
            out[1] = d1 + 0.25f*y;
            out[2] = d1 + 0.375f*y;
            out[3] = d1 + 0.5f*y;
            out[4] = d1 + 0.625f*y;
            out[5] = d1 + 0.75f*y;
            out[6] = d1 + 0.875f*y;
            out[7] = in;
            d1 = in; // store delay
    }

    // 16x interpolator
    // out: pointer to float[16]
    inline void process16x(float const in, float *out) {
            float y = in-d1;
            out[0] = d1 + (1.0f/16.0f)*y;   // interpolate
            out[1] = d1 + (2.0f/16.0f)*y;
            out[2] = d1 + (3.0f/16.0f)*y;
            out[3] = d1 + (4.0f/16.0f)*y;
            out[4] = d1 + (5.0f/16.0f)*y;
            out[5] = d1 + (6.0f/16.0f)*y;
            out[6] = d1 + (7.0f/16.0f)*y;
            out[7] = d1 + (8.0f/16.0f)*y;
            out[8] = d1 + (9.0f/16.0f)*y;
            out[9] = d1 + (10.0f/16.0f)*y;
            out[10] = d1 + (11.0f/16.0f)*y;
            out[11] = d1 + (12.0f/16.0f)*y;
            out[12] = d1 + (13.0f/16.0f)*y;
            out[13] = d1 + (14.0f/16.0f)*y;
            out[14] = d1 + (15.0f/16.0f)*y;
            out[15] = in;
            d1 = in; // store delay
    }

    // 32x interpolator
    // out: pointer to float[32]
    inline void process32x(float const in, float *out) {
            float y = in-d1;
            out[0] = d1 + (1.0f/32.0f)*y;   // interpolate
            out[1] = d1 + (2.0f/32.0f)*y;
            out[2] = d1 + (3.0f/32.0f)*y;
            out[3] = d1 + (4.0f/32.0f)*y;
            out[4] = d1 + (5.0f/32.0f)*y;
            out[5] = d1 + (6.0f/32.0f)*y;
            out[6] = d1 + (7.0f/32.0f)*y;
            out[7] = d1 + (8.0f/32.0f)*y;
            out[8] = d1 + (9.0f/32.0f)*y;
            out[9] = d1 + (10.0f/32.0f)*y;
            out[10] = d1 + (11.0f/32.0f)*y;
            out[11] = d1 + (12.0f/32.0f)*y;
            out[12] = d1 + (13.0f/32.0f)*y;
            out[13] = d1 + (14.0f/32.0f)*y;
```

(continues on next page)

(continued from previous page)

```
            out[14] = d1 + (15.0f/32.0f)*y;
            out[15] = d1 + (16.0f/32.0f)*y;
            out[16] = d1 + (17.0f/32.0f)*y;
            out[17] = d1 + (18.0f/32.0f)*y;
            out[18] = d1 + (19.0f/32.0f)*y;
            out[19] = d1 + (20.0f/32.0f)*y;
            out[20] = d1 + (21.0f/32.0f)*y;
            out[21] = d1 + (22.0f/32.0f)*y;
            out[22] = d1 + (23.0f/32.0f)*y;
            out[23] = d1 + (24.0f/32.0f)*y;
            out[24] = d1 + (25.0f/32.0f)*y;
            out[25] = d1 + (26.0f/32.0f)*y;
            out[26] = d1 + (27.0f/32.0f)*y;
            out[27] = d1 + (28.0f/32.0f)*y;
            out[28] = d1 + (29.0f/32.0f)*y;
            out[29] = d1 + (30.0f/32.0f)*y;
            out[30] = d1 + (31.0f/32.0f)*y;
            out[31] = in;
            d1 = in; // store delay
    }

private:
    float d1; // previous input
};

#endif
```

5.51.1 Comments

- **Date**: 2013-03-31 20:45:21
- **By**: moc.liamg@jdcisumff

```
I incorporated the 32x interpolator with something along this

void process32x(float const in_l, float const in_r, float *out_l, float *out_r)
{
        float y_l = in_l-f_d1_l;
        out_l[0] = f_d1_l + (1.0f/32.0f)*y_l;    // interpolate
        out_l[1] = f_d1_l + (2.0f/32.0f)*y_l;
        out_l[2] = f_d1_l + (3.0f/32.0f)*y_l;
        out_l[3] = f_d1_l + (4.0f/32.0f)*y_l;
        out_l[4] = f_d1_l + (5.0f/32.0f)*y_l;
        out_l[5] = f_d1_l + (6.0f/32.0f)*y_l;
        out_l[6] = f_d1_l + (7.0f/32.0f)*y_l;
        out_l[7] = f_d1_l + (8.0f/32.0f)*y_l;
        out_l[8] = f_d1_l + (9.0f/32.0f)*y_l;
        out_l[9] = f_d1_l + (10.0f/32.0f)*y_l;
        out_l[10] = f_d1_l + (11.0f/32.0f)*y_l;
        out_l[11] = f_d1_l + (12.0f/32.0f)*y_l;
        out_l[12] = f_d1_l + (13.0f/32.0f)*y_l;
        out_l[13] = f_d1_l + (14.0f/32.0f)*y_l;
        out_l[14] = f_d1_l + (15.0f/32.0f)*y_l;
        out_l[15] = f_d1_l + (16.0f/32.0f)*y_l;
        out_l[16] = f_d1_l + (17.0f/32.0f)*y_l;
```

(continues on next page)

(continued from previous page)

```
        out_l[17] = f_d1_l + (18.0f/32.0f)*y_l;
        out_l[18] = f_d1_l + (19.0f/32.0f)*y_l;
        out_l[19] = f_d1_l + (20.0f/32.0f)*y_l;
        out_l[20] = f_d1_l + (21.0f/32.0f)*y_l;
        out_l[21] = f_d1_l + (22.0f/32.0f)*y_l;
        out_l[22] = f_d1_l + (23.0f/32.0f)*y_l;
        out_l[23] = f_d1_l + (24.0f/32.0f)*y_l;
        out_l[24] = f_d1_l + (25.0f/32.0f)*y_l;
        out_l[25] = f_d1_l + (26.0f/32.0f)*y_l;
        out_l[26] = f_d1_l + (27.0f/32.0f)*y_l;
        out_l[27] = f_d1_l + (28.0f/32.0f)*y_l;
        out_l[28] = f_d1_l + (29.0f/32.0f)*y_l;
        out_l[29] = f_d1_l + (30.0f/32.0f)*y_l;
        out_l[30] = f_d1_l + (31.0f/32.0f)*y_l;
        out_l[31] = in_l;
        f_d1_l = in_l; // store delay_l

            float y_r = in_r-f_d1_r;
        out_r[0] = f_d1_r + (1.0f/32.0f)*y_r;    // inrterpolate
        out_r[1] = f_d1_r + (2.0f/32.0f)*y_r;
        out_r[2] = f_d1_r + (3.0f/32.0f)*y_r;
        out_r[3] = f_d1_r + (4.0f/32.0f)*y_r;
        out_r[4] = f_d1_r + (5.0f/32.0f)*y_r;
        out_r[5] = f_d1_r + (6.0f/32.0f)*y_r;
        out_r[6] = f_d1_r + (7.0f/32.0f)*y_r;
        out_r[7] = f_d1_r + (8.0f/32.0f)*y_r;
        out_r[8] = f_d1_r + (9.0f/32.0f)*y_r;
        out_r[9] = f_d1_r + (10.0f/32.0f)*y_r;
        out_r[10] = f_d1_r + (11.0f/32.0f)*y_r;
        out_r[11] = f_d1_r + (12.0f/32.0f)*y_r;
        out_r[12] = f_d1_r + (13.0f/32.0f)*y_r;
        out_r[13] = f_d1_r + (14.0f/32.0f)*y_r;
        out_r[14] = f_d1_r + (15.0f/32.0f)*y_r;
        out_r[15] = f_d1_r + (16.0f/32.0f)*y_r;
        out_r[16] = f_d1_r + (17.0f/32.0f)*y_r;
        out_r[17] = f_d1_r + (18.0f/32.0f)*y_r;
        out_r[18] = f_d1_r + (19.0f/32.0f)*y_r;
        out_r[19] = f_d1_r + (20.0f/32.0f)*y_r;
        out_r[20] = f_d1_r + (21.0f/32.0f)*y_r;
        out_r[21] = f_d1_r + (22.0f/32.0f)*y_r;
        out_r[22] = f_d1_r + (23.0f/32.0f)*y_r;
        out_r[23] = f_d1_r + (24.0f/32.0f)*y_r;
        out_r[24] = f_d1_r + (25.0f/32.0f)*y_r;
        out_r[25] = f_d1_r + (26.0f/32.0f)*y_r;
        out_r[26] = f_d1_r + (27.0f/32.0f)*y_r;
        out_r[27] = f_d1_r + (28.0f/32.0f)*y_r;
        out_r[28] = f_d1_r + (29.0f/32.0f)*y_r;
        out_r[29] = f_d1_r + (30.0f/32.0f)*y_r;
        out_r[30] = f_d1_r + (31.0f/32.0f)*y_r;
        out_r[31] = in_r;
        f_d1_r = in_r; // store delay_r
}

Unfortunately, I am doing something crazy wrong. When I close my plug-in, my DAW
→freezes. I'm fairly new to programming and am not sure what I'm doing wrong.

I'm using your function to write to an audio buffer which is being set to a delay.
→This is what the call looks like.
```

(continues on next page)

(continued from previous page)

```
process32x((fLeftAudioBuffer[i] * decay), (fRightAudioBuffer[i] * decay),
                    &fCircularLeftAudioBuffer[(fCircularBufferPosition + delayFrames)
↪%kCircularBufferSize],
                    &fCircularRightAudioBuffer[(fCircularBufferPosition + delayFrames)
↪%kCircularBufferSize]);

I would love to use this function because it's one line less than how I usually do it.
↪ It works great with no problems.
fCircularLeftAudioBuffer[(fCircularBufferPosition + delayFrames)%kCircularBufferSize]␣
↪= fLeftAudioBuffer[jh] * decay;
fCircularRightAudioBuffer[(fCircularBufferPosition + delayFrames)
↪%kCircularBufferSize] = fRightAudioBuffer[jh] * decay;


Everything seems simple, but I'm puzzled as to what may be wrong. Thanks.
```

- **Date**: 2013-03-31 22:18:36
- **By**: moc.liamg@jdcisumff

```
Never mind. I was calling the function from the wrong place. Works like a charm.

Thank you.
```

5.52 Lock free fifo

- **Author or source:** Timo
- **Created:** 2002-09-13 16:21:59
- **Linked files:** `LockFreeFifo.h`.

Listing 94: notes

```
Simple implementation of a lock free FIFO.
```

5.52.1 Comments

- **Date**: 2003-04-15 11:17:31
- **By**: moc.oohay@SIHTEVOMER_ralohcshsoj

```
There is a good algorithm for a lock free (but multiprocessor safe) FIFO.  But the␣
↪given implimentation is flawed in a number of ways.  This code is not reliable. ␣
↪Two problems on the surface of it:
1. I can easily see that it's possible for two threads/processors to return the same␣
↪item from the head if the timing is right.
2. there's no interlocked instructions to make sure that changes to the shared␣
↪variables are globally visible
3. there's not attempt in the code to make sure that data is read in an atomic way,␣
↪let alone changed in one...

The code is VERY naive
```

(continues on next page)

(continued from previous page)

```
I do have code that works, but it's not so short that will post it in a comment.  If␣
↪anyone needs it they can email me
```

- **Date**: 2003-05-15 19:16:24
- **By**: Timo

```
This is only supposed to work on uniprocessor machines with _one_ reading and _one_␣
↪writing thread
assuming that the assignments to read and write idx are simple mov instructions (i.e.␣
↪atomic). To be sure you'd need to write the update parts in hand-coded asm; never␣
↪know what the compiler comes up with. The context of this code was omitted (i.e.␣
↪Bram posted my written in 1 min sketch in a discussion on IRC on a lock-free fifo,␣
↪not production code).
```

5.53 MATLAB-Tools for SNDAN

- **Author or source:** Markus Sapp
- **Created:** 2002-01-17 03:11:05
- **Linked files:** `other001.zip`.

Listing 95: notes

```
(see linkfile)
```

5.54 MIDI note/frequency conversion

- **Author or source:** ed.bew@raebybot
- **Type:** -
- **Created:** 2002-11-25 18:14:17

Listing 96: notes

```
I get often asked about simple things like MIDI note/frequency conversion, so I␣
↪thought I
could as well post some source code about this.
The following is Pascal/Delphi syntax, but it shouldn't be a problem to convert it to
almost any language in no time.

Uses for this code are mainly for initializing oscillators to the right frequency␣
↪based
upon a given MIDI note, but you might also check what MIDI note is closest to a given
frequency for pitch detection etc.
In realtime applications it might be a good idea to get rid of the power and log2
calculations and generate a lookup table on initialization.

A full Pascal/Delphi unit with these functions (including lookup table generation)␣
↪and a
```

(continues on next page)

(continued from previous page)

```
simple demo application can be downloaded here:
http://tobybear.phreque.com/dsp_conv.zip

If you have any comments/suggestions, please send them to: tobybear@web.de
```

Listing 97: code

```
// MIDI NOTE/FREQUENCY CONVERSIONS

const notes:array[0..11] of string= ('C ','C#','D ','D#','E ','F ','F#','G ','G#','A
↪','A#','B ');
const base_a4=440; // set A4=440Hz

// converts from MIDI note number to frequency
// example: NoteToFrequency(12)=32.703
function NoteToFrequency(n:integer):double;
begin
 if (n>=0)and(n<=119) then
  result:=base_a4*power(2,(n-57)/12)
 else result:=-1;
end;

// converts from MIDI note number to string
// example: NoteToName(12)='C 1'
function NoteToName(n:integer):string;
begin
 if (n>=0)and(n<=119) then
  result:=notes[n mod 12]+inttostr(n div 12)
 else result:='---';
end;

// converts from frequency to closest MIDI note
// example: FrequencyToNote(443)=57 (A 4)
function FrequencyToNote(f:double):integer;
begin
 result:=round(12*log2(f/base_a4))+57;
end;

// converts from string to MIDI note
// example: NameToNote('A4')=57
function NameToNote(s:string):integer;
var c,i:integer;
begin
 if length(s)=2 then s:=s[1]+' '+s[2];
 if length(s)<>3 then begin result:=-1;exit end;
 s:=uppercase(s);
 c:=-1;
 for i:=0 to 11 do
  if notes[i]=copy(s,1,2) then
  begin
   c:=i;
   break
  end;
 try
  i:=strtoint(s[3]);
  result:=i*12+c;
```

(continues on next page)

(continued from previous page)

```
 except
  result:=-1;
 end;
 if c<0 then result:=-1;
end;
```

5.54.1 Comments

- **Date**: 2002-11-29 17:34:28
- **By**: ed.bew@raebybot

```
For the sake of completeness, here is octave fraction notation and pitch class
↪notation:

// converts from MIDI note to octave fraction notation
// the integer part of the result is the octave number, where
// 8 is the octave starting with middle C. The fractional part
// is the note within the octave, where 1/12 represents a semitone.
// example: NoteToOct(57)=7.75
function NoteToOct(i:integer):double;
begin
 result:=3+(i div 12)+(i mod 12)/12;
end;

// converts from MIDI note to pitch class notation
// the integer part of the number is the octave number, where
// 8 is the octave starting with middle C. The
fractional part
// is the note within the octave, where a 0.01 increment is a
// semitone.
// example: NoteToPch(57)=7.09
function NoteToPch(i:integer):double;
begin
 result:=3+(i div 12)+(i mod 12)*0.01;
end;
```

- **Date**: 2002-12-03 12:36:05
- **By**: moc.noicratse@ajelak

```
I thought most sources gave A-440Hz = MIDI note 69. MIDI 60 = middle C = ~262Hz, A-
↪440 = "A above middle C". Not so?
```

- **Date**: 2003-05-14 03:24:58
- **By**: DFL

```
Kaleja is correct. Here is some C code:

  double MIDItoFreq( char keynum ) {
    return 440.0 * pow( 2.0, ((double)keynum - 69.0) / 12.0 );
  }

you can double-check the table here:
http://tomscarff.tripod.com/midi_analyser/midi_note_frequency.htm
```

5.55 Matlab Time Domain Impulse Response Inverter/Divider

- **Author or source:** moc.sdohtemenacra@enacra
- **Created:** 2005-01-19 22:27:15

Listing 98: notes

```
Matlab code for time domain inversion of an impulse response or the division of two of
them (transfer function.)  The main teoplitz function is given both as a .m file and␣
↪as a
.c file for Matlab'w MEX compilation.  The latter is much faster.
```

Listing 99: code

```
function inv=invimplms(den,n,d)
% syntax inv=invimplms(den,n,d)
%         den - denominator impulse
%         n   - length of result
%         d   - delay of result
%         inv - inverse impulse response of length n with delay d
%
% Levinson-Durbin algorithm from Proakis and Manolokis p.865
%
% Author: Bob Cain, May 1, 2001 arcane[AT]arcanemethods[DOT]com

    m=xcorr(den,n-1);
    m=m(n:end);
    b=[den(d+1:-1:1);zeros(n-d-1,1)];
    inv=toepsolve(m,b);



function quo=divimplms(num,den,n,d)
%Syntax quo=divimplms(num,den,n,d)
%       num - numerator impulse
%       den - denominator impulse
%       n   - length of result
%       d   - delay of result
%           quo - quotient impulse response of length n delayed by d
%
% Levinson-Durbin algorithm from Proakis and Manolokis p.865
%
% Author: Bob Cain, May 1, 2001 arcane@arcanemethods.com

    m=xcorr(den,n-1);
    m=m(n:end);
    b=xcorr([zeros(d,1);num],den,n-1);
    b=b(n:-1:1);
    quo=toepsolve(m,b);


function hinv=toepsolve(r,q)
% Solve Toeplitz system of equations.
%    Solves R*hinv = q, where R is the symmetric Toeplitz matrix
%    whos first column is r
%    Assumes all inputs are real
```

(continues on next page)

(continued from previous page)

```
%    Inputs:
%       r - first column of Toeplitz matrix, length n
%       q - rhs vector, length n
%    Outputs:
%       hinv - length n solution
%
%   Algorithm from Roberts & Mullis, p.233
%
%   Author: T. Krauss, Sept 10, 1997
%
%   Modified: R. Cain, Dec 16, 2004 to remove a pair of transposes
%             that caused errors.

    n=length(q);
    a=zeros(n+1,2);
    a(1,1)=1;

    hinv=zeros(n,1);
    hinv(1)=q(1)/r(1);

    alpha=r(1);
    c=1;
    d=2;

    for k=1:n-1,
       a(k+1,c)=0;
       a(1,d)=1;
       beta=0;
       j=1:k;
       beta=sum(r(k+2-j).*a(j,c))/alpha;
       a(j+1,d)=a(j+1,c)-beta*a(k+1-j,c);
       alpha=alpha*(1-beta^2);
       hinv(k+1,1)=(q(k+1)-sum(r(k+2-j).*hinv(j,1)))/alpha;
       hinv(j)=hinv(j)+a(k+2-j,d)*hinv(k+1);
       temp=c;
       c=d;
       d=temp;
    end


-----What follows is the .c version of toepsolve--------

#include <math.h>
#include "mex.h"

/*  function hinv = toepsolve(r,q);
 *  TOEPSOLVE  Solve Toeplitz system of equations.
 *    Solves R*hinv = q, where R is the symmetric Toeplitz matrix
 *    whos first column is r
 *    Assumes all inputs are real
 *    Inputs:
 *       r - first column of Toeplitz matrix, length n
 *       q - rhs vector, length n
 *    Outputs:
 *       hinv - length n solution
 *
 *   Algorithm from Roberts & Mullis, p.233
```

(continues on next page)

(continued from previous page)

```
 *
 *   Author: T. Krauss, Sept 10, 1997
 *
 *   Modified: R. Cain, Dec 16, 2004 to replace unnecessasary
 *             n by n matrix allocation for a with an n by 2 rotating
 *             buffer and to more closely match the .m function.
 */
void mexFunction(
    int nlhs,
    mxArray *plhs[],
    int nrhs,
    const mxArray *prhs[]
)
{
   double (*a)[2],*hinv,alpha,beta;
   int c,d,temp,j,k;

   double eps = mxGetEps();
   int n = (mxGetN(prhs[0])>=mxGetM(prhs[0])) ? mxGetN(prhs[0]) : mxGetM(prhs[0]) ;
   double *r = mxGetPr(prhs[0]);
   double *q = mxGetPr(prhs[1]);

   a = (double (*)[2])mxCalloc((n+1)*2,sizeof(double));
   if (a == NULL) {
       mexErrMsgTxt("Sorry, failed to allocate buffer.");
   }
   a[0][0]=1.0;

   plhs[0] = mxCreateDoubleMatrix(n,1,0);
   hinv = mxGetPr(plhs[0]);
   hinv[0] = q[0]/r[0];

   alpha=r[0];
   c=0;
   d=1;

   for (k = 1; k < n; k++) {
       a[k][c] = 0;
       a[0][d] = 1.0;
       beta = 0.0;
       for (j = 1; j <= k; j++) {
           beta += r[k+1-j]*a[j-1][c];
       }
       beta /= alpha;
       for (j = 1; j <= k; j++) {
           a[j][d] = a[j][c] - beta*a[k-j][c];
       }
       alpha *= (1 - beta*beta);
       hinv[k] = q[k];
       for (j = 1; j <= k; j++) {
           hinv[k] -= r[k+1-j]*hinv[j-1];
       }
       hinv[k] /= alpha;
       for (j = 1; j <= k; j++) {
           hinv[j-1] += a[k+1-j][d]*hinv[k];
       }
       temp=c;
```

(continues on next page)

(continued from previous page)

```
        c=d;
        d=temp;
    } /* loop over k */

    mxFree(a);

    return;
}
```

5.56 Millimeter to DB (faders...)

- **Author or source:** James McCartney
- **Created:** 2002-04-09 16:55:26

Listing 100: 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.)
```

Listing 101: 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;
}

```

(continues on next page)

(continued from previous page)

```
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;
}
```

5.56.1 Comments

- **Date**: 2004-01-29 21:31:18
- **By**: ed.luosfosruoivas@naitsirhC

```
Pascal Translation...

function MMtoDB(Milimeter:Single):Single;
var mm: Single;
begin
  mm:=100-Milimeter;
  if mm = 0 then Result:=10
  else if mm < 48 then Result:=10-5/12*mm;
  else if mm < 84 then Result:=-10-10/12*(mm-48);
  else if mm < 96 then Result:=-40-20./12*(mm-84);
  else if mm < 100 then Result:=-60-35*(mm-96);
  else Result:=-200.;
end;

function DBtoMM(db:Single):Single;
begin
  if db>=10 then result:=0;
  else if db>-10 then result:=-12/5*(db-10);
  else if db>-40 then result:=48-12/10(db+10);
  else if db>-60 then result:=84-12/20(db+40);
  else if db>-200 then result:=96-1/35(db+60);
  else result:=100.;
  Result:=100-Result;
end;
```

- **Date**: 2010-03-11 22:31:06
- **By**: moc.liamg@rellomehcssih.retuow

```
Flash ActionScript translation:

/**
 * Maps normalized value between 0 and 1 to decibel from -200 to 10.
 * @param normalizedValue: Value between 0 and 1.
 * @return Number: Value in decibel from -200 to 10.
 */
public function normalizedToDecibel(value : Number) : Number
{
    value = (1 - value) * 100;

    if(value <= 0.0) var db : Number = 10.0;
    else if(value < 48.0) db = 10.0 - 5.0 / 12.0 * value;
    else if(value < 84.0) db = -10.0 - 10.0 / 12.0 * (value - 48.0);
    else if(value < 96.0) db = -40.0 - 20.0 / 12.0 * (value - 84.0);
    else if(value < 100.0) db = -60.0 - 35.0 * (value - 96.0);
    else db = -200.0;

    return db;
}

/**
 * Maps decibel from -200 to 10 to normalized value between 0 and 1.
 * @param decibel: Value in decibel from -200 to 10.
 * @return Number:
 */
public function decibelToNormalized(decibel : Number) : Number
{
    if(decibel >= 10.0) var normalizedValue : Number = 0.0;
    else if (decibel > -10.0) normalizedValue = -12.0 / 5.0 * (decibel - 10.0);
    else if (decibel > -40.0) normalizedValue = 48.0 - 12.0 / 10.0 * (decibel + 10.0);
    else if (decibel > -60.0) normalizedValue = 84.0 - 12.0 / 20.0 * (decibel + 40.0);
    else if (decibel > -200.0) normalizedValue = 96.0 - 1.0 / 35.0 * (decibel + 60.0);
    else normalizedValue = 100.0;

    return (100.0 - normalizedValue) / 100.0;
}
```

5.57 Motorola 56300 Disassembler

- **Author or source:** moc.ngisedigid@dnesnwot_sirhc
- **Type:** disassembler
- **Created:** 2005-05-24 07:08:07
- **Linked files:** `Disassemble56k.zip`.

Listing 102: notes

```
This code contains functions to disassemble Motorola 56k opcodes.  The code was␣
↪originally
created by Stephen Davis at Stanford.  I made minor modifications to support many␣
↪56300
opcodes, although it would nice to add them all at some point.  Specifically, I added
```

(continues on next page)

(continued from previous page)

```
support for CLB, NORMF, immediate AND, immediate OR, multi-bit ASR/ASL, multi-bit LSL/
↪LSR,
MAX, MAXM, BRA, Bcc, BSR, BScc, DMAC, MACsu, MACuu, and conditional ALU instructions.
```

5.57.1 Comments

- **Date**: 2005-05-24 20:33:25
- **By**: jawoll

```
nice! let's disassemble virus c, nord lead 3, ...
```

- **Date**: 2005-09-29 19:51:55
- **By**: moc.liamg@rhajile

```
Very nice.  How would one get ahold of the OS for one of these synths, to disassemble␣
↪it?  I've got a Nord Micro, with a single 56303... question is.... how to get the␣
↪OS from the flash?
```

- **Date**: 2011-07-11 08:58:10
- **By**: ach nö

```
After a first look inside the c file i found out that the header file "Utility56k.h"␣
↪is missing which is included in the code file...
```

5.58 Noise Shaping Class

- **Author or source:** ude.anaidni@iehsc
- **Type:** Dithering with 9th order noise shaping
- **Created:** 2002-04-23 06:49:06
- **Linked files:** NS9dither16.h.

Listing 103: 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 -
```

(continues on next page)

(continued from previous page)

```
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)
```

5.58.1 Comments

- **Date**: 2003-05-14 14:01:22
- **By**: mail[ns]@mutagene.net

```
I haven't tried this class out, but it looks like there's a typo in the unrolled loop␣
↪-- shouldn't it read "c[2]*EH[HistPos+2]"?  This might this also account for the 3␣
↪clock cycle improvement on a P-III.

// This arrangement seems to execute 3 clock cycles faster on a P-III
samp -= c[8]*EH[HistPos+8] + c[7]*EH[HistPos+7] + c[6]*EH[HistPos+6] +
c[5]*EH[HistPos+5] + c[4]*EH[HistPos+4] + c[3]*EH[HistPos+3] + c[2]*EH[HistPos+1] +
c[1]*EH[HistPos+1] + c[0]*EH[HistPos];
```

- **Date**: 2005-02-15 15:18:55
- **By**: moc.tnemarkas@vokiahc

```
Great class! But I found one more mistake: function my_mod(9, 9) gives 9 instead of 0␣
↪(9 % 9 = 0).

HistPos = my_mod((HistPos+8), order);
EH[HistPos+9] = EH[HistPos] = output - samp;

 -> HistPos + 9 = 18 (max EH index is 17) which leads to out of array boundary and␣
↪crash.

So my_mod should look like:

__inline my_mod(int x, int n)
{
    if(x >= n) x-=n;
    return x;
}
```

5.59 Nonblocking multiprocessor/multithread algorithms in C++

- **Author or source:** moc.oohay@ralohcshsoj
- **Type:** queue, stack, garbage collection, memory allocation, templates for atomic algorithms and types
- **Created:** 2004-04-07 09:38:12
- **Linked files:** ATOMIC.H.

Listing 104: notes

```
see linked file...
```

5.59.1 Comments

- **Date**: 2008-01-10 17:01:39
- **By**: ten.xliamnahx@xmagie

```
This code has a problem with operation exceeding 4G times.  If you do more then 4G of
↪Put and Get with the MPQueue, "AtomicUInt & Index(int i) { return data[i & (len-1)];
↪} will cause BUG.
Modular operation with length of 2^n is OK, but not for other numbers.
My email does not have any "x" letters.
```

- **Date**: 2011-05-23 22:22:21
- **By**: ude.fcu.sc@awajuk

```
In MPCountStack::PopElement, what's to prevent another thread from deleting was.
↪Value().ptr between assigning was and reading was.Value().ptr->next?
```

5.60 Piecewise quadratic approximate exponential function

- **Author or source:** Johannes M.-R.
- **Type:** Approximation of base-2 exponential function
- **Created:** 2007-06-18 08:09:58

Listing 105: notes

```
The code assumes round-to-zero mode, and ieee 754 float.
To achieve other bases, multiply the input by the logarithmus dualis of the base.
```

Listing 106: code

```
inline float fpow2(const float y)
{
    union
    {
            float f;
            int i;
    } c;

    int integer = (int)y;
    if(y < 0)
            integer = integer-1;

    float frac = y - (float)integer;

    c.i = (integer+127) << 23;
    c.f *= 0.33977f*frac*frac + (1.0f-0.33977f)*frac + 1.0f;
```

(continues on next page)

(continued from previous page)

```
    return c.f;
}
```

5.61 Pow(x,4) approximation

- **Author or source:** Stefan Stenzel
- **Created:** 2002-01-17 03:09:20

Listing 107: notes

```
Very hacked, but it gives a rough estimate of x**4 by modifying exponent and mantissa.
```

Listing 108: 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;
}
```

5.62 Rational tanh approximation

- **Author or source:** cschueler
- **Type:** saturation
- **Created:** 2006-11-15 17:29:12

Listing 109: notes

```
This is a rational function to approximate a tanh-like soft clipper. It is based on
↪the
pade-approximation of the tanh function with tweaked coefficients.

The function is in the range x=-3..3 and outputs the range y=-1..1. Beyond this range
↪the
output must be clamped to -1..1.

The first to derivatives of the function vanish at -3 and 3, so the transition to the
↪hard
clipped region is C2-continuous.
```

Listing 110: code

```
float rational_tanh(x)
{
    if( x < -3 )
        return -1;
    else if( x > 3 )
        return 1;
    else
        return x * ( 27 + x * x ) / ( 27 + 9 * x * x );
}
```

5.62.1 Comments

- **Date**: 2006-11-24 16:24:54
- **By**: uh.etle.fni@yfoocs

```
Works fine. If you want only a little overdrive, you don't even need the clipping,␣
↪just the last line for faster processing.

float rational_tanh_noclip(x)
{
  return x * ( 27 + x * x ) / ( 27 + 9 * x * x );
}

The maximum error of this function in the -4.5 .. 4.5 range is about 2.6%.
```

- **Date**: 2006-11-30 15:59:33
- **By**: uh.etle.fni@yfoocs

```
By the way this is the fastest tanh() approximation in the archive so far.
```

- **Date**: 2006-12-08 21:21:02
- **By**: cschueler

```
Yep, I thought so.
That's why I thought it would be worth sharing.

Especially fast when using SSE you can do a 4-way parallel implementation, with MIN/
↪MAX and the RCP instruction.
```

- **Date**: 2007-01-26 12:13:50
- **By**: mdsp

```
nice one

BTW if you google about "pade-approximation" you'll find a nice page with many␣
↪solutions for common functions.

there's exp, log, sin, cos, tan, gaussian...
```

- **Date**: 2007-02-18 03:35:13
- **By**: uh.etle.fni@yfoocs

```
Yep, but the RCP increases the noise floor somewhat, giving a quantized sound, so I'd␣
↪refrain from using it for high quality audio.
```

5.63 Reading the compressed WA! parts in gigasampler files

- **Author or source:** Paul Kellett
- **Created:** 2002-02-18 00:51:08
- **Linked files:** `gigxpand.zip`.

Listing 111: 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
```

5.64 Really fast x86 floating point sin/cos

- **Author or source:** moc.nsm@seivadrer
- **Created:** 2003-11-25 17:43:28
- **Linked files:** `sincos.zip`.

Listing 112: notes

```
Frightful code from the Intel Performance optimization front. Not for the squeamish.

The following code calculates sin and cos of a floating point value on x86 platforms␣
↪to 20
bits precision with 2 multiplies and two adds. The basic principle is to use sin(x+y)␣
↪and
cos(x+y) identities to generate the result from lookup tables. Each lookup table takes
care of 10 bits of precision in the input. The same principle can be used to generate
sin/cos to full (! Really. Full!) 24-bit float precision using two 8-bit tables, and␣
↪one
10 bit table (to provide guard bits), for a net speed gain of about 4x over fsin/fcos,
↪ and
8x if you want both sin and cos. Note that microsoft compilers have trouble keeping
doubles aligned properly  on the stack (they must be 8-byte aligned in order not to␣
↪incur
a massive alignment penalty). As a result, this class should NOT be allocated on the
stack. Add it as a member variable to any class that uses it.

e.g.
  class CSomeClass {
      CQuickTrig m_QuickTrig;
    ...
       mQuickTrig.QuickSinCos(dAngle,fSin,fCos);
    ...
    }
```

Listing 113: code

```
(see attached file)
```

5.65 Reasonably accurate/fastish tanh approximation

- **Author or source:** Fuzzpilz
- **Created:** 2004-08-17 22:56:29

Listing 114: notes

```
Fairly obvious, but maybe not obvious enough, since I've seen calls to tanh() in code
snippets here.

It's this, basically:

tanh(x) = sinh(x)/cosh(x)
        = (exp(x) - exp(-x))/(exp(x) + exp(-x))
        = (exp(2x) - 1)/(exp(2x) + 1)

Combine this with a somewhat less accurate approximation for exp than usual (I use a
third-order Taylor approximation below), and you're set. Useful for waveshaping.

Notes on the exp approximation:

It only works properly for input values above x, but since tanh is odd, that isn't a
problem.

exp(x) = 1 + x + x^2/(2!) + x^3/(3!) + ...

Breaking the Taylor series off after the third term, I get

1 + x + x^2/2 + x^3/6.

I can save some multiplications by using

6 + x * (6 + x * (3 + x))

instead; a division by 6 becomes necessary, but is lumped into the additions in the␣
↪tanh
part:

(a/6 - 1)/(a/6 + 1) = (a - 6)/(a + 6).

Accuracy:

I haven't looked at this in very great detail, but it's always <= the real tanh (>=␣
↪for
x<0), and the greatest deviation occurs at about +/- 1.46, where the result is ca. .95
times the correct value.


This is still faster than tanh if you use a better approximation for the exponential,␣
↪even
```

(continues on next page)

(continued from previous page)

```
if you simply call exp.
There are probably additional ways of improving parts of this, and naturally if you're
going to use it you'll want to figure out whether your particular application offers
additional ways of simplifying it, but it's a good start.
```

Listing 115: code

```
/* single precision absolute value, a lot faster than fabsf() (if you use MSVC++ 6 Standard - others' implementations might be less slow) */
float sabs(float a)
{
int b=(*((int *)(&a)))&0x7FFFFFFF;
return *((float *)(&b));
}

/* approximates tanh(x/2) rather than tanh(x) - depending on how you're using this, fixing that could well be wasting a multiplication (though that isn't much, and it could be done with an integer addition in sabs instead)  */
float tanh2(float x)
{
float a=sabs(x);
a=6+a*(6+a*(3+a));
return ((x<0)?-1:1)*(a-6)/(a+6); /* instead of using <, you can also check directly whether the sign bit is set ((*((int *)(&x)))&0x80000000), but it's not really worth it */
}
```

5.65.1 Comments

- **Date**: 2004-09-19 03:08:02
- **By**: ten.xmg@zlipzzuf

```
Not sure why this didn't occur to me earlier, but you can easily save another two adds as follows:

a*=(6+a*(3+a));
return ((x<0)?-1:1)*a/(a+12);
```

- **Date**: 2004-10-06 22:46:23
- **By**: moc.noicratse@ajelak

```
You shouldn't need the sabs() on VC6 - you just need to add:

#pragma intrinsic( fabs )

before calling fabsf(), and it should go optimally fast.
```

- **Date**: 2004-10-07 10:56:45
- **By**: Laurent de Soras

```
You can optimise it a bit more by using the fact that tanh (x) = 1 - 2 / (exp (2*x) + 1)
```

- **Date**: 2004-10-07 11:38:00
- **By**: ten.xmg@zlipzzuf

```
AFAIK intrinsics aren't supported by VC6 Standard, but limited to Professional and␣
↪Enterprise. Might be wrong, though, in which case I am a silly person. (no time to␣
↪check now)
```

- **Date**: 2005-03-23 22:48:34
- **By**: ed.luosfosruoivas@naitsirhC

```
Delphi Code:

// approximates tanh(x/2) rather than tanh(x) - depending on how you're using
// this, fixing that could well be wasting a multiplication
function tanh2(x:single):Single;
var a : single;
begin
 a:=f_abs(x);
 a:=a*(12+a*(6+a*(3+a)));
 if (x<0)
  then result:=-a/(a+24)
  else result:= a/(a+24);
end;
```

- **Date**: 2005-03-23 23:01:49
- **By**: ed.luosfosruoivas@naitsirhC

```
Laurent de Soras wrote:
"You can optimise it a bit more by using the fact that tanh (x) = 1 - 2 / (exp (2*x)␣
↪+ 1)"

It's not faster, because you'll need 3 more cycles. The routine would then look like␣
↪this:

function tanh2(x:single):Single;
var a : single;
begin
 a:=f_abs(x);
 a:=24+a*(12+a*(6+a*(3+a)));
 if (x<0)
  then result:= (-1+24/a)
  else result:=  (1-24/a);
end;
```

- **Date**: 2005-05-09 02:37:33
- **By**: eb.tenyks@didid

```
I must have missed this one..
but why is the comparison needed?

a simpler version would be:
a:=Abs(x)
Result:=x*(6+a*(3+a))/(a+12)

no?
```

(continues on next page)

(continued from previous page)

```
So in asm:

function Tanh2(x:Single):Single;
const c3 :Single=3;
      c6 :Single=6;
      c12:Single=12;
Asm
        FLD     x
        FLD     ST(0)
        FABS
        FLD     c3
        FADD    ST(0),ST(1)
        FMUL    ST(0),ST(1)
        FADD    c6
        FMULP   ST(2),ST(0)
        FADD    c12
        FDIVP   ST(1),ST(0)
End;

..but almost all the CPU is wasted by the division anyway
```

- **Date**: 2005-05-09 03:11:16
- **By**: eb.tenyks@didid

```
wait.. has anyone tested this function?

Here's a test plot:
http://www.flstudio.com/gol/tanh.gif

Red=TanH

Green=the approximation suggested in this thread

Blue=another approximation that does:

function TanH3(x:Single):Single;
Begin
Result:=x - x*x*x/3 + 2*x*x*x*x*x/15;
end;

If I didn't do anything wrong, the green one is VERY far from TanH. Blue is both␣
↪closer & computationally more efficient.

But ok, this plot is for a normalized 0..1. When you go above, the blue like goes␣
↪crazy.
But now considering that -1..1 is what matters the most for what we do, the input␣
↪could still be clipped.
```

- **Date**: 2005-05-09 03:23:10
- **By**: eb.tenyks@didid

```
forget all this :)

it's all embarrassing bullshit and I obviously need some sleep :)
```

- **Date**: 2005-05-09 03:46:09
- **By**: eb.tenyks@didid

```
Ok ignore my above crap that I can't delete, I swear that this one does work :)

First I hadn't seen that this function was assuming x*2, so my graph was scaled by 2..

Second, the other algo (blue line) is still not to be neglected (because no FDIV)
↪when the input is in the -1..1 range, and it does work.

Third, I'm suggesting here a version without the comparison/branching, but still, the
↪CPU difference is neglectable because of the FDIV.


Here it is (this one does NOT assume a premultiplied x)..

plain code:

function Tanh2(x:Single):Single;
var   a,b:Single;
begin
x:=x*2;
a:=abs(x);
b:=(6+a*(3+a));
Result:=(x*b)/(a*b+12);
end;


asm:

function Tanh22(x:Single):Single;
const c3  :Single=3;
      c6  :Single=6;
      c12 :Single=12;
      Mul2:Single=2;
Asm
        FLD     x
        FMUL    Mul2
        FLD     ST(0)
        FABS                 // a
        FLD     c3
        FADD    ST(0),ST(1)
        FMUL    ST(0),ST(1)
        FADD    c6           // b
        FMUL    ST(2),ST(0)  // x*b
        FMULP   ST(1),ST(0)  // a*b
        FADD    c12
        FDIVP   ST(1),ST(0)
End;
```

- **Date**: 2005-05-10 00:37:16
- **By**: ed.luosfosruoivas@naitsirhC

```
Any suggestions about improving the 3DNow Divide-Operation??? I simply hate my code...

procedure Transistor2_3DNow(pinp,pout : PSingle; Samples:Integer;Scalar:Single);
const ng   : Array [0..1] of Integer = ($7FFFFFFF,$7FFFFFFF);
```

(continues on next page)

(continued from previous page)

```
      pg   : Array [0..1] of Integer = ($80000000,$80000000);
      c2   : Array [0..1] of Single = (2,2);
      c3   : Array [0..1] of Single = (3,3);
      c6   : Array [0..1] of Single = (6,6);
      c12  : Array [0..1] of Single = (12,12);
      c24  : Array [0..1] of Single = (24,24);
asm
 shr ecx,1
 femms
 movd       mm1, Scalar.Single
 punpckldq  mm1, mm1
 movq       mm0, c2
 pfmul      mm0, mm1

 movq       mm3, c3
 movq       mm4, c6
 movq       mm5, c12
 movq       mm6, c24

@Start:
 movq       mm1, [eax] //mm1=input
 pfmul      mm1, mm0   //mm1=a
 movq       mm2, mm1   //mm1=a,   mm2=a

 pand       mm2, ng    //mm1=a,   mm2=|a|

 pfadd      mm3, c3    //mm1=a,   mm2=|a|, mm3=|a|+3
 pfmul      mm3, mm2   //mm1=a,   mm2=|a|, mm3=|a|*(|a|+3)
 pfadd      mm3, c6    //mm1=a,   mm2=|a|, mm3=6+|a|*(3+|a|)
 pfmul      mm3, mm2   //mm1=a,   mm2=|a|, mm3=|a|*(6+|a|*(3+|a|))
 pfadd      mm3, c12   //mm1=a,   mm2=|a|, mm3=b=12+|a|*(6+|a|*(3+|a|))
 pfmul      mm1, mm3   //mm1=a*b, mm2=|a|, mm3=a*(12+|a|*(6+|a|*(3+|a|)))
 pfmul      mm2, mm3   //mm1=a*b, mm2=|a|*b
 pfadd      mm2, c24   //mm1=a*b, mm2=|a|*b+24


 movq       mm3, mm2
 pfrcp      mm4, mm3
 punpckldq  mm3, mm3
 pfrcpit1   mm3, mm4
 pfrcpit2   mm3, mm4
 movq       mm4, mm2
 punpckhdq  mm4, mm4
 pfrcp      mm5, mm4
 pfrcpit1   mm4, mm5
 pfrcpit2   mm4, mm5
 punpckldq  mm4, mm5
 pfmul      mm1, mm4
 movq       [edx],mm1
 add        eax,8
 add        edx,8
 loop    @Start
 femms
end;
```

- **Date**: 2005-05-10 02:26:36
- **By**: eb.tenyks@didid

```
mmh why the loop? You can't process more than 2 Tanh in parallel in this filter, can
↪you?
What CPU gain did you get btw?

Anyway, sucks that 3Dnow doesn't provide division.. SSE does, though.. DIVPS (or
↪DIVPD to get a double accuracy in this case) would work here. Only problem is that
↪on an AMD I usually get better performances out of 3DNow than SSE/SSE2.
```

- **Date**: 2005-05-10 11:00:10
- **By**: ed.luosfosruoivas@naitsirhC

```
The loop works perfectly well, but it's of course for Tanh() processing of a whole
↪block instead of inside the moog filter.

The thing, that 3DNow doesn't provide division really sucks. Anyway, this way i will
↪save a small amount of performance, but it's not huge. But i believe one can
↪optimize the 12 lines of division further more. Also data prefetching might help a
↪little. Or restructuring, because on AMD the order does matter!

I'll SSE/SSE2 the code tonight. I think SSE gives a good performance boost, but SSE2
↪precisition would be needed, if the thing is inside the moog filter (IIR Filter
↪coefficients should allways stay 64bit to avoid digital artifacts).

Cheers,

Christian
```

- **Date**: 2005-05-11 16:41:26
- **By**: ed.luosfosruoivas@naitsirhC

```
Here's the Analog Devices "Sharc" DSP translation of the tanh function (inline
↪processing of register f0):

f11 = 2.;
f0  = f0 * f11;
f11 = abs f0;
f3  = 3.;
f12 = f11+f3;
f12 = f11*f2;
f3  = 6.;
f12 = f12+f3;
f0  = f0*f12;
f12 = f11*f12;
f7  = 12.;
f12 = f12+f3;
f11 = 2.;
f0 = recips f12, f7=f0;
f12=f0*f12;
f7=f0*f7, f0=f11-f12;
f12=f0*f12;
f7=f0*f7, f0=f11-f12;
f12=f0*f12;
rts(db);
f7=f0*f7, f0=f11-f12;
f0=f0*f7;
```

(continues on next page)

(continued from previous page)

```
it can be optimized further more, but hey...
```

- **Date**: 2006-02-25 09:57:21
- **By**: Gene

```
tanh(x/2)~ x/(abs(x)+3/(2+x*x))

better...
```

- **Date**: 2006-02-25 11:53:34
- **By**: Gene

```
tanh(x/2) ~ x/(abs(x)+2/(2.12-1.44*abs(x)+x*x))

Maximum normalized difference 0.0063 from real tanh (x/2) - good enough now.
```

5.66 Resampling

- **Author or source:** ed.corm@liam
- **Type:** linear interpolated aliased resampling of a wave file
- **Created:** 2004-04-07 09:39:12

Listing 116: notes

```
som resampling stuff. the code is heavily used in MSynth, but do not lough about ;-)

perhaps, prefiltering would reduce aliasing.
```

Listing 117: code

```
signed short* pSample = ...;
unsigned int sampleLength = ...;

// stretch sample to length of one bar...
float playPosDelta = sampleLength / ( ( 240.0f / bpm ) * samplingRate );

// requires for position calculation...
float playpos1 = 0.0f;
unsigned int iter = 0;

// required for interpolation...
unsigned int i1, i2;

float* pDest = ....;
float* pDestStop = pDest + len;
for( float *pt=pDest;pt<pDestStop;++pt )
{
    // linear interpolation...
    i1 = (unsigned int)playpos;
    i2 = i1 + 1;
    (*pt) = ((pSample[i2]-pSample[i1]) * (playpos - i1) + pSample[i1]);
```

(continues on next page)

(continued from previous page)

```

    // position calculation preventing float sumation error...
    playpos1 = (++iter) * playposIncrement;
}
...
```

5.66.1 Comments

- **Date**: 2004-02-15 12:01:50
- **By**: Gog

```
Linear interpolation normally introduces a lot of artefacts. An easy way to improve␣
↪upon this is to use the hermite interpolator instead. The improvement is _dramatic_!
```

- **Date**: 2004-05-04 16:57:06
- **By**: moc.sulp.52retsinnab@etep

```
i1 = (unsigned int)playpos;
i2 = i1 + 1;

would this be better as:

i1 = (unsigned int) floor(playpos);
i2 = (unsigned int) ceil(i1 + playposIncrement);

?

if you are actually decimating rather than interpolating (which is what would give␣
↪aliasing), then the second interpolation point in the input could potentially be␣
↪more than i1 + 1.
```

- **Date**: 2004-05-05 11:26:57
- **By**: moc.sulp.52retsinnab@etep

```
no, sorry it wouldn't :%|
```

- **Date**: 2004-08-13 23:45:16
- **By**: ed.corm@liam

```
yes, a more sophisticated interpolation would
improve the sound and prefiltering would
terminate the aliasing. but everything with
hi runtime overhead.
```

5.67 Saturation

- **Author or source:** Bram
- **Type:** Waveshaper
- **Created:** 2002-09-19 14:27:46

Listing 118: 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 ;-)
```

Listing 119: 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)));
    }
}
```

5.67.1 Comments

- **Date**: 2002-10-15 17:22:22
- **By**: moc.oohay@yrret

```
              But My question is
BUT HAVE YOU TRIED YOUR CODE!!!!!!!!!!!!!!!!????
I think no, 'cos give a compiling error.
the right (for sintax) version is this:
```

(continues on next page)

(continued from previous page)

```
float sigmoid(float x)
{
    if(fabs(x)<1)
        return x*(1.5f - 0.5f*x*x);
    else
        return x > 0.f ? 1.f : -1.f;
}

float saturate(float x, float t)
{
    if(abs(x)<t)
        return x;
    else
    {
        if(x > 0.f)
            return t + (1.f-t)*sigmoid((x-t)/((1-t)*1.5f));
        else
            return -(t + (1.f-t)*sigmoid((-x-t)/((1-t)*1.5f)));
    }
}
```

- **Date**: 2003-11-18 10:16:14
- **By**: moc.liamtoh@tnuhhcaebmi

```
except for the missing parenthesis of course =)
the first line of saturate should be either

if(fabs(x)) return x;

or

if(abs(x)) return x;

depending on whether you're looking at the first or second saturate function (in the
↪orig post)
```

- **Date**: 2021-01-01 11:50:14
- **By**: DKDiveDude

```
```

The first function seems to be only a unnecessary complicated brick limit function. See below how I implemented the first function's code. Left is a sample between -1 and 1, positiveThreshold and negativeThreshold should be self explanatory.

if (left > positiveThreshold) left = positiveThreshold + (1 - positiveThreshold) * tanh ((left - positiveThreshold) / (1 - positiveThreshold));

else if (left < negativeThreshold) left = -(positiveThreshold + (1 - positiveThreshold) * tanh ((-left - positiveThreshold) / (1 - positiveThreshold)));

5.68 Sin(x) Aproximation (with SSE code)

- **Author or source:** moc.kisuw@kmailliw
- **Created:** 2004-07-14 10:13:26

Listing 120: notes

```
Sin Aproximation: sin(x) = x + ( x * (-x * x / 6));

This is very handy and fast, but not precise. Below you will find a simple SSE code.

Remember that all movaps command requires 16 bit aligned variables.
```

Listing 121: code

```
SSE code for computing only ONE value (scalar)
Replace all "ss" with "ps" if you want to calculate 4 values. And instead of "movps"
↪use "movaps".

movss       xmm1,   xmm0                                    ; xmm0 = x
mulss       xmm1,   Filter_GenVal[k_n1]             ; * -1
mulss       xmm1,   xmm0                                    ; -x * x
divss       xmm1,   Filter_GenVal[k_6]              ; / 6
mulss       xmm1,   xmm0
addss       xmm0,   xmm1
```

5.68.1 Comments

- **Date**: 2004-10-06 22:47:58
- **By**: moc.noicratse@ajelak

```
Divides hurt. Change your constant 6 to a constant (1.0/6.0) and change divss to
↪mulss.
```

- **Date**: 2005-05-31 05:04:05
- **By**: little%moc.loa@ykee02

```
error about 7.5% by +/- pi/2
you can improve this considerably by
fitting cubic at points -pi/2, 0, pi/2 i.e:
sin(x) = x - x^3 / 6.7901358
```

5.69 Sin, Cos, Tan approximation

- **Author or source:** http://www.wild-magic.com
- **Created:** 2003-04-26 00:17:56
- **Linked files:** `approx.h`.

Listing 122: notes

```
Code for approximation of cos, sin, tan and inv sin, etc.
Surprisingly accurate and very usable.

[edit by bram]
this code is taken literaly from
http://www.wild-magic.com/SourceCode.html
Go have a look at the MgcMath.h and MgcMath.cpp files in their library...
[/edit]
```

5.69.1 Comments

- **Date**: 2002-09-01 00:06:40
- **By**: moc.oi@htnysa

```
It'd be nice to have a note on the domain of these functions. I assume Sin0 is meant
↪to be used about zero and Sin1 about 1. But a note to that effect would be good.
Thanks,

james mccartney
```

- **Date**: 2003-05-31 18:39:50
- **By**: ten.xmg@mapsedocm

```
Sin0 is faster but less accurate than Sin1, same for the other pairs. The domains are:

Sin/Cos [0, pi/2]
Tan [0,pi/4]
InvSin/Cos [0, 1]
InvTan [-1, 1]

This comes from the original header file.
```

5.70 VST SDK GUI Switch without

- **Author or source:** ti.oohay@odrasotniuq
- **Created:** 2004-09-08 12:49:11

Listing 123: notes

```
In VST GUI an on-vaue is represented by 1.0 and off by 0.0.
```

Listing 124: code

```
Say you have two signals you want to switch between when the user changes a switch.
You could do:

if(fSwitch == 0.f) //fSwitch is either 0.0 or 1.0
    output = input1
```

(continues on next page)

(continued from previous page)

```
else
    output = input2

However, you can avoid the branch by doing:

output = input1 * (1.f - fSwitch) + input2 * fSwitch

Which would be like a quick mix. You could make the change clickless by adding a simple one-pole filter:

smooth = filter(fSwitch)
output = input1 * (1.f - smooth) + input2 * smooth
```

5.70.1 Comments

- **Date**: 2004-09-17 04:12:08
- **By**: moc.liamg@knuhcnezitic

```
Not trying to be incredulous, but ... Is this really worth it? Assuming that you pre-calc the (1-fSwitch), you still have 2 multiplies and 1 add, instead of just an assignment. Are branches really bad enough to justify spending those cycles?

Also, does it matter where in the signal flow the branch is? For instance, if it were at the output, the branch wouldn't be such a problem. But at the input, with many calculations downstream, would it matter more?

Also, what if your branches are much more complicated--i.e. multiple lines per case?
```

- **Date**: 2004-09-24 17:46:57
- **By**: ti.oohay@odrasotniuq

```
I use it when I have to compute the (1-fSwitch) signal anyway.
Example: apply a LFO to amplitude and not to frequency. I compute LFO anyway, then I apply (1-fSwitch) to frequency and (fSwitch) to amplitude.

Yes, branches are really bad!:-)
This is because you "break" your cache waiting for a decision
Even if the branch is at the end of your routine, you are leaving a branch to successive code (i.e. to host)

Anyway, this is not ever worth to use, just consider single cases...
```

- **Date**: 2004-10-07 03:29:01
- **By**: moc.noicratse@ajelak

```
Kids, kids, you're both wrong!

chunk: Two multiples and an add are really cheap on modern hardware - P2/P3 take about 2 clocks for fmul and 1 clock for fadd.

quintosardo: Modern hardware also has good branch prediction, so if the switch is, e.g., a VST parameter that only changes once per process() block, it will branch the same way on the order of 100 times in a row. Correcly predicted branches are basically free; mispredicted branches blow the instruction pipeline, which is a penalty of about 20 cycles or so. If you spread the cost of a single missed prediction over 100 samples, it's cheap enough to not worry about. So yes, use this "predicate transform" to optimize away branches which are unpredictable, but don't worry about branches which are predictable.
```

(continues on next page)

(continued from previous page)

```
```

- **Date**: 2014-02-07 08:22:56
- **By**: ten.rotaniliam@akcalabbuh

```
And today it is even more ridiculous to think about cycles!
```

CHAPTER 6

Indices and tables

- genindex
- search