# Psycho Acoustic Bass Extension Plugin
# Project Report

Sakari Bergen

November 9, 2009

## Contents

# 1 Background

Modern recording techniques and synthesizers allow recorded music and special effects to contain arbitrarily low frequencies. All speaker systems, however, have a limited low frequency response, resulting in decreased sound quality. In music, the lowest significant frequencies are usually around 30 to 40 Hz. The low B-string in a 5-string bass for example, has a fundamental frequency of 30.87 Hz, while the lowest frequencies in some pipe organs and grand pianos are even lower.

Due to the way the human auditory system works, humans can perceive lower frequencies than the ones produced by audio reproduction systems. This is due to the missing fundamental phenomenon, which states that the pitch of a tone is determined according to the overtone series, more or less regardless of the actual lowest frequency perceived [1].

By amplifying or synthesizing overtones for low frequency components of a signal, the perceived bass content can be enhanced. This method, among others, has been discussed in the AES conference paper *A unified approach to low- and high-frequency bandwidth extension* [2], which is used as the basis of this project. The result of this project is a plugin for enhancing bass reproduction using the LV2 audio plugin specification[3].

The adjectives *natural* and *unobtrusive* are used in this report to describe sounds. When comparing the input and output of the plugin, *natural* refers to an output where the timbre of the processed portion has not changed much, while *unobtrusive* refers to an overall signal with little or no disturbing artifacts created by the plugin.

# 2 Methods

Processing the signal consists of the following phases:

1. band-pass filter a portion of the signal
2. create harmonic content from the band-pass signal through a non-linear process
3. filter the produced signal to shape the harmonic content and remove unwanted components
4. mix the filtered signal with the original signal

Biquad IIR filters are used for all filtering. Thus, only the general characteristics of the filters will be discussed in this section, while the non-linear process will be presented in more depth. The block diagram of the algorithm is presented in Figure 1.

## 2.1 Band-pass filtering

The objective of this stage is to select the right frequencies to be processed non-linearly. To create the wanted harmonics, the portion of the signal to be processed should be below the reproduction systems low cut-off frequency ($f_c$). Thus, the higher corner frequency of the first filter should be set at $f_c$. To avoid excessive intermodular distortion, the bandwidth of the filter should not be too large. The bandwidth depends on the signal material and $f_c$, but should not be wider than a few octaves in general, as discussed in [2]. Moreover, the lower cut-off frequency should not be below 16-20Hz, because of the limitiations of the human auditory system. Also, the latency of the system is dependent of the lower cut-off frequency (discussed in Section 3.2).

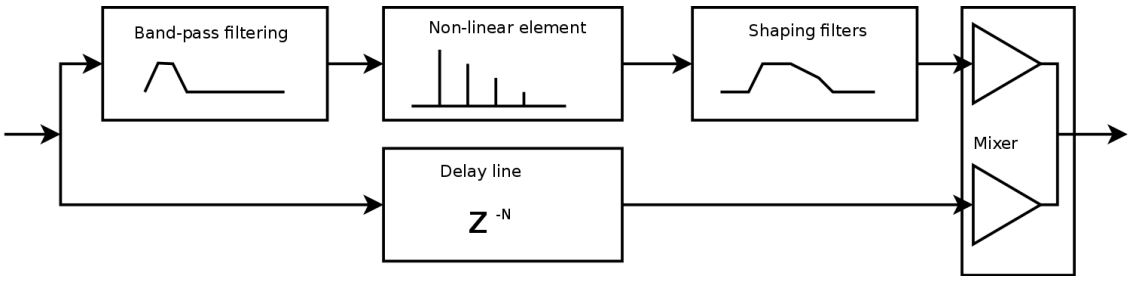


Figure 1: Block diagram

## 2.2 Non-linear processing

The objective of this stage is to synthesize harmonics which create a sensation of the missing fundamental. Experiments have shown that for complex tones (tones with a harmonic structure) having a fundamental frequency below 200Hz, the perceived pitch is judged mainly by the difference between higher harmonics, namely the fourth and fifth harmonic [1]. This means that it is crucial to produce both even and odd harmonics in this stage. To keep the overall sound as natural as possible, the dynamics of the created harmonics should follow the dynamics of the source signal.

A simple integrate-and-dump algorithm is used to fill the above mentioned objectives [2]. The algorithm can be described in pseudo code as follows:

1. start with output at zero
2. add the absolute of the sample value to output
3. if previous sample value was negative and current value is positive, set output to current sample value
4. repeat from step 2

For periodic signals, this is equal to integrating over one period of the the absolute of the signal. The results of the algorithm will be analyzed using a simple sine wave

$$x(t) = A\sin(\omega t) \tag{1}$$

as an example, where $A$ is the amplitude of the signal, $\omega$ the angular frequency and $t$ time. The power over one period is

$$P(in) = \int_0^{\frac{2\pi}{\omega}} (A\sin(\omega t))^2 dt = \frac{A^2\pi}{\omega}. \tag{2}$$

Integrating over one period, the output of the nonlinear process is

$$y(t) = \int_0^t |A\sin(\omega t)|\, dt = \frac{A}{\omega}\left(1 - |\sin(\omega t)|\cot(\omega t)\right) \tag{3}$$

which strongly resembles a sawtooth when plotted. This signal has a power of

$$P(out) = \int_0^{\frac{2\pi}{\omega}} \left( \frac{A}{\omega} \left(1 - |\sin(\omega t)| \cot(\omega t)\right) \right)^2 dt = \frac{3A^2\pi}{\omega^3} \tag{4}$$

over one cycle. The power amplification factor of the algorithm as a function of frequency, can be calculated from Eq. 2 and 4.

$$\frac{P(out)}{P(in)} = \frac{\frac{3A^2\pi}{\omega^3}}{\frac{A^2\pi}{\omega}} = \frac{3}{\omega^2}. \tag{5}$$

Judging by the result of Eq. 5, it is clear that the output of this algorithm has to be scaled depending on the fundamental frequency of the processed signal.

### 2.3 Filtering of the created harmonics

In order to create an unobtrusive and effective result, the harmonics created by non-linear processing need to be filtered. Frequency components below $f_c$ should be removed, as they would not be reproduced anyway. Also, the harmonics created should be filtered with a shaping filter. In addition to maximized bass perception, the objectives of the filtering are naturalness and unobtrusiveness of the final sound. Since naturalness and unobtrusiveness are subjective matters, the filter parameters have been left adjustable for the user.

### 2.4 Mixing and final adjustments

In this stage, the harmonic content created in the previous stages is mixed with the original signal, to produce one output signal. Because new frequency components are created in the process, it is necessary to adjust the level of the signal in order to avoid clipping. Also, all delay introduced in the processing stages is compensated for, by delaying the original signal.

## 3 Implementation

### 3.1 Filters

Biquad IIR filters from the Calf audio plugin pack [4] were used to implement all filters in the project. These filters use formulae by Robert Bristow-Johnson [5] to calculate the filter coefficients and are all second order filters. The filters in the plugin consist of

1. Two band-pass filters, both using the same parameters, configurable for a high cutoff frequency of 40 to 300 Hz and a bandwidth of 0,5 to 5,0 octaves. These filters are used for selecting the material to be processed by the non-linear element discussed in section 3.2.

2. Two low-pass filters for shaping the harmonics created by the non-linear element. These have a cutoff frequency configurable from 40 to 600 Hz and a Q adjustable from $\frac{1}{\sqrt{2}}$ to 5.

3. One high-pass filter configurable for a cutoff frequency from 40 to 300 Hz and a Q adjustable from $\frac{1}{\sqrt{2}}$ to 5. This filter is used to remove dc-offset and excess low-frequency material. It can also be used to boost frequencies around the cutoff, by setting a high Q-value.

### 3.2 Non-linear element

The non-linear element implementation uses the algorithm described in Section 2.2. The output has to be scaled according to the fundamental frequency of the waveform being processed. Since this frequency can only be known after one period of the signal has passed, the amplitude has to be adjusted afterwards, one period at a time. For this purpose, the output values are stored in a ringbuffer before being output. This ringbuffer imposes a limitation on the lowest frequency that can be tracked by the element. On the implementation level the algorithm can be described in pseudo code as follows:

1. **Check for zero-crossing from negative to positive**
   (a) If present
      i. **Calculate relative angular frequency based on counter.** The frequency $\omega = \frac{n}{f_s/2}$, where $n$ is the counter value (samples since last reset) and $f_s$ is the sample rate.
      ii. **Adjust ringbuffer gain based on frequency and counter.** The gain applied to $n$ (samples since last reset) last samples is relative to $\frac{1}{\omega^2}$, as stated in Equation 5.
      iii. **Reset accumulator and counter.**
   (b) Else
      i. **If counter value is equal to ringbuffer size, reset ringbuffer and counter.** Frequencies lower than the lowest trackable frequency are ignored, and the ringbuffer is thus reset.
      ii. **Else increment counter.** The counter is equal to the amount of samples since the last reset, or zero if the frequency is too low to track.
2. **Add the absolute value of sample to accumulator.**
3. **Output last value from ringbuffer and add accumulator value to ringbuffer.** The value output from the ringbuffer has been scaled depending on the frequency found, and has been delayed as many samples as the ringbuffer is long. The value put into the ringbuffer is not yet scaled, but is guaranteed to either be scaled or reset before being output.

It should be noted that the delay and and lowest trackable frequency are related: The delay is equal to one period of the lowest trackable frequency $f$, i.e. $\frac{1}{f}$ seconds.

### 3.3 Mixing

At the very end the processed and original signal are mixed. The original signal is however delayed with a simple delay line, in order to compensate for the delay present in the non-linear element. Both processed and direct signal levels can be adjusted from zero to unity gain. It should be noted, however, that the non-linear element contains some rather non-scientifc gain adjustments hard-coded during listening tests while developing the plugin.

## 4 Results

### 4.1 Informal listening tests

In informal listening tests performed by me, the plugin showed very different characteristics depending on the source material and system cutoff frequency. Different cutoff frequencies were both simulated

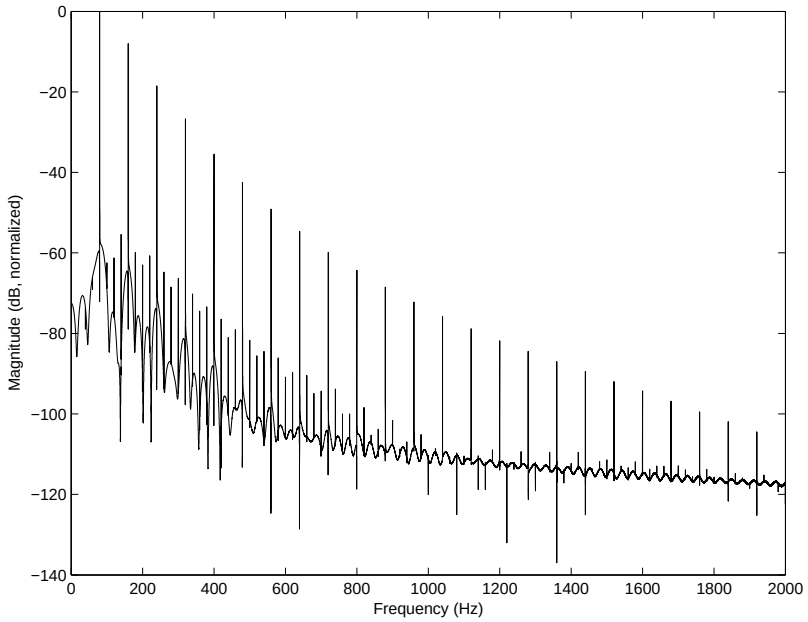


Figure 2: Processed 80 Hz sine wave spectrum

with a high-pass filter and tested on small speakers and headphones.

As could be expected, the plugin was less obtrusive and sounded most natural in material with rich harmonic content. Especially distorted electric guitar played together with bass produced good results. Synthesized pure tones, on the other hand, often sounded very different after processing. Some very low synthesized bass tones which were not intended to have a definite pitch, acquired one during processing, causing very disturbing side effects. These tones had a low fundamental frequency and very little harmonic content, so their timbre was drastically affected by the plugin.

The system cutoff frequency (low-end) affected the optimal low-pass filter settings of the plugin. With lower frequencies, also the filter cutoffs could be set lower. The best results were achieved around 150 Hz, with lower values making the effects of the plugin insignificant, while unobtrusiveness was difficult to achieve with higher values.

## 4.2 Measurements

The plugin was analyzed by examining its output in both time and frequency domain. The non-linear element was tested with a simple sinusoidal input, producing a waveform very close to a sawtooth, just as expected. Feeding the plugin with a 80Hz sine wave produced the output spectrum presented in Figure 2. The harmonic structure produced can be clearly seen with very little additional artifacts present.

Effects of intermodulation were studied by feeding the plugin a 80Hz sine wave together with either a 300 or 500Hz sine, and setting the plugins cutoff frequency at 150Hz. The amplitudes of the higher sine waves were adjusted to have approximately equal perceived loudness. The 300Hz tone (about -8dB relative to the 80Hz tone) caused noticeable intermodulation artifacts, peaking around -28dB (Figure

3), while the 500Hz tone (about -12dB relative to the 80Hz tone) caused no significant artifacts (Figure 4).

In the informal listening tests, material with a strong bass drum was noticed to produce rather bad results. A bass drum sample was thus analyzed, selecting a 100 ms long chirp starting at 70 Hz and ending at 30 Hz as a good approximation for it. The chirp was combined with a constant 400 Hz tone, and the plugin output was analyzed with a peak frequency spectrogram in Sonic Visualiser [6] (44.1kHz samplerate, 4096 point FFT, Blackman-Harris window, 93.75% overlap). The following sreenshots from Sonic Visualiser (Figures 5 and 6) display local peak frequencies in a time-frequency plot. The cursor shows a selected frequency at a given time along with its harmonics, and should be rather self-explanitory.

Figure 5 shows how the plugin has created a proper harmonic structure for the beginning of the chirp, while Figure 6 shows some problems. The harmonic structure in the end of the chirp is delayed, since the corresponding missing fundamental (at cursor) matches that of the middle of the chirp. Fast changes in fundamental frequency appear to cause a structure in the processed sound, where the harmonics created do not match the current fundamental frequency. The fundamental reason for this phenomenon is unclear. It could be due to the non-linear phase of the harmonic shaping filters, or it could have something to do with the characteristics of the sawtooth wave generated by the non-linear element.

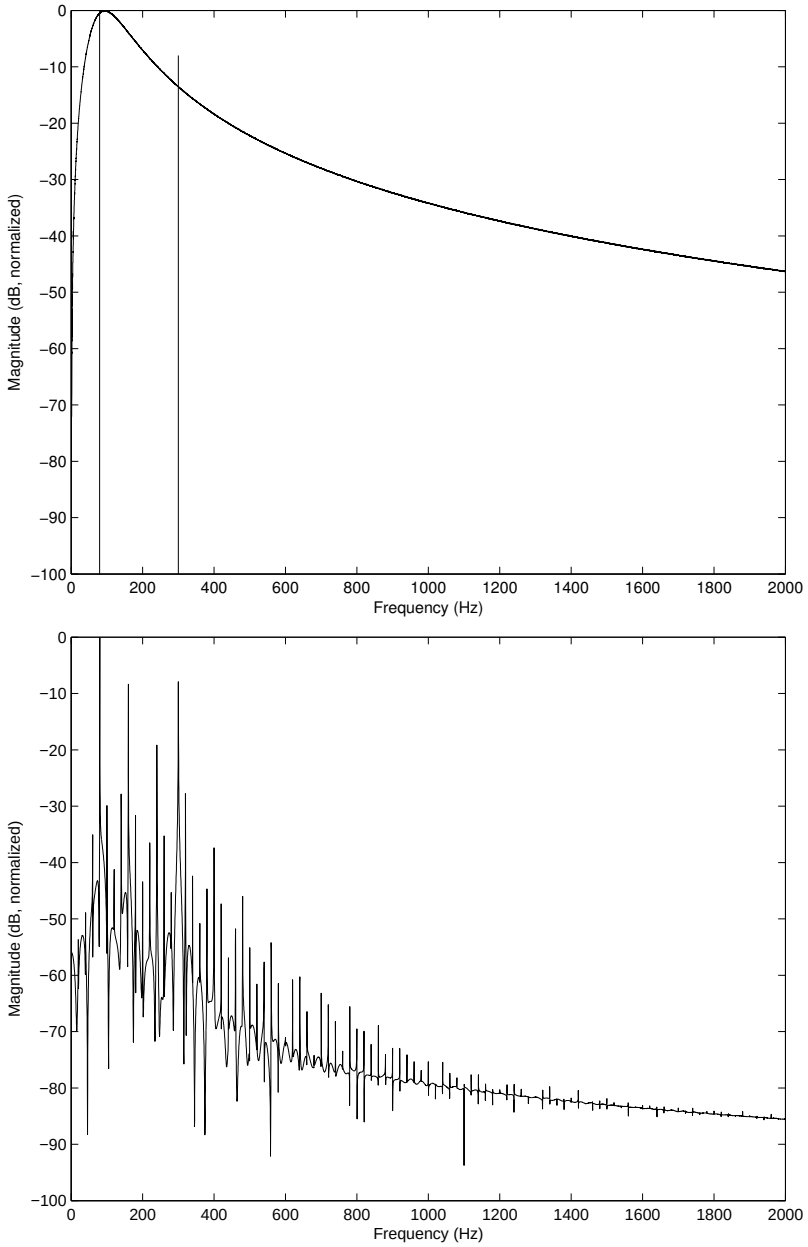


Figure 3: Input spectrum with pre-filter response and output spectrum, 80 + 300 Hz

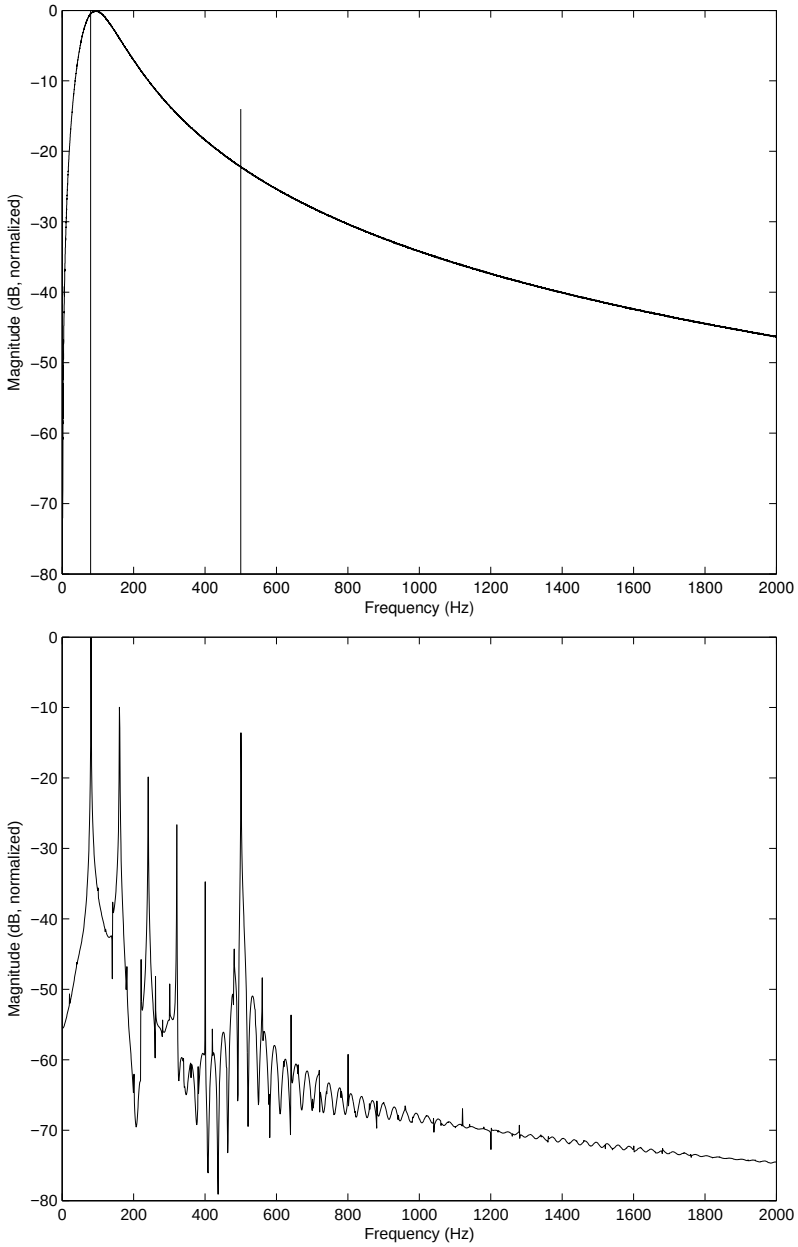


Figure 4: Input spectrum with pre-filter response and output spectrum, 80 + 500 Hz

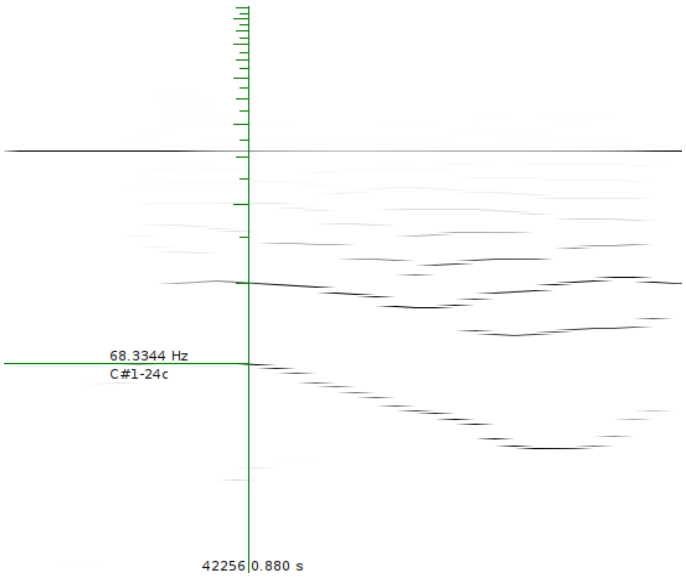


Figure 5: Proper tracking of chirp

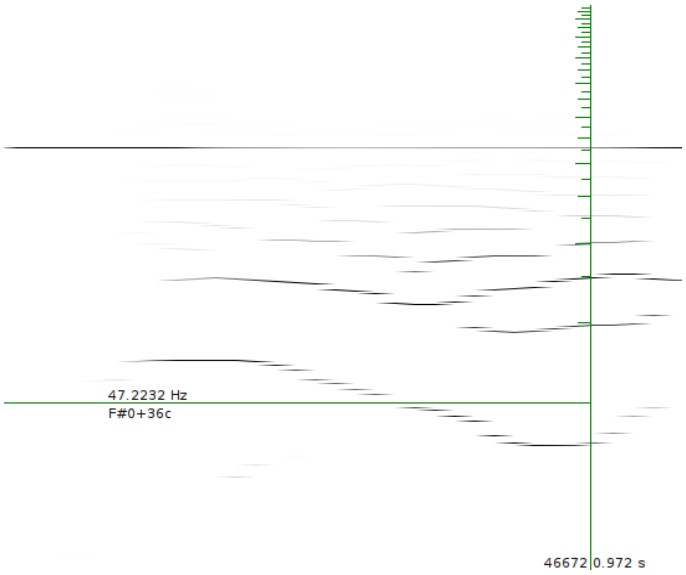


Figure 6: Delayed tracking of chirp

# 5 Conclusions

Implementing a psycho acoustic bass extension plugin using an integrate-and-dump algorithm and biquad filters proved to be a rather simple task. The resulting LV2 plugin showed varying levels of performance. The perceived bass levels in harmonically rich material were easily boosted, while material with less overtones, often synthesized, was harder to keep sounding natural.

Measurements showed that the plugin did what it was supposed to do in theory with relatively steady signals. Signals with a rapidly changing fundamental frequency were however not tracked properly, causing inharmonic content in the output. The fundamental reason for this remained unclear.

The non-linear element in the plugin worked well. Its main accomplishment was the buffered frequency dependent scaling method, while other parts of the plugin used rather standard DSP procedures. This element could be used as-is in future work for creating harmonically rich signals that track the fundamental frequency of a signal. It could also, with small modifications, be used to create an output some octaves higher or lower than the fundamental frequency of its input.

The only parts of the algorithm which were not specifically designed for this use, were the filters. Using linear phase filters was suggested in [2]. However, sufficient filtering at low frequencies was evaluated not to be efficient enough using FIR filters, and no implementation for *linear phase IIR filters* (also suggested in [2]) was found. Using filters designed especially for this algorithm, could make it perform better. Some improvements could also be made by compensating the group delay of the filters in the direct signal delay line. A fractional delay could provide even more phase coherency.

All in all, the project was successful. The range of source material and system cutoff frequencies was not quite as large as I personally expected. On the other hand, the results with harmonically rich content were better than expected. Because of its limited use, the plugin is better suited as a tool used in creative music production, rather than used as a general purpose audio enhancer.

## References


[1] Rossing, Moore & Wheeler. *The Science of Sound*, chapter 7.4. Addison Wesley, 2002.

[2] R.M. Aarts, Erik Larsen and O. Ouweltjes. A unified approach to low- and high-frequency bandwidth extension. In *AES Convention Paper 5921*, October 2003.

[3] LV2 Audio Plugin Standard. http://lv2plug.in.

[4] Calf audio plugin pack. http://calf.sourceforge.net.

[5] Robert Bristow-Johnson. Cookbook formulae for audio EQ biquad filter coefficients. http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt.

[6] Sonic Visualiser - An application for viewing and analysing the contents of music audio files. Developed at the Centre for Digital Music, Queen Mary, University of London. http://www.sonicvisualiser.org.

# Appendices

## A Realized schedule (excluding preparing this report)

| Task | date | hours |
|---|---|---|
| preliminary planning and research | week 45 | 7 |
| writing plan | 17-19.11. | 3 |
| integrate-and-dump mathematical analysis | 21.11. | 4 |
| integrate-and-dump numerical analysis | 25.11. | 2 |
| Matlab implementation and listening tests | 25.11. | 5 |
| LV2 Preparations | 30.11. | 2 |
| first working LV2 implementation | 1.12. | 2 |
| frequency tracking | 6.12. | 2 |
| buffered frequency tracking and amplitude adjustment | 7.12 | 4 |
| LV2 latency reporting | 7.12. | 1 |
| ringbuffer separation, research on filters, biquad implementation | 4.1. | 6 |
| parametrization of filter coefficients | 25.1. | 3 |
| total | 7.11.–25.1. | 41 |

# B Relevant source code

## B.1 psycho_bass.h

```
#ifndef PSYCHO_BASS_H
#define PSYCHO_BASS_H

#include <lv2plugin.hpp>

#include "types.h"
#include "ringbuffer.h"
#include "integrate_and_dump.h"

#include "calf/biquad.h"

using namespace LV2;

class PsychoBass : public Plugin<PsychoBass> {
  public:

        PsychoBass (double rate);
        void run (nframes_t nframes);

  private:

        class Filter
        {
          public:
                enum Type { LP, BP, HP };

                Filter (Type type, float srate);
                void update (float _freq, float _q);
                float process (float val) { return filter.process_d1 (val); }
                void sanitize () { filter.sanitize_d1(); }

          private:
                bool equals (float a, float b) { return (std::fabs(a - b) < 0.01); }

                Type          type;
                float         freq;
                float         q;
                float         rate;
                dsp::biquad<> filter;
        };


        IntegrateAndDump iad;

        // Filters

        inline void update_pre_filters ();
        inline void run_pre_filtering (float * in, float * out, nframes_t nframes);
        std::vector<Filter> pre_filters;

        inline void update_post_filters ();
        inline void run_post_filtering (float * in, float * out, nframes_t nframes);
        std::vector<Filter> post_filters;

        // Delay handling + mixing

        inline void mix_with_direct (float * processed, float * out, nframes_t nframes);
        Ringbuffer<float> delay_line;
        std::vector<float> orig;
};

#endif
```

## B.2 psycho_bass.cc

```
#include "psycho_bass.h"

#include <iostream>
#include <cstring>

#include "calf/primitives.h"

PsychoBass::PsychoBass (double rate) :
  Plugin<PsychoBass> (13),
  iad (static_cast<nframes_t> (rate)),
  delay_line (iad.get_latency())
{
        // FIXME
        orig.resize (1024);

        pre_filters.push_back (Filter (Filter::BP, rate));
        pre_filters.push_back (Filter (Filter::BP, rate));

        post_filters.push_back (Filter (Filter::HP, rate));
        post_filters.push_back (Filter (Filter::LP, rate));
        post_filters.push_back (Filter (Filter::LP, rate));
}

void
PsychoBass::run(nframes_t nframes)
{
        // FIXME
        for (nframes_t i = 0; i < nframes; ++i) {
                orig[i] = p(0)[i];
        }

        // Update filters
        update_pre_filters();
        update_post_filters();

        // Run pre filtering into output buffer
        run_pre_filtering (p(0), p(1), nframes);

        // Run integrate and dump in-place
        iad.run (p(1), p(1), nframes);

        // Run post filtering in-place
        run_post_filtering (p(1), p(1), nframes);

        // Mix direct with processed (compensating latency)
        mix_with_direct (p(1), p(1), nframes);

        // latency
        *p(2) = iad.get_latency();
}

PsychoBass::Filter::Filter (Type type, float srate) :
  type (type),
  freq (0),
  q(0),
  rate (srate)
{ }

void
PsychoBass::Filter::update (float _freq, float _q)
{
        if (_freq == freq && _q == q) {
                return; // Nothing changed
        }

        freq = _freq;
```

```
        q = _q;

        switch (type) {
        case LP:
                filter.set_lp_rbj (freq, q, rate);
                break;
        case BP:
                filter.set_bp_rbj (freq, q, rate);
                break;
        case HP:
                filter.set_hp_rbj (freq, q, rate);
                break;
        }
}

void
PsychoBass::update_pre_filters ()
{
        float cutoff_freq = *p(5);
        float bandwidth = *p(6);
        float filter_freq = 0;
        float filter_q = 0;

        filter_q = sqrt(pow(2, bandwidth)) / (pow(2, bandwidth) - 1);
        filter_freq = ((cutoff_freq / pow(2, bandwidth)) + cutoff_freq) / 2;

        for (std::vector<Filter>::iterator it = pre_filters.begin(); it != pre_filters.end(); ++it) {
                it->update (filter_freq, filter_q);
        }
}

void
PsychoBass::run_pre_filtering (float * in, float * out, nframes_t nframes)
{
        for (nframes_t i = 0; i < nframes; ++i) {
                out[i] = pre_filters[1].process (pre_filters[0].process (in[i]));
        }

        pre_filters[0].sanitize();
        pre_filters[1].sanitize();
}

void
PsychoBass::update_post_filters ()
{
        std::vector<Filter>::iterator it;
        unsigned int i;

        for (it = post_filters.begin(), i = 7; it != post_filters.end(); ++it, i += 2) {
                it->update (*p(i), *p(i + 1));
        }
}

void
PsychoBass::run_post_filtering (float * in, float * out, nframes_t nframes)
{
        for (nframes_t i = 0; i < nframes; ++i) {
                out[i] = post_filters[2].process (
                          post_filters[1].process (
                          post_filters[0].process (in[i])));
        }

        post_filters[0].sanitize();
        post_filters[1].sanitize();
        post_filters[2].sanitize();
}

void
PsychoBass::mix_with_direct (float * processed, float * out, nframes_t nframes)
```

```
{
        float d_vol = *p(3);
        d_vol = dsp::clip01 (d_vol);

        float p_vol = *p(4);
        p_vol = dsp::clip01 (p_vol);

        float orig_compensated;
        for (nframes_t i = 0; i < nframes; ++i) {
                delay_line.run (orig[i], orig_compensated);
                out[i] = d_vol * orig_compensated + p_vol * 0.5 * processed[i];
        }
}

static int _ = PsychoBass::register_class("http://lv2.beatwaves.net/psycho_bass");#include "psycho_bass.h"

#include <iostream>
#include <cstring>

#include "calf/primitives.h"

PsychoBass::PsychoBass (double rate) :
  Plugin<PsychoBass> (13),
  iad (static_cast<nframes_t> (rate)),
  delay_line (iad.get_latency())
{
        // FIXME
        orig.resize (1024);

        pre_filters.push_back (Filter (Filter::BP, rate));
        pre_filters.push_back (Filter (Filter::BP, rate));

        post_filters.push_back (Filter (Filter::HP, rate));
        post_filters.push_back (Filter (Filter::LP, rate));
        post_filters.push_back (Filter (Filter::LP, rate));
}

void
PsychoBass::run(nframes_t nframes)
{
        // FIXME
        for (nframes_t i = 0; i < nframes; ++i) {
                orig[i] = p(0)[i];
        }

        // Update filters
        update_pre_filters();
        update_post_filters();

        // Run pre filtering into output buffer
        run_pre_filtering (p(0), p(1), nframes);

        // Run integrate and dump in-place
        iad.run (p(1), p(1), nframes);

        // Run post filtering in-place
        run_post_filtering (p(1), p(1), nframes);

        // Mix direct with processed (compensating latency)
        mix_with_direct (p(1), p(1), nframes);

        // latency
        *p(2) = iad.get_latency();
}

PsychoBass::Filter::Filter (Type type, float srate) :
  type (type),
  freq (0),
  q(0),
```

```
  rate (srate)
{ }

void
PsychoBass::Filter::update (float _freq, float _q)
{
        if (_freq == freq && _q == q) {
                return; // Nothing changed
        }

        freq = _freq;
        q = _q;

        switch (type) {
        case LP:
                filter.set_lp_rbj (freq, q, rate);
                break;
        case BP:
                filter.set_bp_rbj (freq, q, rate);
                break;
        case HP:
                filter.set_hp_rbj (freq, q, rate);
                break;
        }
}

void
PsychoBass::update_pre_filters ()
{
        float cutoff_freq = *p(5);
        float bandwidth = *p(6);
        float filter_freq = 0;
        float filter_q = 0;

        filter_q = sqrt(pow(2, bandwidth)) / (pow(2, bandwidth) - 1);
        filter_freq = ((cutoff_freq / pow(2, bandwidth)) + cutoff_freq) / 2;

        for (std::vector<Filter>::iterator it = pre_filters.begin(); it != pre_filters.end(); ++it) {
                it->update (filter_freq, filter_q);
        }
}

void
PsychoBass::run_pre_filtering (float * in, float * out, nframes_t nframes)
{
        for (nframes_t i = 0; i < nframes; ++i) {
                out[i] = pre_filters[1].process (pre_filters[0].process (in[i]));
        }

        pre_filters[0].sanitize();
        pre_filters[1].sanitize();
}

void
PsychoBass::update_post_filters ()
{
        std::vector<Filter>::iterator it;
        unsigned int i;

        for (it = post_filters.begin(), i = 7; it != post_filters.end(); ++it, i += 2) {
                it->update (*p(i), *p(i + 1));
        }
}

void
PsychoBass::run_post_filtering (float * in, float * out, nframes_t nframes)
{
        for (nframes_t i = 0; i < nframes; ++i) {
                out[i] = post_filters[2].process (
```

```
                        post_filters[1].process (
                        post_filters[0].process (in[i])));
        }

        post_filters[0].sanitize();
        post_filters[1].sanitize();
        post_filters[2].sanitize();
}

void
PsychoBass::mix_with_direct (float * processed, float * out, nframes_t nframes)
{
        float d_vol = *p(3);
        d_vol = dsp::clip01 (d_vol);

        float p_vol = *p(4);
        p_vol = dsp::clip01 (p_vol);

        float orig_compensated;
        for (nframes_t i = 0; i < nframes; ++i) {
                delay_line.run (orig[i], orig_compensated);
                out[i] = d_vol * orig_compensated + p_vol * 30.0 * processed[i];
        }
}

static int _ = PsychoBass::register_class("http://lv2.beatwaves.net/psycho_bass");
```

## B.3 integrate_and_dump.h

```
#ifndef INTEGRATE_AND_DUMP_H
#define INTEGRATE_AND_DUMP_H

#include "types.h"
#include "ringbuffer.h"

class IntegrateAndDump {
  public:
        IntegrateAndDump (nframes_t samplerate);
        void run (float * in_port, float * out_port, nframes_t nframes);
        float get_latency () { return latency; }

  private:

        float out_val;

        /* frequency counter */

        inline void handle_zero_crossings (float in);

        float last_in;
        nframes_t reset_timer;
        bool freq_below_cutoff;

        /* General stuff */

        nframes_t samplerate;
        nframes_t latency;
        static const int cutoff_freq = 16;

        /* Ringbuffer */

        Ringbuffer<float> buffer;
};

#endif
```

## B.4 integrate_and_dump.cc

```
#include "integrate_and_dump.h"

#include <cassert>
#include <cmath>

#include <iostream>

IntegrateAndDump::IntegrateAndDump (nframes_t samplerate) :
  out_val (0),
  last_in (0), reset_timer (0), freq_below_cutoff (false),
  samplerate (samplerate),
  latency (samplerate / 2 / cutoff_freq),
  buffer (latency)
{
}

void
IntegrateAndDump::run (float * in_port, float * out_port, nframes_t nframes)
{
        for (nframes_t i = 0; i < nframes; ++i) {
                handle_zero_crossings (in_port[i]);
                out_val += std::fabs (in_port[i]);
                buffer.run (freq_below_cutoff ? 0.0 : out_val, out_port[i]);
        }
}

void
IntegrateAndDump::handle_zero_crossings (float in)
{
        if (last_in > 0 && in < 0) { // zero crossing

                freq_below_cutoff = false;

                // calculate relative angular frequency based on last reset
                float freq = (float) reset_timer / (samplerate / 2);
                buffer.apply_gain (50.0 * pow (freq, 2), reset_timer);

                // reset summation and timer
                out_val = 0;
                reset_timer = 0;

        } else {

                if (reset_timer == latency) {
                        freq_below_cutoff = true;
                        buffer.reset ();
                }

                if (freq_below_cutoff) {
                        reset_timer = 0;
                } else {
                        reset_timer++;
                }
        }

        last_in = in;
}
#include "integrate_and_dump.h"

#include <cassert>
#include <cmath>

#include <iostream>

IntegrateAndDump::IntegrateAndDump (nframes_t samplerate) :
  out_val (0),
```

```
  last_in (0), reset_timer (0), freq_below_cutoff (false),
  samplerate (samplerate),
  latency (samplerate / 2 / cutoff_freq),
  buffer (latency)
{
}

void
IntegrateAndDump::run (float * in_port, float * out_port, nframes_t nframes)
{
        for (nframes_t i = 0; i < nframes; ++i) {
                handle_zero_crossings (in_port[i]);
                out_val += std::fabs (in_port[i]);
                buffer.run (freq_below_cutoff ? 0.0 : out_val, out_port[i]);
        }
}

void
IntegrateAndDump::handle_zero_crossings (float in)
{
        if (last_in > 0 && in < 0) { // zero crossing

                freq_below_cutoff = false;

                // calculate relative angular frequency based on last reset
                float freq = (float) reset_timer / (samplerate / 2);
                buffer.apply_gain (pow (freq, 2), reset_timer);

                // reset summation and timer
                out_val = 0;
                reset_timer = 0;

        } else {

                if (reset_timer == latency) {
                        freq_below_cutoff = true;
                        buffer.reset ();
                }

                if (freq_below_cutoff) {
                        reset_timer = 0;
                } else {
                        reset_timer++;
                }
        }

        last_in = in;
}
```

## B.5 ringbuffer.h

```
#ifndef RINGBUFFER_H
#define RINGBUFFER_H

#include "types.h"
#include <vector>
#include <cassert>

template <typename T>
class Ringbuffer
{
  public:
        Ringbuffer (nframes_t size) :
          buffer (size), pos (0), size (size)
        {
                assert (size > 0);
        }
```

```
        void run (float in, float & out)
        {
                out = buffer[pos];
                buffer[pos] = in;

                pos = (pos + 1) % size;
        }

        // Apply gain to nframes last frames
        void apply_gain (float gain, nframes_t nframes)
        {
                nframes_t i = pos;

                while (nframes > 0) {

                        if (i == 0) {
                                i = size - 1;
                        } else {
                                i--;
                        }

                        buffer[i] *= gain;

                        nframes--;
                }
        }

        void reset ()
        {
                apply_gain (0.0, size);
        }

  private:
        std::vector<T> buffer;
        nframes_t      pos;
        nframes_t      size;

};

#endif
```