Imperial College London

Final Year Project

Sound Resynthesis with a Genetic Algorithm

Author:
Adam Johnson

Supervisor:
Dr. Iain Phillips

17th June, 2011

Abstract

In music production, software synthesizers are used to create many of the sounds heard in songs today. They present the user with many different parameters to control the sound they produce. In this project we use a Genetic Algorithm to optimize a synthesizer model to match an input sound file, in order to automate the programming of a synthesizer.

Acknowledgements

First and foremost I would like to thank Dr Iain Phillips for supervising me from the start, supporting me throughout and helping tune my ideas into a workable project. I also offer my gratitude to Professor Murray Shanahan for being my second marker.

I would also like to thank my Personal Tutor Professor Abbas Edalat for reading through and providing feedback on a draft copy of my project, and for supporting me throughout my years at Imperial.

Also thanks Dad for doing what you've always done for me – reading through the project with true interest and giving your own electrical-engineer-project-manager opinion. It's great.

Lastly, I would like to thank all my friends from Imperial.

Contents

Chapter 1

Introduction

In the world of Music production, synthesizers have become a mainstay for creating sounds ever since the commercial release of the Moog in 1967. With the growing market of software synthesizers there is a staggering number of synthesis techniques being explored, giving a wide array of possible sounds.

For a producer aiming to synthesize a particular sound, the programming of each option and parameter can take a very long time. Additionally the producer must have deep knowledge about how the synthesizer works, which can be very hard to comprehend in a world of so many models. There is even a market for pre-programmed synthesizer configurations, known as presets [1].

The problem this project seeks to solve is the automatic programming of a software synthesizer to match a recorded sound. We achieve this using a genetic algorithm to optimize the many parameters in parallel. Our motivation is twofold:

- **The reproduction of sounds** – approximating a real life sound with a synthesizer to avoid the painstaking sampling process. We aim to emulate both real world instruments and other synthesizers. By recreating a sound we can resynthesize it at other pitches, so even if only one note exists as a sample, we can recreate it across the entire desired pitch range.
- **The exploration of sounds** – traversing the potential soundscape of a synthesizer in a partially directed manner. In this case the algorithm does not need to provide a 100% match to the target sound to give a useful output – partial matches are still interesting sonic material for a music producer to work with, as they may have some features from the real world instruments whilst sounding new and fresh.

We build upon previous works that use frequency modulation synthesis to match sounds and explore further with more features typically found in a synthesizer. Particularly, previous works investigated only matching a steady-state sound, whilst we are interested in now matching a sound's evolution over time.

1.1 Contributions

In this report we complement the state of the art in sound resynthesis with the following contributions:

- We apply the GA technique to reproduce sounds that evolve over time, where earlier systems did not
- We present some problems encountered in the system's development and describe how they were overcome
- We show two techniques, normalization and regulatory genes, that improve the speed at which sounds can be matched in such a system
- We optimize a fitness function used in previous literature by parametrizing and then optimizing it
- We evaluate our system and show its limitations, and make suggestions towards future work that could result in a system that could 'match any sound'

1.2 Report Outline

We start with the background in Chapter 2, detailing the state of the art in both synthesis techniques and previous work towards resynthesizing using a genetic algorithm.

Chapter 3 then follows, presenting an overview of our entire system. We expand upon this in detail in both the synthesizer and genetic algorithm parts in Chapters 4 and 5 respectively, also covering implementation problems that were solved.

In 6 we present the results of our experimentation to optimize the system and quantify improvements we have made over past systems.

We provide an evaluation of the system in Chapter 7 and finish with a conclusion in Chapter 8, with suggestions of further work to be undertaken.

A glossary is provided in Appendix 1.

Chapter 2

Background

In this Chapter we detail some necessary background knowledge from the two key areas – synthesizers and genetic algorithms. We then outline three previous works that use a GA to optimize a synthesizer for resynthesis and discuss their drawbacks to justify our motivation.

2.1 Synthesis techniques

The world of synthesizers has exploded in terms of technology ever since the 'Moog' became commercially available in 1964 [9]. Many different techniques have been explored, and the digital realm has offered consideration of a wide range of models beyond those possible in analogue. We present here a few techniques that are necessary to understand this project.

The most basic model of synthesis is an oscillator that generates a repeating waveform – sine, sawtooth and square waves are the easiest to generate. For a sine wave, the level of the signal x at time t can be expressed as $x(t) = A \sin(\omega t)$, where A is the amplitude and ω the frequency. Most synthesis models will use one or more oscillators as starting points.

Additive Synthesis is based upon the concept of the Fourier series, which states that any sound can be represented by the summation of only sine waves, or partials. Therefore additive synthesis focusses on creating a sound 'from the ground up', adding together sine wave partials at their respective amplitudes until the desired timbre is achieved. In theory, an exact match of any sound can be achieved, if enough partials are used. This is similar to the way an organ works with stops, each stop representing the addition of extra partials. An example is shown in Figure 2.1.

As real-world tones tend to vary their harmonic content over time, each partial may have its own amplitude envelope, changing its volume over time from the onset of the note. Any sound can be represented this way – a sine wave requires just one partial, whilst white noise would be the addition of infinite partials at all frequencies.

Subtractive Synthesis is an alternative to additive synthesis, in which a complex waveform is taken and then filtered to remove some of its partials. An example of this would be feeding a sawtooth generator through a low pass filter to emulate a stringed instrument, as in Figure 2.2. The perfect sawtooth wave is an infinite summation of the harmonic

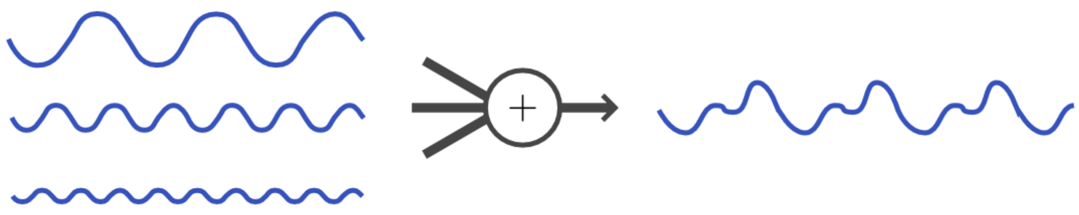


Figure 2.1: Generating a tone by adding 3 partials together

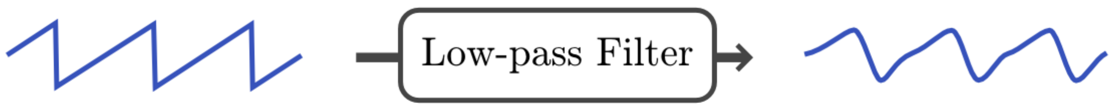


Figure 2.2: Sawtooth wave with upper partials reduced by a low pass filter

series, whilst a stringed instrument is physically limited to which partials are present. Using a low pass filter like this dampens the upper partials and removes some of the 'synthetic' qualities of the sound.

Often this idea of filtering the sound path is used in combination with other synthesis techniques that generate complex waveforms, removing frequency content that is unpleasant/unrealistic to 'smooth over' the sound. There are thus many different synthesis techniques that can be combined with subtractive synthesis in this way.

Frequency Modulation Synthesis is based upon the combination of two waveforms by setting one to frequency-modulate the other, such as in Figure 2.3. This produces complex extra harmonics with a sidebanding effect.

For one sine wave frequency-modulated by another, the sample at time t is given [3] as:

$$x(t) = A\sin[\alpha t + I_1 \sin(\omega_1 t)] \tag{2.1}$$

For the modulator sine wave there are two parameters; its **index** I_1 and **ratio** ω_1. FM

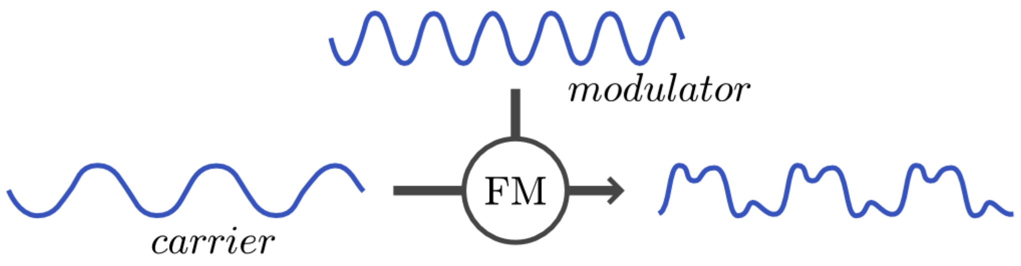


Figure 2.3: Carrier sine frequency-modulated by sine. For equation 2.1, $I_1 = 1$ and $\omega_1 = 2\alpha$

is efficient for real time synthesis of complex spectra. Unlike additive synthesis, only a few sine functions are required to produce many partials. Research in the Audio Engineering Society in 1994 states that “FM has deservedly become recognized as one of the most efficient methods of digitally generating musical sounds with rich harmonic components” [15].

Dual Frequency Modulation, or *DFM*, is a particular application of Frequency Modulation. A 0Hz carrier sine is modulated by two different sine waves with individual frequency ratios and modulation indices.

The DFM formula can be obtained from a simplification of the FM formula for two modulators:

$$x(t) = A\sin[I_1\sin(\omega_1 t) + I_2\sin(\omega_2 t)] \tag{2.2}$$

As opposed to ‘plain’ FM with a single modulator, much richer harmonic textures can be produced. It is a very efficient method of obtaining complex spectra with just four parameters: I_1, I_2, ω_1, and ω_2. Indeed it has been stated that “Theoretically, we may synthesize any kind of sound by tuning and tweaking these parameters” [17].

2.2 Dynamic Spectra in FM Synthesis

For an FM carrier, the timbre it produces can be easily modified by changing the index of one of its modulators (I_1 in Equation 2.1) [3]. By smoothing this change over time with an envelope, we can model the natural change in timbre that most instruments produce over the course of a note.

This harmonic change is not as finely controlled as is possible in additive synthesis, where each harmonic may have its own amplitude envelope, but it allows complex changes to be defined in simple terms. Experiments in imitating real-world instruments show FM seems to provide enough control to create a good impression.

As we saw above in Equations 2.1 and 2.2, the modulation index, or I value, of each modulator scales how much it affects the carrier. As a modulation index changes, the partials it creates increase and decrease in accordance with the Bessel function [3]. Whilst this would seem to give control, reflected side frequencies stack up and the change of the index becomes less predictable with some frequencies rising or decreasing in amplitude. Figure 2.4 shows the effect of changing the modulation index on the partials present for one carrier and one modulator.

It is this unpredictable, intuition-based behaviour of FM synthesis that we believe makes it ideal for adaptation using a genetic algorithm.

2.3 ADSR envelopes

As explained earlier, real instruments do not generate steady-state tones, but evolve over the duration of the note. This evolution is often imitated in synthesis using *envelopes* to alter synthesis parameters in relation to time since the onset of a note.

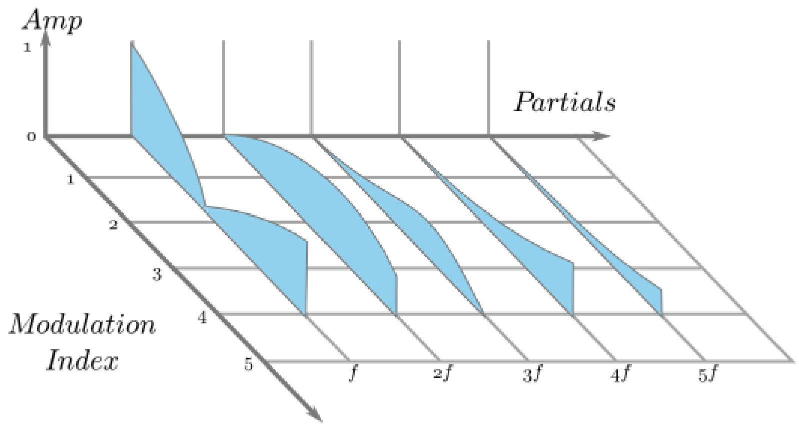


Figure 2.4: Effect of changing the modulation index on the partials present

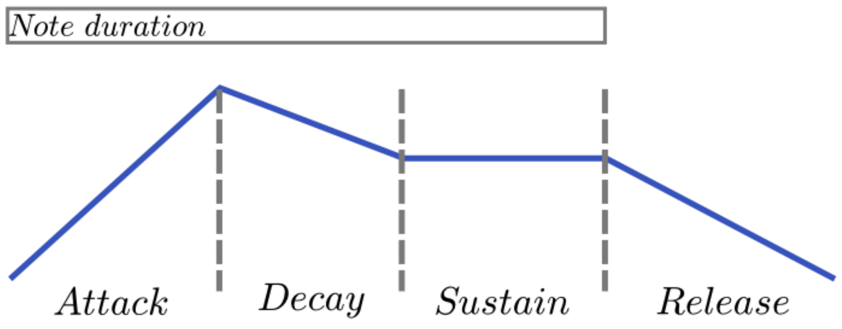


Figure 2.5: The general shape of an ADSR envelope over time

A simple and very popular enveloping method is the **ADSR** envelope model [5], standing for '*Attack, Decay, Sustain, Release*'. It is a method of describing the evolution of a synthesis parameter over time, most commonly applied to the amplitude. Whilst quite simplistic, the limited control it provides is often 'good enough' for the human ear.

Each of the four stages of the ADSR is controlled by a parameter. The 'attack' value is how long the attack phase takes before the envelope reaches its peak. The 'decay' value is how long the envelope takes to fade from peak to the 'sustain' level, which is given as a fraction of the peak value. The 'release' value is how long the envelope takes to fade to 0, after the note has been released. The shape of the envelope in its four stages can be seen in Figure 2.5.

When applied to the amplitude of a note, the envelope will start at $-\infty dB$ and peak at $0dB$, but in other applications base and amplitude parameters are added to control the envelope's effect. Synthesizers typically offer many envelopes for control, on amplitude, filter frequency, pitch, timbre, etc.

The ADSR model is flexible at achieving a variety of different envelope shapes. By setting the attack to 0 and having a low sustain level, percussive-style envelopes can be made. With a sustain level of 100%, the decay phase can be eliminated, as is in wind instruments.

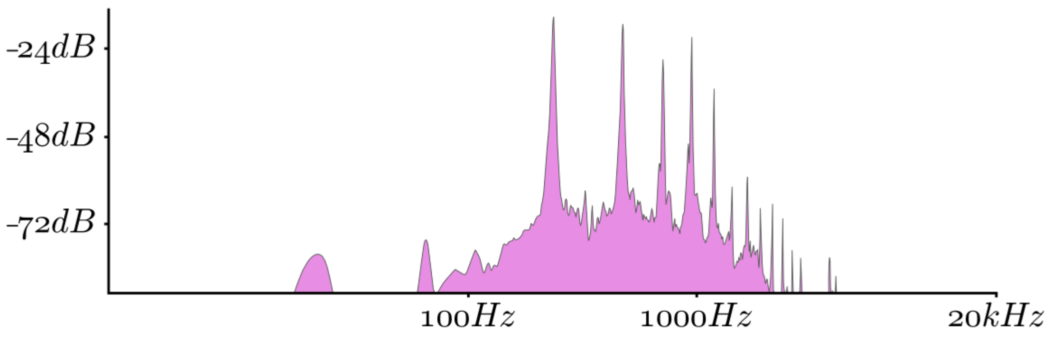


Figure 2.6: Sonic spectrum of a constant Vuvuzela note

2.4 Spectra and Spectrograms

The **Fast Fourier Transform**, or **FFT**, applied to audio signals converts a time-amplitude sample to a frequency-amplitude sample [2], or *spectrum*. It efficiently measures the presence of different frequency bands across the given sample. For speed the sample size is normally given as a power of 2.

The spectrum of a sound at a given point in time can be considered its 'shape' – it is a much more effective visual representation of a sound than the waveform, as two waveforms can sound the same but look completely dissimilar due to phase differences in the constituent partials. An example spectrum is shown in Figure 2.6.

One thing to note about the FFT is that it is not a perfect frequency-amplitude reference of the sound, due to the finite nature of the material analysed. A pure sine wave should only have presence in a single frequency bin, but it will leak into the surrounding bins. This can be seen in Figure 2.6 – a Vuvuzela tone is only made of only a few distinct partials, but these are not manifested as pure lines but instead are spikes with a wide base. This blurring means that algorithms using the FFT are limited to the accuracy of their detection.

To analyze the changing frequency content of a sound over time, windowing is performed. This means dividing the sound into power-of-2 size windows and analysing their spectra individually. A large amount of overlap is used between the windows to reduce boundary artifacts and obtain more detail in the time domain. The different spectra over the duration of the signal are combined into a *spectrogram*.

The spectrogram of a sound can be considered its overall 'fingerprint', and allows us to examine a sound's characteristics in many ways. An example spectrogram is shown in Figure 2.7. There is a trade-off when forming spectrograms though; the window size affects the time and frequency resolutions. A larger window provides more accuracy in frequency information (which frequencies are present at what magnitude), whilst a shorter window will provide more accuracy in time (when frequencies appear in the sound).

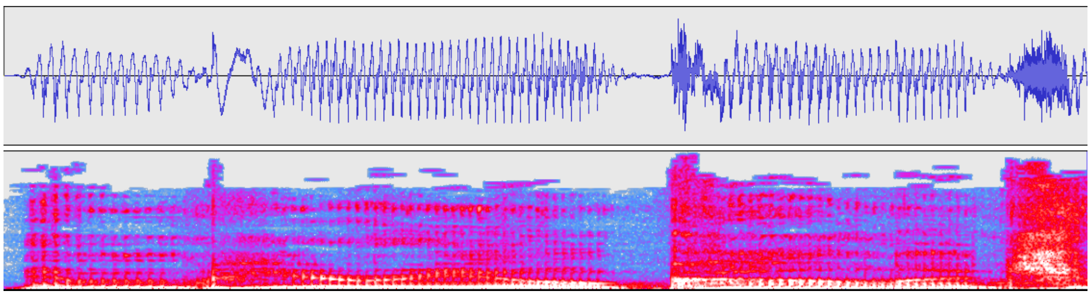


Figure 2.7: Waveform of male voice saying ‘Imperial College’ side-by-side with its spectrogram

2.5 Genetic Algorithms

A genetic algorithm is a search technique for problems with a large search space, abstracting evolution in the natural world [12]. It is often used where a normal search algorithm would be impractical and no straightforward approach is obvious. It does not guarantee an optimal solution, but converges towards one. The general Genetic Algorithm is outlined here, but there are many variations.

The first step is to represent potential solutions as a ‘Gene’ sequence, similar to DNA as used in nature. This involves encoding the parameters in a concise, sensible manner so that a single data structure can be used. At the most basic level this is done with bitstrings – long binary strings which are interpreted by splitting them up into the relevant fields, as shown in Figure 2.8. They are only suitable for problems where there are a fixed number of parameters.

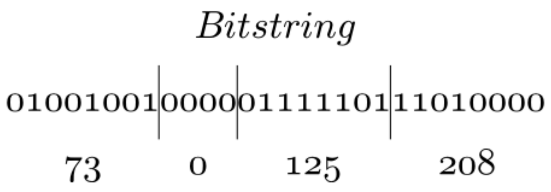


Figure 2.8: A bit string with values for 4 parameters

A *population* of many solutions is considered at each point in the algorithm. In this way, many different values are considered for each parameter in parallel. Typically the starting population features randomly generated individuals, but sometimes a well-known good selection is used.

At each stage of the algorithm, all the individuals are evaluated for fitness. This means interpreting the genes to produce the ‘real solution’ it represents, and then using a *fitness function* to calculate a fitness value for that individual. The fitness function must be continuous and assign a value to any individual, as it is used to compare the population.

Following this, a *selection* of the fittest individuals is performed, for example using a roulette algorithm. This entails selecting individuals randomly in proportion to their

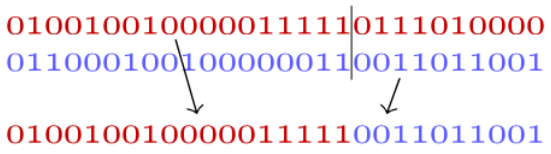


Figure 2.9: One-point crossover in action

fitness over the sum fitness of the population. Thus fitter individuals have a higher chance of being selected. Non-deterministic selection like this can be preferred in GAs to prevent being caught in *local maxima*, where fitness is high compared to the similar individuals, but not at a global maximum.

The selected individuals then *breed* to produce the next generation. Often they are be paired up randomly to produce offspring until the population is re-filled with new individuals. With bitstrings this can be done with one-point crossover – a random point is selected and the bits selected from either parent on the two sides of the split point, such as in Figure 2.9.

The problem facing a breeding strategy is that it must avoid dividing successful parts of an individual's strategy whilst also allowing the inclusion of (possibly) better genetic material from the other parent. Whilst one-point crossover is often used by default, better performance often comes from more elaborate schemes [12]. There are many other options, such as n-point crossover, probabilistic crossover, and tree-based schemes. However, the simplest extension for the most gain is two-point crossover, as shown in Figure 2.10.

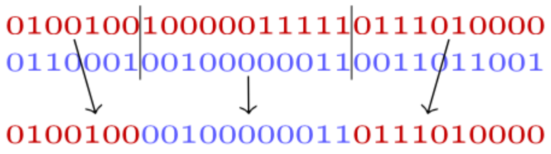


Figure 2.10: Two-point crossover in action

After combination, *mutation* may be applied – randomizing a few genes – in order to avoid stagnation of the 'gene pool'. The new individuals then contain a variety of combinations of previously successful sequences of genes, and have a chance at being even fitter.

The algorithm then loops, repeating the evaluation, selection, and breeding for each generation. Termination can be set up to occur for an application-dependent condition; either a 'good enough' fitness is found, a maximum generation count is reached, or any other test that can ensure the usefulness of the solution. In some applications, an optimal solution may be easily recognizable. The overall loop can be seen in Figure 2.11.

The Genetic Algorithm is not fixed, and a specialized GA can feature many different problem-specific or experimental changes. Some of these include alternative representation schemes (e.g. trees), different selection algorithms, other breeding schemes, etc. For example, the genes could be represented in a two-chromosome format, as in nature, allowing each individual to carry more genetic information than it expresses.

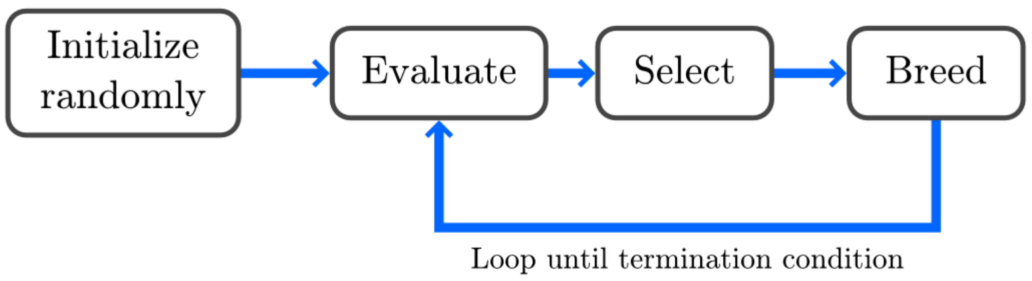


Figure 2.11: Summary of the Genetic Algorithm

2.6 Pre-existing Work

As far we have found, there are three prior published works that use a GA to optimize a synthesizer model for resynthesis. They are all based around FM synthesis.

2.6.1 Tan, 1996

This paper [16] describes using a modification of the basic Genetic Algorithm known as the *Genetic Annealing Algorithm* to optimize a simple DFM synthesizer to match sound spectra obtained from steady-state waveforms of real-world instruments. It shows that resynthesis via a GA is plausible, but has some drawbacks:

- Only the steady-state spectrum of the target sound is matched – not its time-based evolution as represented by its *spectrogram*.
- The synthesizer model used features only a single DFM carrier which severely restricts the types of sounds that can be matched.
- Most of the evolved individuals feature too many high partials, a side effect of using purely FM synthesis – these could be eliminated by combining subtractive synthesis with a low pass filter.

2.6.2 Lai, 2006

This paper [10] is a fresh reimagining of the concept as the authors seem unfamiliar with the Tan paper. However it describes a very similar system with a few enhancements:

- The FM synthesis model uses more waveforms than sine waves, such as a sawtooth waves.
- An enhanced fitness function is used with a weighted sum of two spectrum-comparison metrics.
- The GA selection and crossover methods are investigated and shown to be good for the problem domain.

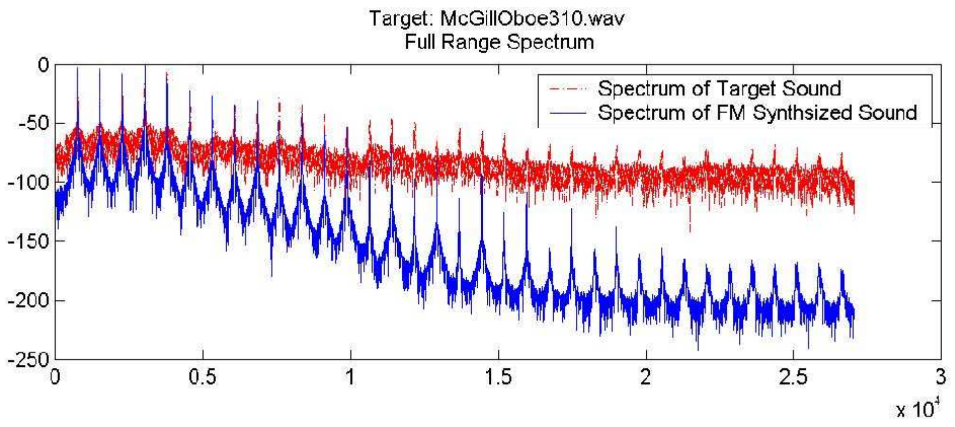


Figure 2.12: Output from Yong's program showing matching of an oboe

The work gathers inspiration from several sources in both the GA and synthesis literature and proves it benefits from the extra techniques. However it is fairly brief and the results shown are fairly biased – they present an 'exact match' but it is for a sound that was generated by the synthesizer to begin with. Other drawbacks include:

- The synthesizer model used is very simple and uses only one FM carrier.
- The fitness function presented has some controlling parameters, but these are only selected arbitrarily and not investigated.
- A larger variety of sounds needs testing.

2.6.3 2007, Yong

This project [17] presents a system implemented in Matlab based upon a combination of the Tan and Lai papers. The DFM synthesizer model used is based upon the Tan paper, and the GA and fitness functions employed from the Lai paper.

The key extension in this work is that multiple DFM carriers are used at once, enabling richer spectra to be produced. Each DFM carrier is encoded separately in the genes, and each individual may have between 1 and 20 such carriers. This allows a large range of sounds to be recreated with more accuracy.

Yong has made the Matlab code available, and we have tested it and found it to be generally effective. An example run takes a few minutes to produce a sound that nicely approximates a tone from an input file. It also outputs the spectrum of the input and output sounds so one can visually compare the accuracy, such as in Figure 2.12.

This work still falls under our major criticism of the other two though – it is only concerned with matching a single sonic *spectrum*, not an overall *spectrogram*. This step deletes most of the information present in a wave file, and cannot match the important evolving characteristics of a sound. At the end of the report Yong highlights this failure by showing the matching of a cat meow sound. Whilst a cat meow has a definite tonal quality, reducing it to a single spectrum overlays all the differently pitched parts of the sound at once, giving it a similar fingerprint to noise, and so the GA fails at matching it.

Therefore we make this one of the main targets for improvement in our project.

Chapter 3

Outline

In this Chapter we describe the basics of our system, before presenting the details in Chapters 4 and 5. We also discuss implementation choices and recognize some drawbacks.

3.1 Overall View

Figure 3.1 gives an overall view of the system – it should look familiar as it expands the basic GA pattern as presented in Section 2.5.

3.2 Synthesizer Model

We implement a synthesizer that combines FM and subtractive synthesis models. We copy the Yong work by using multiple DFM carriers as the basis for the synthesis. The subtractive side comes from adding a low pass filter in the signal path – this is intended to cut down the large number of high partials that DFM synthesis can generate.

Following the research from Chowning on Dynamic Spectra [3], we add the scope for dynamic spectra with independent ADSR envelopes on the index of each modulator, in each DFM carrier. This allows them to change their modulation index over time, and with each envelope taking its own shape there is a lot of scope for complex evolving textures that can match the target sound.

We also have ADSR envelopes applied to the filter's cut-off frequency, pitch control, and overall amplitude. This gives the synthesizer a lot of scope to produce sounds with evolutions over time. Figure 3.2 summarizes the entire signal path of the synthesizer.

3.3 Genetic Algorithm

Most of the GA outline is described in Figure 3.1. We use a random initial population and loop over evaluation, selection, and breeding until we reach the terminating generation. We implement the fitness metric and selection method suggested in the Lai paper [10], but with extensions.

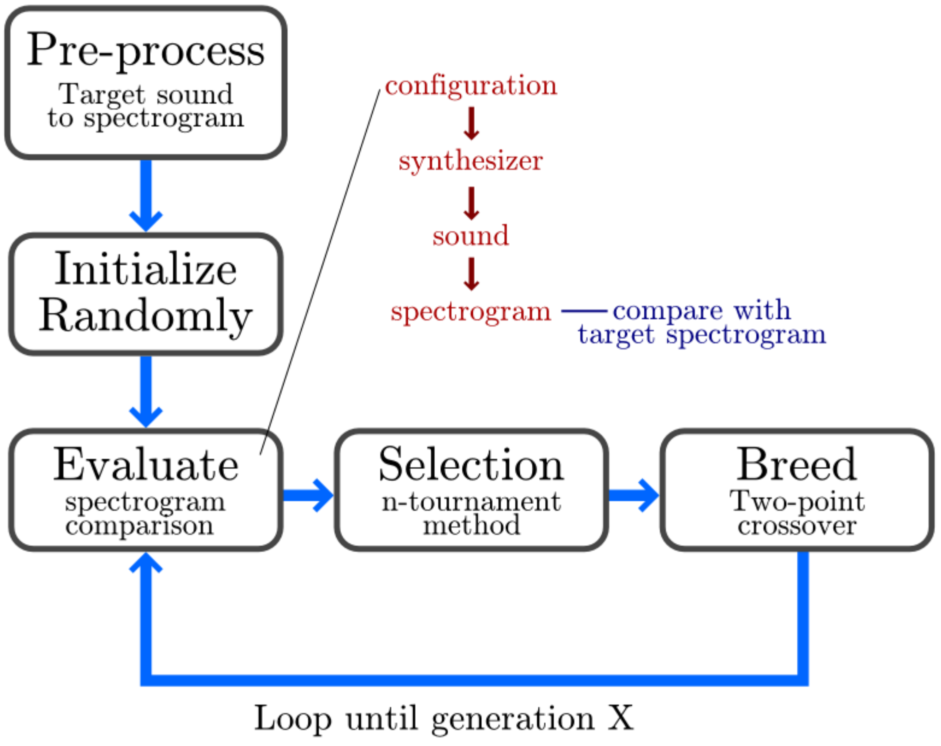


Figure 3.1: An overall view of the system

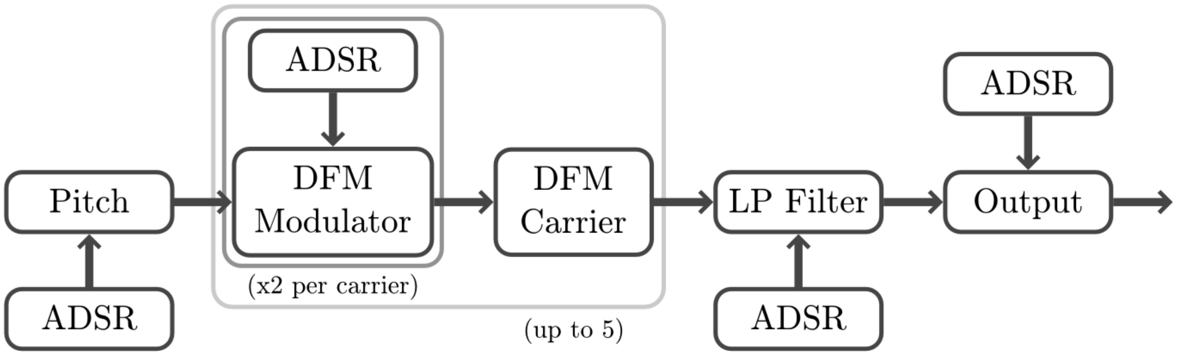


Figure 3.2: Arrangement of the components in the synthesizer model

Individuals in the population represent a synthesizer configuration by a series of 64 floating point and integer 'genes'. We evaluate the fitness of such a member by constructing the synthesizer with the represented configuration, recording its output, and comparing this spectrogram with the spectrogram of the target sound.

We use a spectrogram comparison function as suggested by Lai et al., which combines two different metrics in a weighted sum. It produces an error value, not a fitness value, so the GA works in terms of minimization rather than maximization. We detail this in Section 5.2, and work to optimize the weighting of the metrics in Section 6.4.

Selection is done with an n-tournament selector, again at the suggestion of the Lai paper; however we improve it with elitism. The GA is set to terminate after a fixed number of generations.

3.4 Missing Detail

The 'missing detail' that prevents 100% automated sound resynthesis is pitch detection. Because the synthesizer must take an input note to work at, we need the pitch of the target sound to properly resynthesize it – but the waveform doesn't directly give us this information. The Yong work deals with this problem by taking an argument of the pitch to resynthesize at alongside the target waveform, meaning the user must know the pitch the target sound is at.

Whilst there is scope for adding an automatic pitch detection algorithm, of which many exist, we do not implement it. Fully automated resynthesis is not the focus of this project, so it would only be a matter of convenience to add it. In most cases we have prior knowledge of a target sound's pitch anyhow.

Therefore we take the pitch of the target sound as a second argument alongside its waveform.

3.5 Implementing in Java

The project is implemented in Java, using two high quality libraries to provide the basics for both parts – *Minim* [7, 6] for the synthesizer and *JGAP* [13] for the GA. Both provide steady bases in their respective areas, and will be described in more detail in the respective chapters.

Whilst an implementation might run faster in a machine-code-target language such as C, or taken less coding in a scripting language such as Python, Java was chosen based on this rationale:

- **It is relatively fast** in comparison to scripting languages, even audio-specific languages such as LAPSDA. Although it runs in a virtual machine this has seen big improvements in recent years so it can nearly match the speed of machine-specific code on many tasks.
- **The object-orientated architecture** is good for dealing with the many synthesizer instances that need to be dealt with in parallel, plus the save/load object capabilities would make storing configurations in files easy.

- **It is cross-platform** so the end result is not limited to a single operating system.
- **The libraries available** for both the synthesizer and GA parts provide ground-level features that speed up development considerably.

Chapter 4

Synthesizer Model

In this Chapter we give a detailed description of the synthesizer model as implemented in Section 4.1, show how we record and analyse the output sounds in Section 4.2, and then detail difficulties overcome in the synthesizer design in Sections 4.3 and 4.4.

4.1 Structure

The synthesizer can be viewed as an isolated system that converts pitch data into audio data, as shown in Figure 4.1. To represent pitch we use a basic form of the MIDI format – a standard data type for representing musical information. For audio we write to a floating point data buffer. Minim provides the conversion from MIDI to frequency and takes the floating point audio data to the speakers for output.

The synthesizer is implemented as a single unit with defined interfaces so that it can be easily integrated with the GA – it has no awareness of where its MIDI data comes from or where its audio output goes to. We can easily play it live with manual keyboard control or automatically record its output for a given note.

In the following subsections we elaborate on the internals of the audio synthesis process as helped by the Minim library and then show the entirety of the synthesis chain.

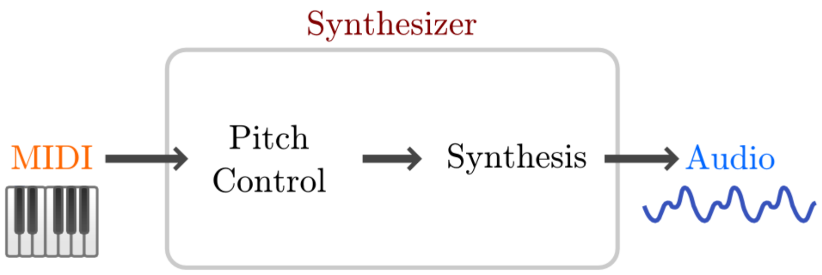


Figure 4.1: The basics of the synthesizer system

4.1.1 Universal Generators

The Minim library presents a modular signal-generating interface using its `UGen` (*'Universal Generator'*) superclass. All data passing is handled by Minim and each `UGen` can take input from others by defining public inputs. A `UGen` only need implement the abstract method `uGenerate(float[] channels)` to define its output on a per-sample basis – for example, the most basic `UGen` that outputs a constant value is implemented as:

```
package ddf.minim.ugens;

public class Constant extends UGen {
  private float value;

  public Constant( float val ) {
    super();
    value = val;
  }

  @Override
  protected void uGenerate( float[] channels ) {
    for(int i = 0; i < channels.length; i++) {
      channels[ i ] = value;
    }
  }
}
```

We build our signal path out of many different `UGen` classes – although there are many provided with the library, we only use a few with adaptation and create the rest from scratch.

4.1.2 UGen Chain

The following `UGen`s are used in the synthesizer:

1. **Glide** – glides towards a target frequency in a given time. Designed for manual monophonic playback of synthesizer.

2. **ADSRRepeatable** – provides an ADSR envelope that can be restarted. Adapted from the included ADSR code that would not allow restarting and thus prevented the synthesizer being used to play more than one note.

3. **ValuePlusADSRPercentage** – takes a value input and scales it percentagewise by the amplitude of its ADSRRepeatable envelope.

4. **ZeroHzFMCarrier** – FM class we implement to provide the DFM functionality. The class contains a list of source UGens that it controls and then performs zero-hz frequency modulation on their sum.

5. **SinePlusADSR** – our own sine wave oscillator class that includes an ADSRRepeatable to change its amplitude.

6. **Summer** – a built-in class that is used to add together several other UGens' outputs.
7. **MoogFilter** – a built-in class that implements the 'Moog-style' low-pass filter, providing input control for its cut-off frequency. This is a 24dB/Oct low-pass filter that is generally considered 'more musical' than a plain low-pass filter.
8. **ValuePlusADSR** – our own control class that provides a constant value that is then modified an absolute amount by the amplitude of its ADSRRepeatable envelope.

Figure 4.2 shows the total structure of the UGen chain as used in the implementation.

4.1.3 Configuration

The synthesizer has a large number of parameters to be configured – a total of 64 in the final implementation. One of the problems faced in Java is providing values for all these in a constructor, as it is impossible to take 64 arguments to a method.

We use a custom configuration class that provides the storage of floating point/integer values for each parameter in a hashmap. The interface allows the discernment between the two data types and the provision of constraints and default values for each parameter. We can then configure the synthesizer by constructing it with such an object, and save a configuration by saving this object using Java's `ObjectOutputStream` class.

Because of the genetic dependence on the parameters, it is only in section 5.6 that we will present the entire list of parameters configurable in the synthesizer.

4.2 Recording and Analysing

To link the synthesizer with the GA we need to record its output. Minim handles passing the output to speakers, but provides no recording method. However, the `UGen` interface makes redirecting the output simple.

We use a recording class that repeatedly calls the synthesizer's output `UGen` to `uGenerate` one sample at a time and store this in a `float[]` array. We then have the synthesized sound in a manipulatable form, and can save it as a wav file or analyze it.

We also implement code to generate a spectrogram from such an audio buffer, so that we can analyse both target and resynthesized sounds. We use Minim's FFT code to generate a single spectrum at a time, in power-of-2 size windows. We use a 75% window overlap to obtain more detail in the time domain and reduce boundary artifacts. Figure 4.3 shows an example spectrogram generated from an oboe recording.

4.3 Envelope Mess

Early versions of the system featured entirely autonomous ADSR (Attack, Decay, Sustain, Release) envelopes for the amplitude, filter, pitch, and independent FM modulator

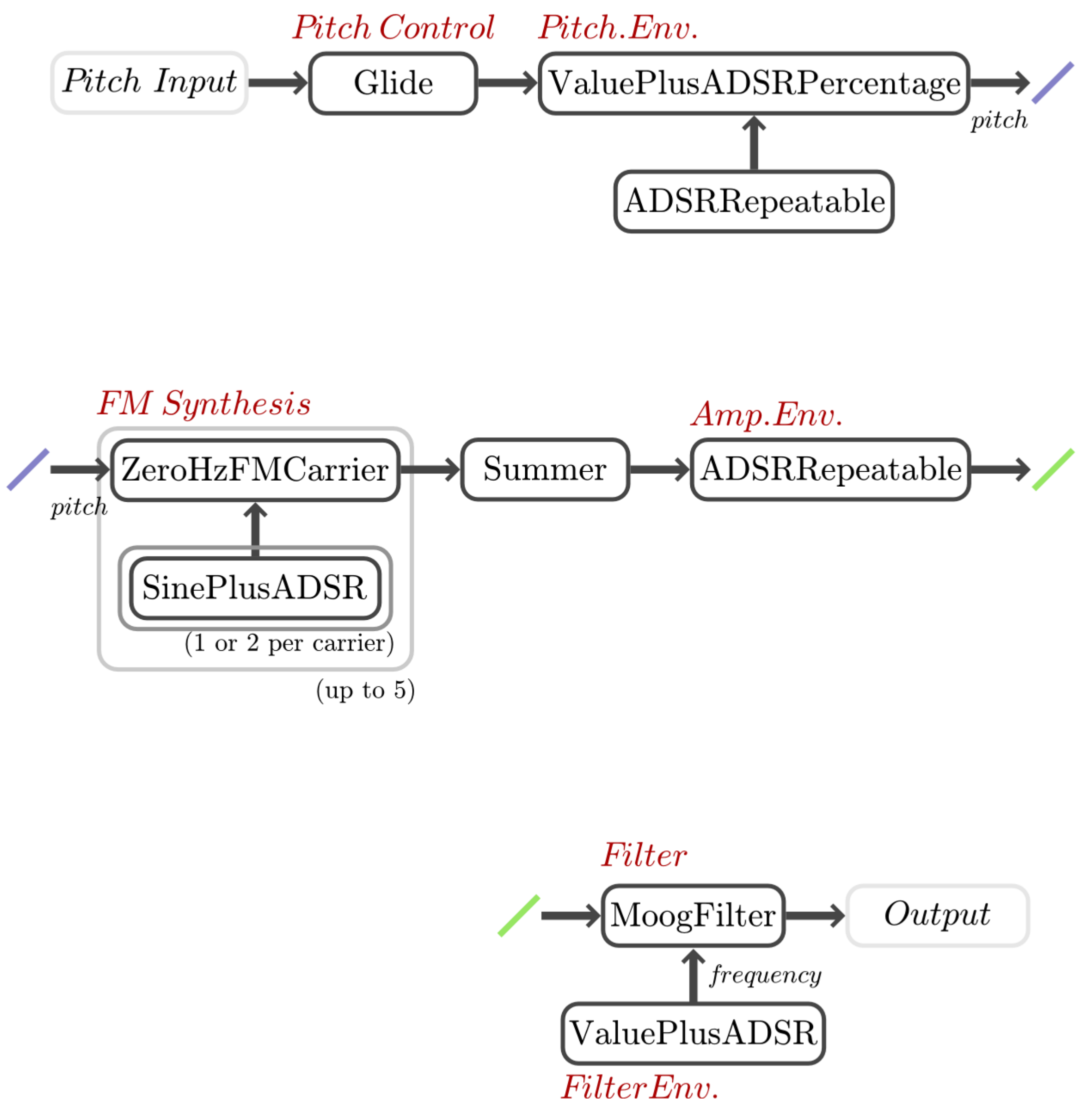


Figure 4.2: The synthesizer implementation in terms of UGens. The diagram has been broken over three lines to make it more compact.

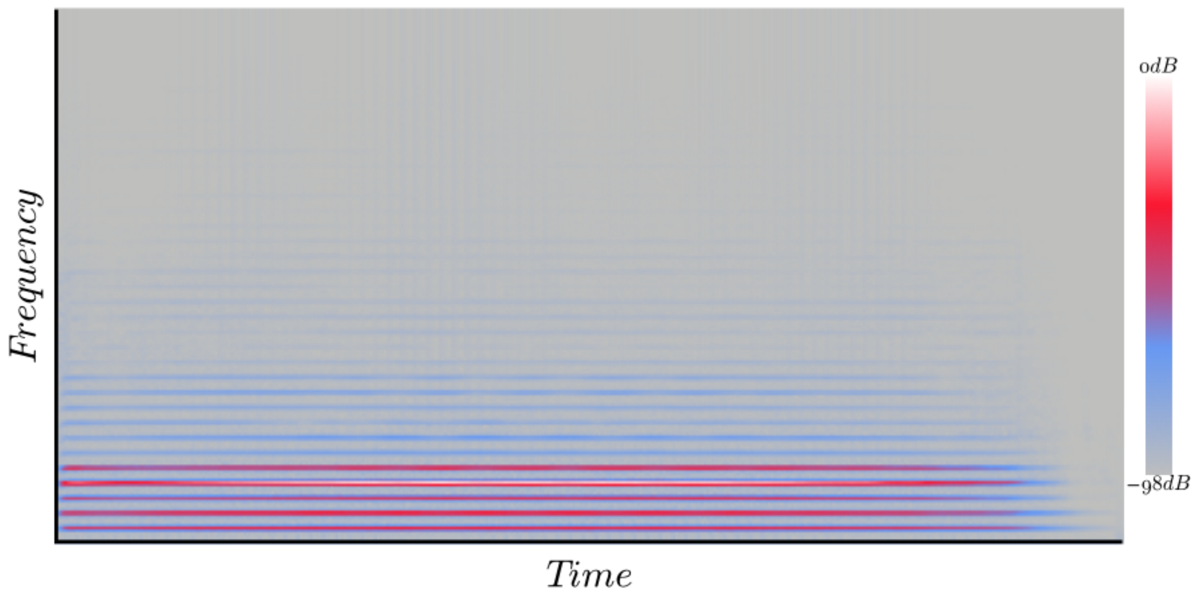


Figure 4.3: The spectrogram of an oboe note

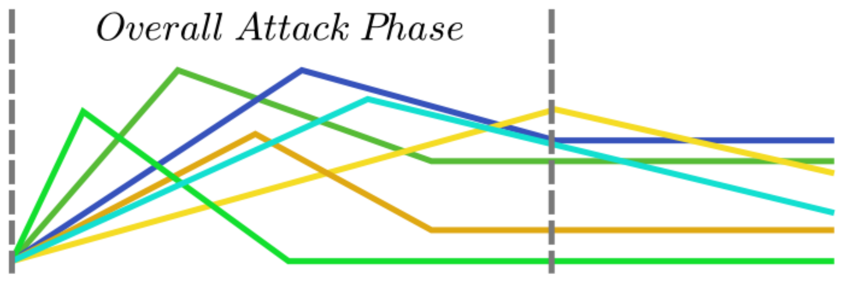


Figure 4.4: Illustration of the ‘messy’ effect of many randomly generated ADSR envelopes interacting together

indices, but this proved to be a bad idea. The output sounds were extremely complex and the GA found convergence hard, even for simple sounds such as a sine wave – because of the complexity of evolving many different envelopes at once, the correct attack/decay/release timings were rarely found.

Multiple unsynchronized ADSR envelopes are very ‘messy’ because they create extremely complex sounds with essentially no fixed character. With many different envelopes being modelled at once, the attack phase is only over after the last one finishes its attack phase – even if the others have started or passed their decay phase already, as indicated in Figure 4.4. This also applies to the decay and release phases, leading to a single long envelope effectively monopolizing the overall sound of the synthesizer.

As most of the real world sounds do not rely on complex harmonic evolutions in their attack phase, we found this model to be overkill. It was simplified to the following:

- There is only one value for each attack, decay, and release that cover all the envelopes.
- Each envelope maintains its own sustain value – as some parameters may need to stay high whilst others should disappear after the decay phase.
- The FM modulator index envelopes were given ‘models’ to select between – see next section.

After this simplification the GA had much better performance optimizing the synthesizer parameters.

4.4 Evolving Spectra

As shown in the background section 2.2, changing the index of a modulator in FM causes rich and varied changes in the resultant spectra. For this reason ADSR envelopes are applied to the indices of each modulator in the synthesizer. However, the ADSR model is enhanced to allow a wider variety of spectral evolutions.

The effect of changing a modulation index is not particularly well defined, but can be estimated intuitively [3]; both increases and decreases can be used to model the harmonic change of certain sounds over time. Thus the envelope applied to the modulation index should work for both increases and decreases at any phase.

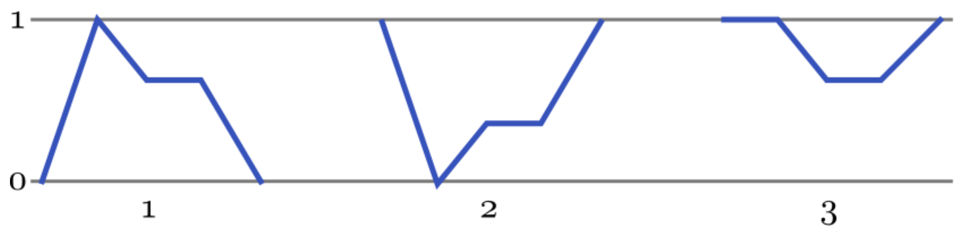


Figure 4.5: The three ADSR models applied to modulation indices

We allow each index to take one of three ADSR envelope models:

- **Low to High** – the envelope starts at 0, peaks at 1, and ends back at 0 after the release phase.
- **High to Low** – the envelope starts at 1, peaks at 0, and ends back at 1 after the release phase.
- **High to High** – the envelope stays at 1 for all of the start, peak, and post-release parts – however, it still falls to its sustain value.

Thus the index ADSR envelopes are controlled by only two parameters – the model (from 1 to 3), and the sustain value (from 0.0 to 1.0 amplitude). They use the global attack/decay/release times, as fixed above. The models are illustrated in Figure 4.5.

Chapter 5

Genetic Algorithm

In this Chapter we cover details of the genetic algorithm, which we implement with the JGAP library. We detail the types of Genes in Section 5.1, the Fitness Function in Section 5.2, and genetic operations in Sections 5.3 and 5.4. We then expose the genome in Section 5.6 and detail the termination in Section 5.7. On the way, we cover two additions we made to significantly improve the GA over previous works, in Sections 5.5 and 5.8.

5.1 Genes

In the JGAP framework we define a Genome for the population – a template which all individuals conform to consisting of an ordered array of gene objects. Each gene object has a datatype (Integer, Floating Point, etc.) and constraints (e.g. a Minimum) that JGAP uses to make sure random individuals are valid to the problem domain.

Effectively, JGAP uses a bitstring implementation with the crossover points limited to be on the boundaries between the individual numbers, which is known to improve convergence when using complex number representations such as floating points [11]. Because of this the framework presents only a limited number of gene types, of which we use only Floating Points and Integers, with Booleans stored as integers constrained to either 0 or 1.

The genome is automatically derived from the synthesizer model, which exposes its parameters as either integers or floats. We have utility functions to map from synthesizer configurations to genes and back, meaning JGAP can work with genes and the respective synthesizer can be easily determined. An example section of the genome is presented in Figure 5.1.

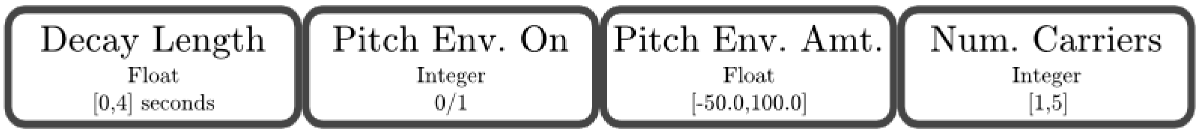


Figure 5.1: An extract of the genome

The following sections elaborate on the internals of the GA and provide justification for design choices made for the Genome before we present the entirety of the Genome in Section 5.6

5.2 Fitness Function

Inside the JGAP framework the fitness function must accept a genome and turn it into a fitness value. We do this by comparing the spectrograms of the target and synthesized sounds to estimate how well they match.

We pre-process the target sound and store its spectrogram. Then for each individual genome in the population, we can calculate a fitness value with a multiple-step procedure:

1. **Chromosome → Synthesizer Configuration**
   The Gene values presented in an individual's Chromosome are reinterpreted as a configuration for the synthesizer.
2. **Synthesizer Configuration → Synthesizer Instance**
   The synthesizer class is instantiated and prepared for playback.
3. **Synthesizer → Waveform**
   The synthesizer has its output redirected to a wavebuffer and is played at the input MIDI note pitch, for a length of time to match the target sound. This is done much faster than realtime playback.
4. **Waveform → Spectrogram**
   The waveform is converted to a spectrogram using the given transform window length.
5. **Spectrogram → Fitness Value**
   The spectrograms of the synthesized and target sounds are compared using the below formulæto arrive at a fitness value.

The fitness formula itself is taken from the Lai paper [10] which was also used in the Yong work [17]. It uses a weighted sum of two metrics that are used to compare different characteristics of spectrograms – **Spectral Norms** and **Spectral Centroids**.

For a spectrogram f_x we define $f_x(w, b)$ to be the magnitude of their Fourier Transform at window w and bin b. We compare two spectrograms f_x and f_y, for the synthesized and target sounds respectively. Let W be the number of Fourier Transform windows to compare, which is the maximum number of either sound as the synthesized sound is created to be the same length as the target. Let B be the number of bins used in the Fourier Transform, which is half the total window length for audible sound [2].

The **Spectral Norm** formula is given as:

$$N(a, b) = \sum_{w=1}^{W} \sum_{b=1}^{B} |f_x(w, b) - f_y(w, b)|^2 \tag{5.1}$$

Which is intuitively the sum of the absolute difference squared for each point of the transforms. This is a typical measure of the difference between two sounds and helps the GA hone in on correct sounds as the error for missing or including a harmonic that should not be present increases the with the square of the difference.

The **Spectral Centroid** formula is given as:

$$C(a, w) = \frac{\sum_{b=1}^{B} f_a(w, b) \times b}{\sum_{b=1}^{B} f_a(w, b)} \tag{5.2}$$

for a single sound a at a single window w. The centroid measure is also called the brightness, as a larger spectral centroid corresponds to an emphasis on higher frequency components, which results in a brighter sound. In practice we compare the sum of the absolute difference between the two sounds across all windows, $C_{dif}(a, b)$.

These two metrics are added as a weighted sum:

$$error(a) = \frac{1}{2}N(a, b) + \frac{1}{2}C_{dif}(a, b) \tag{5.3}$$

As this is an error term, the GA works in term of minimization rather than the traditional maximization process.

This sum is taken at the suggestion of the Lai paper, but there is no rigorous proof as to why it is the ideal weighting at $\frac{1}{2}$ and $\frac{1}{2}$. We later run experiments to find the optimum weighting, in Section 6.4.

5.3 Selection Method

The selection mechanism used is a tournament strategy as suggested in the Lai paper [10], but with elitism as an extension as it improves the performance of GAs [12].

The n-tournament selection method creates small tournaments between n random individuals from the population. The fittest in the tournament gains the right to breed, and the winners from two tournaments are combined into a new individual with two-point crossover. Tournaments continue until the new population is full again.

The method is meant to simulate real world natural selection where individuals compete to breed, but only against a limited set of others as opposed to the global population. The restricted tournament size means the fit individuals that come out top are not necessarily amazing relative to the entire population, but are definitely better than some. Thus fresh combinations of genetic material are more likely than in a selection mechanism that overly favours the globally fittest individuals.

At the suggestion of the Lai paper, the tournament size is set to 5. However they again provide no backing as to why this is the ideal number, so we experiment with this to try and find the optimum in Section 6.5.

Elitism is added, copying the top individuals directly into the next population. This preserves the best individuals, protecting their highly fit traits from being deleted due to

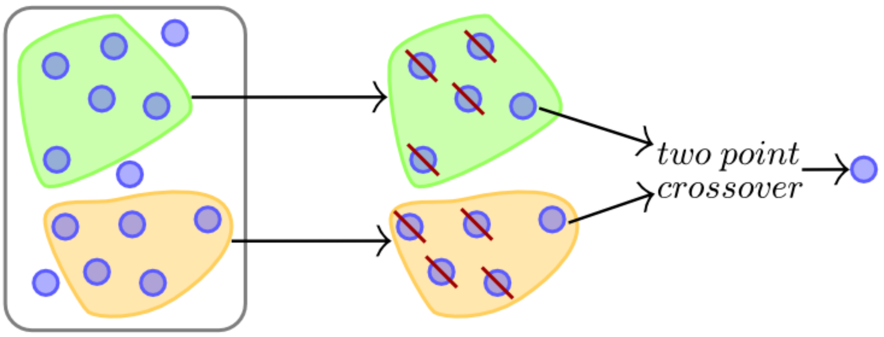


Figure 5.2: 5-tournament selection

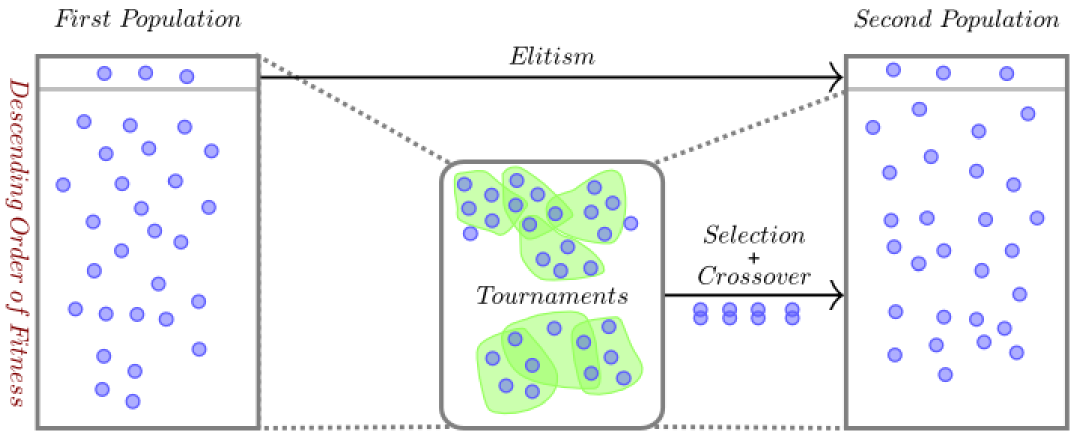


Figure 5.3: The combined elitism and tournament methods

crossover. This occurs before the tournament process, but the fit individuals are left in the pool so they can also be selected for tournaments.

Elitism adds a stronger 'forward crank' to the genetic algorithm by preventing the crossover operation from destroying the best configurations that have been discovered. However it can also lead to local minima being exacerbated, but the mutation is designed to deal with this.

5.4 Genetic Operators

The individuals are combined with two-point crossover, which is believed preferable to one-point crossover as the combination of different carrier properties from individuals is desired. JGAP does not manipulate raw bitstrings, but only allows crossover on the boundaries between the independent genes, so we preserve the sanity of the values.

Mutation is applied to all individuals after the selection and crossover process, with a mutation rate μ defining the chance of a particular gene being mutated and replaced with a new random value. For each individual, each gene has a random probability evaluated with $p = \frac{1}{\mu}$ for the gene to mutate; thus about 1 in every μ genes is mutated. We set μ to 200 at the recommendation of [12].

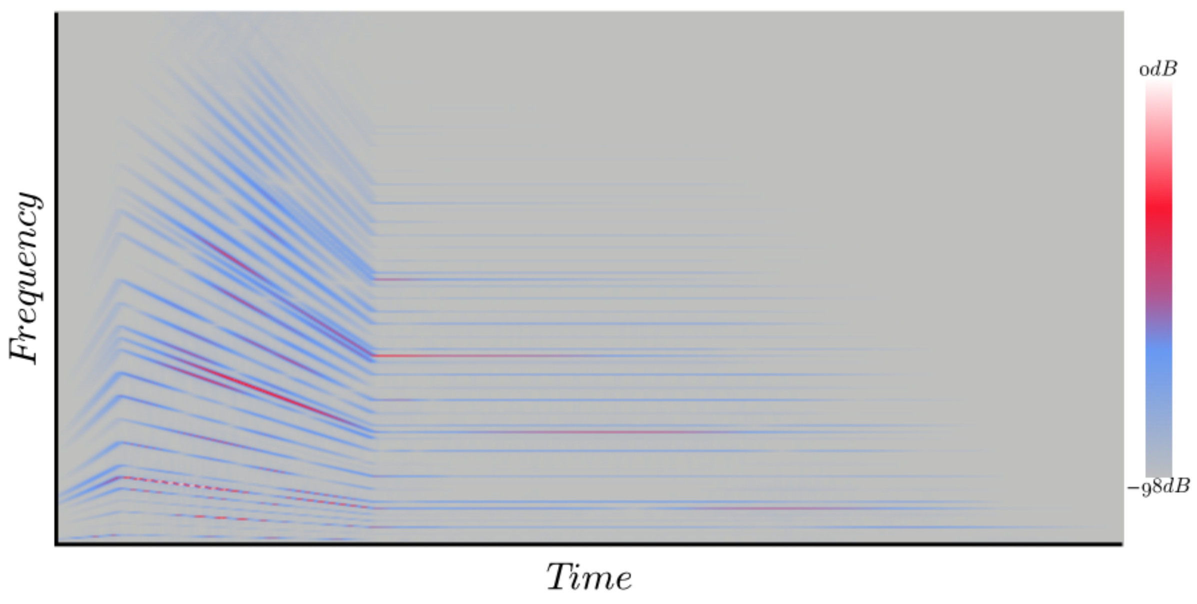


Figure 5.4: Unnecessary pitch envelope use in an individual that remained highly fit due to its correct second-half

5.5 Regulatory Genes

Initial runs of the GA were promising, but the generated synthesizer patches were not very well aligned with the target sounds' envelopes. This was especially clear in sounds that have no envelope, such as a sustained synthesized pitch.

For example, Figure 5.4 shows the spectrogram of a resynthesized sound that was an attempt to match a pure sawtooth wave. The second half is mostly correct, but the start has a very excessive use of the pitch envelope parameter to generate a large bend which is not present in the target sound. If the pitch envelope amount was set to 0% in this same individual, it would be a very good match at almost twice the fitness.

The problem was that the probability of the GA finding the optimum pitch envelope – 0% – was low, as when mutated it could fall anywhere in the $[-50, 100]$ range. This problem also extended to the other envelopes for the amplitude, filter, and modulation indexes. For example, some sounds feature no attack phase for any of these parameters, but it would still persist throughout the population due to the low probability of becoming 'turned off' and set to 0.

The solution was to add regulatory genes, a suggestion found in [12]. In nature, genes form a complex network of control, with regulatory genes affecting the expression of other genes, which is what leads to cells differentiating their types. Adapting this concept to our GA meant adding genes that control whether or not another gene takes effect.

The regulatory genes added were, in JGAP terms, integer genes which take either 0 or 1. When constructing the synthesizer these are treated as a boolean value controlling the expression of their target gene. For example, if the pitch envelope on/off regulatory gene is off, the pitch envelope amount is set to 0; but if it is on, the pitch envelope amount is derived from the respective gene. Two of the regulatory genes added are demonstrated in Figure 5.5.

The following is the full list of regulatory genes implemented:

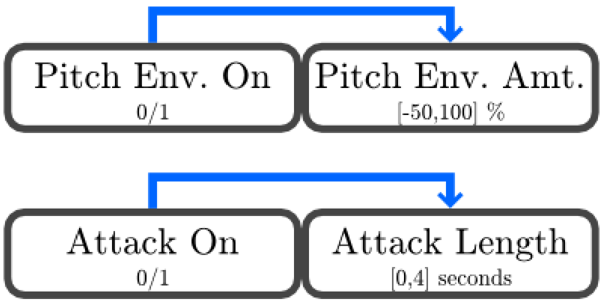


Figure 5.5: Regulatory genes in the genome

- Attack Phase On/Off
- Release Phase On/Off
- Pitch Envelope On/Off
- Filter Envelope On/Off
- Number of Carriers Active
- Per Carrier : Number of Modulators Active

The 'Number of Carriers' gene is the highest-level regulatory gene, as it affects a lot of genes at once. Whilst each individual contains configuration parameters for the maximum of 5 carriers, only the number specified by the number of carrier genes will ever be active. Thus a 1-carrier individual is mostly carrying 'junk' genes which encode for the spare 5 carriers – a parallel to the human genome which is estimated to be 95% 'junk', encoding for unexpressed proteins.

Following this addition, accuracy improved greatly. The resynthesized sounds stopped featuring the pitch envelope when unnecessary, as well as no attack/release phases for non-fading sounds.

5.6 Full Genome

Table 5.6 shows the entire listing of the final genome used. For floating point genes we show decimal points in the value ranges, whilst for integers only round numbers are shown. The per-carrier section is repeated for each of the five carriers; the per-modulator section is repeated two times for each modulator

With 4 genes per modulator, 2 modulators per carrier with an extra 2 genes each, and 5 carriers with a further 14 global genes, the genome has a total size of $((4\times2)+2)\times5+14 = 64$. However not all take effect in each individual as regulatory genes come into effect, turning off modulators, carriers, and/or envelopes.

Name & Effect	**Range**
Attack On/Off – regulatory gene for attack	[0,1]
Attack – length of attack phase	[0.0,4.0] seconds
Decay On/Off – regulatory gene for decay	[0,1]
Decay – length of decay phase	[0.0,4.0] seconds
Amplitude Sustain – level amplitude sustains at	[0.0,1.0]
Pitch Envelope On/Off – regulatory gene for pitch envelope	[0,1]
Pitch Envelope Amount – range of pitch envelope as percentage, where -50% gives -1 octave and +100% gives +1 octave	[-50.0,100.0] %
Filter Frequency – base frequency for filter	[30.0,20000.0] Hz
Filter Resonance – resonance factor for filter	[0.1,0.5]
Filter Envelope On/Off – regulatory gene for filter envelope	[0,1]
Filter Envelope Amount – scales the effect of the filter envelope	[-10,000.0Hz,10,000.0] Hz
Filter Envelope Sustain – the sustain value for the filter envelope ADSR model	[0.0,1.0]
Number of Active Carriers – out of the total possible 5, how many should be active	[1,5]

Per Carrier :

Carrier's Amplitude – volume multiplier applied to signal generated by carrier	[0.0 to 1.0]
Carrier's Active Modulators – either 1 or 2, gives the possibility to use a single modulator instead of the 2 in the DFM model	[1,2]

Per Modulator (Per Carrier) :

Modulator Index – defines the peak amplitude the modulator operates at, referred to as its index in the formulæ	[0.0,10.0]
Modulator Ratio – the multiplication ratio of the modulator's pitch from the fundamental frequency	{0.25,0.5,1,2,3,4,5,6,7,8}
Modulator ADSR Model – selector for the ADSR model used for this modulator's index (as per Section 4.4)	[1,3]
Modulator Sustain – the sustain value for the ADSR envelope applied to the index	[0.0,1.0]

Table 5.1: The full genome

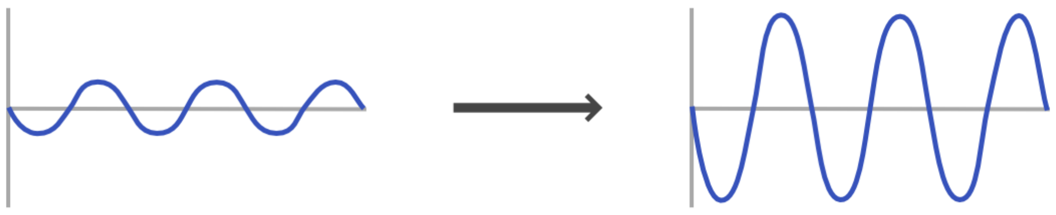


Figure 5.6: Normalization applied to a waveform

5.7 Termination Condition

As explained in the background Section 2.5, the generic GA does not define a termination condition as it needs selecting on a per-application basis. For this system we choose to terminate after a fixed number of generations – a very simple scheme, but we have a rationale:

- Some sounds never converge that satisfyingly, so having a minimum error threshold would not make sense
- Some sounds slowly converge forever, so stopping after convergence stops could still lead to very long execution times
- A fixed, predictable runtime is desirable which no other termination condition can provide
- Most sounds reach a satisfying convergence after 30 generations, as we will show in Section 6.2

We later do an experiment to find the optimum cut-off generation, in Section 6.2.

5.8 Normalization

First iterations of the system had problems matching the sound due to the challenge of the amplitude. Because each carrier's modulators have independent amplitudes and a non-fixed number of carriers are active, the basic output of the synthesizer can fall in a wide amplitude range. Thus a sound could be entirely correct in its frequency content but come with the incorrect volume level – which adds a lot of error due to the **Spectral Norms** metric as defined in equation 5.1. This problem was overcome by adding normalization to all waveforms present in the GA.

Normalization is the process of stretching a waveform so that its peak amplitude matches a target amplitude; when this is set to 1.0 it will make the entire range of the sound fall within the [-1.0,1.0] range. We implement it by scanning the waveform for its maximum absolute value, then dividing all values throughout. This can scale a sound up or down in amplitude, and is demonstrated in Figure 5.6.

When added to the GA this significantly improved performance. A normalized form of the resynthesized sound is compared with a normalized form of the target sound. This greatly reduces the error between similar sounds as the volume ratios are not required to accurately evolve, only the tonal quality. In addition, certain balances of partials are only possible with low amplitude FM modulators, so the normalization process is necessary to bring these up to the right level.

We quantify the improvement that adding normalization adds with experimental results in Section 6.3.

After adding this to the GA, it was necessary to extend the synthesizer, as a fit configuration could also be excessively quiet or loud. We added a new volume scaling parameter in the synthesizer which was set to the scaling factor from normalization so when played back the whole synthesizer sound is itself normalized.

Chapter 6

Experimentation

In this Chapter we experiment with our system to optimize our parameters and prove the new features we have added are true improvements. In Section 6.1 we describe how we run our experiments, and then in Sections 6.2, 6.3, 6.4, and 6.5 we present our experiments.

6.1 Experimental Setup

We run several experiments in the following sections, all of which run with the same base and then we vary a parameter. The basic constants are:

- Population Size = 100
- Number of Generations = 30 (this is chosen in)
- Elitism Percentage = 10%
- Mutation Rate = 200 (1 in 200 genes will mutate)
- FFT Window Size for Spectrogram Analysis = 8192

Each experimental configuration is tested on a suite of 7 sounds which have been selected because they are fairly approximable with the synthesizer and cover a range of complexity. They are shown in table 6.1.

All experiments are repeated 5 times for each target sound and averaged to smooth out the stochastic nature of the GA.

6.1.1 Distribution

Due to the circa 20 minute runtime per sound, we distribute our experiments across multiple machines. The Java program is compiled to a JAR file and accepts parametric arguments on the commandline. We then use a perl wrapper script to run the experiments across the **Condor** distributed environment [4], performing many CPU-hours of tests at once on the DoC computing pool.

With 7 sounds at 5 repeats, each experimental configuration takes 35 runs. With a mean runtime of 6 minutes per 10 generations of 100 individuals, a typical experiment can

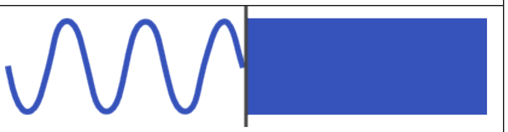
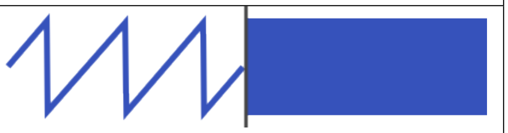
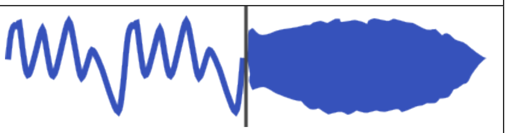
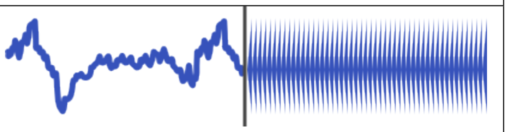
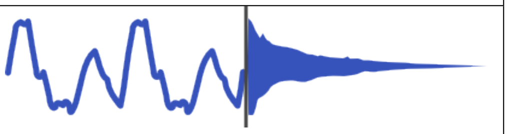
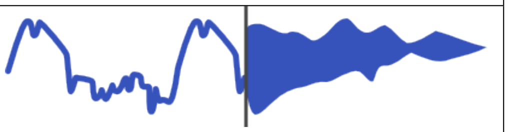
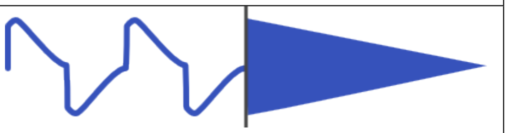
Sound & Description	Waveform Detail + Overview
Sine Wave continuous amplitude and pitch	
Sawtooth Wave continuous amplitude and pitch	
Oboe Note from BBC Orchestral Archive	
Electric Organ Tone produced by another FM Synthesizer	
Piano Note from BBC Orchestral Archive	
Detuned Synth Lead produced by another synthesizer	
Modified Square Wave with a 2 second volume decay	

Table 6.1: All the test sounds used in our experiments

easily require 9 hours of CPU time per data point. This might seem like an excess of computing time to get our results, but a lot of averaging is required to find trends without being affected by the stochastic nature of the GA.

6.2 Number of Generations

As explained in Section 5.7, the GA is set to terminate after a fixed number of generations. However, choosing the cut-off point is not easy without data. We experiment by running the GA for a long time and then choose the cut-off point from the results.

We run the standard test configuration as defined above for 100 generations. The results for each test sound are plotted in Figure 6.1.

Analyzing the graph, we see each sound exhibiting the standard GA convergence curve similar to an exponential decay, as we expected from background literature [12, 11]. Each experiment tends towards the best possible resynthesis at 0 error, with each downwards jump indicating a beneficial mutation spreading throughout the population.

Most of the sounds show only a small amount of improvement over their starting point, reaching about 50 to 100 error. However as this is mean error it is slightly misleading, and most of the sine wave runs end with a best individual error of only 1.0, a near-perfect match. The sawtooth wave is an outlier on the graph, and its analysis is presented in Subsection 6.2.1 below.

The termination generation for the experiments that follow is chosen to be 30 generations, as this is where most of the sounds reach an acceptable error value. Excluding the sawtooth wave, the mean error of the sounds is 75.6 at generation 30, which is within 22% of the mean error of 61.8 at generation 100.

6.2.1 Sawtooth Wave

Interestingly the sawtooth wave never reaches a very good error value and has the most continuous improvement over time, which is believed to be because of its infinite series of partials becoming slowly matched. The DFM equation 2.2 that the synthesizer model operates upon can only generate a finite number of partials, so no exact match is ever possible. It appears though that over time the GA finds ways to add more and more of those partials accurately.

Figure 6.2 shows the results of one of the 100 generation runs compared with the target sawtooth wave. The waveform and sound are very similar, but the spectra have a lot of difference. The blue FFT of the pure sawtooth has many artifacts due to its infinite nature – reflected frequencies and aliasing cause the appearance of extra features which aren't actually present in the pure tone [2].

Therefore the sawtooth is hard for the system to be matched with DFM, but this is due to its infinite nature not being easily recognized by our similarity metric.

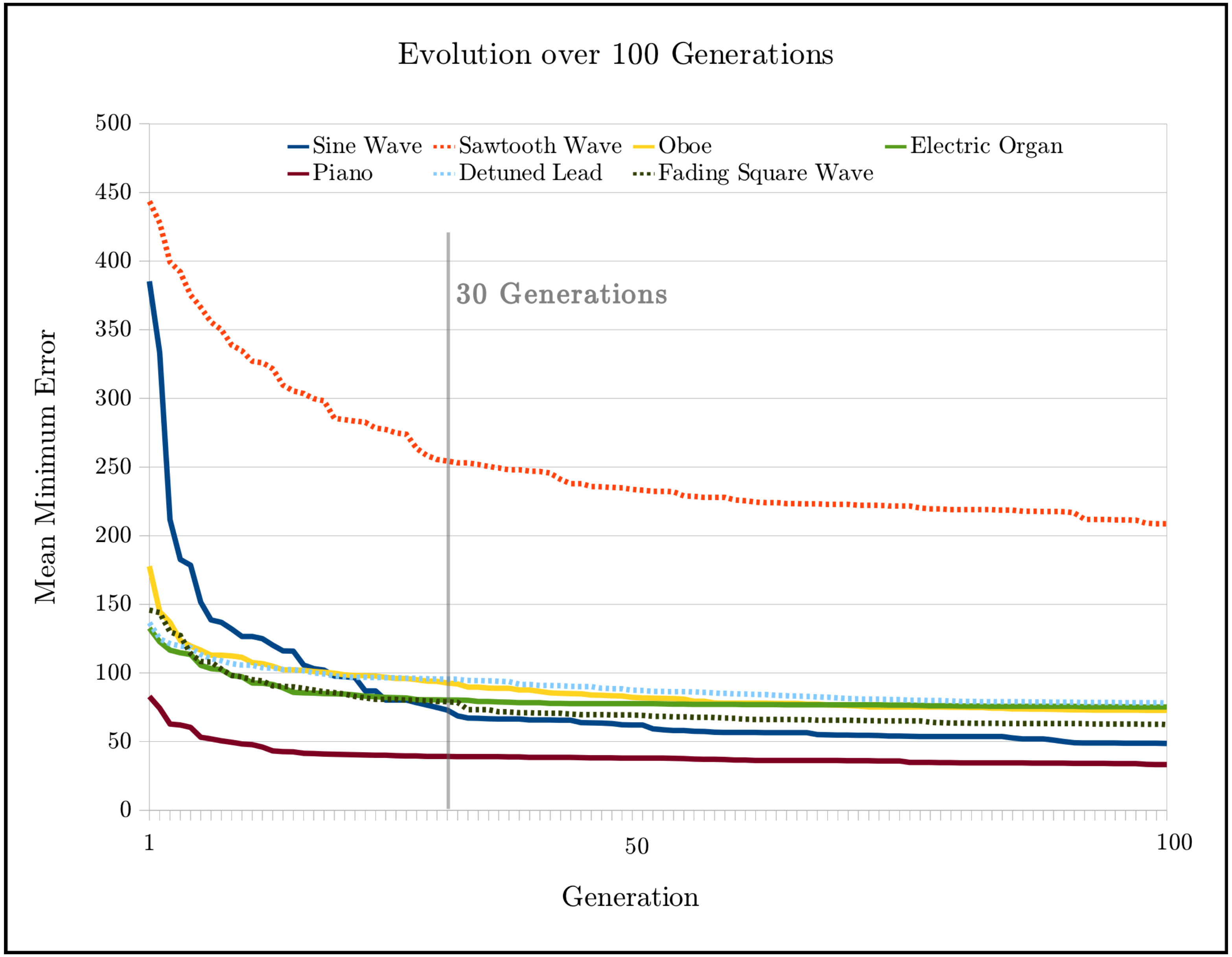


Figure 6.1: 100 generations of convergence for the 7 test sounds

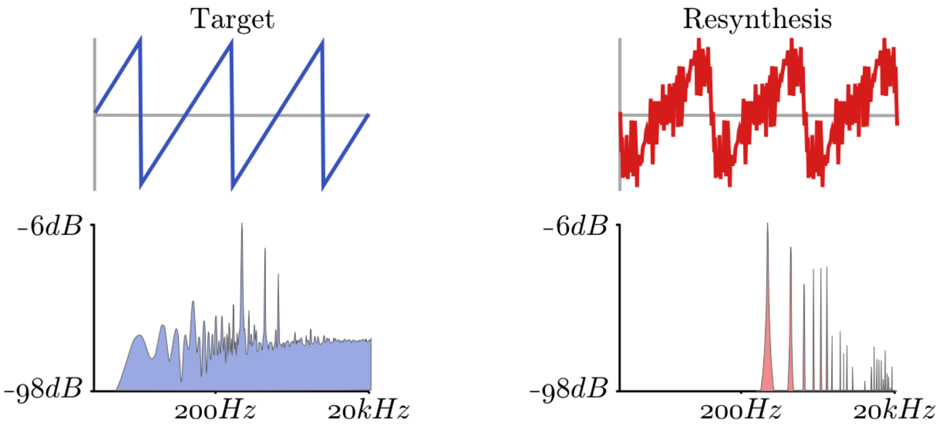


Figure 6.2: Waveforms and spectrums of the target and resynthesized sawtooth wave compared

6.3 Normalization

The GA implements normalization on both the target and resynthesized sounds to improve the quality of matching, as explained in Section 5.8. We show here that it is a valuable addition to the system by measuring performance with and without it.

We test the convergence rate both with and without normalization for the standard 7 test sounds and take the average of 5 repeats. The below table summarizes the mean fitness of the population after 30 generations for both tests:

	Without Normalization	**With Normalization**
Mean error at Gen 30	176	100

Normalization clearly adds a better convergence to the GA, and the mean fitness across generations is show in Figure 6.3. The mean error starts higher with normalization, but converges at a faster rate, indicating the GA has a better ability to 'discern' between the quality of individuals.

6.4 Balancing the Fitness Function

Recall that the error formula, Equation 6.1, is a two-part weighted sum of the error from the norms and centroids formulæ, as taken from the Lai paper [10]. The paper shows that a fitness function summing both error metrics is better than either individually. However it only ever uses an equal weighting of $\frac{1}{2}$ for each metric, but there is no evidence presented in the paper that this is the optimal weighting. We aim here to find the perfect balance for the weightings. We rewrite it as:

$$error(a) = \alpha N(a,b) + (1-\alpha)C_{dif}(a,b) \tag{6.1}$$

This gives us a new parameter α we vary between 0 and 1 to find the optimal weighting to help the convergence of the GA. We test values from 0 to 1 in increments of 0.1 and record the minimum norms error N and centroids error C_{dif} for each configuration. Again

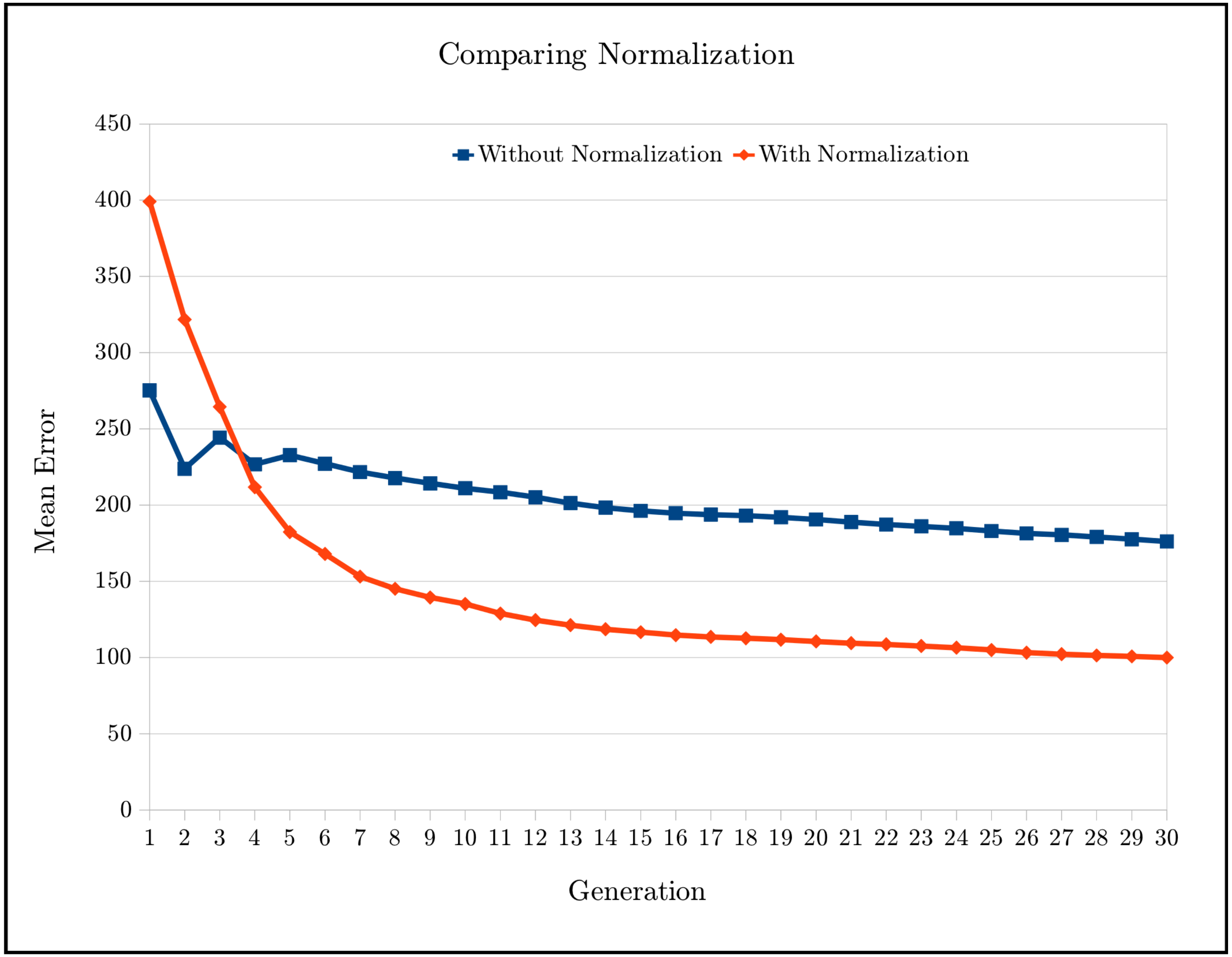


Figure 6.3: The performance increase from using Normalization in the Fitness Function

Tournament Size	**First Generation within 30% of Final Error**
2	13
3	11
4	10
5	8
6	9
7	9
8	7

Table 6.2: Effect of Tournament Size on convergence

these results are averaged across the 7 sounds and 5 repeats for each. Figure 6.4 shows our findings.

Unsurprisingly, when $\alpha = 0$ and only the Centroids error is considered, the Norms error remains very high; conversely when $\alpha = 1$ the Centroids error is very high. It is also clear the optimal weighting to minimize both error measures is not around 0.5, but more around 0.8. Thus we use $\alpha = 0.8$.

6.5 Finding the Optimum Tournament Size

The last parameter we experiment with is the tournament size used in the selector, as detailed in Section 5.3. This was again copied from the Lai paper with no proof of its optimality, where they suggest using a tournament size of 5.

We experiment with the tournament size between 2 and 10, using the optimal fitness function as achieved above. We use the same test suite of 7 sounds for 5 repeats. Figure 6.5 shows our findings.

We note two things on this graph. Firstly a larger tournament size leads to faster convergence, as the successive curves for larger tournament sizes fall underneath each other. Secondly, the tournament size does not seem to affect the final generation's error measure, as all the curves have approximately the same value at generation 30, and the variation between them seems random.

Table 6.5 shows our results re-arranged to show the speed of convergence. For each tournament size we present the number of generations it took to reach within 30% of the final error achieved at the 30th generation. There is a clear downwards trend as the tournament size increases to faster convergence.

We conclude that a tournament size of 7 or 8 seems ideal. Because each individual is evaluated independently before the tournaments, there is no performance hit from increasing the tournament size. The main problem with increasing is that the GA becomes more likely to converge to only local minima, rather than the globally best solution. This is because with larger tournaments, fewer solutions can win, so the genetic material in the population gets limited and specialized rapidly.

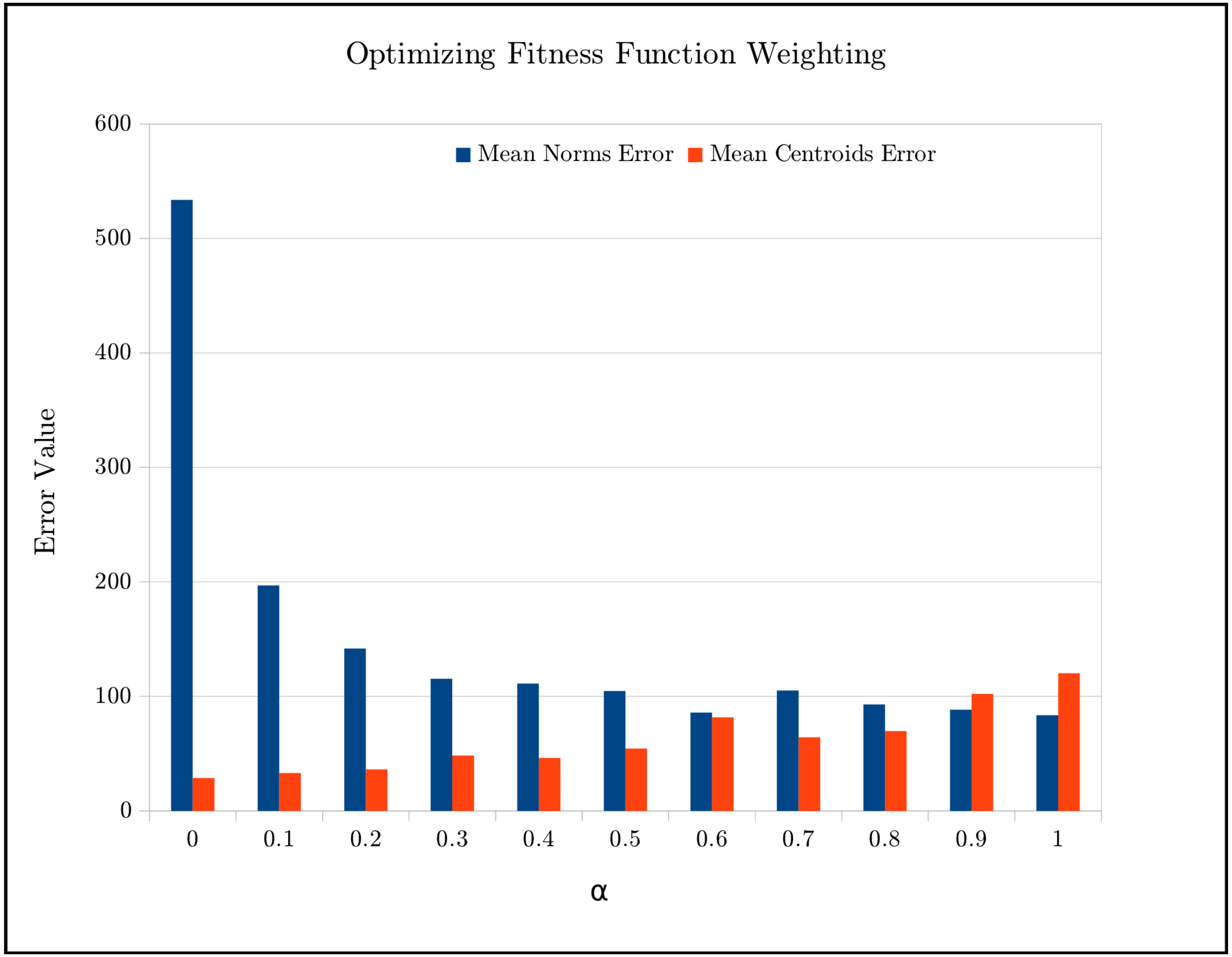


Figure 6.4: The effect of changing α on the mean Norms and Centroids error measures

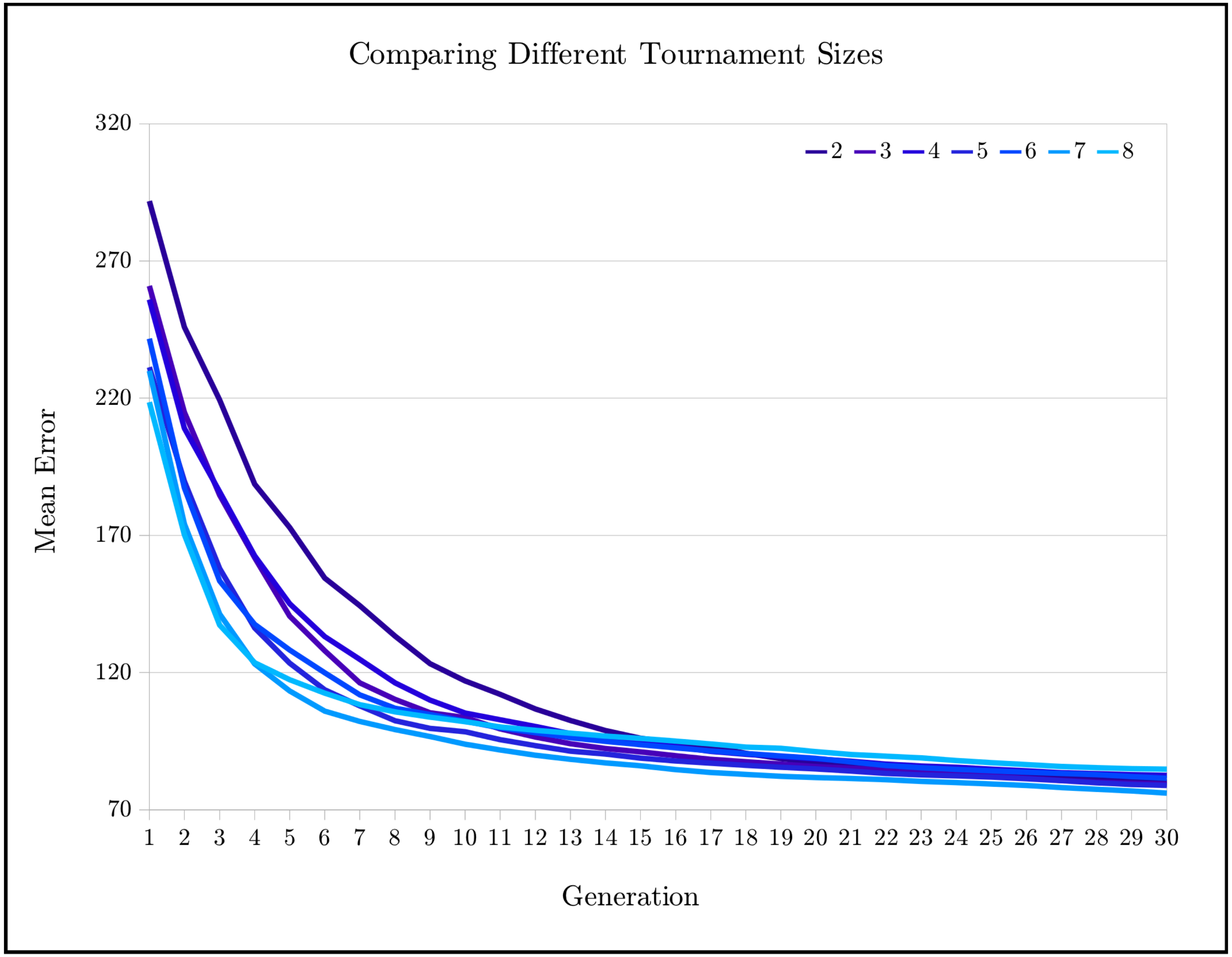


Figure 6.5: The effect of changing tournament size on convergence

Chapter 7

Evaluation

In this Chapter we evaluate the system, questioning how well it can resynthesize sounds in Section 7.1, discussing the runtime in Section 7.2, and evaluating against the goals we set in the introduction in Section 7.3.

7.1 Resynthesizing Sounds

The main question we must ask when evaluating the system is: how well can it reproduce sounds? We analyse this in two ways below, statistically and perceptually, drawing upon mathematical and human measurement of the sonic difference.

7.1.1 Statistical Difference

One of the problems in drawing any statistical measure of the overall resynthesized sounds' similarity is that the only formula we have is the one used in the fitness function. If there were a better measure, we would have used that instead. This makes our measurement circular, as it is not really surprising that the GA finds sounds which are similar by the same fitness function it uses.

Regardless, we can still draw conclusions from the mean error value the system typically accomplishes. For our test suite we typically achieve an error between 50 and 70 for each sound, indicating a fairly good match. Compared to early iterations of the system which could never drop below 100, it is relatively accurate. Unfortunately, we cannot compare our error measures with the Lai and Yong works as their implementations differ and use a different scale to ours.

For the simplest sound, the sine wave, an error of around 1 can be obtained, indicating a perfect match. However, as explored in Subsection 6.2.1, the basic sawtooth wave is problematic to match, and at generation 30 still has an error of around 250.

7.1.2 Perceptual Difference

The best measure for such a system is in human perception – we ask: how similar are the resynthesized sounds to a human listener? Although we cannot present any numbers for this measure, we believe that the similarity is definitely there. For example, in the oboe analysis below we show that the harmonic fingerprint is imitated very well, and at a basic level they sound the same to human listener, even though one can still tell the difference.

Most of the perceptual difference is in the much finer details in the sound that cannot be replicated by the synthesizer model. Sounds such as the resonance of a piano body, the slight noise from wind instruments, or slight pitch variations due to vibrato are all missing from the resynthesized sounds. This is typical of synthesizers though, and compared to third party programs we believe our resynthesized sounds are favourable. Finding a way to resynthesize perfectly would indeed be solving a large problem in synthesizers in general – making them sound realistic.

One of the things that can mask the difference is not listening to a single note, but instead playing back an entire melody with the resynthesized sound. When there is more motion it is generally harder to tell the difference, as several pitches can be overlaid at once.

We aim to show off our demonstration sounds in the presentation.

7.1.3 Oboe

If we examine a successful sonic match, the Oboe, we can see how well the system performs. This is a sound also matched in the Yong work [17], so we can compare the output. The harmonic profile of the oboe is fairly constant throughout, so taking the spectrum is a useful measure. Figure 7.1 shows waveform, spectrum, and spectrogram details for both the target sound and its resynthesis.

The waveforms for the sounds appear fairly similar, indicating the same low-partials presence and high-partials absence. The spectrum rigorises this detail, and proves that they have very similar harmonic profiles. Indeed, it would only appear to be the slightly noisy elements, the 'filler' on the spectrum between the partials, that the resynthesis is missing. This is 'breath' noise made by the blown nature of the Oboe, sounds that the FM synthesizer model cannot really replicate.

However, we have only considered the matching of the spectrum, which is the same accomplishment that the three previous works achieved. To show our system has some improvements, we must also analyse the time-based evolution of the sounds.

The amplitude envelope of resynthesis is fairly true to the original Oboe, as can be seen in the full-scale waveform. The attack of the resynthesized sound is a little too long, but the start of the oboe is more complex than a simple attack and we must consider that the ADSR model only gives limited control. Overall the target sound is fairly complex and 'wavey', so the smoother envelope in the resynthesis is a good approximation.

For the harmonic evolution, the resynthesis features more change than the original oboe does, especially with a two high partials that appear and disappear. These add to the non-oboe quality, but are fairly quiet and so do not have too much of an effect. Perceptually, this adds a slight metallic, synthetic quality in the sound.

The oboe is one instrument in the woodwind family, and they all share the basic characteristics. Because the oboe matches so well, we believe the rest of the family can also be resynthesized well.

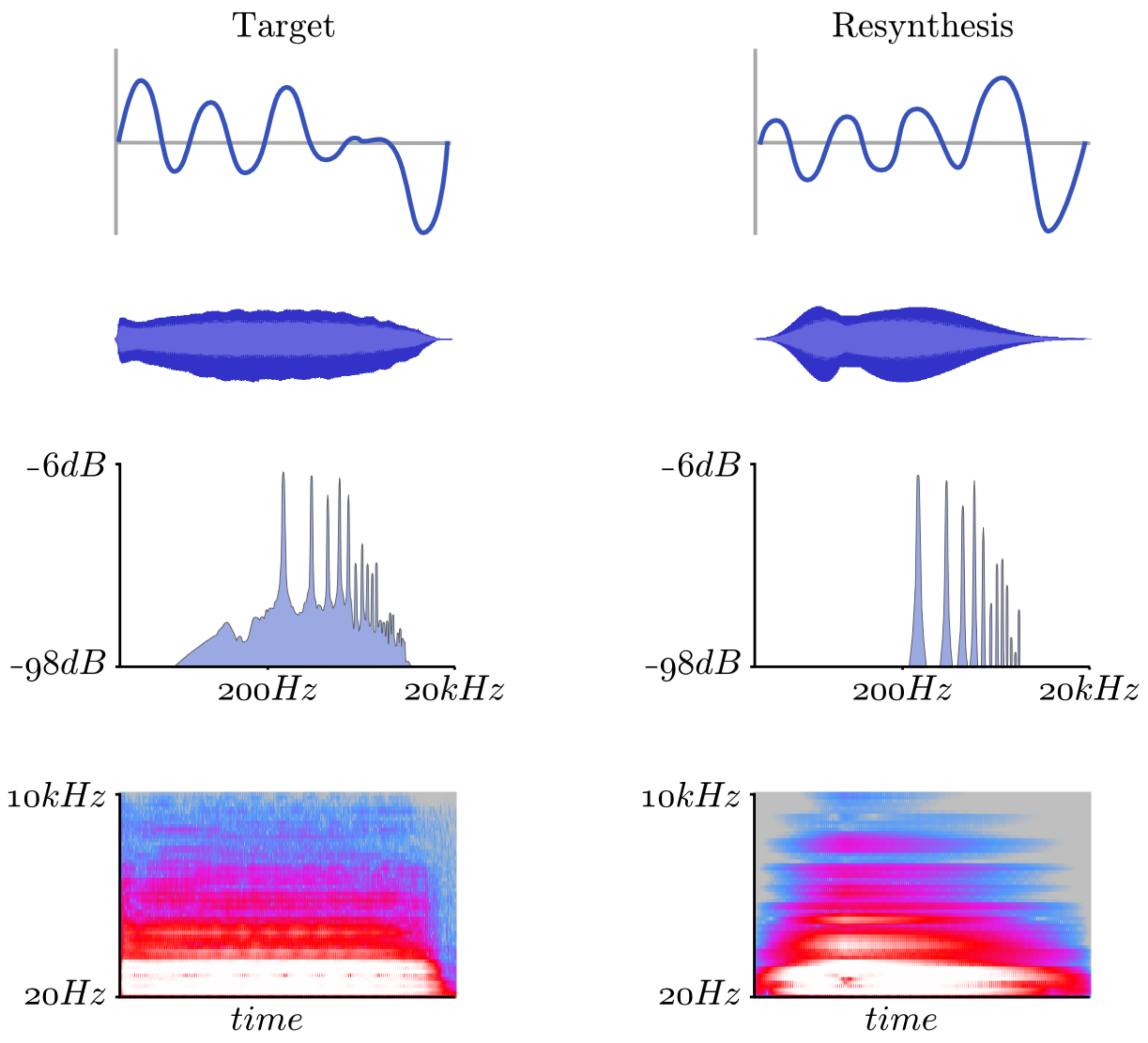


Figure 7.1: For both Target and Resynthesized Oboe sounds: waveform detail, total waveform, spectrum, and spectrogram

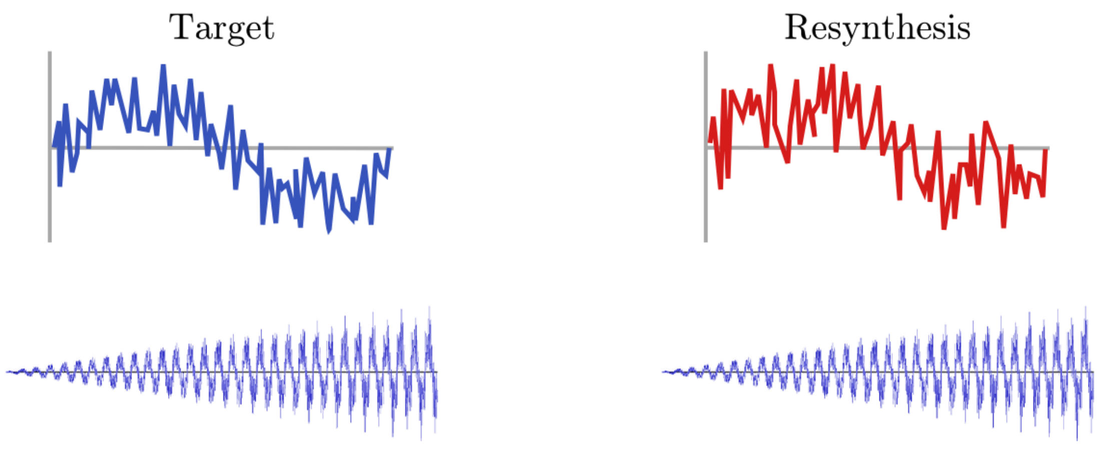


Figure 7.2: A synthetic sound created by our synthesizer almost perfectly resynthesized

7.1.4 Self-Reproduction

As we have now noted, the DFM synthesizer model is not ideal for all sounds, and is not even capable of accurately reproducing a sawtooth wave. However, there is one surefire set of sounds it can reproduce, and that is those sounds made with the same synthesizer model to begin with.

After some experimentation, we find that most sounds created by the synthesizer can be resynthesized from their waveform in 30 generations. Often the end error measure for the last individual is under 20. For example, Figure 7.2 shows a random sound from the synthesizer re-produced very accurately. The harmonic profiles are very similar and the envelope matching is very accurate.

This is a slightly less useful application of the technique, but still has potential. For example, a music producer listening to a published piece of music might want to use the same synthesizer sound; if it was produced by this model it can be replicated very easily. This result also promises that with a more complex synthesizer model, a wider range of sounds could be matched to a lesser degree of error.

7.2 Runtime

A typical run of the system (a population of 100 for 30 generations) takes approximately 10-15 minutes on a laptop with an Intel Core 2 Duo processor. Note that the runtime is linear with the length of the sound to be matched – a longer sound means more frames in the spectrogram, meaning both synthesized sound generation and comparison take longer.

This runtime is satisfactory, especially since it needs only be done once for a target sound and runs offline. Of course the user can choose to run for fewer generations to decrease the runtime at the sacrifice of some accuracy; conversely, as the GA seems to continue converging beyond the 100th generation, the user could run for as long as wanted. On the other hand, for potential commercial applications of the concept only runtimes of under a minute would be acceptable – a music producer wanting to automatically program a synthesizer would not want a long interruption to their creative flow.

However, speed was never the primary aim of the project, as we were aiming more to prove our concept. We believe that the system can be optimized to run a lot faster, and provide some suggestions on how to do so in the conclusion Section 8.2.

7.3 Fulfilling Goals

In the introduction we stated that we are driven by two motivations – both the reproduction and exploration of sounds. We now analyse if the end result is useful to these goals.

7.3.1 Sound Reproduction

As we have shown throughout, some sounds are much more reproducible than others; the oboe works well, but the sawtooth wave not so much. However this is not necessarily due to a weakness in the overall technique, only the synthesizer model employed. We did quote Yong earlier in 2.1, stating that DFM is theoretically able to reproduce any sound; we have found on the way that this is not necessarily true in real-world application.

We cannot therefore reproduce *any* sound. As we have shown, the sounds that can be reproduced the best are those that were produced with the synthesizer model itself to begin with. This is not that surprising though.

Compared to the other works, we have extended sound reproduction to the time domain, often succeeding at matching much of the harmonic evolution of sounds. This is the main achievement of our project.

However, the sound reproduction is still limited; but usefully, the GA and fitness function seem extensible to other synthesizer models. We will suggest further work in the conclusion that could extend the system to many sounds in the soundscape. Therefore, whilst we have not hit the mark on being able to 'reproduce any sound', we have shown the same GA-synthesizer model from previous works can be extended to time-evolving sounds, and paved the way for future work.

7.3.2 Sound Exploration

As stated in the introduction, fully accurate reproduction is not the goal when exploring, as sounds that are only partially similar are more desirable. We have achieved this as a side effect of aiming for full reproduction.

Some sounds do not provide a very convincing resynthesis, such as the sawtooth wave, but the matches still have potential for use. We can also create partial matched sounds by stopping the GA early before full convergence takes place, creating synthesizer configurations that have started to match up the characteristics of the target without adding every detail.

Additionally, the sounds reproduced are never 100% accurate either, due to limitations from the synthesizer model.

Therefore, whilst we did not strictly aim for an exploration system, our result is useful in this regard.

Chapter 8

Conclusion

In this Chapter we review all that we have covered in Section 8.1 and suggest further work that could enhance the sound-matching capabilities in future systems in Section 8.2.

8.1 Review

In this report we detailed our system that resynthesizes sounds using a Genetic Algorithm to adapt a synthesizer model and described several innovations that we add on top of previous work to improve convergence and quality of the sounds.

We gave background knowledge to the problem in Chapter 2, detailing concepts from both synthesizers and genetic algorithms. We also described the three previous works on the matter and pointed out their drawbacks upon which we wanted to build.

An outline of our system was presented in Chapter 3 with a diagram and a short description of both the synthesizer and GA parts. We also noted the detail that we chose to exclude from the start – automatic pitch detection.

In Chapter 4 we detailed our synthesizer model, describing our implementation and giving a diagram showing the audio flow. We also described two major problems that were overcome in implementation, firstly with 'Envelope Mess' in Section 4.3, detailing how we had to actually restrict our synthesizer so it made 'more sensible' sounds. We then described our innovation in controlling the evolution of the FM modulators in Section 4.4.

Following this we presented the details of our GA in Chapter 5, outlining its fitness function, selection method, and genetic operators. We then showed our use of regulatory genes as an extension to the state of the art in 5.5 and described how they improve convergence. We also summarized how adding Normalization improved the fitness function in Section 5.8.

In Chapter 6 we showed our experimental findings. We first outlined how our experiments were run on a distributed platform consuming hundreds of hours of CPU time, before describing how we optimized the system for rapid convergence in Sections 6.2 and 6.5. In Section 6.3 we proved that adding Normalization definitely improves the fitness function, and in Section 6.4 we improved the fitness function as drawn from the Lai paper [10].

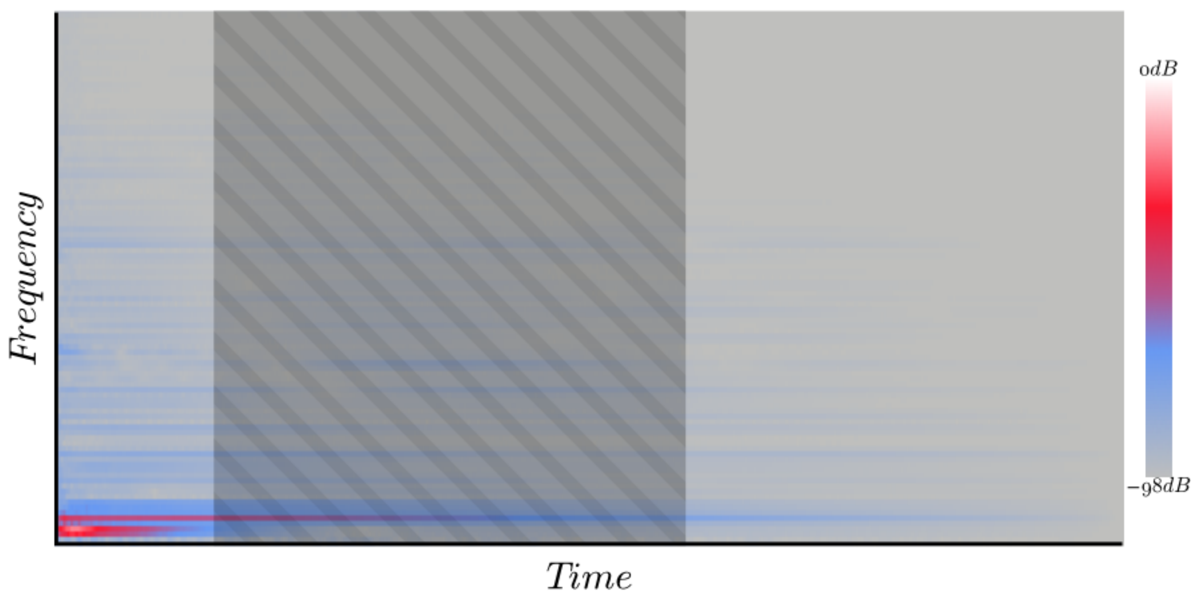


Figure 8.1: In this spectrogram of a piano note there is little variation between frames in the blocked-out sustain phase, so only a single comparison would be necessary for this range

We provided an evaluation of our system in Chapter 7, noting where it works well and where it fails. We noted that there is a wide scope for further work, and now present our suggestions.

8.2 Further work

There are many possible lines of investigation to follow from this work – here we suggest three directions that seem promising.

8.2.1 Optimized/Co-Evolving Fitness Function

The runtime of the fitness function is linear with the length of the target sound, but it could be optimized to run faster for most sounds. As many of the matched sounds reach a steady state in their sustain phase, it is not necessary to compare every frame of the spectrogram at this point.

For example, the spectrogram of a piano note in Figure 8.1 shows that the blocked-out sustain phase does not have much variance from frame-to-frame. By pre-processing the target sound, we could save the fitness function a lot of work by identifying ‘key frames’ in the sound and comparing only these. In this example, we would ignore perhaps 40% of the frames, which cuts 40% of the runtime off of comparison.

This technique could be further improved by making the fitness function a ‘parasite’ in a coevolution framework [14]. This would mean making the fitness function also subject to genetic operations, allowing it to adapt over time to ‘prey’ on the weaknesses in the general population. It could then focus on the parts of the sound that are generally badly matched, forcing the evolution to adapt in these areas faster. The fitness function would then only need to evaluate a handful of spectrogram frames per generation, giving both a speed increase and enhanced convergence.

8.2.2 Evolving Fitness Formula

In section 6.4 we experimented with and optimized the weighting used in the fitness formula suggested by the Lai paper [10]. It is likely that rather than using a fixed weighting but allowing it to adjust itself per generation in a second co-evolution would yield better performance, focussing the GA on different characteristics in the sound in different generations.

Additionally, we note that the amplitude envelope is relatively poor at being matched, in comparison with the tonality aspect of the synthesizer. An extra error metric based purely upon the per-window amplitude difference of the target and resynthesized sounds would likely help in the evolution of this – and this would add another weighting to the fitness formula which would need optimizing.

8.2.3 More Complex Synthesizer Model

In Chapter 4 we outlined our synthesizer model and indicated DFM synthesis is the sole technique used here. Whilst the Yong paper [17] indicated that "Theoretically, we may synthesize any kind of sound by tuning and tweaking these parameters", in practice it has shown itself to not be so capable. For example, even the sawtooth wave is not very well matched as we saw in Subsection 6.2.1.

A more complex synthesizer model is suggested for usage, although it would also bring with it a more complex set of parameters requiring a longer time to evolve. From knowledge of the commercially-available FM synthesizer **'Operator'** [1] we suggest several enhancements:

- **Add modulator waveforms** other than the sine, such as a sawtooth or square. A sawtooth modulator on a sine carrier produces an extremely rich texture that takes many sine modulators to reproduce.
- **Allow the modulator/carrier structure to vary** in more complex ways. Figure 8.2 shows a comparison of possible structures of our synthesizer against **'Operator'**. A more complex FM structure leads to a larger set of possible tonalities, but would need a suitable genetic encoding.
- **Add an LFO** – a low frequency oscillator – that can modify another parameter at a rate typically around 1-10Hz. For example, applied to pitch this can recreate vibrato easily – a sonic feature that is unmatchable in the presented implementation.

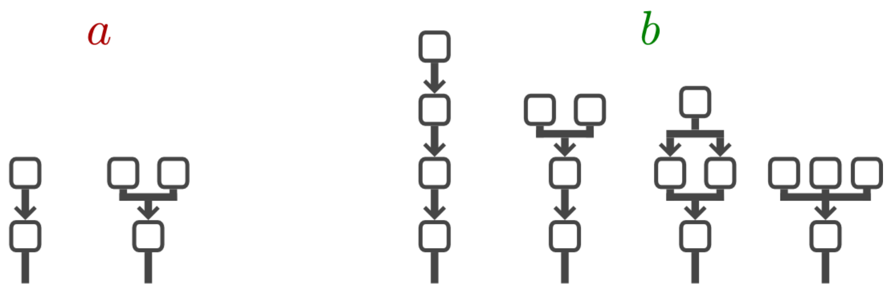


Figure 8.2: **a)** The only two possible carrier/modulator structures in our synthesizer model **b)** 4 of the 19 structures possible in **'Operator'**, each using 3 modulators + 1 carrier

Bibliography

[1] Ableton AG, *Ableton Live User Manual.* 2010

[2] C. Sidney Burrus, *Fast Fourier Transforms.* Rice University e-Book `http://cnx.org/content/col10550/latest/`, Creative Commons License 2010

[3] John M Chowning, *The Synthesis of Complex Audio Spectra by Means of Frequency Modulation.* Journal of the Audio Engineering Society, September 1973, Volume 21, Number 7, page 526

[4] Condor Team, *Condor Website.* `http://www.cs.wisc.edu/condor/`, 2011

[5] Ralph Deutsch et al, *Patent : ADSR envelope generator.* G01H 102, 1978

[6] Damien Di Fede, *Minim Website.* `http://code.compartmental.net/tools/minim`, 2011

[7] Damien Di Fede et al, *Music Programming in Minim.* Proceedings of the 2010 Conference on New Interfaces for Musical Expression, 2010

[8] Harris, F.J., *On the use of windows for harmonic analysis with the discrete Fourier transform.* Proceedings of the IEEE, Jan 1978, Volume 66, Page 51

[9] Damien Karras *Sound Synthesis Theory.* Wikibooks e-Book `http://en.wikibooks.org/wiki/Sound_Synthesis_Theory`, 2011

[10] Yuyo Lai et al, *Automated Optimization of Parameter for FM Sound Synthesis with Genetic Algorithms.* 2006 International Workshop on Computer Music and Audio Technology

[11] Zbigniew Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs.* ISBN 3-540-60676-9, Springer-Verlag Berlin Heidelberg, 1992

[12] Melanie Mitchell, *An Introduction to Genetic Algorithms.* ISBN 0262133164, MIT Press, 1996

[13] Klaus Meffert et al, *JGAP (Java Genetic Algorithms Package).* `http://jgap.sourceforge.net/`, 2011

[14] W. Rand, *Genetic Algorithms in Dynamic and Coevolving Environments.* Thesis 2006, University of Michigan

[15] B.T.G. Tan, S.L. Gan, S.M. Lim, and S.H. Tang, *Real-Time Implementation of Double Frequency Modulation (DFM) Synthesis.* Journal of the Audio Engineering Society, November 1994, Volume 42, Number 11, page 918

[16] B.T.G. Tan, and S.M. Lim, *Automated Parameter Optimization for DFM Synthesis using the Genetic Annealling Algorithm.* Journal of the Audio Engineering Society, January/February 1996, Volume 44, Number 1/2, page 3

[17] Shi Yong, *Automatic FM Synthesis Based On Genetic Algorithm.* Music Technology, McGill University, April 2007, `http://www.music.mcgill.ca/~yong/mumt307/mumt307.html`

Appendix A

Glossary

ADSR - Attack / Decay / Sustain / Release envelope model. Explained in Section 2.3.

DFM - Dual Frequency Modulation synthesis technique. Explained in Section 2.1.

FFT - Fast Fourier Transform. Explained in Section 2.4.

FM - Frequency Modulation synthesis. Explained in Section 2.1.

GA - Genetic Algorithm. Explained in Section 2.5.

JGAP - Java Genetic Algorithms Package - library used for basis of genetic algorithm. Explained in Chapter 5.

LFO - Low Frequency Oscillator - a method used to add subtle modulation to a synthesizer parameter. Explained in Subsection 8.2.3.

MIDI - Musical Instrument Digital Interface - format used to represent musical notes. Explained in Section 4.1.

Minim - Java-based audio library used for basis of synthesizer. Explained in Chapter 4.