NTNU – Trondheim
Norwegian University of
Science and Technology

Intelligent Guitar Processor

An Evolutionary Prototype

Jonas Gutvik Korssjøen

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Asbjørn Thomassen, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Investigate whether Evolutionary Computation can be utilized to simulate guitar sounds. Study the open-source library Csound and its capabilities to apply guitar effects in the context of guitar sound simulation. Implement a prototype to integrate these aspects into one functioning system.

Assignment given: 16. January 2012
Supervisor: Asbjørn Thomassen

Page intentionally left blank

Abstract

In this thesis we propose a system with capabilities of simulating guitar sounds with evolutionary methods. Every guitarist has probably at some point had a great desire to be able to recreate the signature sound of their favorite guitar player, that be Mark Knopfler or Kirk Hammett. However, this is not a trivial task to perform manually, and can prove to be both time consuming as well as expensive, considering the wide range of existing guitar effects.

We implement a somewhat simple prototype with a relatively small number of guitar effects. Considering that the perceived sound of the guitar is affected by the order chain of the effects as well as the effects in itself, a small number of effects is preferred to maintain a reasonably low complexity.

An Evolutionary Algorithm and a Sound Synthesis module (consisting of several scripts applying the effects), employing the open-source library Csound, cooperate to evolve individuals representing guitar sounds. The fast Fourier transform is employed in a spectral comparison between the evolved candidate sounds and the specified target sound. In this comparison the *frequency domain* of the sounds are analyzed and compared to 'grade' the candidates.

Varying results concerning the evolved sounds are presented, identifying which effects the system successfully handles, and which ones it struggles to deal with. However, the overall results are promising, and exact matches are eventually found in the majority of the different target sound experiments.

Page intentionally left blank

Sammendrag

I denne oppgaven presenterer vi et system med evnen til å simulere gitarlyder ved hjelp av evolusjonære metoder. Enhver gitarist har mest sannsynlig en gang i livet hatt et stort ønske om å rekonstruere signaturlyden til hans/hennes favorittgitarist, det være Mark Knopfler eller Kirk Hammett. Men dette er ikke en triviell oppgave å utføre manuelt, og kan vise seg å være både tidkrevende og kostbar, tatt i betraktning den store mengden av gitareffekter som finnes i dag.

Vi implementerer en noe enkel prototype med relativt få gitareffekter. Tatt i betraktning at den endelige oppfatningen av gitarlyden påvirkes av rekkefølgen på effektene i tillegg til effektene i seg selv, er få effekter å foretrekke for å opprettholde en kompleksitet innenfor rimelighetens grenser.

En Evolusjonær Algoritme og en lydsyntesemodul (bestående av flere skript som legger på effektene), som benytter open-source biblioteket Csound, samarbeider om å utvikle individer som representerer gitareffekter. Fouriertransformasjonen benyttes i en spektralsammenligning mellom de utviklede lydene og den spesifikke lyden vi ønsker å oppnå. I denne sammenligningen er frekvensdomenet av lydene analysert og sammenlignet for å gi kandidatene en 'karakter'.

Varierte resultater vedrørende de utviklede lydene presenteres, og med dette identifiserer vi hvilke effekter systemet klarer å håndtere, og hvilke det strever med. Likevel, resultatene i sin helhet er lovende, og eksakte kopier er funnet i de aller fleste av eksperimentene for de ulike gitarlydene.

Page intentionally left blank

Preface

This thesis is a continuation of my specialization project carried out in autumn 2011, a preliminary study to this master's thesis. My main motivation for this project is simply that I am a hobby guitarist myself, and can see the high potential of this idea, in addition to my fascination for Evolutionary Computation in general. This project was carried out in the second semester of my final year at the Norwegian University of Science and Technology (NTNU), from January to June 2012, at the Department of Computer and Information Science (IDI).

First, I would like to thank my supervisor, Asbjørn Thomassen. which has provided guidance and support throughout the project period, both in an academic and social manner.

Credits go to Iain McCurdy for writing the *.csd* file MultiFX.csd, providing very helpful tips in the manners of implementing guitar effects in Csound.

Thanks to Geir Josten Lien, who worked with me on an earlier project, which some of the code in this master's thesis stems from.

I would also show my gratitude to fellow students Stian Pedersen Kvaale and Trond Bøe Engell for daily conversations and advice at both a social and technical level.

Finally, I would like to thank my family for their constant text messages and phone calls of encouragement, motivating me to perform at a higher level, and to finish this thesis in a good manner.

Sincerely,

Jonas Gutvik Korssjøen

Page intentionally left blank

Contents

Chapter 1

Introduction

The term Artificial Intelligence has its roots all the way back to Greek myths (Talos the bronze giant), humanoid automatons and different kinds of "artificial beings" such as Mary Shelley's Frankenstein. Today we think of Artificial Intelligence as a branch of computer science, more specifically machines that are able to perform tasks normally conducted by humans. When Darwinian principles first were used for automated problem solving, terms such as "Genetic Algorithms" and "Evolution Strategies" appeared. Evolutionary Computation has then, since it appeared in the late 1950s, been applied to virtually every imaginable area of optimization, from areas such as antenna and aircraft design, to game playing and market forecasting.

In the music business, the search for new and exciting sounds is perpetual. And in time, the desire to be able to create innovative music and simulate or resemble a specific sound has increased. The latter task of replicating a specific sound can usually be performed by hand, by using the respective instrument's interface, that be a guitar, synthesizer or any other sound source. However, with a wide range of exsisting effect units, it can be troublesome and very time consuming to do this manually. This led to the entry of Artificial Intelligence, more specifically Evolutionary Computation (EC), in this research area. EC has eventually been applied to the tasks of music composition [8] [9] [16], sound creation [12] [10] and sound simulation [13] [14] [2].

James McDermott et al. in [14] say that "sound synthesis is a natural domain in which to apply Evolutionary Computation (EC). The EC concepts of the genome, the phenotype, and the fitness function map naturally to the synthesis concepts of control parameters, output sound, and comparison with a desired sound". There have been several attempts on trying to create and simulate sounds on synthesizers [13] [14] [4] [2] [12], but the same can not be said for guitars. As far as we know, the only research concerning EC and 'guitars' is Janne Riionheimo et al. in [17]. There are similarities between the two cases, with both having an amount of adjustable parameters affecting the output sound. The synthesizer has an amount of preset sounds that can be mixed, and the guitar has separate effect units such as stomp boxes and amplifiers. However, the guitar output is not only affected by which effects that are applied, but the order of the effects is also decisive. This increases the complexity significantly, considering the rising amount of existing effect units.

The idea of an intelligent guitar processor that is able to replicate any given guitar sound is intriguing to us, being guitar players. And considering that the guitar is a very popular, if not the most popular instrument of them all, one would think that such a product would have been greatly appreciated in the music industry. This claim, in addition to the successfull results concerning EC and simulation of synthesizer sounds, is the main motivation for this project.

Can Evolutionary Computation be used to search for guitar effect parameters to resemble a desired target guitar sound?

In this thesis we develop a system that tries to resemble a target guitar sound (to a certain degree) by applying effects to a clip consisting of a clean guitar signal. An Evolutionary Algorithm is implemented to optimize the effect parameters, including the order of the effects. The system is mainly inspired by papers concerned with optimization of parameters in connection with sound in some form (usually synthesizers), as well as papers dealing with matching of evolved sounds against target sounds. To reduce the complexity we have limited the system to four popular guitar effects, and mono sound files, i.e single channeled sound files. Additionally, the audio programming language CSound is selected as a tool to handle the sound processing part of the system.

1.1 Overview of the document

This thesis is structured as follows. Chapter 1 introduces the problem area within the scope of this thesis, as well as the motivation and goal of the project. A brief description of the system is presented along with its predefined limitations. In Chapter 2 we cover important background material used throughout this thesis, which includes common techniques within the problem area and brief descriptions of systems used to solve similar problems. Chapter 3 describes the design of the implemented system as a whole as well as the key components individually. In Chapter 4 we present the results of the system run with various settings, and discuss the results presented. Chapter 5 sums up the project and its results, while discussing the results in relation to the task presented in Chapter 1. Finally we suggest areas of future work.

Page intentionally left blank

Chapter 2

Background

2.1 Guitars and Effects

This section gives an introduction to guitars, effect units and how they both affect the perceived output sound.

Since the first successfull magnetic pickup for a guitar was invented by George Beuschamp in the 1930s, the music industry has produced a huge variety of effect units. The very first guitar effects, however, were built into instruments themselves. In the 1930s, Rickenbacker made a clunky Vibrola Spanish guitar with motorized pulleys that jiggled the bridge to create a vibrato effect. In the 1940s, DeArmond manufactured the world's first standalone effect, a type of tremolo [6].

When effect units started stabilizing in the market, musicians soon realized that they could combine different effects to create their own unique sound. As a result, guitar players started to become famous for their signature sound, and not only their music. Today guitar players still experiment on finding innovative sounds, and have identified *three key factors* affecting the output sound: (1) The choice of effects, (2) the order of the effects, (3) the choice of guitar (and its respective pickups). The choice of amplifier also has an effect on the sound, as the manufacturers tend to 'colour' the sound in their own way, but this is also classified as effects.

It is easy to understand that different types of effects equal different sounds, but understanding that the order of the effects is just as crucial, is perhaps not as trivial. However, when considering that the effects actually alter the signal of the guitar one by one, one should be aware of what the effects actually do with the signal (in fact, the second effect in the chain might double the effect already applied by the first one). There are, however, some "unwritten rules" on how guitar players usually set up their effects after one another, but these are usually just a guideline to novice users. Experienced guitar players tend to play around with both the effects in itself as well as the order of them to discover innovative sounds.

2.1.1 Guitar Effects

To understand the basics of guitar effects, we look at the principles behind the various existing categories.

- *Distortion* effects is created by compressing the peaks of the sound wave and by adding overtones. "Distortion" literally refers to any modification of the sound wave of a signal, and in terms of music amplification this technique is also known as clipping. When an amplifier is pushed to create a signal more powerful than it can handle, the signal "clips" at the maximum capacity of the amplifier (referenced as *threshold* in Figure 2.1), resulting in a sine wave becoming a distorted square-wave-type waveform. This creates a "noisy" sound, but is also characterized as "warm", "dirty" or "fuzzy".

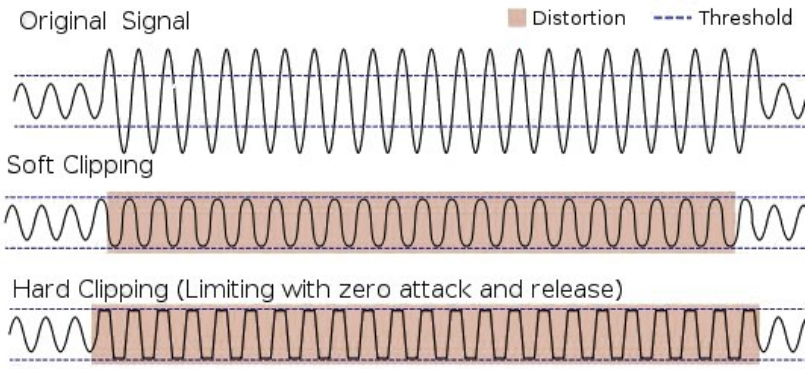

Figure 2.1: A waveform plot showing two types of clipping. Soft clipping can be produced by overdriving the valves in an amplifier. Hard clipping can be produced by overdriving a transistor amplifier.

- *Modulation* effects combine multiple audio signals to create sounds with certain tonal characteristics. The signal is combined with one or several modified or delayed copies of itself, resulting in a somewhat richer tone. Chorus, flanger, phaser, tremolo and vibrato are all examples of modulation effects.

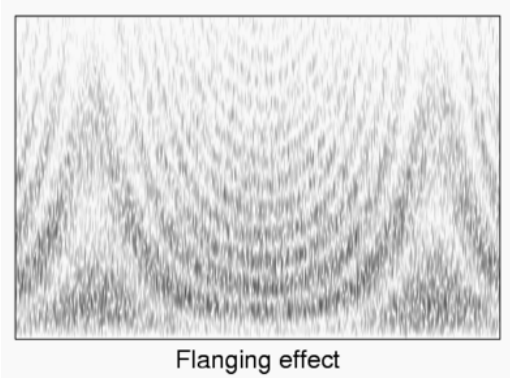

Figure 2.2: Spectogram of the flanging effect, resulting in a comb-like trace.

- *Dynamic* effects, also known as volume and amplitude effects, simply modifies the volume, and were the first effects used by guitarists.

- *Filter* effects alter the frequency content of the signal by strengthening or weakening certain frequencies. The equalizer, talk box and the wah-wah

pedal are all filter effects. These effects usually require live input from the guitar player, specifying which frequencies to alter.

- *Pitch* effects modifies pitch by altering the frequency of a sound wave or by adding harmonies. Harmonizers can be found in synthesizers or electric organs, these combines the altered pitch with the original pitch to create a two or more note harmony.
- *Time-based* effects delay the signal or add echoes, such as the well-known reverb effect. This effect simulates sounds produced in echo chambers or large concert halls (to simulate acoustic spaces) by creating a number of echoes that gradually fade.

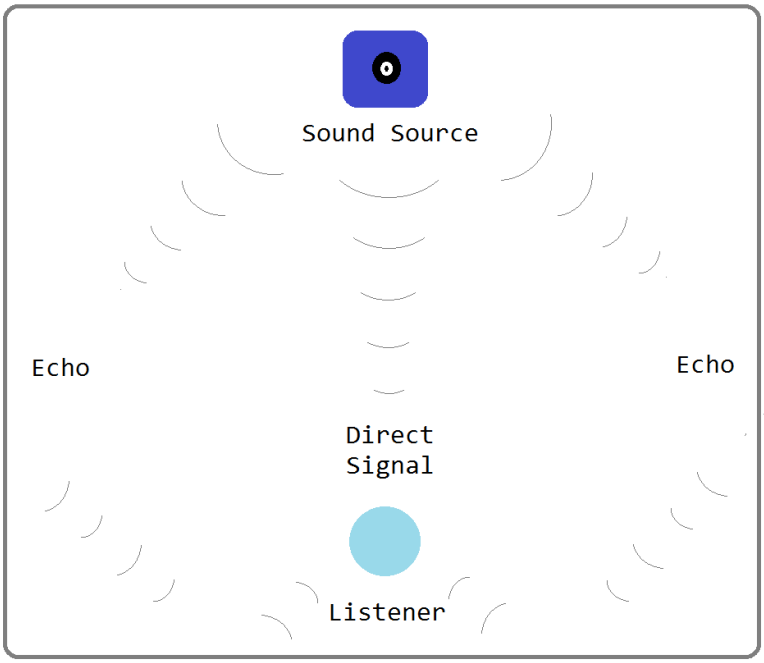

Figure 2.3: A demonstration of how the reverb effect works. Signals are being sent in many directions from the sound source. They eventually reach the listener, but not at the same time, as some are reflected one or several times by obstacles before reaching the listener.

Effect units representing the aforementioned effects typically have one or several knobs to adjust the rate (or degree) of the given effect. Some of these effect units

are depicted in Figure 2.4. These map very well with the concept of *genome* parameters in Evolutionary Computation, as we will look at later in this chapter.

(a) Boss Distortion pedal (b) DigiTech Chorus pedal (c) Boss Reverb pedal

Figure 2.4: Three types of effect units (stompboxes)

Details regarding the effects used in our implementation are further explained in Chapter 3.

2.1.2 Guitars and Pickups

In addition to the effects and the order in which they are applied, the guitar itself, its pickups and electronics have a major impact on the output sound. A wide range of guitars exist today, from old Renaissance and Baroque guitars to axe-looking electrical guitars, and they all have their own characteristics when it comes to sound. However, there are mainly electrical guitars and some types of acoustic guitars that most commonly are used together with effect units. In the following paragraphs we cover the basics of these guitars.

Acoustic guitars can be split into several subgroups, but the most commonly used are classical, steel-string and twelve-string guitars. Sound is shaped by the characteristics of the guitar body's resonant cavity, and this body is hollow on most acoustic guitars. The guitar top is a finely crafted and engineered element made of tonewoods, and is considered by many as the most dominant factor in deter-

mining the sound quality. Acoustic guitars have either nylon or steel strings, but steel strings are usually prefered when working with amplifiers and effects. Such guitars are either equipped with a pickup inside the guitar, or they are recorded through a separate microphone. This aspect is also a decisive factor affecting the sound quality.

Electric guitars have either a solid, semi-hollow or hollow body, and can barely produce sound without amplification. They are equipped with steel strings, and electromagnetic pickups to convert the vibration of these strings into signals, which are fed to an amplifier (usually through a cable). In contrast to the acoustic guitar, that primarily relies on the body to determine the sound quality, the electric guitar heavily relies on the pickups. They are transducers attached to the guitar to "pick up" string vibrations and convert the mechanical energy of the string into electrical energy. Guitar manufacturers tend to have their own preferences when it comes to pickups, which eventually have affected the sound characteristics of their guitars. Two of the most well-known electric guitars are the Fender Stratocaster that generally utilizes three single-coil pickups, and the Gibson Les Paul model that uses humbucker pickups.-

(a) Fender Stratocaster (b) Gibson Les Paul (c) Martin, acoustic guitar

Figure 2.5: Three types of guitars

The guitar players choice of guitar is, however, outside of the scope of this thesis. As to why this is the case, will be further explained in Chapter 3.

2.2 Evolutionary Computation

The concept of evolution has grown to become a general term of how biological populations tend to develop. Since Charles Darwin first introduced the theory of evolution by natural selection, the acceptance of this concept has increased into the different branches of biology. Eventually it reached the field of Artificial Intelligence and computer science, thus the term *Evolutionary Computation* emerged.

Evolutionary Computation (hereinafter EC) deals with automated problem solving based on the principles of evolution and biology. It incorporates the concept of maintaining a population of solutions and evolving them through generations, in contrast to the typical approach of trying to make improvements to a single design. It consists of an iterative process of *selection*, *recombination* and *mutation* based on a problem specific *fitness function*. In our system we implement an Evolutionary Algorithm (hereinafter EA), which is a subset of EC. In this section we look at the different aspects of an EA and how it solves the given problem.

Evolution is a process that does not operate on organisms directly, but on *chromosomes* [1]. An EA encodes chromosomes as *genomes* (also known as *genotypes*), where a genotype represents the chromosomes of an individual. These represent the possible *solutions* of the problem at hand, usually in the simple form of bitvectors or floating point arrays. EAs operate on populations of individuals, and typically the initial population of genomes are randomly generated with a uniform probability distribution. In some cases, though, it might be beneficial to 'guide' the system in the right direction, by limiting the range of the chromosome values assigned to the initial population (by means of some heuristic seeding procedure [1]). These are typically problems where satisfactory results are difficult to achieve with a large search space.

As the initial population is generated, the 'circle of life' begins. In an encoding/decoding process the EA performs a mapping between the chromosomes and candidate solutions, which transforms the genotypes into *phenotypes*. Each individual then receives a *fitness value*: a measure of how good the solution it represents is for the problem being considered. This is an essential component of the EA, to the point that some early (and nowadays discredited) views of EAs considered it as the

unique point of interaction with the problem that is intended to be solved. This interpretation has given rise to several misconceptions, the most important being the equation 'fitness = quality of a solution'. A much more reasonable choice, though, is defining fitness as the number of satisfied clauses in the formula by a certain solution. This introduces a gradation that allows the EA 'climbing' in search of near-optimal solutions [1].

After this 'grading' of the individuals a *selection* process is commenced, and *adults* that will represent this particular generation are chosen through a selection protocol. Three protocols are typically assessed:

- *Full Generational Replacement* - All adults from the previous generation are removed (i.e., die), and all children gain free entrance to the adult pool [3].
- *Over-production* - All previous adults die, but the maximum size of the adult pool is smaller than the number of children. Hence, the children must compete among themselves for the adult spots, so selection pressure is significant [3].
- *Generational Mixing* - The adults from the previous generation do not die, so they and the children compete in a free-for-all for the adult spots in the next generation [3].

At this point the EA typically performs a control of the adults against the target solution and decides whether the criteria to stop are met or not. This tends to be when a certain degree of similarity to the target solution have been reached (when the highest rated fitness has surpassed a specific threshold), or when a maximum number of generations have been reached. Further in the selection process, *parents* are selected from the previously chosen adults. These individuals are selected to produce offsprings for the next generation, and are chosen through selection mechanisms. Here we cover three common mechanisms:

- *Fitness proportionate* - Scales fitnesses so that they sum to 1. Selection is then based on roulette wheel spin. This mechanism does not alter the selective advantages/disadvantages inherent in the original values and thus does not implicitly change the fitness landscape [3].

- *Sigma scaling* - Modifies the selection pressure inherent in the raw fitness values by using the population's fitness variance as a scaling factor. Hence, unless this variance is 0 (in which case all fitnesses scale to expected values of 1.0), the conversion is [3]:

$$ExpVal(i,g) = 1 + \frac{f(i) - f^{-}(g)}{2\sigma(g)} \tag{2.1}$$

  where g is the generation number of the EA, f(i) is the fitness of individual i, $f^{-}(g)$ is the population's fitness average in generation g, and $\sigma(g)$ is the standard deviation of population fitness.

- *Tournament selection* - Random groups of K adults are chosen and their fitnesses compared. With a probability of 1 - ϵ, the best fit of the K is chosen for the next mating, while the choice is random with a probability of ϵ (the parameter ϵ is a user-defined value). This process is reapeted till the desired number of parents are chosen [3].

Following the parent selection, a *reproduction* process is initiated. In this final stage of the cycle (or loop), two (or more) parent individuals are chosen for mating, and their genetic material is combined to produce one or more offspring [9]. This process involves *inheritence* of genetic material from parents to offspring. The genes of the parents are either directly transfered to the offspring (cloned) or combined in a *crossover* operation, providing exploration of the search space. The recombination also involves some form of *mutation* to the newly produced genotypes, which means that a small component of an individual is randomly changed (e.g. one bit in a bitvector). Mutation is applied to maintain *diversity* within the population.

The recombination phase concludes the generation loop, and the loop starts over again. The new offspring are transformed (decoded) to phenotypes and assigned fitness values to be able to compete with the adults from the previous generation.

Figure 2.6 depicts an overview model of the evolutionary cycle.

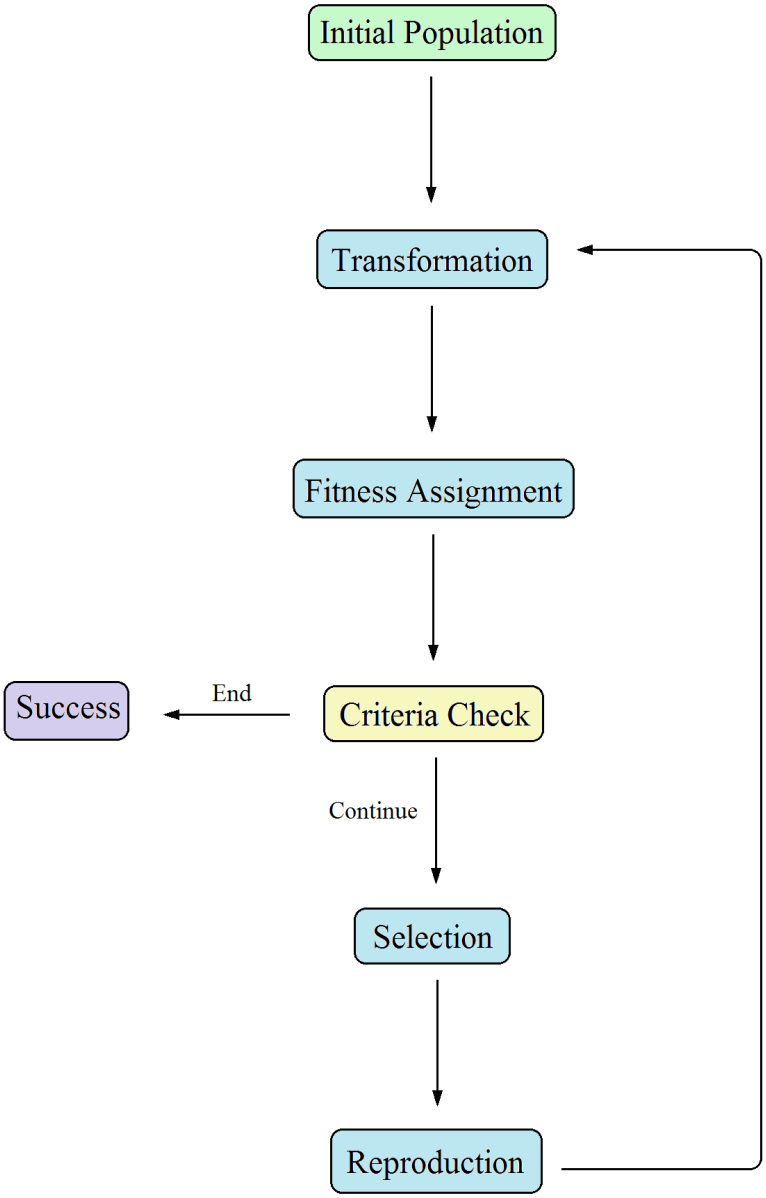

Figure 2.6: An overview of the evolutionary cycle. The 'Transformation' stage involves the genotype-to-phenotype process.

The task of implementing an EA can roughly be done by following the steps depicted in Figure 2.6. However, there are some aspects that are greatly problem dependent and need to be designed based on the specific problem:

- The encoding (representation) of the genotype and the phenotype, as well as the decoding process (geno-to-pheno process) is highly problem dependent.
- The fitness function indicates how good the individual represents the specific problem and needs to fit well with the representation. This is to be able to measure the quality in a satisfactory manner.

2.3 Sound Parameter Representation

The concept of the genome and chromosomes maps very well with the synthesis concept of control parameters [14]. Previous work concerning EC and synthesizer sounds [10] [14] [12] typically deals with a specific number of parameters (in a fixed order) found in the given synthesizers. This means that the genotype only has to consist of a vector that can hold the given amount of parameters (either in the form of binary or floating point values). This simplifies the task of designing the genotype in EAs, in contrast to more complex conditions where the amount and order of the parameters vary.

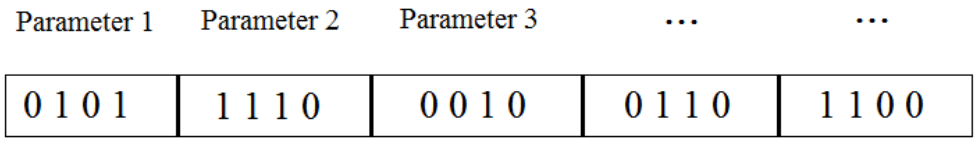

Figure 2.7: The design of a typical genotype representing a synthesizer sound as a bitvector.

The guitar effects described previously in this chapter can all be controled by adjusting their respective *parameters* (or knobs, on their corresponding effect units). These effect parameters can easily be represented by the EAs genotype. However, effects tend to have multiple control parameters. Additionally, when considering

the order of the effects on top of it, the complexity increases significantly. *We will look further into this problem in Chapter 3.*

2.4 Sound Analysis

Since the introduction of the compact disc (CD) in the early 1980s, the digital format has provided increasingly greater storage capacity and the ability to store audio information at an acceptable quality [18]. The music digitizing opened another world to music technicians. This new *representation* created a wealth of opportunities, and have eventually simplified, and yet enhanced the tools to create and modify sound in general.

When stored on computers, sound is typically represented linearly (as bitvectors or floating point arrays). These express the pressure wave-forms as a sequence of symbols. However, this low-level representation is very rarely used in the EC literature in this context [14].

2.4.1 Discrete Fourier Transform

The discrete Fourier Transform (hereinafter DFT) transforms a time-domain signal into the *frequency* domain. It takes a finite sequence of numbers as input, making the DFT ideal for processing information stored in computers (i.e. a discrete sound signal). This is a lossless, invertible transform, and represents the energy present in frequency *bins*, together with their associated phases [14]. The DFT is defined as follows:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi k \frac{n}{N}} \tag{2.2}$$

where a sequence of N numbers is transformed into another sequence of N numbers representing the frequency domain.

2.4.2 Spectral Analysis

The majority of research concerning sound matching using EAs (e.g. McDermott et al. in [14] and Mitchell in [15]) has consistently based their distance functions on spectral comparison of the target and candidate sounds by using the DFT, while Mitchell in [15] also states that it offers an excellent balance between detail and execution speed.

McDermott et al. in [14] point out that the DFT involves a trade-off, describing that increased transform length gives better frequency resolution, which also means that different DFT lengths better characterize different aspects of sound. Based on this, they define a DFT distance function that takes the average over multiple lengths applied to 2x-overlapping *Hann windows*. A Hann window is one of several known *window functions* and is multiplied to the input signal in the time domain before applying the DFT, and is used to remove discontinuity. Many window functions have eventually been defined, and they usually involve some compromise between the width of the resulting peak in the frequency domain, the amplitude accuracy and the rate of decrease of the spectral leakage into other frequency bins [7]. A typical approach has eventually been to calculate the DFT of one or several (sometimes overlapping) windows of the input signal. *For further details concerning window functions and Hann windows, see* [7].

Another approach, is Wun et al. in [19] which calculates the DFT over several *snapshots* of the input signal. This is done to reduce the workload, and is justified by the assumption that their respective synthesis technique will not produce sounds which change very quickly over time [14].

The *regular* formula of spectral distance comparison by the DFT is based on the mean squared error (MSE) metric:

$$Dist = \sqrt{\sum_{i=0}^{Bins} (X_i - T_i)^2} \tag{2.3}$$

where X and T is the DFT of the candidate and target solution, respectively. This can be adjusted to fit a *window* or a *snapshot* solution, by adding a loop on the outside, iterating through the windows or snapshots.

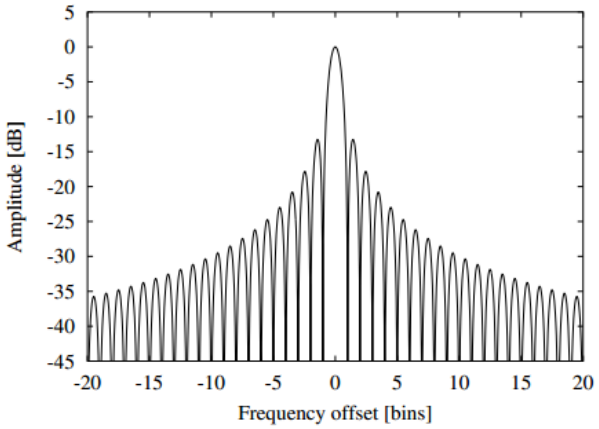

Figure 2.8: The spectral response of a rectangular window, which is equivalent to replacing all but N values of a data sequence by zeros, making it appear as though the waveform suddenly turns on and off. (Figure borrowed from [7])

2.5 Csound

This section describes the basics of Csound as a sound synthesis tool. Since the Csound part plays such a vital role in our system, and we spent a considerably large amount of time understanding and figuring out how to integrate this part with the EA, we thought the reader might benefit from a brief introduction.

Today there exist many available options of sound synthesis software. The open-source library CSound was chosen for this approach because it provides many built-in functions, such as effects, filters and algorithms. CSound is a very powerful system, providing facilities for composition and performance over a wide range of platforms. One of its greatest strenghts is that it is completely modular and extensible by the user. The fact that it has bindings to Python, Java, Lisp, Tcl and C++ in addition to the basic C API, makes it versatile and easy to use across different platforms [11].

A working Csound module consists of a *.orc* (orchestra) file containing "instruments", and a *.sco* (score) file containing "notes". However, these can be unified into a single structured *.csd* file using markup language tags. Csound offers an editor and a complete development environment called *QuteCsound*, and is a nat-

ural choice as it comes with the current Csound installation. It provides access to the Csound reference guide and *opcode* (operational codes to build instruments or patches) overview, thus reducing the learning curve initially required.

Examples of code snippets with corresponding descriptions are presented in Chapter 3.

2.6 Related Work

In this section we look at previously developed systems and research related to the problem in this thesis.

Parallel Evolutionary Optimization of SS Parameters

Bozkurt et al. in [2] developed a system that employs a Genetic Algorithm (hereinafter GA) to optimize sound synthesis parameters. They ultimately want to match parameters of different sound synthesizer topologies to target sounds, and at the same time maintain a reasonable runtime by using a parallelizable evolutionary architecture.

The evolutionary framework is implemented in the SuperCollider (SC) programming language, and optimizes the set of parameters required to approximate a target sound given an arbitrary sound synthesizer created by the user. The GA is implemented very similarily to an EA loop described earlier in this chapter, but they incorporate a server/client solution which allows a single client to control multiple instances of servers via the Open Sound Control (OSC) protocol. The evaluation of the entire population is divided between several servers, which optimizes the workflow of the system.

Their GA applies *tournament selection* to provide a reasonable relationship between convergence rate and selection pressure. The analytical spectral distance metric proposed by Garcia in [5] is implemented, but they exclude the phase information while focusing only on the magnitudes. This metric involves the mean

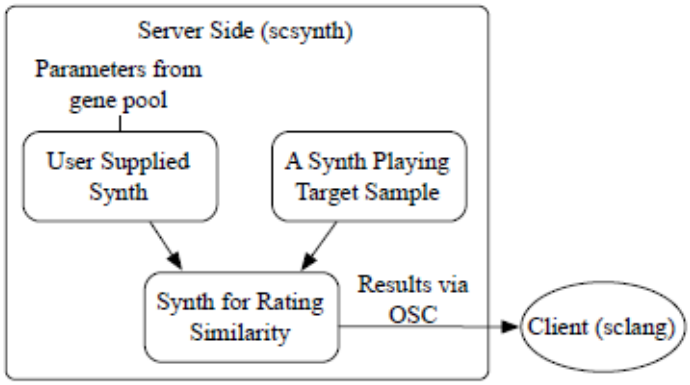

Figure 2.9: The server handles the fitness evaluation and genetic operations. (Figure borrowed from [2])

squared error (MSE) extended with a weight matrix to curtail the errors at spectral regions with more energy.

The paper indicates that this system yields good results when directing the algorithm to a set of possibly related parameter ranges, while greatly decreasing the complexity of the search. However, the primary contribution of this work is the improved convergence time obtained through the parallel architecture.

Plucked String Synthesis Model with GA

Riionheimo et al. in [17] developed a system that employs a GA to estimate control parameters for a plucked string synthesis model. However, the tuning of parameters are performed by a semi-automatic method, requiring hand adjustment with human listening.

They use a synthesis model controled by nine parameters affecting the resulting sound:

Parameter	Control
$f_{0,h}$	Fundamental frequency of the horizontal string model
$f_{0,v}$	Fundamental frequency of the vertical string model
g_h	Loop gain of the horizontal string model
a_h	Frequency-dependent gain of the horizontal string model
g_v	Loop gain of the vertical string model
a_v	Frequency-dependent gain of the vertical string model
m_p	Input mixing coefficient
m_o	Output mixing coefficient
g_c	Coupling gain of the two polarizations

The model involves a fairly regular GA, very much alike the one Bozkurt et al. in [2]. However, they use a selection mechanism where the individuals are ranked according to their fitness, and a user-defined value q denotes the probability of selecting the best individual.

The most intriguing part in this paper, though, is the fitness calculation. Their fitness function uses a comparison of a perceptually transformed spectra of the candidate and target sounds by applying a window function to the regular MSE metric. This window function takes account of psychoacoustic masking effects (by giving less weight to errors likely to be masked by louder parts of the sound) and "the frequency-dependent sensitivity of human hearing" [14]. By doing this they achieved (by some degree) to imitate human perception of sound differences, so that high fitness values actually indicate well-sounding results.

Sound Synthesis Interface using Interactive GA

Colin G. Johson in [10] created a system that involves the user in the steps of a GA to produce sounds with the Csound *FOF synthesizer*. This Csound module has more parameter fields than other modules, and is a very flexible module (*see The Csound Book for further description*). The genomes are represented as floating point arrays, which are translated into sounds by the FOF synthesizer. Typical of interactive GAs, he do not operate with specific target sounds, but simply want

to produce sounds desired by the user, and is therefore an exploratory approach.

The system lets the user assign fitness to the sounds through a Graphical User Interface (GUI) by clicking buttons and moving sliders. This aspect of the system results in a significantly longer runtime than non-interactive approaches, considering the amount of mouse clicks on top of processing time of the human brain. However, a relatively low number of generations is reported to be required for 'successful' evolution. Additionally, it lets novice users learn the capabilities of the given synthesizer.

Johnson expresses a personal desire to eventually be able to evolve sounds with certain characteristics, such as melancholy sounds.

The Genophone

The Genophone is, much like Johnsons approach described above, a system with a goal of aiding the user in designing sounds without the necessity for a high level of knowledge of Sound Synthesis Techniques (SSTs). Mandelis et al. in [12] developed a 'hyper instrument' for sound synthesis using the evolutionary concept of selective breeding. The system as a whole consists of a data glove (the user input), an evolutionary software interface, a MIDI keyboard and a synthesizer, as depicted in the following figure. The evolutionary aspect of the system do not involve standard operators, but involves interpolating and a fitness-weighted probabilistic crossover. The user chooses individuals interactively and assign fitness to each of them. Additionally, the operation of reproduction and mutation is applied on discretion of the user, and the offspring are assigned preliminary fitness values according to their parents.

Due to the difficulty of quantitative analysis in exploration approaches, Mandelis et al. do not carry out such experiments in their paper. However, qualitative analysis are described, in which users where adviced to work systematically through controlled techniques. Non-technical users seemed to understand the concepts of diversity, inheritance and recombination. The authors state that the users enjoyed playing with the Genophone, and in overall, considered the project a success.

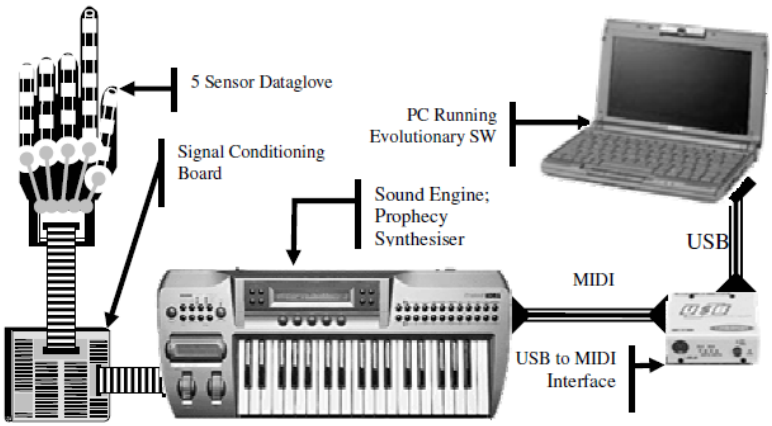

Figure 2.10: An overview of the Genophone components. (Figure borrowed from [12])

Improving Interactive and Non-interactive EC Techniques

James Michael McDermotts partial Ph.D. thesis [14] is a comprehensive work discussing how the field of interactive as well as non-interactive EC can be improved. He aims to improve performance through new GUIs and rigorous methods to study new and exsisting techniques. He also defines a new computational fitness function for matching sounds. This fitness function is compared to other fitness functions based on performance. Since the non-interactive part is far more related to our approach than the interactive part, we only cover this part of McDermotts work.

His non-interactive EC study is largely focused on measuring distance between sounds. His distance metric is based on the idea of *attributes*, a low-dimensional semantic representation of sound. He is the first one to apply this kind of metric to non-interactive EC, and showed to drive target-matching evolution successfully.

Because different DFT lengths emphasise differences at different time and frequency resolutions (mentioned in section 2.4.2), his DFT distance function takes the average over multiple (256, 1024 and 4096) lengths:

where L is the DFT length, X_j and Y_j are the normalized outputs from the jth transform of the sound signals x and y. N (the number of of transforms for each

$$d_F(x,y) = \frac{\sum_{L\in\{256,1024,4096\}} d_{F_L}(x,y)}{3}$$

$$d_{F_L}(x,y) = \frac{\sum_{j=0}^{N}\left(\sum_{i=0}^{L/2}|X_j(i) - Y_j(i)|\right)}{N}$$

sound) is on the basis of 2x-overlapping Hann windows.

His experiments showed that a time-domain fitness function performed badly, while the typical DFT fitness function and his own *attribute* fitness function performed equally well (as did a composite function composed of the three).

Exploration of EC to Audio Synthesis Parameter Optimization

This is also a comprehensive thesis written by Thomas James Mitchell, which performs a wide exploration in the field of automated parameter estimation with Evolutionary Computation (EC). He studies the potential for EC to automatically map known sound qualities onto the parameters of frequency modulation synthesis. Based on his wide research, he develops a system that evolves sound matches using conventional frequency modulation synthesis models.

The part of his thesis that intrigues us, and relates the most to our problem is his sound similarity measures. He employs a distance metric to indicate the individuals' fitness, and he compares the individuals by computing the *relative spectral error* between spectra of the target and candidate sounds. He justifies his choice on the basis that it has proved effective in previous evolutionary matching studies, while it also offers an excellent balance between detail and execution speed. This relative spectral error is computed by accumulating the normalised difference between each frequency component (bin) of the candidate spectrum against their corresponding components in the target spectrum:

where E is the relative error, T is a vector of target spectrum amplitude coefficients, S a vector of synthesized candidate spectrum amplitude coeffisients, N_{frames}

$$E = \frac{1}{N_{frames}} \sum_{t=0}^{N_{frames}} \sqrt{\frac{\sum_{b=0}^{N_{bins}} (T_{tb} - S_{tb})^2}{\sum_{b=0}^{N_{bins}} T_{tb}^2}}$$

the number of static spectra analysed over the sound duration, and N_{bins} the number of frequency bins produced by the spectrum analysis.

These spectrums are extracted by the Short-Time Fourier Transform (STFT), which is a Fourier-related transform used to determine the sinusoidal frequency and phase content of local sections of a signal as it changes over time. Simplified, it divides a continuously-sampled target signal into frames, which then are transformed into the frequency domain by the DFT. Mitchell employs 10 frames of size 1024 at uniform intervals throughout the duration of the target, and for static tones a single frame of size 1024 is taken.

Additionally, Mitchell developed a window function that allows the energy from each frequency partial to bleed into surrounding bins. Both the target and candidate spectra are modified by this window function.

His first tests showed a highly positive relationship between spectral similarity (calculated by the relative spectrum error) and perception of human listeners. However, variation in the results in the forthcoming tests confirmed that the relative spectrum error could not be assumed as an exact measure of perceptual similarity.

Page intentionally left blank

Chapter 3

Methodology

In this thesis we attempt to construct a system with capabilities of *resembling* or simulating a target guitar sound. We propose an Evolutionary approach by designing an Evolutionary Algorithm (EA) whose task is to evolve sounds *similar* (to a certain degree) to, or preferrably a *true replica* of the desired target sound. Essentially, we face a search for a specific set of parameters for guitar effects.

Our system consists of two separate modules: (1) A Python implemented EA and (2) a collection of Csound scripts. The EA (naturally) handles the evolution of the candidate sounds, and the Csound scripts handles the sound synthesis.

The idea is to *feed* the system with a short sound sample containing nothing but the sound of a guitar (preferably a chord, a single note or a few notes), representing the target solution. Through evolution and sound synthesis the system will then attempt to produce sounds with a certain degree of resemblance to the target sound until a user defined number of generations have been reached.

The system do not account for the fact that the choice of guitar is a decisive factor for the perceived output sound. This aspect is omitted from this project simply because of the high amount of complex factors that comes with it. Details, such as pickups, material, strings and brand of the guitar are too much to handle within our scope, and are in fact a part of a completely different area of research. However, the order in which the effects are applied is included in our implementation.

In Section 3.1 we look at the system pointwise as a whole, before we look at the individual aspects. In Section 3.2 we cover how we represent the guitar sounds as genotypes and phenotypes, as well as how the EA operates on them. In Section 3.3 we look at our Csound module, the different guitar effects we employ, how we apply these effects, and how the communication between the EA and the Csound scripts works. In Section 3.4 we describe details concerning our fitness function and discuss some alternatives to the choice we made.

3.1 Overview

In this section we present an overview of the system.

To get an overall understanding of the system, we go through all the steps at a high level from start to finish.

1. The user specifies a *target sound* file location along with all the other user specified variables (EA parameters, such as population size, crossover rate, etc.).

2. An initial population of sounds (genotypes) are created with values generated randomly with a uniform probability distribution.

3. The genotypes are transformed into phenotypes by converting their binary parameters into floating point values.

4. A loop iterating through every phenotype is initiated:

   (a) The first effect (according to the order vector) is applied to a clean guitar signal audio file (by the *Csound scripts*) corresponding to the parameter value(s) in the parameter vector. The rest of the effects are applied onto the output audio file of the previously applied effect.

   (b) The resulting audio file is analyzed and compared against the target sound by *spectral comparison*. The phenotype is assigned a *fitness value* based on the spectral distance to the target sound.

5. Adults to keep for the next generation are selected through a *selection mechanism*.

6. Parents to produce new offspring for the next generation are selected through a *selection protocol*.

7. New offspring are produced by combining genetic material of the parents (in pairs).

   Step 3 to 7 are repeated until the maximum number of generations has been reached.

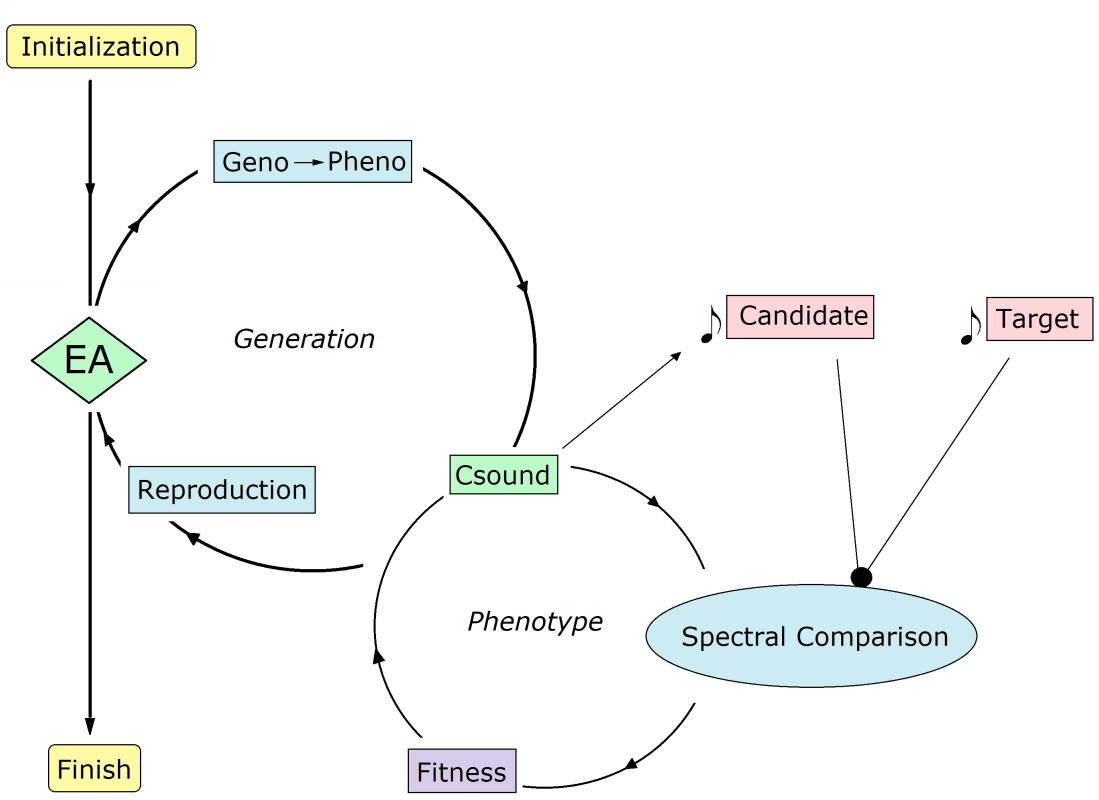

Figure 3.1: Overview of the system.

3.2 Representing Guitar Sounds and Effects

In the field of sound synthesis and previous work attempting to mimic desired sounds [2] [14] [15], a vector-based representation has clearly been the typical way of encoding sound parameters. As parameters simply are numbers within specific ranges, a vector of bits or real numbers is the natural way of representing a sequence of parameters.

In this thesis we employ a genotype represented as a bitvector of guitar effect parameters. This is justified by the fact that it is simple to perform genetic operations (such as crossover and mutation) onto bitvectors, and can be done by simply flipping a bit (between 0 and 1) or swapping two bits. A floating point array might have been an alternative, as it do not have the need for a transformation to real numbers. However, genetic operations tend to be a bit more complicated with real numbers than bits.

3.2.1 Guitar Effects

In our implementation we have included a carefully selected set of guitar effects. These are based on popularity among guitar players, as well as if/how they could be applied by Csound:

- **Distortion** - "Clips" the signal at the maximum capacity of the amplifier, resulting in a distorted square-wave-type waveform.
- **Flanger** - Mixes the signal with a copy which is delayed by a small and gradually changing period. Referred to as having a "jet plane-like" characteristic.
- **Tremolo** - Rapidly turns the volume up and down, creating a "shuddering" effect.
- **Chorus** - Several delayed copies of a signal are mixed with the original, resulting in a rich, shimmering effect.

We wanted to involve a small number of effects to keep the complexity at a reasonably low level. This was preferred to decrease the runtime of the system, in addition to compensate for the increase of complexity when adding the order of the effects to the picture.

In addition to the aforementioned effects, we (initially) included the *reverb* effect in our system. However, it turned out that it had no effect on the *frequencies* of the signal, and was, as a result, removed from the system.

3.2.2 Effect Parameters

The selected guitar effects proved to be achievable/possible to implement in Csound, and they consist of the parameters listed below. Additionally, we have limited each effect parameter within a specific range. This range is carefully chosen by ourselves, solely based on our own perception of at what parameter values the effects no longer sound like they are supposed to. Although the Csound API suggests a 'usual' range to some of the parameters, they do not provide a definite limit and do not necessarily represent a natural range in the context of guitar effects. However, the API informs that the distortion effect starts clipping when the *distortion amount* exceeds 0.7. After testing different values, we ended up with 1.0 as the lower boundary, as it did not seem to have any noticeable impact before it exceeded this value.

- Distortion amount [0.9 - 6.0]
- Flanger delay time [-0.001 - 0.100]
- Tremolo rate [-0.1 - 1.0]
- Tremolo depth [-0.1 - 1.0]
- Chorus rate [-0.1 - 1.0]
- Chorus depth [-0.1 - 1.0]

It turned out that the the effects actually had an impact on the resulting spectral analysis when inserting '0.0' (zero) parameter values (that '0.0' seems to be a minimal value to the parameters). Based on this, we decided to extend the range with one value (e.g. change [1.0 - 6.0] to [0.9 - 6.0] for distortion, where 0.9 turns off the effect entirely and 1.0 represents the first parameter value in the range) for each parameter. Concerning the effects with several parameters, if only one or several of the parameters are outside the range (set to 'off'), these parameters are set to the minimal value within the range (e.g. 0.0 in the tremolo range).

3.2.3 Order of the Effects

As stated previously in this thesis, the order of the effects is an essential aspect of this project. This is normally not an issue in these kinds of EC approaches, and is therefore not a commonly discussed aspect within this field of research. An effective way of including this aspect in our representation was early identified as an obstacle. However, eventually we came up with a solution where we added a second vector to the genotypes, representing the order of the effects. With a register containing a numerical key corresponding to each guitar effect, the *order vector* could simply consist of these keys (the solution is depicted in Figure 3.2).

However, this expansion of the genotype has its consequences. Not only do the general complexity of the search increase, but the genetic operations (such as crossover and mutation) need adjustments to be able to involve the order vector in the inheritance and mutation process. This issue was solved by implementing a separate mutation operation that swaps the order of two effects with the probability of a user specified rate. This operation also swaps the parameters bit sequence(s) inside the parameter vector according to the changes in the order vector. Additionally, when performing a crossover operation, our system simply clones the order vector from parent to offspring. This choice was based on the fact that a crossover operation on the order vector would risk redundancy of one effect (inside the vector) while removing another.

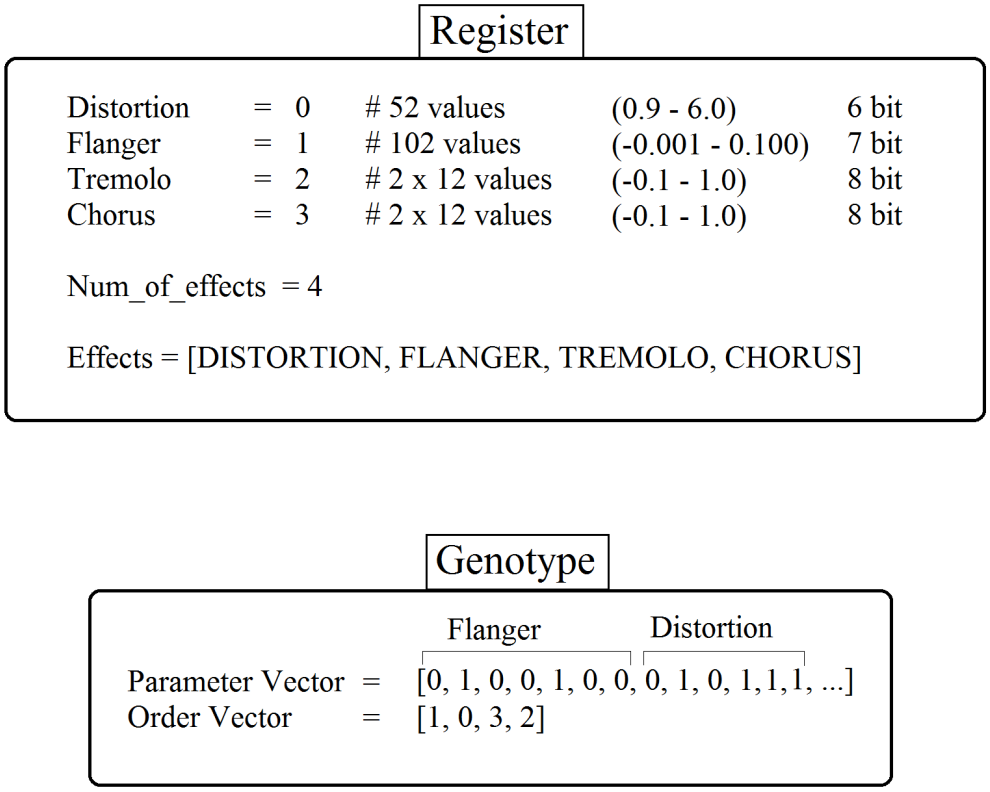

Figure 3.2: The mapping between the register and the genotypes. The number of values is the values within the range subsequently specified in parentheses, followed by the number of bits required to represent the number of values.

3.2.4 Genetic Operations

Typical genetic operators are employd in the system. The initial population is generated randomly with a uniform probability distribution (both parameter and order vector). Our system employs a genotype-to-phenotype process which transforms the bitvector into a floating point array. As depicted in Figure 3.2, the system is familiar with the range of the different effect parameters, as well as the number of bits required to represent this range properly. Each parameters binary value is converted to a floating point value, and then normalized within the corresponding parameter range.

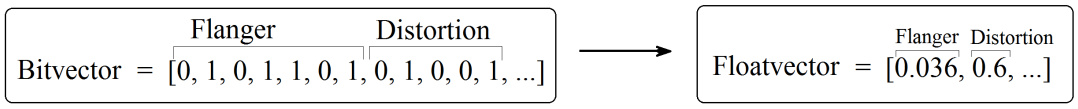

Figure 3.3: Binary values are converted to floating point values corresponding to the limit of the given parameter.

Additionally, traditional *crossover* and *mutation* operations are employed with a probability distribution specified by the user. Our system provides the selection of either a one-, or two-point crossover operation, and the mutation operator consists of a selection between flipping and swapping of bits. The reproduction operation takes two parents as input, and produces two offspring based on the parents genetic material. This operation has several user defined variables: Crossover type (one- or two-point), crossover rate, mutation type (flip or swap), mutation rate, order mutation (yes or no) and order mutation rate.

3.3 Sound Synthesis and Csound

The system possess a separate Csound module which handles the *sound processing* part of the system. In this section we present how the Csound *scripts* are implemented to be able to apply the selected guitar effects, as well as how the scripts and the Python implemented EA communicate.

3.3.1 Csound Handler

As mentioned in Chapter 2, the Csound API offers bindings to Python as well as several other programming languages. The EA module was implemented in Python, where a *Csound handler* was included, functioning as the *link* between the two modules. This handler defines an instance of a Csound session, and is therefore able to compile the Csound scripts, send messages and exchange data with the them.

During the research phase of the project, we were looking for a resource-efficient technique to handle the communication between the two modules. This involved transmission of parameter values from the EA module to the Csound scripts, following a transfer of an audio file back to the EA module. A solution consisting of a channel with the capabilities of sending an audio file directly was preferred to the simpler alternative of writing/reading to/from the hard drive. Unfortunately, we were unable to find such a tool, thus resulting in the latter solution.

When implementing this solution, it turned out that Csound was not able to write to the very same file it read from (in the same script). This issue was solved by producing a *copy* of the sound file at all times. The copy functioned as input to the scripts, and the regular sound file functioned as output.

Another observation was that Csound failed to compile the scripts from time to time, even though the scripts were unchanged. We are not able to explain this behaviour, but it might be that it has trouble handling the high amount of compilations after one another. Nevertheless, this was solved by simply recompiling if it happened to fail.

```
def applyFlanger(inputFile, flangerDelay):
    csdFile = "csd/flangerTest.csd"
    csound = csnd.Csound()  #Csound instance
    compiled = csound.Compile(csdFile)
    if compiled != 0:
        print "Failed to compile ",csdFile
        return False
    csound.SetChannel("flanger", flangerDelay)
    csound.SetChannel("filename", inputFile)
    csound.Perform()
    copySuccess = createCopy("sound.wav")
    if copySuccess:
        return True
    else:
        return False
```

Figure 3.4: The method handling the flanger effect. Sends *input file* and *delay time* as arguments to the Csound script through communication channels.

3.3.2 Csound Scripts

As we had minimal experience with Csound, the implementation of these scripts are greatly affected by examples from the API and tutorials. However, some adjustments were necessary to reach the functional level we wanted.

In the scripts we define the details regarding the sound files, such as samplerate and number of audio channels. The specific sound is defined within the *instrument* tag. Here we read the input audio file, define the communication channels that exchange data with the EA module, create and apply the given effect based on the arguments received through the channels, and finally write the result to a *.wav* file (defined within the *CsOptions* tag). Within the *CsScore* tag the 'composing' happens. Here the duration of the instrument(s) are specified (together with its parameters). In our case we have just one instrument playing for one short duration, but usually this section is significantly longer (when dealing with a high amount of instruments).

Each script represents an effect, which are run and compiled by the Csound handler (as depicted in Figure 3.4). The flanger effect script is depicted in Figure 3.5,

including descriptive comments regarding the essential parts.

```
<CsoundSynthesizer>
<CsOptions>
-o D:/Fordypning/Kode/Prototype/csd/sound.wav -W ;Render file
</CsOptions>
<CsInstruments>

sr = 44100               ;Sampling rate
ksmps = 32
nchnls = 1               ;Number of audio channels
0dbfs = 1

instr 1
idelay chnget "flanger"          ; Communication channel
Sfile chnget "filename"          ; Communication channel

asnd diskin2 Sfile,    1,       0,         1  ;Read file

kfeedback = p4
adel linseg 0, p3, idelay
aflg flanger asnd, adel, kfeedback   ;Apply flanger effect
     out aflg+asnd             ;Output original with flanger
endin
</CsInstruments>
<CsScore>
i 1 0 2 .1       ;instrument, start time, end time, feedback
e
</CsScore>
</CsoundSynthesizer>
```

Figure 3.5: The script applying the flanger effect. The variables *idelay* and *Sfile* are received through communication channels from the EA module.

3.4 Fitness

Some form of quality measurement is essential to be able to properly compare two individuals, a factor to *distinguish* the two from one another. Based on the fact that we operate on guitar sounds in this approach (more specifically the influence of guitar effects), we made a decision to base our fitness function solely on the *frequency* properties of the sounds. Nearly all guitar effects alter the frequencies of the signal in some way, and the frequency domain is therefore a natural choice of area to investigate. No factor of *human percepton* is included in the fitness function, as it would cause a significantly longer runtime as well as an increase in complexity.

The majority of systems involving sound comparison employ the discrete Fourier Transform (DFT) to transform the signal into the frequency domain, and a spectral analysis of the candidate and target sound to compare them. Among others, McDermott et al. in [14] reports that a clean DFT-based fitness function performed equally well as two other more complex metrics. The fast Fourier Transform (FFT) has proved to be an efficient algorithm of computing this transform, and is the reason why we employ it in our system. Furthermore, we use the mean squared error (MSE) metric to compare the candidate and target sound. This metric (in different variations) has consistently been used to compare sound, e.g. by Bozkurt et al. in [2], Mitchell in [15] and McDermott et al. in [14], to mention some.

We have also added the alternative of multiplying a *Hamming window* onto the signal before computing the DFT. This window is applied to remove discontinuity in the sound signal (see [7] for further information concerning window functions). The Hamming window formula is defined as follows:

$$w(n) = 0.54 - 0.46cos(\frac{2\pi n}{N-1}) \ , \quad 0 \leq n \leq N-1 \tag{3.1}$$

where N is the transform size and n is the length of the signal vector.

We employ the FFT (formula 2.2 described in Section 2.4.1), and extract the real numbers from the results by taking the *modulus* of the output of the FFT.

Furthermore, we calculate the MSE by formula 2.3 described in Section 2.4.2.

The formula of the *relative spectral error* employed by Mitchell in [15] (without his window function) was tested, but proved to be far too resource demanding, resulting in the system to run very slowly (details is presented in Chapter 4). To achieve a satisfactory amount of result data we decided to stick with the regular MSE metric of one frame with the same length as the transform. Additionally, Mitchell in [15] states that a single frame of size 1024 proved to be adequate for producing accurate matches for static tones (as we are working with in this system).

The distance metric returns the actual calculated distance to the target sound, which means that lower is better. Normalization of the distance was omitted to minimize the runtime of the system, and was found to have no actual impact on the results.

Page intentionally left blank

Chapter 4

Results and Discussion

In this chapter we present the results of running the system with various target sounds and settings. We will discuss the individual results within each section, as well as make an overall discussion at the end of the chapter.

Runtime results of the sound synthesis part (sending parameters to Csound and writing to disk) are presented in Section 4.1. Runtime results of the distance metrics are presented in Section 4.2. Accuracy results of the fitness function are presented in Section 4.3, while the primary results concerning the outcome of the system are presented in Section 4.4.

4.1 Sound Synthesis

This part of the system was predicted to be computationally heavy, as it physically writes/reads to/from the hard drive whenever applying an effect to an individual in the population. Here we present the runtime of the methods applying the effects (can be found in the CsoundHandler).

Method	Runtime (s)
Apply Flanger	0.0682
Apply Distortion	0.0676
Apply Tremolo	0.0748
Apply Chorus	0.0757

The runtime result is an average of the respective method executed 100 times.

Considering that each individual in the EA population potentially utilizes all four methods every generation, a relatively high runtime of the system is expected. If we operate on a population of 100 individuals and a 100 maximum number of generations (as we do throughout in our experiment), we get the approximate runtime calculated in equation 4.1 (in minutes) for each run. Note that this runtime is exclusive the spectral distance metric and the rest of the operations in the EA.

$$Runtime = \frac{(0.0682 + 0.0676 + 0.0748 + 0.0757) * 100 * 100}{60} = 47,72\ min \quad (4.1)$$

This is a considerably high runtime, and it was also expected to be relatively high from the start. The runtime is, however, not a critical aspect of the system, as this system not is supposed to run live (while playing guitar). Nevertheless, this issue is definitely a technical drawback, and is perhaps the systems biggest weakness.

We believe there may exist other alternatives to this technique, e.g. some sort of open *direct channel* (working both ways) between the Csound module and the CsoundHandler with capabilities to send parameters, commands and complete sound files (*.wav*). Such a solution would most likely enhance the runtime significantly.

4.2 Distance Metrics

We assessed two distance metrics in this system; (1) a FFT calculation followed by a *regular MSE distance metric*, and (2) a FFT calculation performed on several

frames of the signal, followed by a *relative MSE distance metric* based on the outcome of the several FFTs (employed by Mitchell in [15]). Before deciding which to use in our experiment, a runtime test of the two was performed. Below we present the results of the runtime tests of the metrics, as well as an approximation of the total runtime of the system (inclusive sound synthesis) using the corresponding distance metrics.

Metric	Metric Runtime (s)	System Runtime (min)
Regular MSE distance	0.0021	48.07
Relative MSE distance	0.1080	65.72

The runtime result is an average of the respective metric executed 100 times.

Looking at the runtime of the metrics alone, the regular MSE metric can be executed more than 50 times in the same time the relative MSE metric is executed once. The difference of the total runtime is just above 17 minutes, and is definitely noteworthy, considering the total amount of times the system is run within the project period.

The actual runtime of the system with the regular MSE metric on a target sound with all effect parameters (using a timer) turned out to be approximately 62 minutes, depending on the workload on the computer from other applications.

4.3 Fitness Accuracy

When applying EC onto an unexplored field of research (within EC), such as guitar sounds and effects, the risk of vain or simply poor results is always present. However, promising results concerning EC and synthesizer sounds is an indication that EC might be a suitable technique to mimic guitar sounds as well.

In this section we present the accuracy of the fitness function, that is how well it reflects the candidates distance to the target effect parameters. Fitness values for a wide range of parameter sets (against several target sounds) are presented in the following tables. Note that the candidates listed in the tables throughout

this section are manually generated with parameters somewhat close to the target sound. We have done this to analyze the fitness value's behavior when comparing sounds with different sets of parameters.

4.3.1 Individual Effects

First we look at candidates with only one active effect, to investigate the accuracy against the individual effects.

Note that lower fitness is better.

D = Distortion, F = Flanger, T = Tremolo, C = Chorus

Distortion

Sound	**Parameters**	**Order**	**Fitness**
Target	**2.6**, -0.1, -0.1, -0.1, -0.1, -0.001	**D**, T, C, F	-
Candidate 1	**2.3**, -0.1, -0.1, -0.1, -0.1, -0.001	**D**, T, C, F	0.3430
Candidate 2	**2.0**, -0.1, -0.1, -0.1, -0.1, -0.001	**D**, T, C, F	0.7843
Candidate 3	**2.9**, -0.1, -0.1, -0.1, -0.1, -0.001	**D**, T, C, F	0.2687
Candidate 4	**3.2**, -0.1, -0.1, -0.1, -0.1, -0.001	**D**, T, C, F	0.4845

Flanger

Sound	**Parameters**	**Order**	**Fitness**
Target	0.9, -0.1, -0.1, -0.1, -0.1, **0.040**	D, T, C, **F**	-
Candidate 1	0.9, -0.1, -0.1, -0.1, -0.1, **0.037**	D, T, C, **F**	1.1502
Candidate 2	0.9, -0.1, -0.1, -0.1, -0.1, **0.034**	D, T, C, **F**	2.1475
Candidate 3	0.9, -0.1, -0.1, -0.1, -0.1, **0.043**	D, T, C, **F**	1.1010
Candidate 4	0.9, -0.1, -0.1, -0.1, -0.1, **0.046**	D, T, C, **F**	2.0026

The fitness value seems to reflect the difference of the distortion and flanger parameter to the target sound fairly well, as it increases with the actual parameter

difference. Another observation is that the fitness value makes bigger leaps when increasing the parameter at low values than higher values. This is not surprising behavior, considering that a specific increase of the parameter value (say 0.3) represents a higher percentage at lower values than at higher values. This indicates that lower parameter values may be easier targets, as the fitness difference is more evident.

Tremolo

Sound	**Parameters**	**Order**	**Fitness**
Target	0.9, **0.2, 0.2**, -0.1, -0.1, -0.001	D, **T**, C, F	-
Candidate 1	0.9, **0.2, 0.1**, -0.1, -0.1, -0.001	D, **T**, C, F	0.8320
Candidate 2	0.9, **0.1, 0.2**, -0.1, -0.1, -0.001	D, **T**, C, F	0.6061
Candidate 4	0.9, **0.1, 0.1**, -0.1, -0.1, -0.001	D, **T**, C, F	0.6246
Candidate 5	0.9, **0.1, 0.0**, -0.1, -0.1, -0.001	D, **T**, C, F	1.7034
Candidate 6	0.9, **0.0, 0.1**, -0.1, -0.1, -0.001	D, **T**, C, F	0.5376
Candidate 7	0.9, **0.0, 0.0**, -0.1, -0.1, -0.001	D, **T**, C, F	1.7034
Candidate 8	0.9, **0.2, 0.3**, -0.1, -0.1, -0.001	D, **T**, C, F	0.7978
Candidate 9	0.9, **0.3, 0.2**, -0.1, -0.1, -0.001	D, **T**, C, F	0.5677
Candidate 10	0.9, **0.3, 0.3**, -0.1, -0.1, -0.001	D, **T**, C, F	0.5206
Candidate 11	0.9, **0.3, 0.4**, -0.1, -0.1, -0.001	D, **T**, C, F	0.9565
Candidate 12	0.9, **0.4, 0.3**, -0.1, -0.1, -0.001	D, **T**, C, F	0.8104
Candidate 13	0.9, **0.4, 0.4**, -0.1, -0.1, -0.001	D, **T**, C, F	0.9650

The fitness value seems to somewhat increase with the actual distance of the parameters. However, a change of the tremolo *depth* seems to affect the fitness value more than a change of the tremolo *rate* by a small margin. This indicates that the depth parameter most likely will be easier to hit than the rate parameter.

There are some noteworthy observations to be made, however. Despite that both candidate 3 and 6 actually is further from the target parameters than candidate 1 and candidate 4 and 5, respectively, they both have better fitness values. This is a somewhat strange behavior of the fitness value, but they actually have one thing in common; their own tremolo parameters are identical (as with the target sound).

This indicates that perhaps it is not just the *distance* that affects the fitness, but also the parameter *pattern* or *relationship* between the parameters within the same effect. However, this 'pattern factor' seems to fade when moving further from the targets parameters (looking at candidate 7 and 13), and it is natural to assume that this will fade even more when moving even further.

Chorus

Sound	**Parameters**	**Order**	**Fitness**
Target	0.9, -0.1, -0.1, **0.2, 0.2**, -0.001	D, T, **C**, F	-
Candidate 1	0.9, -0.1, -0.1, **0.2, 0.1**, -0.001	D, T, **C**, F	2.1936
Candidate 2	0.9, -0.1, -0.1, **0.1, 0.2**, -0.001	D, T, **C**, F	0.2264
Candidate 3	0.9, -0.1, -0.1, **0.1, 0.1**, -0.001	D, T, **C**, F	2.2105
Candidate 4	0.9, -0.1, -0.1, **0.2, 0.3**, -0.001	D, T, **C**, F	2.3275
Candidate 5	0.9, -0.1, -0.1, **0.3, 0.2**, -0.001	D, T, **C**, F	0.2655
Candidate 6	0.9, -0.1, -0.1, **0.3, 0.3**, -0.001	D, T, **C**, F	2.3856

We can see the same pattern here as with the tremolo effect. The chorus *depth* generally seems to have a greater effect on the fitness value than the chorus *rate*, and this difference is notably larger in the chorus than the tremolo effect. However, the 'pattern factor' observed in the tremolo effect is not evident here.

4.3.2 Effect Relationship

In this section we look at the fitness value's behavior when comparing sounds with different effects. Since three of the four selected effects are in the same category (modulation), we suspect those (in particular) to perhaps 'disturb' each other in the search for the target sound parameters.

With a target sound consisting of only the tremolo effect, we investigate if a sound with only the chorus or flanger effect would get a better fitness value than a sound with the tremolo effect, but with slightly wrong parameters.

Sound	**Parameters**	**Order**	**Fitness**
Target	0.9, **0.1, 0.1**, -0.1, -0.1, -0.001	D, **T**, C, F	-
Candidate 1	0.9, **0.3, 0.3**, -0.1, -0.1, -0.001	D, **T**, C, F	1.0074
Candidate 2	0.9, -0.1, -0.1, **0.1, 0.1**, -0.001	D, T, **C**, F	6.2203
Candidate 3	0.9, -0.1, -0.1, -0.1, -0.1, **0.005**	D, T, C, **F**	15.3103

The results tell us that the system do not seem to be 'disturbed' by the effects of the same category. We had a suspicion that especially the chorus and tremolo effect (in some situations) could be mistaken for each other. But the fitness values presented above clearly states that this is not the case.

4.3.3 Order of the Effects

In the following tables we study the behavior of the fitness value when comparing sounds with identical effect parameters, but placed in a different order. We look at sounds with two, three and four effects individually.

Sound	**Parameters**	**Order**	**Fitness**
Target	**2.0**, -0.1, -0.1, -0.1, -0.1, **0.020**	**D**, T, C, **F**	-
Candidate	**0.020**, -0.1, -0.1, -0.1, -0.1, **2.0**	**F**, T, C, **D**	1.7356

Sound	**Parameters**	**Order**	**Fitness**
Target	**2.0**, **0.3, 0.1**, -0.1, -0.1, **0.020**	**D**, **T**, C, **F**	-
Candidate 1	**0.020**, **0.3, 0.1**, -0.1, -0.1, **2.0**	**F**, **T**, C, **D**	1.2190
Candidate 2	**0.020**, **2.0**, -0.1, -0.1, **0.3, 0.1**	**F**, **D**, C, **T**	1.4988

Sound	**Parameters**	**Order**	**Fitness**
Target	**2.0**, **0.3, 0.1**, **0.2, 0.2**, **0.020**	**D**, **T**, **C**, **F**	-
Candidate 1	**0.020**, **0.3, 0.1**, **0.2, 0.2**, **2.0**	**F**, **T**, **C**, **D**	2.3260
Candidate 2	**0.020**, **2.0**, **0.2, 0.2**, **0.3, 0.1**	**F**, **D**, **C**, **T**	1.4657
Candidate 3	**0.2, 0.2**, **0.020**, **2.0**, **0.3, 0.1**	**C**, **F**, **D**, **T**	2.2273
Candidate 4	**0.2, 0.2**, **0.020**, **0.3, 0.1**, **2.0**	**C**, **F**, **T**, **D**	2.2263

We have arranged the tables, such that the number of swaps increases downwards in the tables (candidate 1 has two effects in the wrong position, candidate 2 has three effects in the wrong position, etc.). When dealing with only two effects, we can see that the difference in fitness value changes to a fairly large extent. However, modifications on the order when dealing with more than two effects resulted in somewhat unexpected fitness values. We assumed that the fitness would increase with the number of misplaced effects, but that do not seem to be the case, and no clear pattern can be found. We suspect that the effects are greatly dependent of their successors in the order chain and their respective parameter values. For example, the flanger effect might be amplified by distortion, and weakened by the chorus.

However, a thorough investigation of the impact of the effect order is a complex task in itself, and further analysis of this aspect would require a deeper study of the effects alone as well as the effects in relation to each other. Such a study is too extensive to cover in this thesis, and is therefore not investigated any further.

4.4 System Results

In this section we present the outcome results of the system. Various plots of the population development through the required number of generations, as well as overall plots of the total number of runs are depicted in the figures listed further below. The following EA settings were used throughout the experiment:

- Population size = 100
- Maximum number of generations = 100
- One-point crossover operation
- Mutation operation done by bit flip
- Clone rate = 0.5 (thus a 0.5 crossover rate)
- Order mutation = true (on)

- Order mutation rate = 0.2
- One selection mechanisms was employed:
  - Tournament selection: $k = 3$, $\epsilon = 0.2$
- One selection protocol was employed:
  - Generational mixing
- The Hamming window was not utilized in our experiments

The target sounds were manually generated , and the system was run multiple times for each target sound. But since the system spent a considerably large amount of time on each run, the number of runs for each target sound had to be somewhat limited. Some key target sounds were therefore chosen to be run 20 times, and the rest were run 5 times. We will look closely at the results of the key target sounds, while loosely discussing the observations we make of the less detailed results of the remaining target sounds. Note that the results we present where only 5 runs have been performed are not classified as statistical significant data. We can only make our own assumptions based on the trends and observations we make of these results.

4.4.1 Individual Effects

Before we see how the system handles target sounds with all the effects, we look at how well the system handles target sounds consisting of only one effect. After observing that the fitness value seemed to behave fairly similar concerning the distortion and flanger effect (in Section 4.3.1), we decided to pick only one of them to study in detail (the *flanger* effect). We had a somewhat similar observation regarding the tremolo and chorus effect, therefore we decided to look closer at only one of them as well (the *tremolo* effect). Below we present results of 20 runs against a flanger and tremolo target sound, followed by 5 runs against a distortion and chorus target sound.

Flanger

	Parameters	**Order**
Target	0.9, **0.055**, -0.1, -0.1, -0.1, -0.1	Distortion, **Flanger**, Chorus, Tremolo

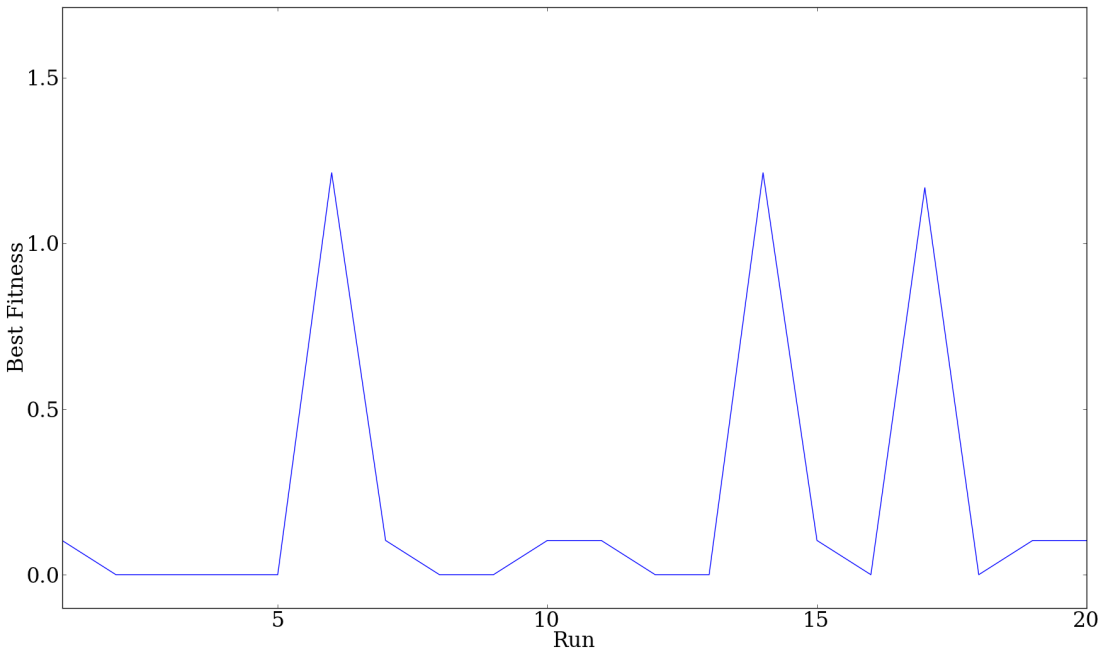

Figure 4.1: Plot of 20 runs against a flanger target sound.

Variance	Mean	Max	Min
0.1725	0.2159	1.2132	0.000

The system managed to find an exact match ten out of twenty runs, was *marginally* close in seven runs (missing 0.001 on the flanger parameter, all other effects excluded), and was somewhat far from a good solution the remaining three runs. The convergence time varied from 17 to just above 80 generations to reach the goal (no clear pattern was evident). Below we present plots of one run from each of the three cases.

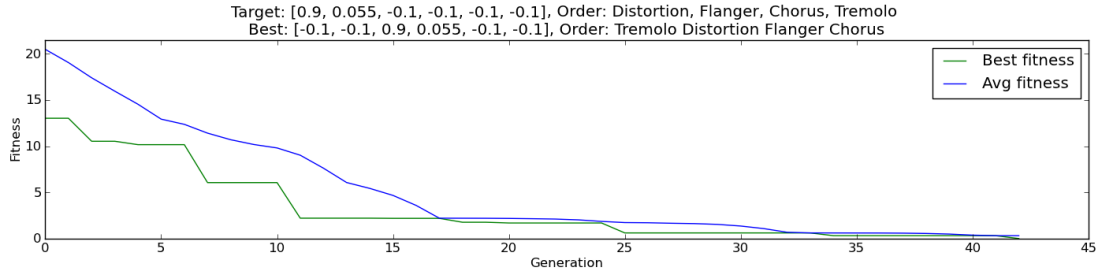

Figure 4.2: Exact match.

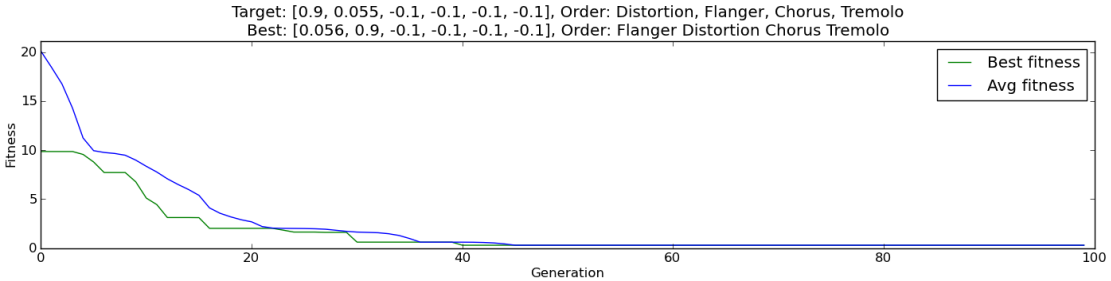

Figure 4.3: Nearly a match.

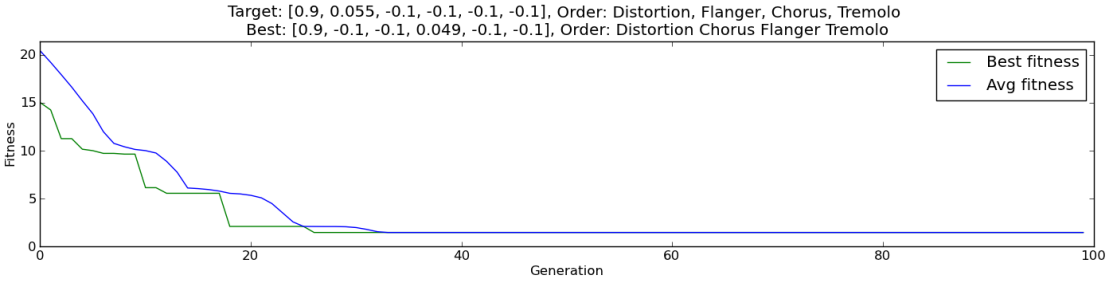

Figure 4.4: Poor match.

Tremolo

	Parameters	**Order**
Target	0.9, -0.001, -0.1, -0.1, **0.1, 0.5**	Distortion, Flanger, Chorus, **Tremolo**

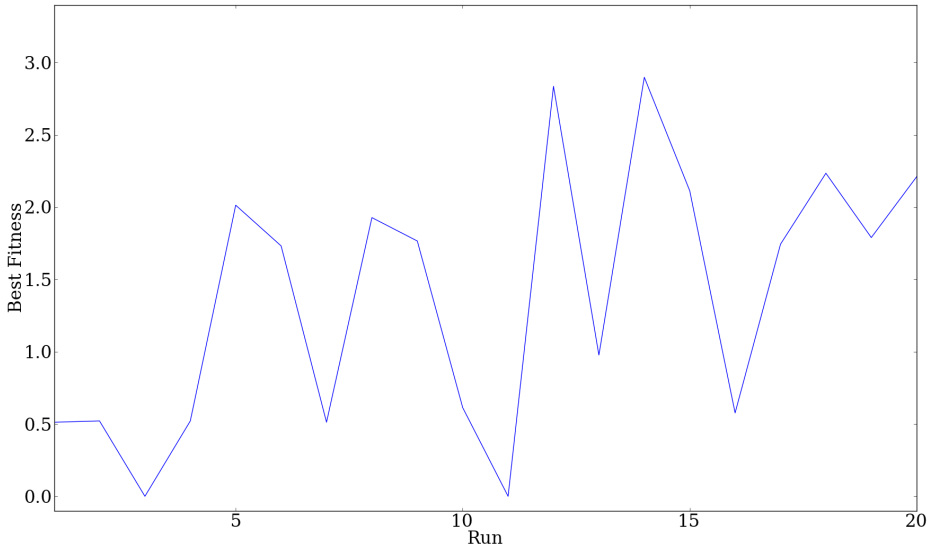

Figure 4.5: Plot of 20 runs against a tremolo target sound.

Variance	Mean	Max	Min
0.7916	1.3747	2.8976	0.000

These results clearly indicate that the system has trouble finding the correct tremolo parameters. We found an exact match only twice of the twenty runs, while the remaining eighteen were pretty poor solutions overall. In about five of the eighteen the system managed to exclude the other effects entirely, only missing the tremolo parameters by some margin. Below we present plots of three selected runs, as with the flanger effect.

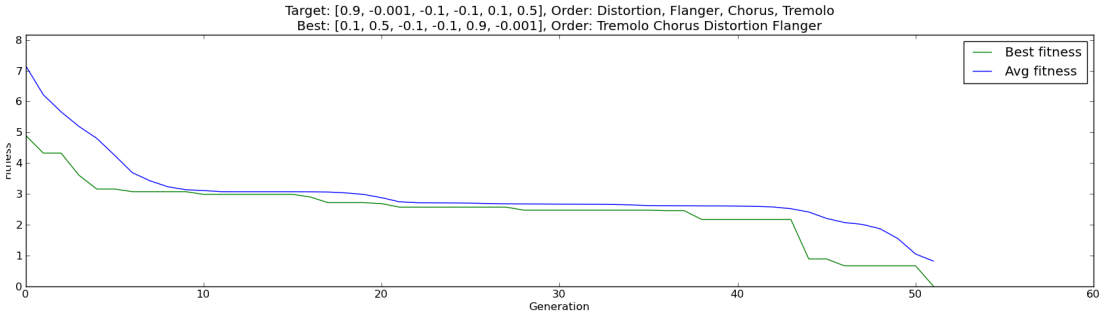

Figure 4.6: Exact match.

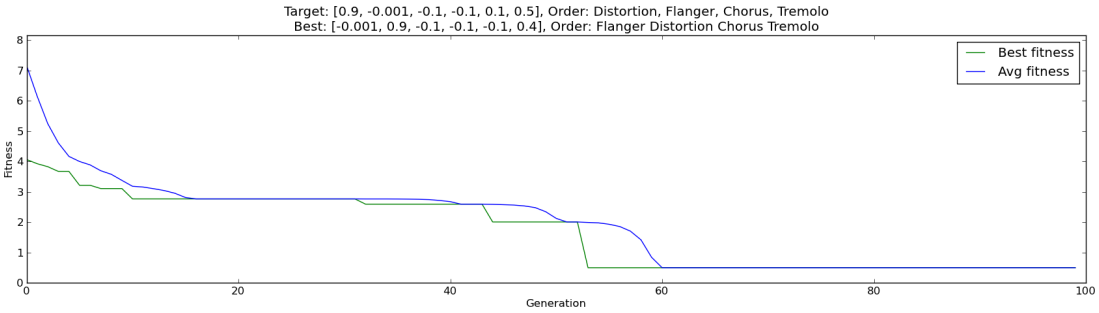

Figure 4.7: Nearly a match.

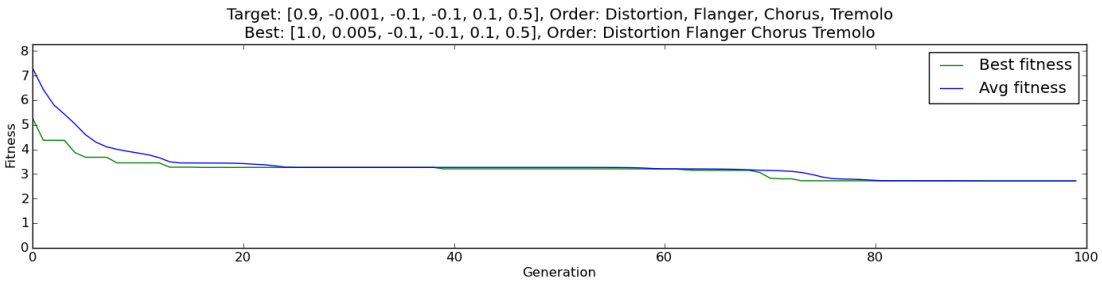

Figure 4.8: Poor match.

Distortion

	Parameters	**Order**
Target	**1.7**, -0.001, -0.1, -0.1, -0.1, -0.1	**Distortion**, Flanger, Chorus, Tremolo

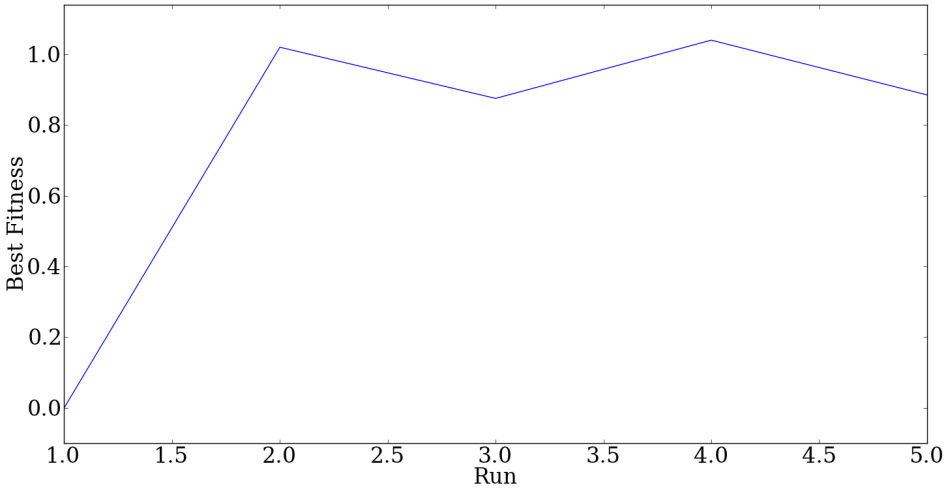

Figure 4.9: Plot of 5 runs against a distortion target sound.

We can see that one exact match was found in five runs, while the other four were further from a good solution. It do not tend to be as easy to hit as the flanger effect, but we would assume that a few more exact matches would have been found if more runs had been performed.

Chorus

	Parameters	**Order**
Target	0.9, -0.001, **0.2, 0.4**, -0.1, -0.1	Distortion, Flanger, **Chorus**, Tremolo

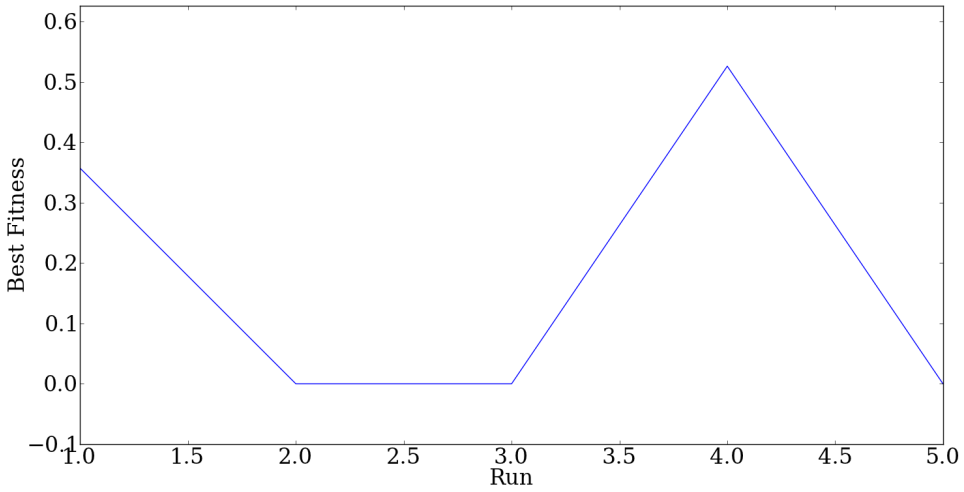

Figure 4.10: Plot of 5 runs against a chorus target sound.

The results of the five runs indicate that the system handles the chorus parameters very well, and hits en exact match in three out of five runs. Since it reaches a perfect match a majority of the runs, it is natural to assume it would continue this trend with a larger amount of runs as well.

4.4.2 Multiple Effects

In this section we present the results when employing target sounds consisting of multiple effects. A detailed study is performed on a target sound with four effects, while we only comment the observations we make of the less detailed results regarding the target sounds with two and three effects.

Two Effects

	Parameters	**Order**
Target	**0.031**, **3.1**, -0.1, -0.1, -0.1, -0.1	**Distortion**, **Flanger**, Chorus, Tremolo

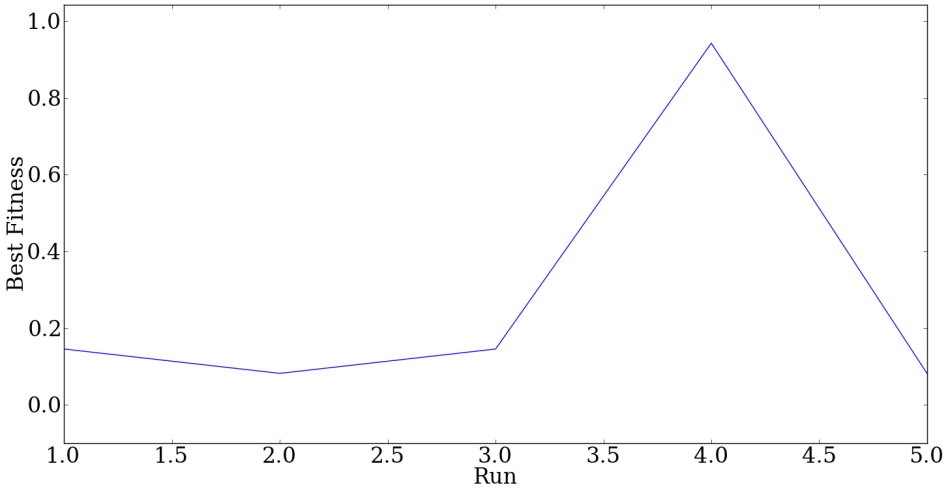

Figure 4.11: Plot of 5 runs against a target sound with two effects.

The five runs resulted in four marginally close solutions (all of them were missing by one decimal in one of the effects), and one fairly poor solution. We would assume that additional runs would result in at least a couple of exact matches, considering that four out of five were as close as it possibly can be an exact match.

Three Effects

	Parameters	**Order**
Target	**1.4**, **0.022**, -0.1, -0.1, **0.1, 0.5**	**Distortion**, **Flanger**, Chorus, **Tremolo**

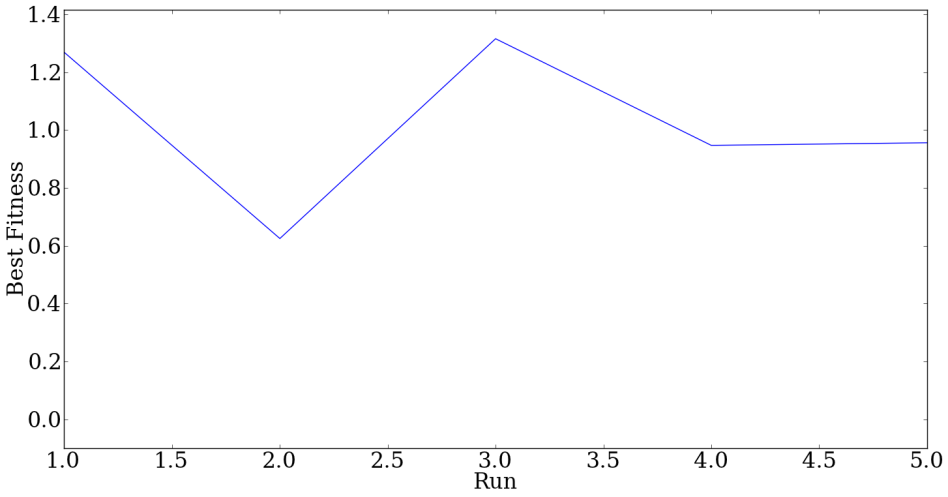

Figure 4.12: Plot of 5 runs against a target sound with three effects.

These five runs resulted in no good solutions, and the system seems to struggle with this set of parameters. It is possible that it might have produced better solutions if a higher amount of runs had been made, but based on these results it is unlikely that an exact match would have been found.

Four Effects

Since this target sound consists of all the effects, it is perhaps the most interesting and complex target sound in this experiment. Next we present the plot of 20 runs against the target sound specified below.

	Parameters	**Order**
Target	**0.008**, **0.2**, **0.3**, **0.1**, **0.1**, **1.5**	**Flanger**, **Tremolo**, **Chorus**, **Distortion**

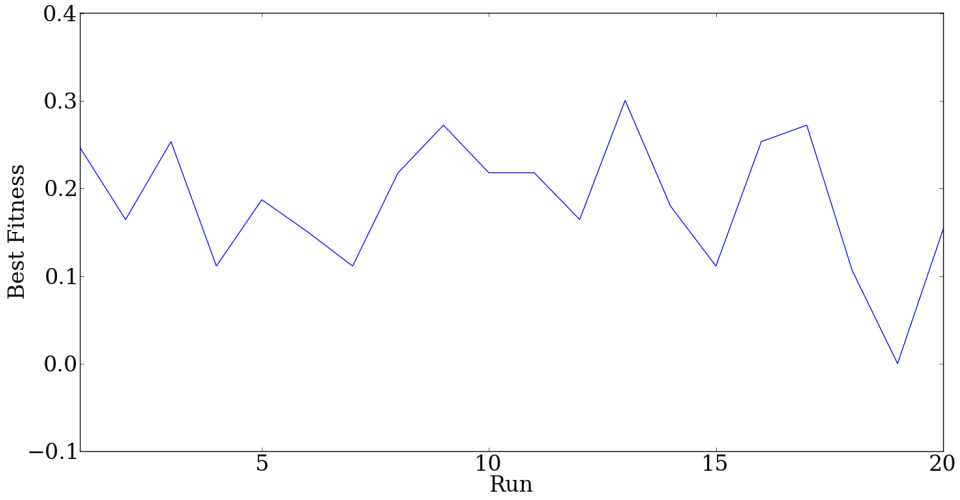

Figure 4.13: Plot of 20 runs against a target sound with all four effects.

Variance	Mean	Max	Min
0.0052	0.1846	0.3006	0.000

The plot of the twenty runs shows very promising results, with all solutions being relatively close to the target sound, and even finding an exact match once. Plots of all the runs are not presented in this report, but we have identified the most consistent aspects of the results:

- The chorus and flanger parameters are identical to the target 16 out of 20 runs.
- Distortion is placed correctly in the order chain 20 out of 20 runs.
- Chorus is placed correctly in the order chain 19 out of 20 runs.
- The system seems to struggle to find the correct position in the order for the flanger and tremolo effect, and these two are (approximately) correctly placed in half of the runs (they tend to swap between first and second position).
- The system seems to struggle to hit the correct distortion and tremolo parameters, as these are identical to the target parameters in only two out of twenty runs.

To get a closer look at the various parameters statistically, we present the variance, mean, max and min values for the different parameter values.

Parameter	Variance	Mean	Max	Min
Distortion amount	0.0216	1.72	1.9	1.4
Flanger delay time	1.9e-07	0.0081	0.009	0.007
Tremolo rate	0.0145	0.295	0.5	0.1
Tremolo depth	0.0082	0.185	0.4	0.1
Chorus rate	0.0026	0.08	0.1	-0.1
Chorus depth	0.0000	0.1	0.1	0.1

The table tells us that the flanger and chorus parameters are very consistent, since the variance value is close to zero. The distortion and tremolo parameters, however, tend to be varied, as the variance value is higher. This result fits very well with the observations we made of the individual effects (Section 4.4.1). The flanger and chorus effect generally tends to be easier to mimic than the distortion and tremolo effect.

Below we present plots of some of the runs performed against this target sound.

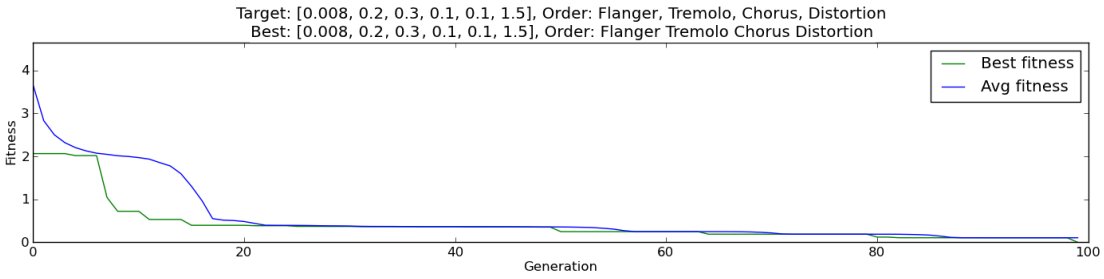

Figure 4.14: Exact match.

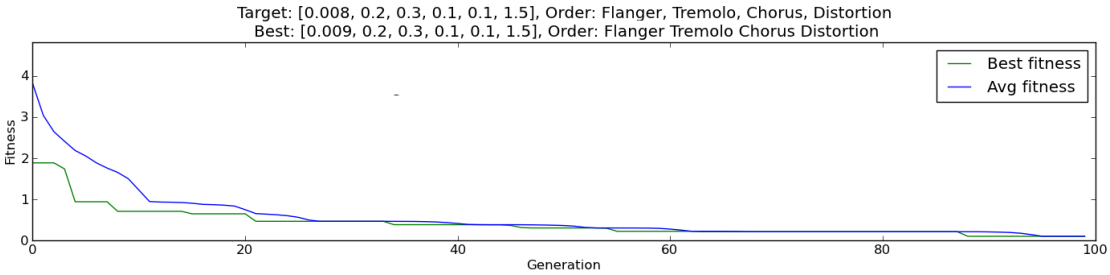

Figure 4.15: Nearly a match.

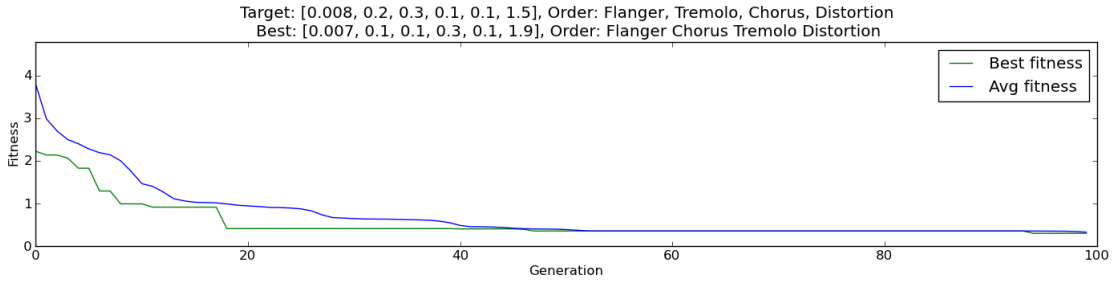

Figure 4.16: Somewhat poor match.

4.5 Discussion

Looking at the overall results, we can definitely say that the system handles the the various effects differently. It tends to struggle more with the distortion and tremolo effect than with the flanger and chorus effect. However, this might be connected with the fitness accuracy to the given effect. The fitness accuracy of the flanger and chorus effect seems to be better than the distortion and tremolo, as an increase of the flanger and chorus parameters clearly results in a bigger increase of fitness compared to an increase of the distortion and tremolo parameters. It might be the case that the distortion and tremolo effect simply are more complex than the flanger and chorus effect, but that would require an entirely different kind of study to prove, and is outside the scope of this thesis.

Another thing to notice is that the target sound with three effects (hereinafter (3)) tends to be harder to mimic than the target with four effects (hereinafter (4)). We expected the complexity to increase somewhat proportionally with the

number of effects, but the results indicate that this might not be the case (at least not consistently). However, we have identified a few things that might have caused this:

- (3) has a higher flanger and tremolo depth value than (4). Some results indicated that lower parameter values were easier to hit than high parameter values.
- (3) do not contain the chorus effect, which proved to be a fairly simple effect to mimic. This might increase the general fitness value of the produced solutions of (3), since the error of each parameter in (3) will have a greater effect on the fitness value than in (4) (considering that each effect represents a larger percentage of the total error).

Further experiments and runs of (3) probably would have made this more clearly.

Aside from the things mentioned above, the system seems to produce fairly positive results. The main target sound (4) is mimicked to a high degree, and the system even produced an exact match in one of the twenty runs. But the results concerning (3) indicate that the system might have trouble with certain parameter settings. Further investigations of such target sounds would hopefully expose the flaws and weaknesses of the system even more, and would certainly be interesting as future work.

Regarding the runtime tests, the system certainly has a high potential for improvement, especially concerning the sound synthesis part. Alternatives, such as a *direct channel solution* between the Csound and EA module would most likely enhance the system significantly. A parallelizable architecture like Bozkurt et al. propose in [2] could also be an improvement on the performance of the system.

The *regular MSE distance metric* was preferred to the more resource demanding *relative MSE distance metric*. This choice was based on the decisive runtime results, as well as on the fact that the regular MSE metric provided a sufficiently accurate measurement tool.

Page intentionally left blank

Chapter 5

Conclusion and future work

In this thesis we have proposed an *Evolutionary Intelligent Guitar Processor*, a system that strives to evolve sounds identical to a given target sound with an Evolutionary Algorithm (EA). With earlier successful attempts being made concerning simulation of synthesizer sounds with Evolutionary methods, we posed the question to whether Evolutionary Computation (EC) could be applied to the task of simulating various guitar sounds (in Chapter 1). The open-source library Csound was initially chosen to handle the sound synthesis functions within the system, and a thorough research and testing period with Csound was conducted to investigate whether it was a suitable tool for the problem at hand as well as to learn how to utilize it.

Our main goal was to implement a somewhat simple prototype employing a small number of carefully selected guitar effects to keep the complexity at a reasonably low level, while still providing popular and well-known guitar effects. The aspect of the order in which the effects are applied is included in the system, which (as far as we know) is an unexplored area within the field of EC. This aspect increased the complexity of the problem, and forced us to expand the design of the genotype and phenotype to be able to properly represent and integrate this aspect into the principles of the EA.

The system relies solely on an automatic fitness function based on spectral com-

parison between the candidate and target sound, and no human perception is involved to assure the quality of the evolved sounds. We employ the fast Fourier transform and the regular mean squared error metric to assign a fitness value to the individuals produced by the EA.

Much effort was put into the *link* between the Csound and EA module. Next to nothing of guides and helpful tips to solve this issue were available online, and thus we hope this thesis (in some manner) can provide guidance on how to communicate between Csound and Python. Our solution was to implement a *CsoundHandler* within the EA module, that handles messages and commands *to* the Csound module. Csound scripts are run and compiled from the CsoundHandler, and the scripts produce sound files which the EA module eventually reads from the hard drive when analyzing the individuals in the EA population. This technical solution proved to be highly resource demanding and time consuming, and is considered as the main weakness of the system. However, the goal of this thesis was not to create an efficient system with a low runtime, but to investigate functionality and whether guitar sounds could be evolved by an EA.

The results of the fitness accuracy experiments indicated that the fitness value reflects the actual difference of the effect parameters in a good way. A few exceptions were identified, though, and a form of 'parameter pattern factor' within one of the effects were discovered. However, this factor proved to be evident at only a few cases, and is only considered a minor issue with marginal impact on the overall results.

The system proved to produce promising results, especially regarding the flanger and chorus effect, while struggling with the distortion and tremolo effect. This was evident in the experiments regarding target sounds with both one and multiple effects. This was somewhat expected from the fitness accuracy results, where the fitness value increased/decreased more evidently with the flanger and chorus effect in comparison with the distortion and tremolo effect. The order of the effects proved to be a decisive factor, and could affect the resulting fitness value just as much as a direct modification of an effect parameter.

The high runtime of the system resulted in less result data than we initially wanted,

and a somewhat less detailed analysis had to be performed on some of the target sounds. This was a bit of a let down, as we hoped to be able to compare all the different target sounds at an equally significant level of detail. We did, however, produce result data of statistical significance regarding the most interesting target sounds (at least in our opinion).

As this only can be considered a prototype of a potential well-functioning intelligent guitar processor, we are far from our long term goal of being able to simulate any given guitar sound sample. The target sounds used in this approach are all manually produced by applying the effects employed in the system. If the system one day shall be able to replicate literally *any* guitar sound (that be an extracted clip from an AC/DC track, or a simple sound produced by your own amplifier), the system should employ *all* existing guitar effects. Additionally, the system should be able to adapt to any type of guitar the user wishes to utilize. These are highly complex aspects to incorporate in the system, and is considered as recommended future work of interest. Other less complex proposals for future work are:

- Further study of the order of the effects, and effect relations in general.
- Alternative techniques to handle the sound synthesis, e.g. an open channel solution.
- Extraction of guitar sound samples from complete musical pieces.
- Some form of human perception in the fitness assignment or as quality assurance, e.g. listening tests.
- Inclusion of an interactive aspect where the user can provide guidance in the sense of which effect he/she thinks are employed, as well as the order he/she believes is utilized.

Finally, we would like to give our recommendations to continue the work in this thesis, and can definitely see this idea reaching a satisfactory functioning level in the future. We have not (and far from it) proved that guitar sounds can be replicated or simulated by the means of EC in any situation, but we present promising results that might encourage and point us in the right direction to eventually

reach our goal of a fully functioning 'intelligent guitar processor'. Additionally, we show that EC *can* be utilized to search for guitar effect parameters equal to the parameters of a target sound, with varying results.

Bibliography

[1] Enrique Alba and Carlot Cotta. Evolutionary Algorithms. *Addresses the concept of Evolutionary Algorithms*, 2004.

[2] Batuhan Bozkurt and Kamer Ali Yüksel. Parallel Evolutionary Optimization of Digital Sound Synthesis Parameters. *Uses genetic algorithms for optimization of various sound synthesis parameters*, 2011.

[3] Keith Downing. Natural and Artificial Selection. *Describes different ways to perform adult and parent selection in Evolutionary Algorithms*, 2011.

[4] Ricardo A. Garcia. Automating the Design of Sound Synthesis Techniques using Evolutionary Methods. *Uses Genetic Programming to design Sound Synthesis Techniques*, 2001.

[5] Ricardo A. Garcia. Growing Sound Synthesizers using Evolutionary Methods. *Evolutionary Methods to suggest topological arrangements for sound synthesis algorithms elements and to optimize their internal parameters*, 2001.

[6] Gerlinda Grimes. How Guitar Pedals Work, March 2011. `http://electronics.howstuffworks.com/gadgets/audio-music/guitar-pedal.htm`.

[7] G. Heinzel, A. Rüdiger, and R. Schilling. Spectrum and spectral density estimation by the discrete fourier transform (dft), including a comprehensive list of window functions and some new flat-top windows. *Gives a practical overview about the estimation of power spectra/power spectral densities using the DFT.*, 2002.

[8] Johannes H. Jensen and Pauline C. Haddow. Evolutionary Music Composition based on Zipf's Law. *Evolutionary Computation applied to music composition*, 2010.

[9] Johannes Høydahl Jensen. Evolutionary Music Composition, A Quantitative Approach. *Finding an automatic fitness function in Evolutionary Music Composition*, 2011.

[10] Colin G. Johnson. Exploring the sound-space of synthesis algorithms using interactive genetic algorithms. *Using genetic algorithms and CSound to evolve and generate desired sounds*, 1999.

[11] Jonas Gutvik Korssjøen. Intelligent Guitar Processor. *Plans a prototype of a system simulating guitar effects using Evolutionary Computation.*, 2011.

[12] James Mandelis and Phil Husbands. Genophone: Evolving Sounds and Integral Performance Parameter Mappings. *Evolutionary Techniques to design novel sounds and their characteristics during performance*, 2006.

[13] James McDermott, Niall J.L. Griffith, and Michael O'Neill. Target-driven genetic algorithms for synthesizer control. *Uses Genetic Algorithms to find synthesizer parameters for a target sound, focus on search performance*, 2006.

[14] James McDermott, Niall J.L. Griffith, and Michael O'Neill. Evolutionary Computation Applied to the Control of Sound Synthesis. *Interactive Evolutionary Computation applied to synthesizer*, 2008.

[15] Thomas James Mitchell. An Exploration of Evolutionary Computation Applied to Frequency Modulation Audio Synthesis Parameter Optimisation. *Using evolutionary computation to automatically map known sound qualities onto the parameters of frequency modulation synthesis*, 2010.

[16] Artemis Moroni, Jônatas Manzolli, Fernando Von Zuben, and Ricardo Gudwin. Evolutionary Computation applied to Algorithmic Composition. *Uses genetic algorithms to generate and evaluate sequences of chords*, 1999.

[17] Janne Riionheimo and Vesa Välimäki. Parameter Estimation of a Plucked String Synthesis Model using a Genetic Algorithm with Perceptual Fitness

Calculation. *Estimating control parameters for a plucked string synthesis model using a Genetic Algorithm*, 2003.

[18] Wikibooks. Sound synthesis theory, December 2011. `http://en.wikibooks.org/wiki/Sound_Synthesis_Theory/Sound_in_the_Digital_Domain`.

[19] Simon Wun, Andrew Horner, and Lydia Ayers. A Comparison between Local Search and Genetic Algorithm Methods for Wavetable Matching. *Compares techniques to select spectral snapshots in wavetable matching of musical instrument tones*, 2005.