Bachelor thesis
Computing Science

Radboud University

GASCAR: Genetically Automated Synthesizer Configuration for Audio Replication

Author:
Hamzah Al Zubi
s1047768

First supervisor/assessor:
Prof. David van Leeuwen
`d.vanleeuwen@science.ru.nl`

Second assessor:
Prof. Djoerd Hiemstra
`hiemstra@cs.ru.nl`

March 26, 2022

Abstract

Digital audio synthesizers revolutionized music production by allowing artists to create almost any sound they desire. However, the robustness of synthesizers often comes at the cost of being complex and difficult to configure. This spawned research attempting to tackle automatic synthesizer configuration to create audio that matches a desired target. One approach of doing this is through the use of genetic algorithms. In this paper we discuss an implementation of a genetic algorithm for this purpose, and test it with a complex modern synthesizer and a diverse set of target audio samples. The results confirm that a genetic algorithm approach is a viable one even for a complex synthesizer, but the performance of the algorithm greatly depends on the nature of the target sounds. Our algorithm is much more successful at replicating synthetic and instrument sound than foley or organic sounds, especially ones that modulate significantly over time or have a lot of background noise. We speculate that replicating these sounds fails because of limitations of the synthesizer we used, as well as biases in our audio similarity measure.

Contents

Chapter 1

Introduction

The emergence of electronic music presents artists with new creative outlets alongside musical composition and arrangement. Electronic music producers have the freedom to create any sound they desire for their music using audio synthesizers. Synthesizers are electronic musical instruments that can be either (partially) physical, or implemented completely in software. They enable the creation of a multitude of sounds by allowing artists to configure a selection of parameters of the audio signal being generated. The use of synthesizers has greatly expanded the variety of sounds present in modern music, far beyond acoustic instruments [1, P. 7]. Audio synthesizers also have uses outside of music. Most notably, they can be used for sound effects in films, video games, and other media [1, P. 273]. Human speech synthesis is also a very prominent field of research that is being used in many modern technologies [2].

There are several types of audio synthesizers (e.g., subtractive, additive, frequency modulation, wavetable, etc.) [3, Ch. 16, 17, 20.3]. Each type has unique capabilities when it comes to sound design. Those types can also be combined to create complex synthesis tools. Using such tools often requires artists to be experienced audio engineers, and to posses advanced knowledge in acoustics.

While the complexity of synthesizers enables the wide range of unique sounds they can produce, it also increases the difficulty of creating a desired sound. A common challenge for audio engineers is creating sounds similar to ones they hear in others' works. Such challenges spawn many conversations in the world of electronic music centered around hypothesizing about the synthesis methods used by popular artists and how to achieve sounds similar to theirs [4]. Synthesizing audio similar to some desired sound gives audio engineers a starting point to freely transpose, customize, and modulate the sound as they desire, instead of only being able to sample the target audio.

Given the complexity of modern synthesizers, and this artistic desire to use them to replicate sounds, there has been some research on automating the process of configuring a synthesizer's parameters in order to match a target sound [5], treating this as an optimization problem. This research showed promising starts for this field, but the problem is far from solved.

In this paper, we show an implementation of an algorithm that tackles the aforementioned challenge. This algorithm, given a synthesizer and a target audio sample, finds a configuration of the parameters of that synthesizer such that the sound it generates is as similar to the target sample as possible. For this implementation, like a lot of previous works, we have used a genetic algorithm to find synthesizer parameters. Genetic algorithms are optimization methods that are inspired by natural selection and biological evolution, which makes them somewhat straightforward to grasp.

To expand on previous similar work, our research investigates the effectiveness of this approach for different types of audio targets using a well-known and commonly used complex synthesizer instead of small examples. We also provide the resulting audio achieved using our algorithm.

To evaluate the efficacy of our algorithm, we have used it to replicate a variety of audio samples we have picked or created. Alongside recording and analyzing the values of a computed similarity measure between the final results and the targets, we also present the results as audio so we can compare them manually to informally study the degree of their perceptual similarity. By implementing and evaluating this algorithm we hope to answer the following question: *how does a genetic algorithm perform for configuring a complex synthesizer's parameters to replicate different types of audio samples?* Our results show that a genetic algorithm approach can be viable for audio replication even when using complex synthesizers. However, we also found that the success of the algorithm varies depending on the nature of the target audio.

Chapter 2

Preliminaries

2.1 Acoustics

Sound is a vibration that propagates as waves. Therefore, any sound can be represented as a *waveform.* For the purpose of analyzing sound, and more specifically, comparing sounds, directly using the waveform is not sufficient. Much more useful information about a sound can be obtained using the field of *Fourier Analysis.* In essence, any sound wave can be approximated sufficiently by representing it as a summation of a set of different sine waves called a *Fourier series.* These sine waves have different frequencies, phases, and amplitudes. This is the basis of the *Fourier transform* [6], which represents any sound wave as a set of sine waves that approximate it when summed together. This way, a waveform is converted into the non-temporal information defining those sine waves, also called *sinusoids* or *harmonics.* **Figure 2.1** shows an example of this approximation for a square wave, and **Figure 2.2** shows the non-temporal representation of the harmonics that make up a square wave as a *spectrum.* The amplitude of the harmonics is often expressed in *Decibel (dB)*, which is a relative and logarithmic unit, i.e., each doubling in Decibels equates to squaring the amplitude. Decibels are used because human perception of sound intensity is approximately logarithmic [7, P. 83].

For long and dynamic sounds, a more useful approach than a simple Fourier transform is segmenting the signal into small time *windows* and applying the Fourier transform on each window. This is known as the *short-time Fourier transform* where the spectrum changes over time [8]. A *spectrogram* is a way of representing the frequencies and intensities of these spectra over time [9]. Essentially, a spectrogram is a two dimensional matrix with a linear axis for frequency ranges and the other linear axis for time windows, and the real values of the matrix correspond to the intensity of a specific frequency band at a specific time interval. The size of this matrix therefore corresponds to the frequency and time resolution of the spectrogram.

A *Mel Spectrogram* is a type of spectrogram that uses the *Mel frequency scale* instead of a linear scale. The Mel scale corresponds more with the non-linear human perception of frequency [10]. A fixed difference measured in Hz is *perceived* as a larger difference *in pitch* in low parts of the frequency spectrum compared to that same difference in higher parts of the spectrum. A Mel spectrogram captures this by grouping frequencies into *Mel bands* that span over a larger range of frequencies in the higher part of the spectrum. The mapping between a frequency on the linear scale and its corresponding Mel band is linear for low frequencies (typically below 1000 Hz) and logarithmic for high frequencies. **Figure 2.3** shows spectrograms with linear and Mel frequency scales representing a square wave sweep (a square wave that is increasing in pitch). For the calculation of our audio similarity measure, we assume that two similar sounds will have similar Mel spectrograms. This is not necessarily perfectly true as some information about the audio can be lost in spectrograms, especially ones with low resolution.

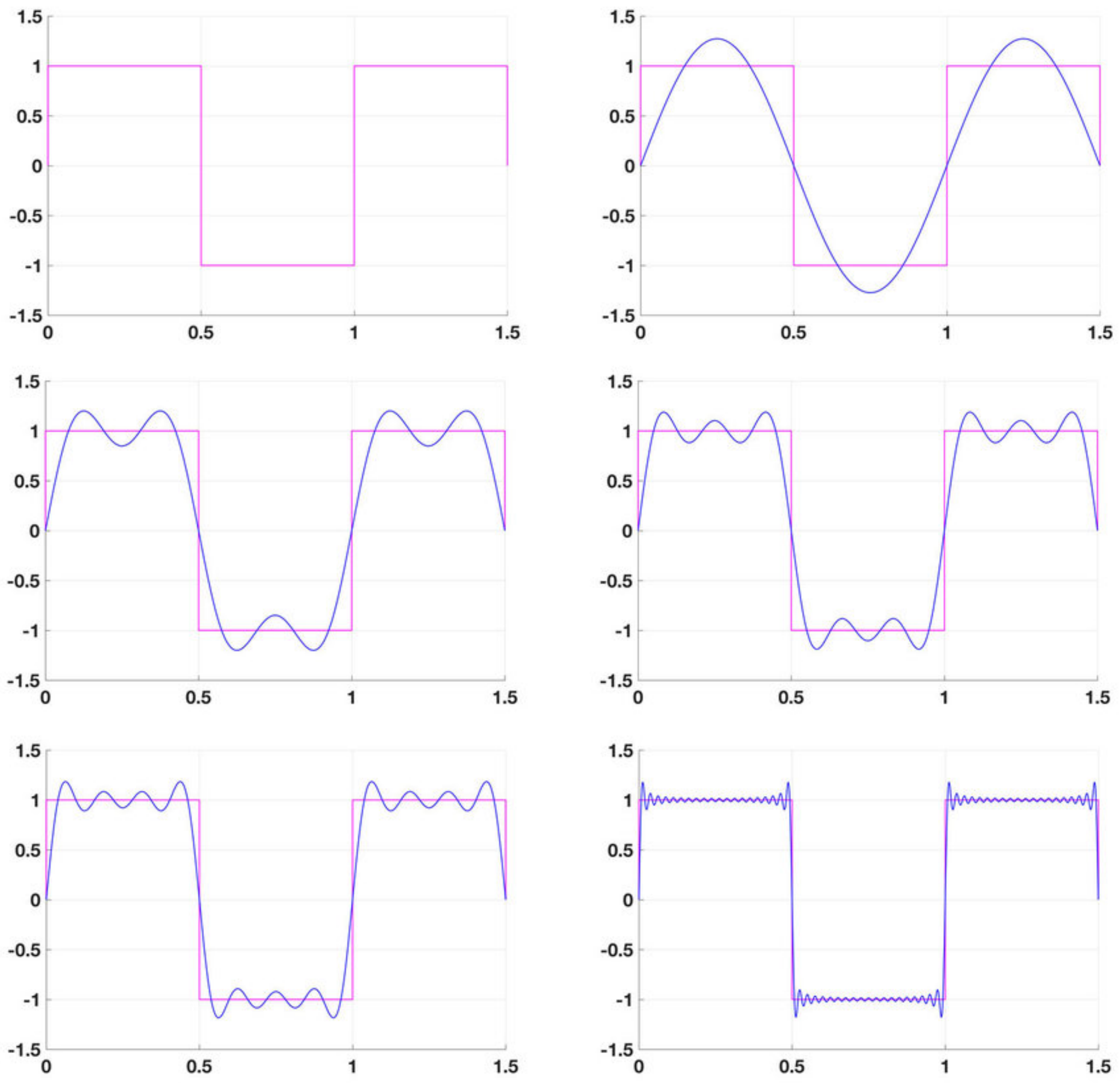


Figure 2.1: A square wave being approximated as a summation of an increasing number of sine waves. [11]

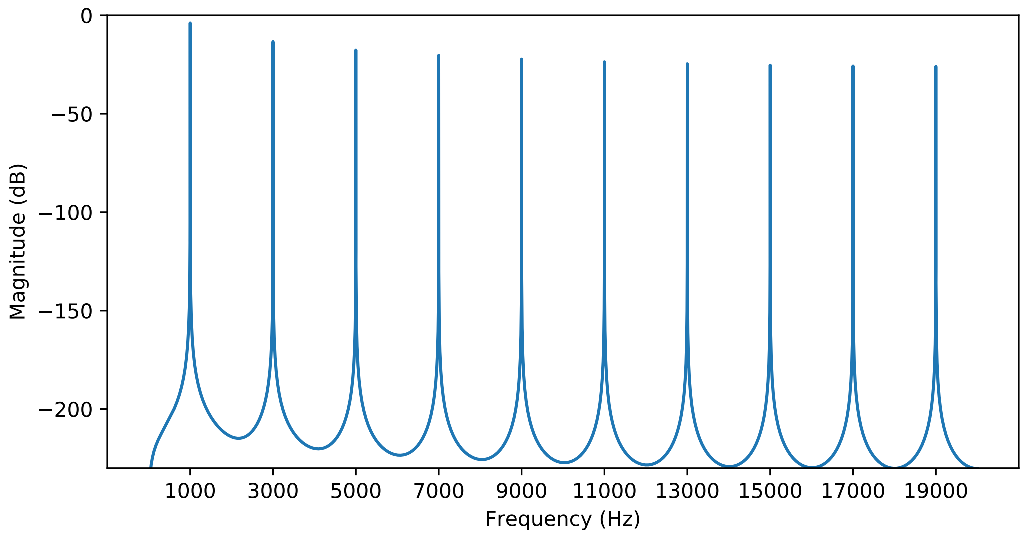


Figure 2.2: The frequencies of the harmonics that add up to a 1000 Hz square wave.

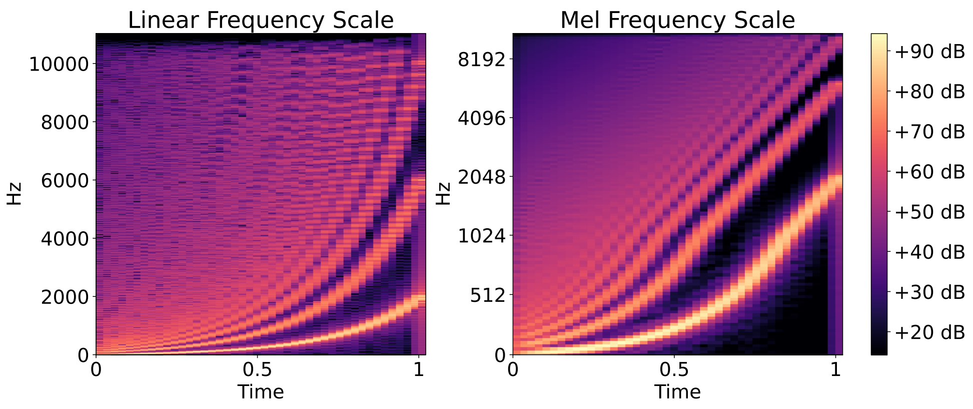


Figure 2.3: Two spectrograms with different frequency scales showing the harmonics of the same audio of a square wave that is increasing linearly in perceptual pitch over time.

2.2 Digital Audio

Our work deals with *digital audio*, which is encoded as a continuous sequence of *samples*[1] [3, Ch. 2.2]. Each sample is simply a number that denotes the amplitude of the audio signal at a certain point. The amount of bits in that number is called the *bit depth*, and it determines the resolution of samples (i.e., higher bit depth allows more possible amplitudes, resulting in higher resolution audio). **Figure 2.4** shows a visualization of sampling. To play or analyze

[1]It is important to note that the term “sample” can be used for two meanings in this field. A sample can refer to one unit of digital audio data, or a full audio clip. In this section, we are referring to the former.

digital audio, a *sampling rate* has to be defined. The sampling rate indicates how many samples one second of audio contains. The Nyquist–Shannon sampling theorem states that a sampling rate of n samples per second is sufficient to represent any signal that contains no frequency higher than $\frac{n}{2}$ [3, Ch. 2.2].

Efficient algorithms exist to perform Fourier analysis on digital audio, and convert digital signals into matrices containing (Mel) spectrograms [8]. We use those algorithms to define audio similarity as a function of the error between two Mel spectrograms.

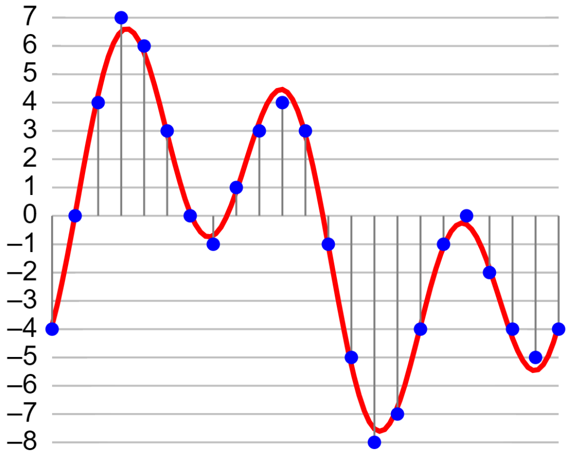


Figure 2.4: A sound wave (in red) represented as digital audio samples (in blue) with a bit depth of 4 bits, hence the 16 possible values for amplitude. [12]

2.3 Digital Audio Synthesis

A digital audio synthesizer is software whose end result is raw audio generated by outputting digital signals in the form of the sample sequences explained in the previous section. Synthesizers can use a wide selection of methods to allow their users to generate this audio. A few examples are:

- *Subtractive* synthesis: synthesizers using this technique have oscillators that generate pre-determined audio waveforms. These synthesizers allow users to modify the output by combining different oscillators and using *filters* that can increase the amplitudes of certain frequencies and decrease others [3, Ch. 17].
- *Additive* synthesis: this technique uses basic waveforms like sine waves as a building block to create more complex waveforms. Additive synthesizers simultaneously produce a large number of those basic waveforms that act as harmonics of the final sound [3, Ch. 16].
- *Frequency modulation (FM)* synthesis: FM synthesizers use waveforms that modulate each other. For example, the oscillation of a sine wave can be used to determine the frequency of a square wave instead of being directly used for audio output. A spectrogram of this example is shown in **Figure 2.5**. Performing FM at high enough frequencies produces complex new sounds [3, Ch. 20].

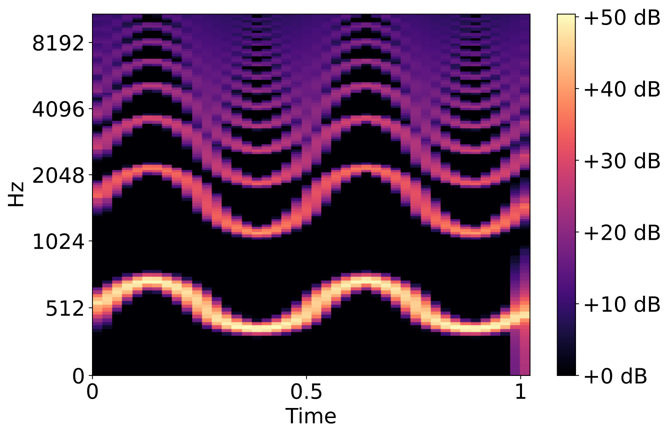


Figure 2.5: Mel spectrogram of a square wave being frequency-modulated by a sine wave at a low frequency.

Some synthesizers combine multiple of those techniques. For example, Sytrus [13] by Image-Line, among other things, is an additive/subtractive FM synthesizer. This means that the waveforms can be generated additively, manipulated subtractively, and then used for frequency modulation. Such complex synthesizers contain numerous parameters to control many qualities of their output. For example, the amplitudes of (groups of) harmonics generated by an additive synthesizer, the filter configurations applied to the signal of a subtractive synthesizer, or various other controls and effects. Many synthesizers have parameters that modify their sound in unique ways as an attempt to innovate and compete with other synthesizers.

In order to replicate a certain given sound using a synthesizer, one must configure the synthesizer's parameters in some way such that it generates audio with maximal similarity to the target sound. Finding the values for those parameters is therefore an optimization problem that can have many dimensions depending on the number of parameters being controlled.

2.4 Genetic Algorithms

Genetic algorithms are optimization methods analogous to natural selection. They operate by creating populations of possible solutions to a problem, evaluating those solutions using some *fitness measure*, and then letting the fittest members of that population create the next population using evolutionary concepts like gene *crossovers* and *mutations* [14].

Naturally, to use a genetic algorithm, one must encode the parameters to be optimized as *genes* and define a fitness measure that the algorithm minimizes or maximizes. In the case of our problem, the genes are simply the values of the parameters of the synthesizer, and the fitness measure is how similar the generated audio is to the target. A genetic algorithm will attempt to maximize this measure by doing the following:

1. Randomly generate an initial population of solutions (with randomized genes).
2. Calculate the fitness of the members of this population using a problem-specific fitness measure (typically a function that maps a member to a (real) number).
3. Create the next population using a *selection* method, and performing crossovers and mutations on the selected members.
4. Repeat steps 2 and 3 until a specified condition is met (e.g., a fixed number of iterations, also known as *generations*, have passed, or a specific fitness value is reached).

Several approaches are possible for selecting members of a population to create the next one. The simplest is just picking the fittest members and duplicating them. However, more sophisticated methods exist for preserving *genetic diversity*. One such method is *tournament selection*. It creates a new population of k members by randomly sampling k subsets known as *tournaments* from the current population and picking the fittest member of each tournament [15]. Usually the *tournament size* is fixed and is specified as a hyperparameter. While this approach helps with maintaining genetic diversity, it can potentially lose the fittest member of a population by not even including it in a tournament in the first place. To counteract this, *elitism* can be used to always carry over a small portion of the best-performing members to the next population.

There are also many methods to perform crossovers and mutations, but most are straightforward. For example, *uniform crossover* simply takes two parents, and sets the genes of the child by choosing each gene from either parent randomly. *k-point crossover* picks k random points in the parents' genes, dividing them into $k + 1$ sections, and then creates children by swapping every other section between parents. It follows that the ordering of the genes is relevant for k-point crossover, but irrelevant for uniform crossover. Most often, synthesizer parameters are ordered such that parameters that control aspects of one effect are grouped together (e.g., a filter's frequency, width, and gain are grouped together). However, the overall ordering is not necessarily coherent. As for mutations, there is usually a chance that a random gene is chosen to be replaced by a random value from its corresponding domain.

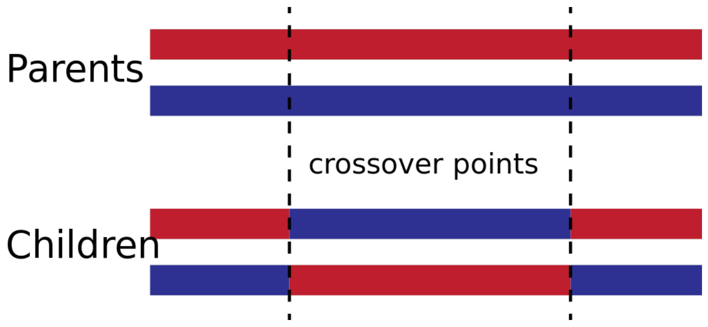


Figure 2.6: A visualization of two-point crossover. [16]

Chapter 3

Related Work

Automatic synthesizer configuration is a field of research that has seen some exploration, but is evolving rather slowly, with beginnings focusing on simplistic FM synthesis. Horner et al. (1993) [17] used a genetic algorithm with a fitness measure involving error between spectrograms, and demonstrated the viability of genetic algorithms for optimizing a small set of parameters of a simple FM synthesizer. Lai et al. (2006) [18] expanded on that work by using a more advanced calculation for the fitness measure that extracts different features of the spectrograms as opposed to simply calculating the error between them.

Recently, there have been works implementing such algorithms in ways that are synthesizer-independent, introducing the ability to use such methods on commercially available synthesizers. Bozkurt & Yuksel (2011) [19] used a genetic algorithm with spectral error as a fitness measure and tested it with FM as well as percussion synthesizers that are custom-built. Tatar et al. (2016) [20] used a similar method on a real-world simple synthesizer with mixed results when it comes to replicating external targets.

A lot of research in this field has been carried out surrounding the creation of a tool called *Synthbot* by Yee-King (2008) [21]. Synthbot is implemented in C++ and can take a user-specified synthesizer of the VST format [22]. However, the full capabilities of this were not explored as it was only tested on a relatively simple synthesizer. Synthbot uses a genetic algorithm as well. Its fitness measure uses the error between Mel Frequency Cepstral Coefficients (MFCCs). MFCCs are features extracted from the Mel spectrograms that capture their content in a reduced form [23]. This differs from previous works that did not use the Mel scale.

Genetic algorithms are not the only methods explored for solving this problem. Heise et al. (2009) [24] used a Particle Swarm Optimization (PSO) approach for optimizing the parameters of a fairly simple synthesizer, also using MFCC error as a fitness measure. PSO differs from the genetic algorithms used in other works because it does not use evolutionary operations such as crossovers and mutations, but instead explores the search space in a more formulaic manner [25]. The aforementioned research of Heise et al. found that this does not present any advantages over using a genetic algorithm. Yee-King & Roth (2011) [5] presented a comparison between using genetic algorithms, neural networks, and hill climbing for synthesizer configuration, and found that genetic algorithms seemed to perform the best.

Similarly to many previous works, we also use a genetic algorithm to configure synthesizer parameters, with a fitness measure involving spectral similarity. However, we can see a few areas where previous work in this field can be expanded:

- Synthesizer complexity: most previous work is tested with limited and simple synthesizers with few parameters (less than 30). In this research, we use a powerful and complex commercial synthesizer with 92 parameters.
- Samples: most works use a limited set of target sounds, and rarely use real-world sounds. We test our approach with synthetic and non-synthetic audio and analyze differences in the performance of our algorithm for different categories of sound.
- Fitness measure: previous work uses differences between spectrograms or features extracted from Mel spectrograms as a fitness measure. In this work we experiment with using difference between Mel spectrograms directly.
- Results: the vast majority of previous research does not provide the audio results of their experiments, and instead simply shows spectrograms or waveforms of some runs. This makes it difficult to perceptually compare the success of the algorithms used. In this work, we focus on presenting results as audio comparisons.

Chapter 4

Research

4.1 Data Set

We have created an audio data set specifically for this research. This data set includes four different sound categories containing 25 samples each. The main focus when creating the data set was diversity of sounds. The categories are: synthetic instruments, acoustic instruments, animal sounds, and miscellaneous sound effects. All audio in our data set is either created using different synthesizers or collected from free-to-use sources such as *Freesound* [26]. Some samples are mono and others are stereo, but our algorithm loads them as mono audio. All samples are 1 second long and are of the uncompressed *wav* file format with a sampling rate of 44100. Each sample is also labeled with the note that the synthesizer should play when replicating it. This eliminates the need to search for this note and focuses the algorithm on matching *timber* as opposed to pitch. However, this has the downside of not being descriptive enough when the sound has unclear or changing pitch. This limitation is mitigated when the synthesizer being used already has a parameter that has some control over the pitch.

4.2 Implementation Details

For this work, it is necessary to be able to interface with software synthesizers. Several API standards exist for synthesizer formats (VST, AU, AAX, etc.) but most of them work by allowing another program (the so called *host*) to dynamically load them and call functions they define. We used the JUCE framework to create a very minimal host using C++ that can load most commercial synthesizers. JUCE provides a single interface that can interact with most synthesizer formats [27]. Our host implementation uses functions provided by JUCE to allow us to load a synthesizer, set its parameters, and record a sample of the audio generated using those parameters. We have wrapped this host program in Python code and implemented it as a dynamic Python library using `Boost.Python` [28]. This allows for using the many Python packages that facilitate implementing our algorithm, most notably including Librosa [29] for audio analysis and DEAP [30] for implementing a genetic algorithm.

All of our experiments were performed using the Image-Line Harmless synthesizer. Harmless is a powerful additive/subtractive synthesizer with many unique effects. The graphical user interface of Harmless is shown in **Figure 4.1**. This is the interface one would interact with to use the synthesizer and adjust its parameters. In-depth documentation of each of the parameters in Harmless can be found in [31]. Our host implementation does not load the graphical interface of the synthesizer and instead adjusts the parameters using function calls. Our implementation also allows generating audio signals much faster than real-time synthesis. i.e., a 1-second audio sample would take a few milliseconds to generate and store.

The genetic algorithm was implemented using the DEAP Python framework [30]. The implementation records a "history" of data for facilitating analysis. This includes the genes of the fittest member of each generation along with their fitness measure. Encoding synthesizer parameters as genes is trivial since they are all represented as real numbers between 0 and 1 already. Therefore, an individual in the population is simply a list of real numbers corresponding to the synthesizer parameter values.

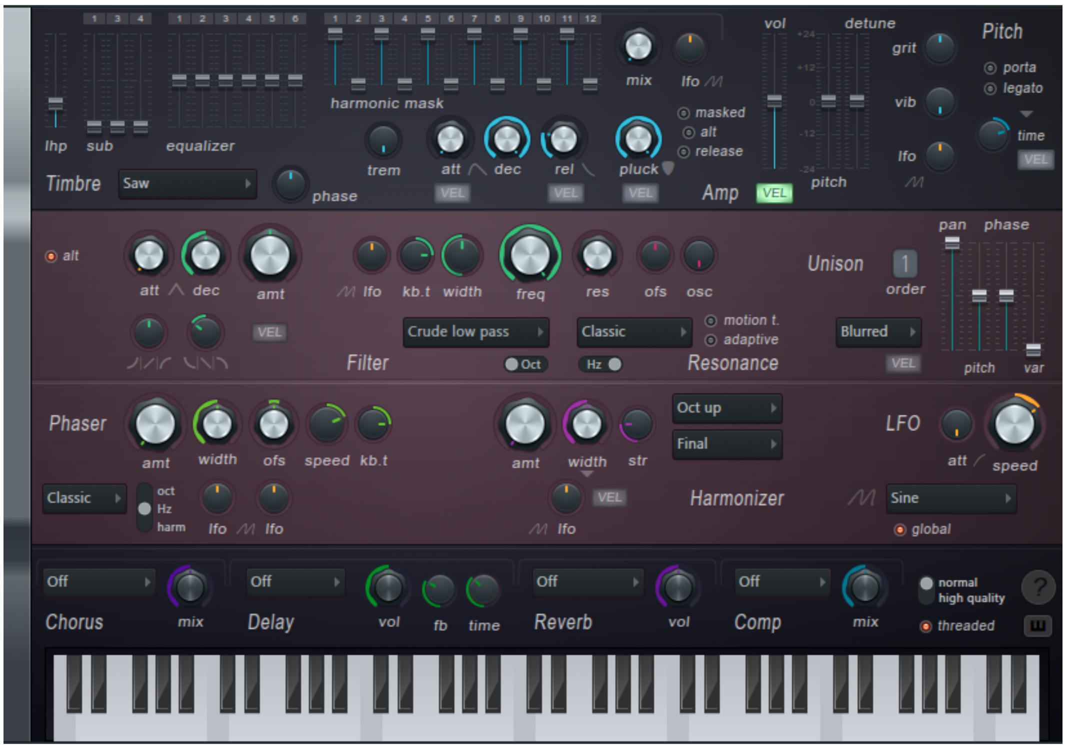


Figure 4.1: The graphical interface of the Harmless synthesizer.

4.3 Fitness Measure

The purpose of our algorithm is genetically evolving a set of synthesizer parameters in order to create sound that replicates some target audio. Therefore, the fitness measure of a member in this genetic algorithm naturally has to correlate with the similarity between the target audio sample and the resulting audio. It is difficult to define the concept of audio similarity, let alone mapping that concept to one numerical value. The approach used in this work is maximizing the similarity by minimizing the difference (or *error*) between the matrices representing the Mel spectra of the two audio samples. Several variations of this calculation can be used, including the type of audio pre-processing performed (if any), and the type of error function (*mean squared error*, *mean absolute error*, etc.).

The procedure for calculating our fitness measure is as follows:

1. Convert the raw audio signals of the target audio sample and the resulting audio sample into Mel spectrograms with a logarithmic intensity (dB) scale. For the exact parameters and code used in this conversion, see **Appendix A**.
2. Calculate the distance $d = \frac{\overline{|target - result|}}{\max(\overline{target}, \overline{result})}$ where $\overline{|target - result|}$ is the mean absolute error between the two spectrograms, and $\max(\overline{target}, \overline{result})$ is the larger mean of all values in each spectrogram. The intuition behind this calculation is that the error is taken as a *proportion* of the total energy. For instance, a change in one harmonic has a much larger perceived impact on the similarity between two sounds that only contain one harmonic, as opposed to sounds that contain a large number of harmonics.
3. Finally, the fitness measure (to be maximized) is $s = 1 - d$. A value of $s = 1$ indicates complete similarity, $s = 0.5$ indicates that only half of the values match in the two spectrograms, and so on.

To establish confidence in the validity and consistency of the similarity measure we use, we tested it by computing it for a few trivial cases. The first case is testing with identity. In this test we simply compute the similarity between an audio sample and a copy of it using the procedure described above. The expectation here is that the result will always be equal to 1 or 100%. This is indeed the case with all samples in our data set. In the second case we compute the similarity between audio samples and silence (which is simply represented as a spectrogram of exclusively zeros). An objective expected result is easily debatable and difficult to conceptualize, and the resulting computation will strongly vary depending on the type of measure being used. For our similarity measure, the expectation is that the similarity between any audible sound and silence is 0. This holds for all samples in our data set. Finally, **Figure 4.2** demonstrates a simple and sensible comparison between two sounds.

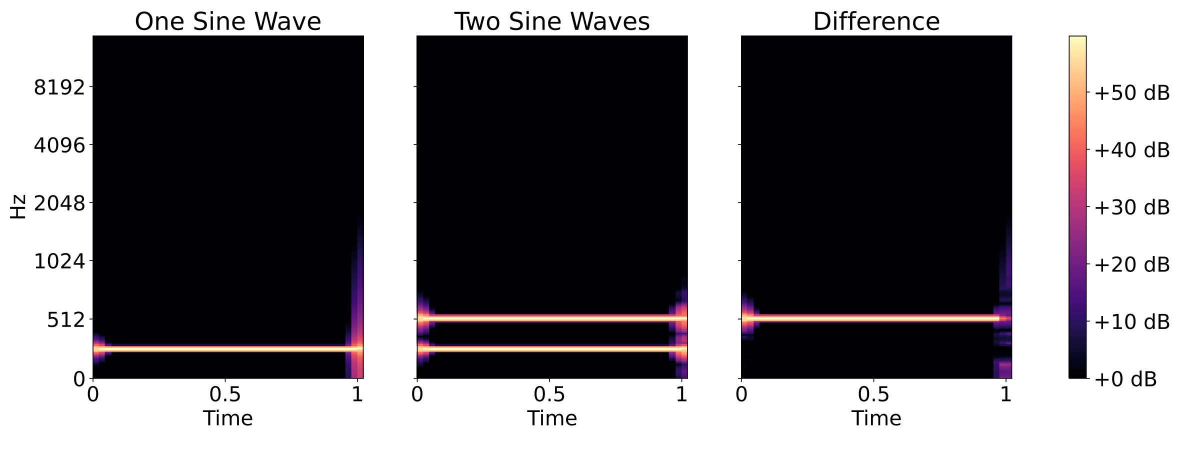


Figure 4.2: Mel spectrograms of one sine wave (left), two sine waves (middle), and the absolute difference between them (right). The computed similarity measure according to the method described above is approximately 0.5 as expected.

4.4 Algorithm Details

Our algorithm follows the general structure of genetic algorithms given in **2.4**, and uses hyperparameters that were found by some preliminary experimentation. The full details are as follows:

1. Load the raw target audio from a wav file with a sampling rate of 44100 and compute its Mel spectrogram as described in **Appendix A**. This Mel spectrogram is stored for use throughout the run of the algorithm without having to recompute it.
2. Load the given synthesizer. In the case of our experiments, the synthesizer is the VST version of IL Harmless 1.0.
3. Create a population of 1000 individuals with randomized genes. In other words, create 1000 lists of random values for the synthesizer's parameters.
4. Compute the fitness for all members of the population. This is done by computing the audio similarity between each one of them and the target audio. This computation is done as described in **4.3**.
5. Form the next population of 1000 members as follows:
   (a) Use tournament selection with a tournament size of 10 to select members from the current population 940 times. The tournaments are created by choosing members from the previous population randomly and with replacement.
   (b) Take the 10 members of the current population with the highest fitness as the elite members.
   (c) Generate 50 new random members to introduce genetic diversity.
6. Randomly partition the new population (except the elites) into pairs. Then, each pair has a 50% chance to exchange genes using uniform crossover.
7. Mutate all members of the new population. Each gene of a member being mutated can be replaced with a new random value between 0 and 1 with a probability of 3%.
8. Repeat from 4. until this is done 100 times.
9. The final output is the individual with the highest fitness measure encountered during the execution of the algorithm.

4.5 Experiments

We conduct two types of experiments to evaluate our algorithm. The first involves using the Harmless synthesizer to replicate 1-second audio samples that were also synthesized using Harmless with completely randomized parameters. The second set of experiments involves using the synthesizer to replicate audio samples in our data set, which are not made using Harmless.

For those experiments, the runtime of the algorithm using an Intel i7 8700k processor is around 45 minutes per sample. The time is roughly equally split between using the synthesizer to generate audio and converting this audio to a Mel spectrogram. Obviously, the runtime is dependent on many of the algorithm's inputs and hyperparameters, such as synthesizer used, sample duration, spectrogram resolution, etc.

4.5.1 Experiment with Randomized Targets

In this experiment, we simply generated 25 sets of randomized parameters, and used those parameters to generate 25 target audio samples that are one second long. Note these samples are separate from the data set described in 4.1. We ran our algorithm on each of these randomized audio samples. Doing this allows us to judge the algorithm's performance knowing that perfect replication is possible. This validates the method being used and also forms a baseline of performance for later comparisons.

Figure 4.3 shows the convergence of the genetic algorithm for randomized samples. It is clear that the algorithm successfully increases the fitness measure over the course of the 100 generations. Although, the majority of that increase happens in the first 50 generations. The average fitness value of all results at the 100th generation is approximately 0.79 with a standard deviation of 0.09. **Figure 4.4** shows the distribution of the fitness measure across the 25 samples replicated in this experiment.

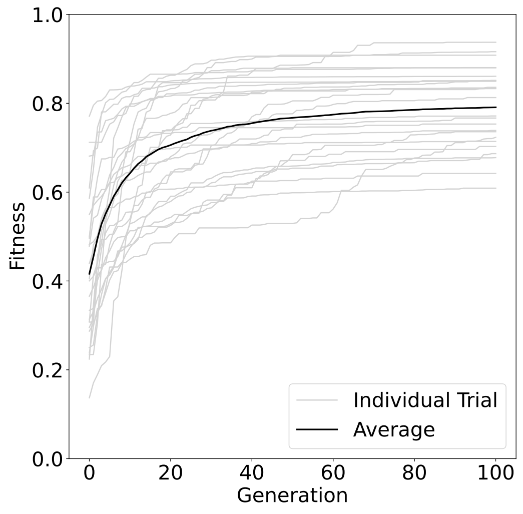


Figure 4.3: The convergence of the fitness measure in the 25 trials of replicating randomized samples over the generations of the genetic algorithm.

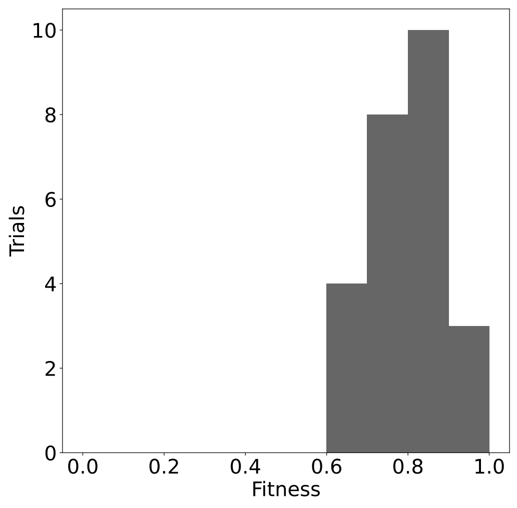
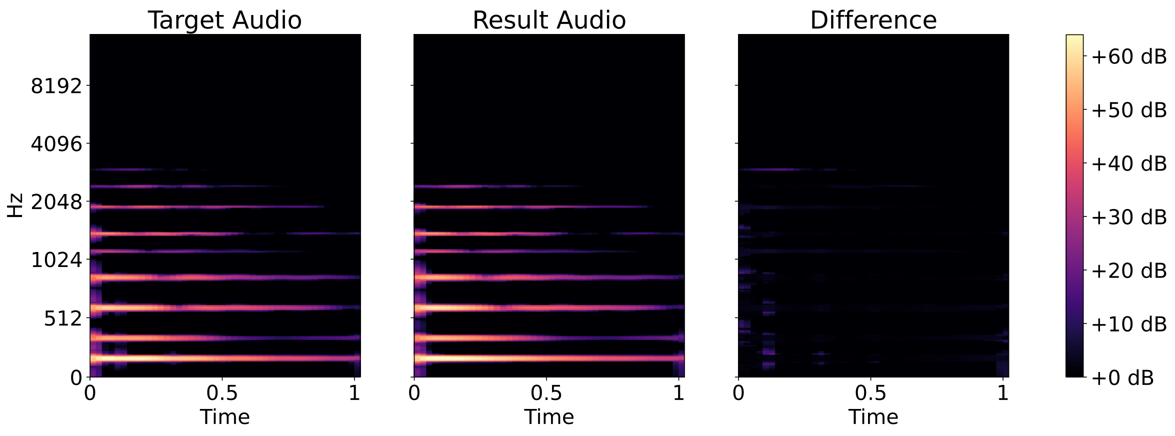


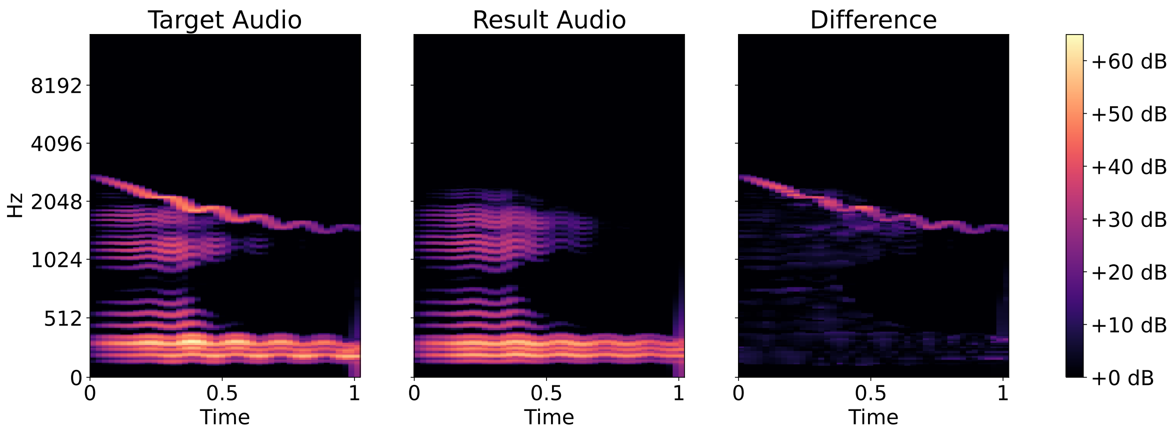
Figure 4.4: Histogram showing the distribution of the fitness measure in the 25 trials of replicating randomized samples.

Figure 4.5 shows comparisons of spectrograms belonging to the trials of this experiment with the best, median, and worst fitness. We can see that the algorithm succeeds in matching the general shape of the spectrograms of those samples, including some changes over time. However, it is clear that replicating finer details in the original sample is not always successful.

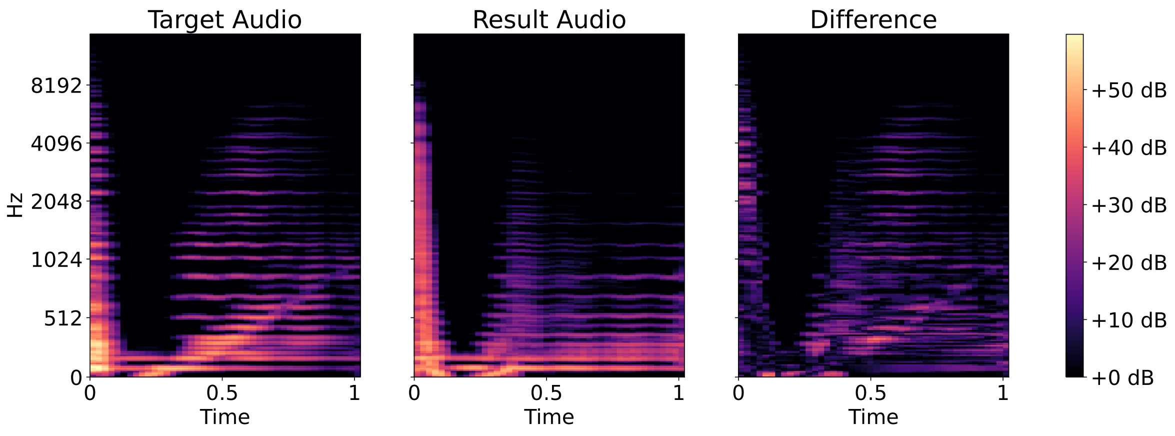
A full compilation of the target and resulting sounds is available in [32]. There are 25 pairs of consecutive samples in the compilation, one for each run of the algorithm. The first sample of each pair is the target audio, and the second is the result of the algorithm. We observe that almost all resulting sounds from this experiment are perceptually similar to the corresponding targets.



(a) Sample with the highest fitness.



(b) Sample with the median fitness.



(c) Sample with the worst fitness.

Figure 4.5: Target (left) and result (middle) audio Mel spectrograms as well as the difference between them (right) for runs of the algorithm replicating randomized targets with the highest, median, and worst fitness values.

4.5.2 Experiments with Categorized Data Set

In these experiments, we run our algorithm with samples from our data set as targets. We analyze the results separately for each category in our data set, and then compare them to assess the utility of such an algorithm for different tasks. We expect that the success of the algorithm will vary between the categories because the synthesizer we use, like most synthesizers, has a finite space of possible sounds that it can create.

Figure 4.7 shows the convergence of the genetic algorithm for the four categories, **Figure 4.8** shows histograms of the final fitness values, and **Table 4.1** includes the average resulting fitness values for each of the experiments (including randomized targets). On average, it can be observed that the algorithm is most successful at replicating randomized targets. Furthermore, the performance when replicating synthetic sounds or acoustic instruments is considerably higher than when replicating animal sounds or other sound effects.

A full compilation of the target and resulting sounds is available in [32]. We can perceptually observe clear differences between data sets in a way that confirms the findings described above. We can also observe that the algorithm succeeds with relatively stable sounds that are not very noisy, which applies to most acoustic instruments and pure synthetic tones. On the contrary, the algorithm is more likely to fail when replicating sounds that change over time (either in pitch or in timber). Based on manual inspection, we speculate that this is mostly due to limitations in the synthesizer we use. The algorithm also does not perform well when replicating audio samples that have a lot of background noise. It appears to attempt to replicate this background noise as well, so the fitness measure being used is likely exaggerating the role of noise in the perceptual qualities of the sound. An example spectrogram comparison for a modulating sound is shown in **Figure 4.6**. The algorithm clearly replicates the main harmonics of the sound, but fails to replicate the volume modulation in its entirety.

Random	Synth	Acoustic	Animal	Effect
0.79 ± 0.09	0.76 ± 0.14	0.70 ± 0.1	0.58 ± 0.12	0.59 ± 0.14

Table 4.1: Mean and standard deviation of the fitness measure of the results across the different data sets.

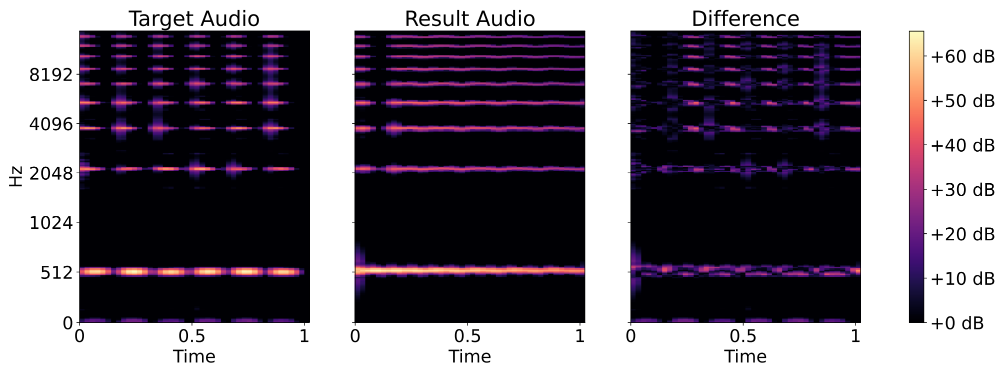
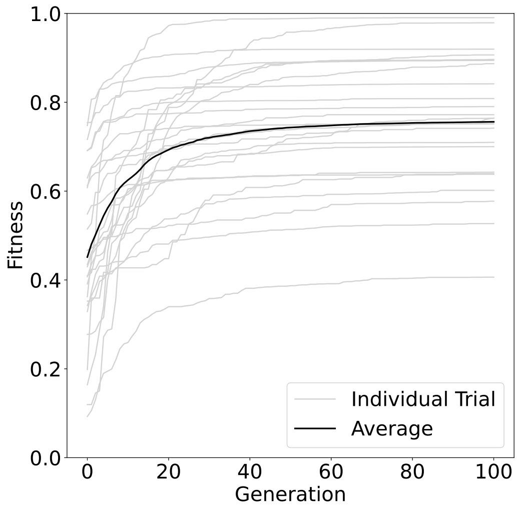
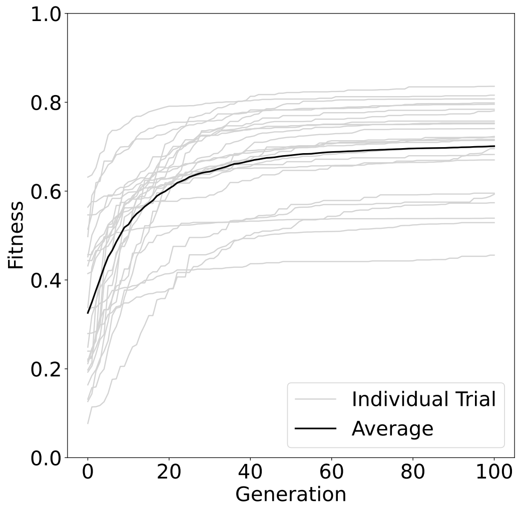


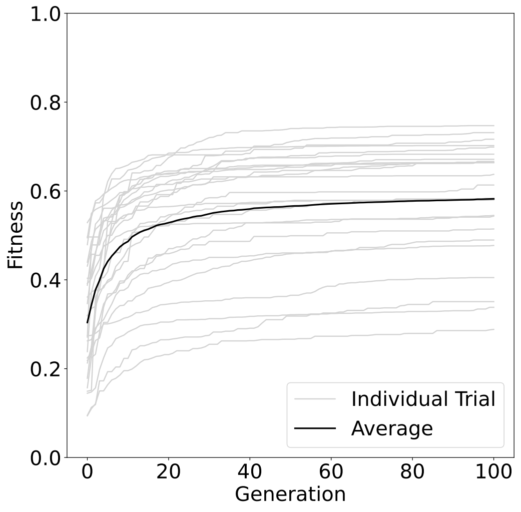
Figure 4.6: Mel spectrogram comparison of replication of a changing sound (modulating in volume).



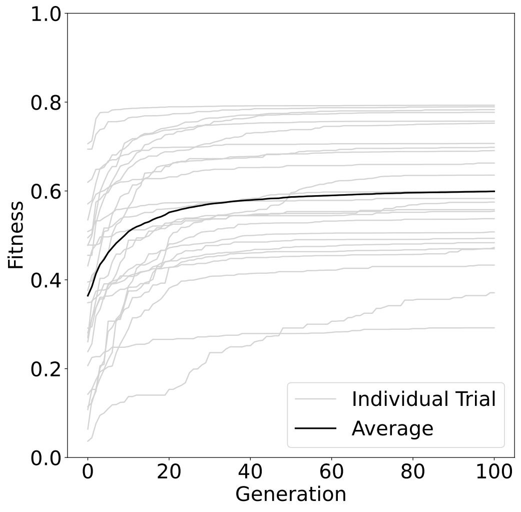
(a) Convergence with synthetic samples.



(b) Convergence with acoustic samples.

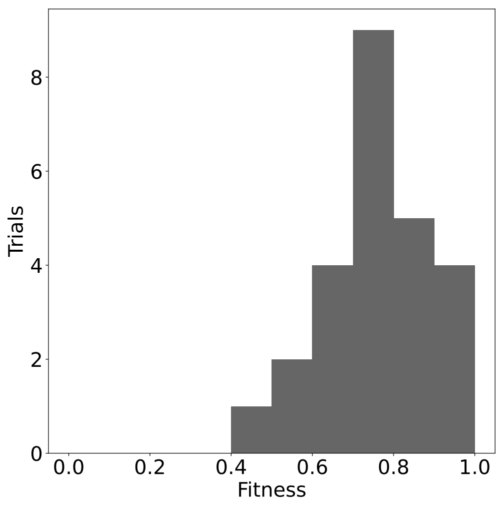


(c) Convergence with animal samples

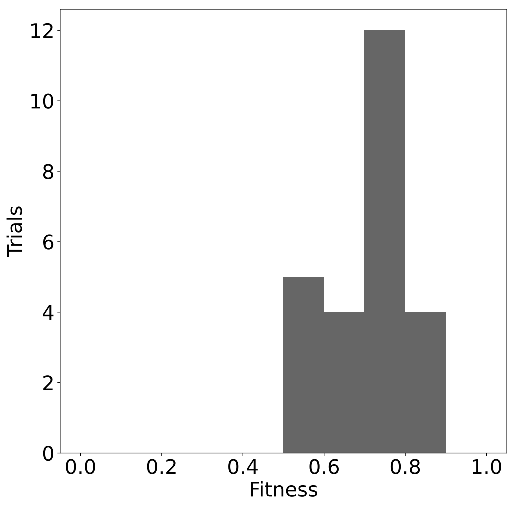


(d) Convergence with effect samples.

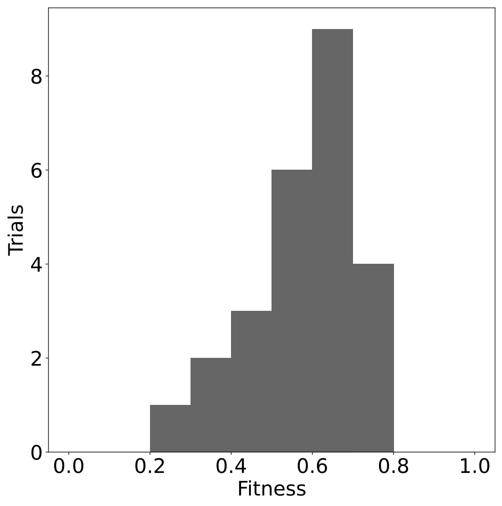
Figure 4.7: The convergence of the fitness measure in the trials of replicating samples of different categories over the generations of the genetic algorithm.



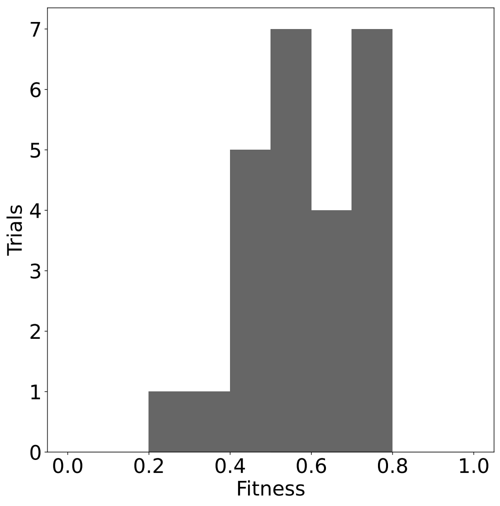
(a) Synthetic samples.



(b) Acoustic samples.



(c) Animal samples



(d) Effect samples.

Figure 4.8: Histograms showing the distributions of the fitness measure in the 25 trials of replicating each data set.

Chapter 5

Conclusions

We showed an implementation of a genetic algorithm approach for tackling the problem of automatic synthesizer configuration for replicating target audio samples. We used the error between the Mel spectrograms of the target and resulting audio samples for computing the fitness measure for our genetic algorithm. We expanded on previous work by experimenting with a complex modern software synthesizer, using a data set comprised of a diverse set of sounds in addition to randomly generated targets, and sharing the resulting audio samples. The results show that a genetic algorithm approach is viable even as the complexity of the synthesizer being used increases. However, it is clear that the algorithm struggles with organic sounds, or sounds that change significantly over time.

We think future work could benefit from a large-scale survey to ensure that the audio similarity measure sufficiently approximates perceptual similarity. Furthermore, algorithms like ours involve a very large number of hyperparameters (including hyperparameters of the genetic algorithm optimizer, and also the many variables used for computing the similarity measure). So far there has not been much research exploring tuning genetic algorithm hyperparameters for this purpose. We think hyperparameter optimization can be greatly beneficial going forward and thus is a worthwhile direction for future research.

Bibliography

[1] Trevor Pinch and Frank Trocco. *Analog days: The invention and impact of the Moog synthesizer.* Harvard University Press, 2004.

[2] John Holmes and Wendy Holmes. *Speech synthesis and recognition.* CRC press, 2002.

[3] David Creasey. *Audio processes: musical analysis, modification, synthesis, and control.* Taylor & Francis, 2016.

[4] "How do I make this sound?" Reddit threads. `https://www.reddit.com/r/edmproduction/search?q=how+do+i+make+this+sound%3F&restrict_sr=on&sort=relevance&t=all`. Accessed: 2022-03-18.

[5] Martin Roth and Matthew Yee-King. A comparison of parametric optimization techniques for musical instrument tone matching. In *Audio Engineering Society Convention 130.* Audio Engineering Society, 2011.

[6] Kenneth B Howell. *Principles of Fourier analysis.* CRC Press, 2016.

[7] George Mather. *Foundations of perception.* Taylor & Francis, 2006.

[8] Ervin Sejdić, Igor Djurović, and Jin Jiang. Time–frequency feature representation using energy concentration: An overview of recent advances. *Digital signal processing*, 19(1):153–183, 2009.

[9] Leon Cohen. *Time-frequency analysis*, volume 778. Prentice hall New Jersey, 1995.

[10] Stanley Smith Stevens, John Volkmann, and Edwin Broomell Newman. A scale for the measurement of the psychological magnitude pitch. *The journal of the acoustical society of america*, 8(3):185–190, 1937.

[11] MT Caccamo and S Magazù. Variable length pendulum analyzed by a comparative fourier and wavelet approach. *Revista mexicana de física E*, 64(1):81–86, 2018.

[12] File:4-bit-linear-pcm.svg — wikimedia commons, the free media repository. `https://commons.wikimedia.org/w/index.php?title=File:4-bit-linear-PCM.svg&oldid=570770731`, 2021. Accessed: 2022-03-18.

[13] Sytrus reference manual. `https://www.image-line.com/fl-studio-learning/fl-studio-online-manual/html/plugins/Sytrus.htm`. Accessed: 2022-03-18.

[14] Melanie Mitchell. *An introduction to genetic algorithms.* MIT press, 1998.

[15] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[16] File:twopointcrossover.svg — wikimedia commons, the free media repository. `https://commons.wikimedia.org/w/index.php?title=File:TwoPointCrossover.svg&oldid=496964021`, 2020. Accessed: 2022-03-18.

[17] Andrew Horner, James Beauchamp, and Lippold Haken. Machine tongues xvi: Genetic algorithms and their application to fm matching synthesis. *Computer Music Journal*, 17(4):17–29, 1993.

[18] Yuyo Lai, Shyh-Kang Jeng, Der-Tzung Liu, and Yo-Chung Liu. Automated optimization of parameters for fm sound synthesis with genetic algorithms. In *International Workshop on Computer Music and Audio Technology*, page 205, 2006.

[19] Batuhan Bozkurt and Kamer Ali Yüksel. Parallel evolutionary optimization of digital sound synthesis parameters. In *European Conference on the Applications of Evolutionary Computation*, pages 194–203. Springer, 2011.

[20] Kıvanç Tatar, Matthieu Macret, and Philippe Pasquier. Automatic synthesizer preset generation with presetgen. *Journal of New Music Research*, 45(2):124–144, 2016.

[21] Matthew Yee-King and Martin Roth. Synthbot: An unsupervised software synthesizer programmer. In *ICMC*, 2008.

[22] VST: Seamless integration for virtual instruments and effects. `https://www.steinberg.net/technology/`. Accessed: 2022-03-25.

[23] Md Sahidullah and Goutam Saha. Design, analysis and experimental evaluation of block based transformation in MFCC computation for speaker recognition. *Speech communication*, 54(4):543–565, 2012.

[24] Sebastian Heise, Michael Hlatky, and Jörn Loviscach. Automatic cloning of recorded sounds by software synthesizers. In *Audio Engineering Society Convention 127*. Audio Engineering Society, 2009.

[25] Mohammad Reza Bonyadi and Zbigniew Michalewicz. Particle swarm optimization for single objective continuous space problems: a review. *Evolutionary computation*, 25(1):1–54, 2017.

[26] Freesound.org. `https://freesound.org/`. Accessed: 2022-03-18.

[27] JUCE framework repository. `https://github.com/juce-framework/JUCE`. Accessed: 2022-03-18.

[28] Boost.Python documentation. `https://www.boost.org/doc/libs/1_71_0/libs/python/doc/html/index.html`. Accessed: 2022-03-18.

[29] Librosa package repository. `https://github.com/librosa/librosa`. Accessed: 2022-03-25.

[30] DEAP repository. `https://github.com/DEAP/deap`. Accessed: 2022-03-18.

[31] Harmless reference manual. `https://www.image-line.com/fl-studio-learning/fl-studio-online-manual/html/plugins/Harmless.htm`. Accessed: 2022-03-18.

[32] Experiment Results. `https://drive.google.com/drive/folders/15guhpEBTnfMVqkx9l6v7iYh4SwHP-Qak?usp=sharing`.

Appendix A

Fitness Measure Calculation

The calculation of our fitness measure is done with the help of the Librosa package for Python, which provides straightforward functions for performing complex audio analysis.

Given two raw audio signals as Numpy arrays, we first get the Mel power spectrograms of each as follows.

```
melS = librosa.feature.melspectrogram(
                                        y=audio,
                                        sr=44100,
                                        n_mels=256,
                                        hop_length=1024,
                                        window=scipy.signal.windows.kaiser(M=8192,
                                            beta=20),
                                        n_fft=8192,
                                        fmin=0,
                                        fmax=15000
                                        )
```

Then, we convert the resulting power spectrograms to Decibel units. We specify a minimum threshold to get rid of any low-level noise and isolate the audible harmonic content of the samples. We use that same threshold as the reference zero-point.

```
melS = librosa.power_to_db(melS, ref=10**-2,amin=10**-2)
```

Finally, we calculate the fitness measure as described in **4.3**.

```
similarity = 1 - (abs(melS1 - melS2).mean())/max(melS1.mean(), melS2.mean())
```