# Reverbtuner - Automatic adjustment of LV2 Reverb plugins

Sakari Bergen

December 13, 2010

**Abstract**

Convolution reverbs are a powerful way for achieving reverberation effects similar to actual physical environments. They are, however computationally inefficient and badly adjustable. Procedural reverbs on the other hand are more efficient and adjustable. To achieve sounds similar to convolution reverbs with procedural reverbs, it is possible to automatically adjust their parameters. The parameter space of procedural reverbs is very large, and such automatic adjustment requires efficient optimization techniques. An implementation for doing this adjustment using genetic evolutionary optimization and particle swarm optimization is presented.

# Contents

**1 Introduction** **4**
1.1 Background . . . . . 4
1.2 Motivation . . . . . 4
1.3 Overview of functionality . . . . . 4

**2 Methods** **5**
2.1 Evaluation . . . . . 5
2.2 Optimization . . . . . 5
2.2.1 Evolutionary optimizer . . . . . 5
2.2.2 Particle swarm optimizer . . . . . 6

**3 Implementation** **6**
3.1 Data abstraction . . . . . 6
3.2 Evaluation . . . . . 7
3.3 Optimization . . . . . 7
3.4 GUI . . . . . 7

**4 Results** **7**

**A Test runs** **10**

**B Installation and running notes** **11**

# 1 Introduction

The original idea and motivation for this project was from an AES Convention paper titled *Automatic Adjustment of Off-the-Shelf Reverberation Effects*[1].

## 1.1 Background

Any LTI (linear time-invariant) signal processing system can be defined with its impulse response. This means that all LTI systems can also be implemented via convolution. As processing power in general purpose computers has increased, using convolution for modelling audio systems has become more and more popular. Reverbs (reverberation effects) and speaker cabinet responses are among the most popular uses for convolution.

Convolution reverbs are usually created by recording the impulse response of the desired environment by using e.g. a gunshot as the excitement signal. Some post-processing is then applied to the recorded response, and the resulting impulse response is used to create a reverb by convolution. In theory, the resulting reverb is equivalent to the reverb present in the modelled environment.

## 1.2 Motivation

Impulse responses used in convolution reverbs are very long (several seconds, hundreds of thousands of samples), which means that the convolution involved is a very computationally intense operation. Most often similar results can be obtained with some other suitable method of processing. Where convolution reverbs are computationally intense, and have little or no adjustability, most procedural reverbs are not only more effective, but also more adjustable.

A versatile procedural reverb will have many adjustable parameters, anything from ten to a few dozen being usual. This makes the parameter space very large and hard to adjust manually. If one would want to have the sound of a convolution reverb with the efficiency and versatility of a procedural reverb, the parameters of the procedural reverb would need to be carefully adjusted, being a very tedious and time consuming process. Using an automated process for matching the procedural reverb's parameters to a convolution reverb would make the task easy.

## 1.3 Overview of functionality

As convolution reverbs are based on the impulse response of a given environment, producing comparable data from a procedural reverb is as simple as measuring its impulse response. This impulse response can be then compared to the target impulse response, and the fitness of the used parameter

set evaluated. Parameters are optimized by evaluating many sets at a time, and performing optimization techniques which create a new set of data. This data is then evaluated and optimized repeatedly, until a predefined number of optimization rounds is reached.

# 2 Methods

## 2.1 Evaluation

The impulse response of the procedural reverb is evaluated using its mel-frequency cepstrum (MFC). An MFC consists of mel-frequency cepstral coefficients (MFCCs), which describe the signal at different frequencies using real values. Each coefficient is calculated using the mel-scale, which models the human auditory system. The impulse response is split into several overlapping regions, and the MFC is calculated for each separately. As a result, we have a series of MFCs, describing the signal frequency content over time.

To calculate the fitness function for the impulse response, each MFC is compared to the target responses MFC. The MFCs are compared by calculating their euclidean distance, treating each MFC as a real vector. Next the differences of each MFC are summed, and finally scaled to indicate their fitness as a percentage of the difference between the target response and the excitement impulse.

## 2.2 Optimization

Two optimization methods are used: genetic evolutionary optimization and particle swarm optimization. Only one of these is used at a time. Both optimizers use a set of parameter sets, later referred to as the population.

### 2.2.1 Evolutionary optimizer

The evolutionary optimizer works by evaluating parameter sets, selecting parents and reproducing:

1. Initialize an initial population with random values
2. Evaluate each parameter set in the population
3. Store the best result
4. Select a bunch from the population with the best fitness
5. Select a bunch from the population by random
6. Reproduce from the selected parameter sets

7. Repeat from stage 2 until the desired number of round has been completed

The reproduction stage (6) takes two or more parents by random, and generates a new parameter set from it. It consists of the following stages:

1. Select each parameter value from a random parent (cross over)
2. Continue to mutations with a given probability, otherwise the child is ready
3. For each parameter, replace it with a random value (mutate) with a given probability

### 2.2.2 Particle swarm optimizer

The particle swarm optimizer was implemented, but not much effort was put into it. The implementation roughly follows the guidelines in an article titled *Limiting the velocity in particle swarm optimization using a geometric series* [2].

# 3 Implementation

The implementation is done in C++. The back end is completely independent from the GUI, and is built as a shared library. The following libraries are used:

- **Boost** General utilities, threading, random number generator[3]
- **SLV2** A library for hosting LV2[4] plugins[5]
- **Aubio** An audio analysis library, used for MFCC evaluation[6]
- **libsndfile** Reading sound files[7]
- **Gtkmm** GUI[8]

## 3.1 Data abstraction

The most central data structures are related to plugin parameters. Each parameter has a minimum, maximum and default value in addition to a type and name. The parameters are owned by a parameter set, which models all the parameters of a given plugin. Actual evaluation uses parameter value sets, which are linked to the corresponding parameters set, but only include the values themselves as data. This makes the evaluation of large sets of data efficient regarding both total memory usage and cache hits/misses.

### 3.2 Evaluation

The MFCC evaluation uses the Aubio library. The parameters for the evaluation are:

- frequency bands: 40
- buffer size: 2048 samples
- hop size: 1024 samples

Evaluations are run via schedulers, which take a set of parameters to evaluate, and return when all of them are evaluated. As most of the processing time is spent in evaluation, the implementation includes single threaded and multi threaded evaluators. The multi threaded evaluator is especially suited for multi core CPUs, which can run several evaluations concurrently, giving an almost linear speed up compared to single threaded evaluation. Memory allocation is also done efficiently, allocating the required memory for evaluation only once.

### 3.3 Optimization

In addition to random number generators, the optimization implementation mostly uses the already described utilities. The actual algorithms were rather straight-forward to implement.

### 3.4 GUI

A simple graphical user interface for selecting both the impulse and target file, and showing progress was made. Parameters can not be adjusted from the GUI, but only by changing the source code.

## 4 Results

Considering the large parameter space of the problem, the program works rather well. The programs performance was tested by feeding it the impulse response of the same plugin it was adjusting. Doing this optimization, the program was able to achieve over 95% fitness in a rather short time. However, looking at the resulting parameter values, it was noted that adjusting some parameters such as *room size* and *reverb time*, caused similar effects. Often one of them was found to compensate for the other, resulting in the optimization ending in slightly different values for these kinds of parameter pairs.

Unfortunately there are only a few versatile LV2 plugins around. The programs performance was tested by listening to results obtained by using the Calf Reverb[9], and GVerb[10] plugins. These plugins were adjusted to

match some impulse responses which came with Jconv[11]. These plugins didn't produce very impressing results on their own, but often mixing the signals from these two plugins produced good results. It might have been a good idea to use LADSPA[12] instead of LV2 at this stage, to have more plugins to test.

Tuning the parameters of the optimizers caused clear effects, but no clear conclusions could be made from the tests performed. Tuning the parameters of optimizers is an optimization task of it's own, and out of the scope of this project.

## References

[1] Sebastian Heise, Michael Hlatky, and Jörn Loviscach. Automatic adjustment of off-the-shelf reverberation effects. In *Audio Engineering Society Convention Paper*, number 7758. 2009.

[2] Julio Barrera and Carlos A. Coello Coello. Limiting the velocity in particle swarm optimization using a geometric series. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1739–1740, New York, NY, USA, 2009. ACM.

[3] Boost c++ libraries. `http://www.boost.org/`.

[4] Lv2, successor of ladspa. `http://lv2plug.in/`.

[5] Slv2, a library to make the use of lv2 plugins as simple as possible for applications. `http://drobilla.net/software/slv2`.

[6] Aubio, a library for audio labelling. `http://aubio.org/`.

[7] Libsndfile, a c library for reading and writing files containing sampled sound through one standard library interface. `http://www.mega-nerd.com/libsndfile/`.

[8] gtkmm - the c++ interface to gtk+. `http://www.gtkmm.org/`.

[9] Calf plugins, including calf reverb. `http://calf.sourceforge.net/`.

[10] Swh plugins, including gverb. `http://plugin.org.uk/`.

[11] Jconv, a convolution engine. `http://www.kokkinizita.net/linuxaudio/`.

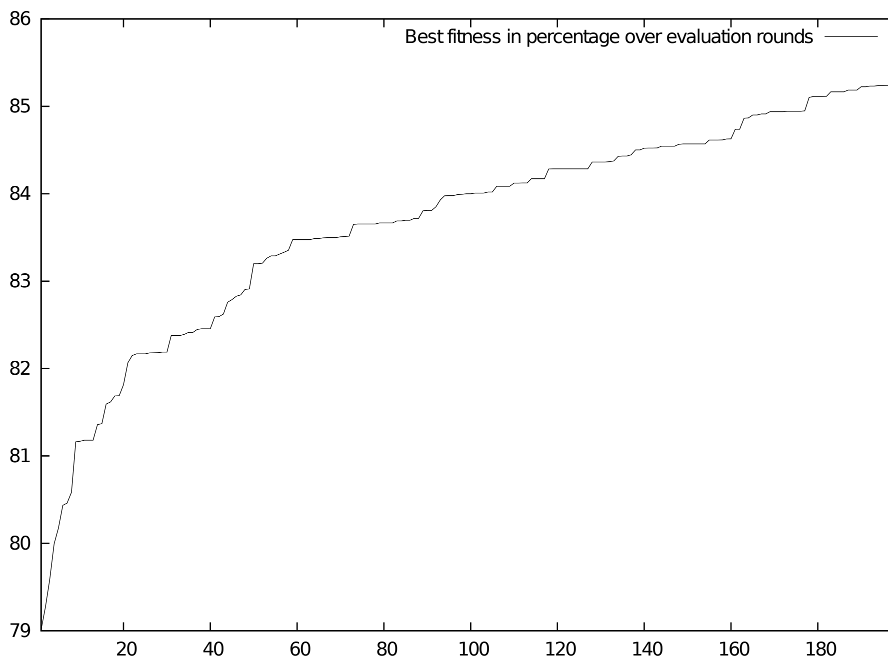
[12] Ladspa, linux audio developer's simple plugin api. `http://www.ladspa.org/`.

# Appendices

## A Test runs

The typical performance of evolutionary optimizer is shown in the following plot. The optimizer was run four times with the following parameters, using an impulse reverb recorded in a chapel, and the Calf Reverb LV2 plugin. The plot shows the average of all runs.

- Rounds: 200
- Population size: 50
- 15 best settings selected for parents
- 7 random settings selected for parents
- 2 parents
- Probability of mutating a parameter set: 0.2
- Probability of mutating an individual parameter in the set set: 0.1




## B Installation and running notes

The program has only been tested on Linux, but should work on any *nix system. To build the project you need to have the libraries listed in Section 3

installed. For Aubio, the Git version is required (`git clone git://git.aubio.org/git/aubio/`). It is recommended to install aubio to `/usr/local/aubio`, which makes it possible to use the provided script for setting relevant environment variables. The command

```
source setup_env
```

will set the necessary environment variables for building and running the program. Obviously you will also need some LV2 reverb plugins and impulse responses at hand. These can be rather easily obtained via the links in the *References* section.

The program uses *waf* for building. Building with waf is as simple as issuing the following commands:

```
./waf configure
./waf
```

After the program is built, it can be run by issuing the command:

```
build/default/reverbtuner-gui
```

Using the GUI should be rather straight forward. Do note, however, that it is recommended to quit the program before starting a new evaluation, to avoid possible problems with old data being left over in memory and not updated.

To change the evaluator type, the two `#ifdef`s in `reverbtuner/runner.h` and `src/runner.cc` shoul be changed. Optimization parameters can be adjusted from the constructors in `src/evolutionary_optimizer.cc` and `src/particle_swarm_optimizer.cc`.