Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

Victor Gabaldon Garcia

30th April 2024

Exploring Genetic Algorithms for Sound Synthesis Parameter Optimisation

Project supervisor: Prof. Richard Watson
Second examiner: Prof. Leslie Carr

A project report submitted for the award of
BSc Computer Science

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
ELECTRONICS AND COMPUTER SCIENCE

A project report submitted for the award of BSc Computer Science

By Victor Gabaldon Garcia

Sound synthesis has become an increasingly relevant subject in the past decades, finding countless applications in fields like music production or game sound design. A large number of synthesiser models exist, many of which consist of convoluted parameters that require manual adjustment. The complexity of synthesisers along with sounds being difficult entities to describe and categorise effectively can make their synthesis challenging even for experienced users. Inspired by the biological mechanisms of evolution, genetic algorithms could be used to find the optimal parameters of a desired sound wave. In this project, an evolutionary sound optimisation library was designed and developed to conduct a wide range of simulations. Using this framework, the efficacy of genetic algorithms in sound synthesis optimisation was explored and assessed. Genetic algorithms demonstrated superior performance over hill climbing in parameter optimisation. Multiple factors influencing their success have been analysed, including the fitness evaluation method used, population size, number of generations, and mutation rate.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/ EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

Some signal processing methods were inspired by libraries such as `dasp-signal` or `synthrs`.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching

staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

Acknowledgments

I am deeply grateful to my grandmother, mother, father, and sister, for their unwavering support and encouragement throughout all these years.

Special thanks to my supervisor, Prof. Richard Watson, for his invaluable advice and guidance, which have been essential in steering this project in the right direction.

Last but not least, I want to sincerely thank my friends Alec and Ed for taking the time to review this work and provide insightful feedback.

Contents

1. Overview

1.1. Motivation

Since the invention of the synthesiser more than six decades ago, sound synthesis has become an increasingly relevant field across multiple audiovisual industries, including music, film, gaming, marketing, and extended reality [1]. This growth in the market has resulted in great advances and developments of new forms of synthesis, leading to a wide variety of products [2].

Synthesisers can be analogue or digital and software or hardware-based. Apart from their type, each manufacturer typically uses a unique architecture and set of components and parameters. Such a complex environment can make it difficult for unfamiliar users to get started with sound production, where even experienced users may struggle to find an adequate timbre [3].

In this project, the use of genetic algorithms (GAs) as a sound optimisation technique is explored by attempting to determine the parameters of a synthesiser given a target sound wave. There has already been a significant amount of research in employing GAs in the field of sound synthesis, and most of the literature concludes on their effectiveness [4], [3], [5].

However, the implementations in these experiments are often very specific to the problems presented and do not provide a wider overview and comparison of GAs with other optimisation techniques. They also do not explore how GAs perform in multiple forms of synthesis, components, configurations, and objective evaluation methods. Finally, there are not many libraries and other software tools available to replicate these experiments and even apply these techniques in industry.

1.2. Problem

Finding the precise synthesiser parameters that produce a desired soundwave can be a challenging and tedious task, especially for novice users and those unfamiliar with a particular electronic instrument. Apart from the differences across multiple designs, challenges arise from the lack of intuitive correspondence between parameter values and the produced sound. This may lead users to settle for an imperfect timbre, which can ultimately have an impact on the quality of the produced musical work or sound effect.

1.3. Goals

This project aims to solve some of the challenges that come with finding the right parameter values in a software or hardware synthesiser. This involves both a design-and-build and research component.

1.3.1. Design and Build Goals

1. Write a library that implements a genetic algorithm for synthesiser parameter optimisation.
2. Support multiple configurations, synthesis methods and components, and approaches to keep the library flexible and general-purpose.
3. Design, implement and document an API that can be later used for research and by third-party apps.
4. Emphasise performance, efficiency, and usability; getting accurate results within seconds.
5. Ensure the library is extensible and follows a modular design, so components can be replaced or upgraded independently.

1.3.2. Research Goals

1. Investigate the effectiveness and performance of genetic algorithms in optimising the parameters of a synthesiser by comparing them to other techniques.
2. Evaluate different configurations and strategies when running genetic algorithms, including the fitness function, population size, mutation rate, and number of random individuals introduced per generation.
3. Explore multiple sound synthesis methods and how each performs and converges for specific target sound waves.
4. Analyse some of the trade-offs that come with genetic algorithms, such as execution time vs accuracy of solutions, or diversity vs convergence.
5. Identify potential limitations and challenges of using genetic algorithms for this purpose and what other areas could be explored in the future.

1.4. Scope

The design and build component consists of a parameter optimisation library that attempts to reverse-engineer synthesiser parameters through the use of genetic algorithms. Both subtractive and additive synthesis are supported, including a filter component, as well as two fitness evaluation methods, based on frequency and time-domain analysis. The library does not currently integrate with actual synthesisers but still facilitates fundamental research by enabling users to customise the synthesis process and genetic algorithm.

The research cases covered encompass a review of parameter estimation and performance across different synthesis methods, optimisation algorithms, fitness evaluation methods, population growth, population sizes, mutation rates, and number of random individuals added per generation.

2. Background

2.1. Signals

A signal is a function $f(t)$ that conveys information about a physical phenomenon, such as voltage, radio waves, or sound. Signals can either occur naturally or be created artificially [6] and be converted from one medium to another through a transducer [7].

Signals can be analogue, such as voltage-controlled oscillators or radio waves, or digital, typically represented as samples, an array of bits where the amount of samples taken per unit of time is known as the *sampling rate*, and the number of bits per sample, the *bit depth*.

When comparing two signals, it is important to obtain the relevant characteristics of each, including frequency composition, amplitude over time, and sometimes phase [8]. Multiple digital signal comparison methods exist, primarily based on the time and spectral domain. In this section, we will examine some of them and when they should be used.

2.1.1. Time-Domain Analysis

An intuitive yet naive approach for comparing the similarity between two waveforms is to perform the Mean Square Error (MSE) over the vector of samples. This method measures the expected quadratic difference between the samples of both signals. Given two discrete signals u and v, each with n samples, the cost function J may be defined as

$$J(u, v) = \frac{1}{n} \sum_{i=1}^{n} (u_i - v_i)^2 \tag{2.1}$$

However, this method has multiple disadvantages. Firstly, both signals must be normalised and the larger signal must be trimmed, resulting in information loss. Subsequently, some form of phase alignment needs to be performed, otherwise, the differences between two samples at the same position would not be representative. In continuous signals with the same frequency, phase alignment can be solved by finding the phase that minimises the MSE.

Time-domain analysis does not give us information about the frequency components of a signal. Since harmonics are what give sound waves their distinctive tone, this method becomes less useful as the complexity of the sound wave increases. Even so, time-domain analysis may still work well at comparing simple signals like the sinusoidal, especially when the amplitude changes in time (such as with envelopes or faders). There exist more advanced techniques like cross-correlation, which performs a *sliding dot product* of two signals [9], or dynamic time warping, which measures the similarity of two sequences varying in speed [10].

2.1.2. Spectral Analysis

A more reliable and efficient approach to comparing the similarity between two waveforms is to look at the frequency spectrum, as this encodes the frequencies present on each signal [4]. The Fourier transform converts a function into the frequencies that compose it. Based on the Fourier inversion theorem, a periodic signal can then be reconstructed from its spectrogram and phase information [11].

Continuous periodic waves may be decomposed into the sum of multiple trigonometric functions. This concept is known as the Fourier series and can be expressed as

$$f(\theta) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty}(a_n \cos(n\theta) + b_n \sin(n\theta)) \tag{2.2}$$

where a_0, a_n, and b_n are the Fourier coefficients, and $f(\theta + 2\pi) = f(\theta)$ [12]. The phase of the trigonometric function n can be recovered by

$$\theta_n = \tan^{-1}\left(-\frac{b_n}{a_n}\right) \tag{2.3}$$

The Fast Fourier Transform (FFT) is an efficient way of computing the Discrete Fourier Transform from a sequence of equally spaced samples. FFT brings substantial improvement from the DFT, reducing the time complexity from $O(n^2)$ to $O(n \log n)$ in the best case, depending on the implementation and factorisation of n.

In order to compare two signals, we can perform the MSE over the frequency domain. Since we are working in a discrete domain, the higher the number of frequencies, the more accurate this measure will be.

$$J(u, v) = \frac{1}{n}\sum_{i=1}^{n}(\hat{u}_i - \hat{v}_i)^2 \tag{2.4}$$

In conclusion, given that harmonics determine the timbre of a sound, spectral analysis seems to be the best way of calculating the similarity between two arbitrary signals. However, in cases where a signal's peak level changes over time

or is not periodic, an analysis method based on the time domain may be more suitable.

2.2. Sound Synthesis

A synthesiser is a musical instrument that generates and manipulates sound electronically, being capable of generating a wide range of sounds, from accurately imitating classical instruments to creating completely new and unique tones.

One advantage of using synthesisers over sampled sounds is that they allow for more expressive control in the amplitude, pitch, rhythm, and timbre of a sound [13]; as well as the absence of noise that might be present in recordings, which leads to cleaner mixes. Digital synthesisers can also be more convenient to work with than actual instruments, as these are often expensive, heavy, or difficult to find, and require a professional sound recording setup. However, some level of technical knowledge is needed to manipulate them effectively, which may result in an entry barrier for inexperienced users [13], [14].

In this section, we will explore how synthesisers came to be, some of their relevant components, and what the most popular synthesis methods are.

2.2.1. Brief History

Synthesisers have revolutionised the way music has been produced since the early 1960s, leading to the creation of genres like Electronic Dance Music (EDM) and the appearance of a perfectly timed rhythm which humans could not manually replicate [15]. The early versions of the synthesiser were completely based on analogue electronics and used voltage-controlled oscillators, filters and amplifiers to generate and shape sound. However, these machines were costly, cumbersome, and difficult to set up and use.

The introduction of the Minimoog in 1970, which was specifically designed for live performance, saw the appearance of affordable, portable synthesisers, making them more accessible to ordinary people and allowing them to produce electronic music virtually anywhere [16]. Subsequent technological advancements led to the development of polyphonic analogue synths, capable of simultaneously playing multiple notes, and later digital synths that emulated analogue ones, such as NordLead. This last category of synthesisers would come to be known as “virtual analogue” [17].

On the other hand, Max Mathews, a pioneer of computer music, developed the first digital sound synthesis program on an IBM 704 in the late 1950s, known as MUSIC I. This technical breakthrough led Mathews and others to write subsequent programs and domain-specific languages such as Csound, which is still in use today[1]. As these digital synthesis tools evolved, they also started

supporting Musical Instrument Digital Interface (MIDI), a protocol that describes the properties of a musical note rather than the sound wave itself. Consequently, Digital Audio Workstations (DAWs) emerged, integrating music sequencing and synthesis into a single interface.

2.2.2. Components

There are multiple components present in a synthesiser, including oscillators, filters, amplifiers, unison, and modulation sources, such as low-frequency oscillators (LFOs) or envelopes. A synthesiser is considered modular if its components are not integrated into a single device, but rather exist across multiple independent devices that are manually interconnected.

In this section, the mathematical foundation and purpose of some of these components will be explored.

2.2.2.1. Oscillators

An oscillator is one of the most important components of a synthesiser, providing the source signal upon which other components can then be applied. Oscillators generate periodic waveforms with different shapes, usually sine, square, sawtooth, or triangle, which can be expressed mathematically.

A sine wave y_1 as a function of time may be defined as

$$y_1 = A\sin(2\pi ft + \varphi) \tag{2.5}$$

where A is the amplitude, f the frequency, φ the phase, and $2\pi f$ corresponds to the angular frequency ω.

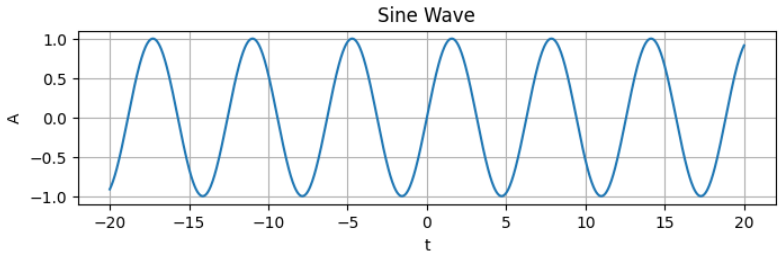


Figure 1: The representation of a sine wave over the time domain.

The sawtooth (y_2) and square (y_3) sound waves are discontinuous functions and can be respectively defined as:

$$y_2(t) = 2\left(t - \left\lfloor t + \frac{1}{2}\right\rfloor\right) \tag{2.6}$$

$$y_3(t) = 2(2\lfloor ft\rfloor - \lfloor 2ft\rfloor) + 1 \tag{2.7}$$

[1]https://csound.com/

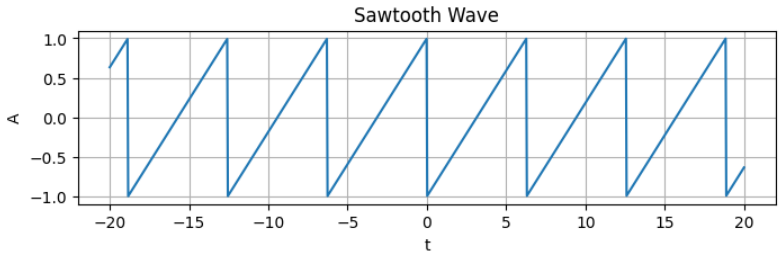


Figure 2: The representation of a sawtooth wave over the time domain.

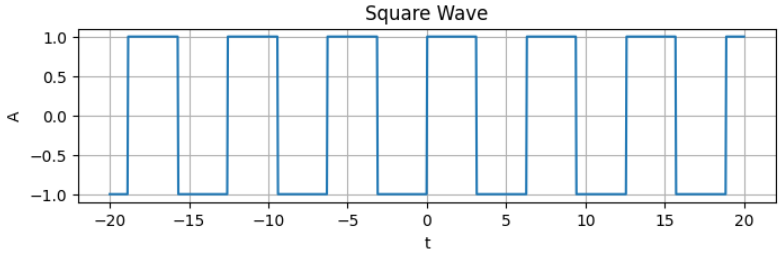


Figure 3: The representation of a square wave over the time domain.

Based on Fourier analysis, any periodic shape can be approximated by summating the harmonics of a fundamental sine wave [12], which is the principle behind additive synthesis.

$$\begin{aligned} {y_2}'(t) &= -\frac{2}{\pi}\sum_{k=1}^{\infty}\frac{(-1)^k}{k}\sin(2\pi kt) \\ &= \frac{2}{\pi}\left(-\sin(2\pi t)+\frac{1}{2}\sin(4\pi t)-\frac{1}{3}\sin(6\pi t)+...\right) \end{aligned} \tag{2.8}$$

$$\begin{aligned} {y_3}'(t) &= \frac{4}{\pi}\sum_{k=1}^{\infty}\frac{\sin(2\pi(2k-1)t)}{2k-1} \\ &= \frac{4}{\pi}\left(\sin(\omega t)+\frac{1}{3}\sin(3\omega t)+\frac{1}{5}\sin(5\omega t)+\frac{1}{7}\sin(7\omega t)+...\right) \end{aligned} \tag{2.9}$$

On the other hand, multiple basic waveforms can be combined into a new one, z, by performing the weighted average of their amplitudes, such that

$$\begin{aligned} z(t) &= \frac{\theta_1 y_1(t+\varphi_1)+\theta_2 y_2(t+\varphi_2)+...+\theta_n y_n(t+\varphi_n)}{\theta_1+\theta_2+...+\theta_n} \\ &= \frac{\sum_{k=1}^{n}\theta_k y_k(t+\varphi_k)}{\sum_{k=1}^{n}\theta_k} \end{aligned} \tag{2.10}$$

where θ is the weight vector and $0 \leq \theta_i \leq 1$, $0 \leq \varphi_i \leq 2\pi$. The resulting wave can then be normalised based on the maximum value in z.

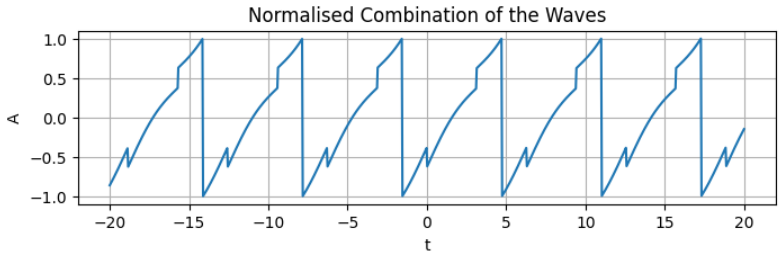


Figure 4: Representation of the resulting normalised sound wave after combining the sine, square and sawtooth waveforms with weights $\theta = [0.1, 0.1, 0.8]^T$ and phases $\varphi = \left[0, 0, \frac{\pi}{2}\right]$

2.2.2.2. Filters

Filters are ubiquitous components in sound production, present in fields from signal processing to mixing. They change the tone of a sound by removing certain frequencies in the signal. When dealing with musical notes, the additional frequencies beyond the fundamental are known as harmonic partials (or simply, harmonics), which are what give instruments their distinctive timbre. For example, the note A4 will always correspond to 440 Hz, however, will sound differently when played on a piano than an oboe.

Depending on the section of frequencies filtered, we can distinguish four main kinds of filters: *low pass*, *high pass*, *band pass*, and *band reject*. Low-pass filters allow frequencies beyond a threshold, known as the cutoff frequency, to pass through. In contrast, a high-pass filter only allows frequencies above the cutoff to be present in the new signal. By combining a low-pass with a high-pass filter a band-pass filter can be produced, which only lets the frequencies between two thresholds pass through. Opposedly, a band-reject filter, also known as a notch, will filter out all the frequencies between the two thresholds.

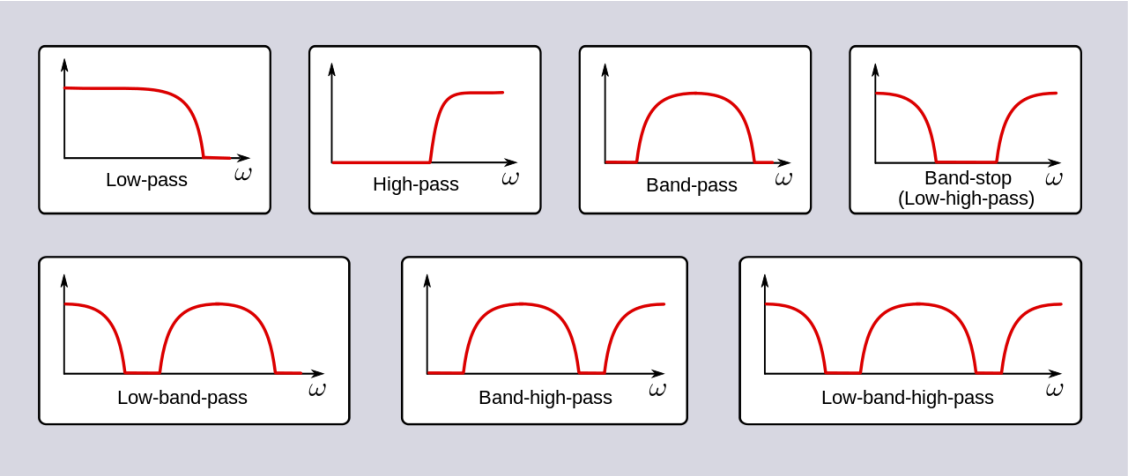


Figure 5: Different types of filters depending on the frequencies passed. *By SpinningSpark real life identity: SHA-1 commitment ba62ca25da3fee2f8f36c101994f571c151abee7 - Self created using Inkscape, CC BY-SA 3.0, https://en.wikipedia.org/w/index.php?curid=20951989*

2.2.2.3. Envelopes

An envelope shapes the characteristics of a sound wave over time, typically in terms of amplitude and timbre. Unlike other components, it is sensitive to user input, such as pressing or releasing a note on a keyboard. The most common kind is the Attack, Decay, Sustain, and Release (ADSR) envelope.

The attack phase defines the time it takes for the signal to reach its peak level. A fast attack creates a sharp and abrupt sound while a slower attack will introduce the sound gradually, producing a fade-in effect. The decay parameter determines how long it takes for the sound to reach the sustain level once the peak is reached. The sustain phase maintains the sound at a certain level for as long as the note is held down. Finally, the release phase determines the time it takes for the sound to fade out after this is no longer active.

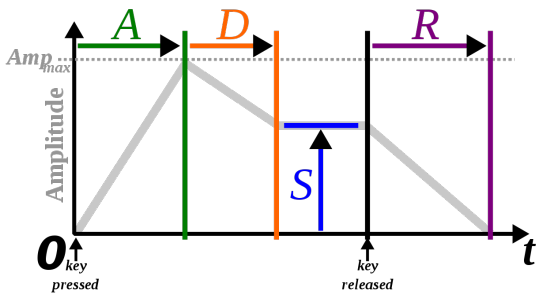


Figure 6: Different phases of the ADSR envelope and what each represents. *Unknown author - Wikimedia Commons, CC BY-SA 3.0, https://commons.wikimedia.org/wiki/File:ADSR_parameter.svg*

2.2.3. Methodology

Different sound synthesis methods exist, including subtractive, additive, FM, wavetable and granular.

2.2.3.1. Subtractive Synthesis

Subtractive synthesis is a sound synthesis technique in which certain frequencies of an audio signal are filtered out. The process starts with an oscillator, usually a combination of sine, square, and sawtooth waves, and then subtractive components like filters are applied to the produced signal to change the harmonics.

2.2.3.2. Additive Synthesis

Additive synthesis is similar to subtractive synthesis but utilises a very different method to construct harmonics. Rather than using a combination of shaped oscillators, it sums the amplitudes of the harmonics of a fundamental sine wave. Similarly, components such as filters can still be applied to remove unwanted frequencies.

2.2.3.3. FM Synthesis

Frequency modulation (FM) synthesis is a technique which involves modulating a carrier frequency with another in order to create complex evolving sounds. The energy of the carrier is then transformed into *sidebands*, which are series of harmonics at both sides of the carrier frequency [18]. Some of the parameters that can be manipulated include the frequency ratio between the carrier and modulator, the modulation index, and the shape of the waveform used.

2.2.3.4. Wavetable Synthesis

Wavetable synthesis uses a set of pre-recorded or generated waveforms, known as *wavetables*, to enable the production of a wide variety of tones and textures. Techniques like interpolation may be used to prevent abrupt transitions between multiple of them [19]. A sound can be shaped over time by modifying parameters such as the wavetable used, position, and speed.

2.2.3.5. Granular Synthesis

Granular synthesis works by decomposing sound into small units called *grains* which are then manipulated to produce other sounds. A granulator transforms an input sound into multiple grains, each usually a few milliseconds long, and processes each grain independently. Grains are organised into *events* and characterised by twelve parameters related to time, waveform, frequency, bandwidth, grain density, and amplitude [20].

2.3. Genetic Algorithms

Evolutionary algorithms (EAs) are population-based optimisation algorithms which use mechanisms inspired by biological evolution, such as reproduction, mutation, and natural selection. They represent a class of adaptive optimisation techniques that are domain-agnostic and can therefore adjust to dynamic environments. Thus, they do not require any training or prior knowledge about the problem in question and are not based on a defined learning model [21].

A genetic algorithm (GA) is a subclass of EAs where the information of a candidate solution is encoded in genes. GAs follow an iterative process where a population evolves over multiple generations. A member of a population is known as an *individual* and contains information encoded in *chromosomes*, which all together form the *genotype*.

Each individual has a *fitness* property which is evaluated through the use of an objective function measuring how good the candidate solution is. The fittest individuals will have more chances of surviving and passing their genetic information to the next generation, analogous to natural selection. A selection strategy needs to be defined in order to decide which individuals will produce offspring. Multiple techniques exist, including proportionate roulette wheel, linear

ranking, exponential ranking, and tournament selection, where the latter performs best in general [22].

We then need a *crossover* or *recombination* operator to combine the genetic information of two parents, producing a new individual. The technique used will differ depending on whether the genotype is represented in binary or real numbers. Some of the methods suited for binary arrays include one-point, two-point, k-point, and uniform crossover [23]. If the chromosomes have integer or real values, intermediate recombination may be used instead, which consists of performing a weighted average between the values of the parents. For each gene i, the intermediate recombination of two individuals u and v is

$$C(u_i, v_i) = \beta u_i + (1 - \beta)v_i \tag{2.11}$$

where

$$\beta_i \in [0, 1] \tag{2.12}$$

[24]. GAs are generally very good at finding global optima, even in non-linear problem spaces, and have been proven useful in a wide range of disciplines, such as designing gas pipeline networks, timetabling, and engineering applications [25]. Nonetheless, their success depends on factors like the chromosome encoding, population size, or diversity introduced in each generation.

2.4. Evolutionary Algorithms in Sound Exploration

Producing sound by manually calibrating parameters on a synthesiser can often be a confusing and tedious task [14]. It can be particularly difficult to find the optimal configuration giving a specific sound wave, which is why an adaptive global optimisation technique like GAs could be well suited for this purpose.

A tool based on evolutionary algorithms that can closely approach a target sound may be helpful during the process of sound production, where the parameters yielded could be used as a guide or default configuration in user-interfaced synthesisers.

2.4.1. Previous Work

Both interactive and non-interactive evolutionary computation in the field of sound synthesis have been explored before and the results are usually positive. Interactive simulations have the advantage of better judgement of similarity while non-interactive simulations are based on objective and faster fitness evaluations [26].

GAs have been shown to perform better than random search on parametric optimisation with FM and subtractive synthesis [13]. One publication considered that hill climbing was not a suitable technique in the context of sound synthesis optimisation, unlike EAs [26].

Even though most of the literature agrees that GAs provide satisfactory results, choosing the right configuration and fitness function is essential for an accurate convergence. For instance, an experiment performed on FM synthesis concluded that "the synthesized sound [was] not that similar to the target when judged by human ears" when using the spectral centroid of the sound wave as the fitness function. However, when combined with the spectral norm and repeated the experiment, the synthesised and target waves were almost identical [4].

A few open-source projects exist in this field, such as `evosynth`[2], but no libraries providing an interface to a general-purpose, synthesis parameter optimisation system have been found.

[2]https://github.com/yeeking/evosynth

3. Proposed Solution

The proposed solution consists of developing a Rust library that facilitates the research and application of genetic algorithms for optimising sound synthesis. In simple terms, instead of producing a sound wave given a set of parameters, this tool produces a set of parameters given a sound wave. A list of requirements, benefits, and constraints have been gathered throughout this project and are presented in this chapter.

3.1. Requirements

The system should allow multiple candidate solutions containing certain encoded parameters to be represented as individuals in a GA simulation. Each individual must be able to be converted into a signal and assigned a fitness value. The genetic operators described above, such as crossover and mutation, must also be supported. The simulation must return the fittest individual after the specified number of generations, which will contain the optimal parameters.

When performing research on GAs, we need a way of objectively determining how successful they are. This can be achieved by comparing them to other optimisation algorithms such as hill climbing. The proposed system should support running simulations on at least one other optimisation technique and provide useful metrics, such as the number of fitness evaluations done.

The performance of a GA depends on multiple factors, including the fitness evaluation method used, the population size and its evolution, the diversity introduced in each generation, the number of generations running for, and the mutation rate in crossover operations. The proposed solution should therefore be able to run simulations with these parameters.

Subtractive and additive synthesis are some of the most commonly found methods in synthesisers and are relatively simple to implement. However, one may be better suited than the other at generating certain sound waves. Consequently, the system should support at least these two synthesis methods and provide a way to compare how they perform with different targets.

Finally, the library should allow researchers to conduct simulations through a usable API. It should also adhere to a modular design and remain open to future expansion.

The list of requirements originally envisioned and whether they have been met is included in the appendix Section C.

3.2. Benefits

This solution aims to assist users, in particular novices, in achieving desired sound characteristics more efficiently. Specifically, when importing a target sound similar to the desired one, the system will provide its corresponding parameters. While some fine-tuning may still be necessary to achieve the ideal sound wave, using this approach as a starting point can already save significant time and effort, enhancing the overall sound design process.

The proposed library offers a certain degree of customisation which increases the number of use cases and sound types that can be explored. This flexibility would also be beneficial for any research carried out, as many cases can be studied.

Furthermore, the existence of an API makes it possible for third-party applications to integrate with the system. The modular design of its implementation will facilitate any future expansion or replacement of certain components without affecting the overall codebase. The absence of a runtime should enable users to run entire simulations in a matter of seconds.

3.3. Constraints

Synthesisers are generally expensive and complex machines that take years to design and build, hence why only a small portion of their functionality can be replicated in this solution. This limitation may affect the accuracy and richness of the produced sounds.

Apart from the above design challenges, iterative optimisation algorithms like GAs are usually computationally expensive, especially when they require evaluating each individual on the frequency decomposition of the samples they represent. However, this cost can be drastically reduced by writing and running the simulation in a compiled environment and parallelising certain operations.

Tuning the parameters of a GA, such as population size, mutation rate or selection criteria, can be a challenging task, as they affect its outcome and effectiveness. GAs can also suffer from premature convergence, yielding a suboptimal solution, and, since they are non-deterministic optimisation methods, it may be difficult to distinguish information from noise. The library should support running multiple concurrent simulations to increase accuracy and statistical significance.

4. Design

4.1. Genetic Representation

Genotypes are typically encoded as binary strings where the alleles are either 0 or 1. However, in this case, we will use floating-point values to represent genes instead. Many parameters in a synthesiser consist of decimal values that must be treated as such, since performing crossover in their raw binary representation would not be safe due to the memory layout of floats (sign, exponent, and fraction). Binary strings are not the most efficient structures to work with either. Even if an individual was encoded as an array of `bool`, each element would have a size of 8 bits rather than 1. Crates like `bincode` may have addressed this issue but would have still complicated the recombination operation. Therefore, parameters will be stored as float values in a struct representing a component.

Individuals will therefore be made of independent optional components, each with a specific set of parameters and effect on the signal, based on the synthesis method they represent. Users will be able to include or omit components as they wish, with some exceptions, and following a preset order of application. For instance, it would not make sense to apply a filter before creating an oscillator. Some components should also be reusable within multiple synthesis methods to avoid duplicate logic.

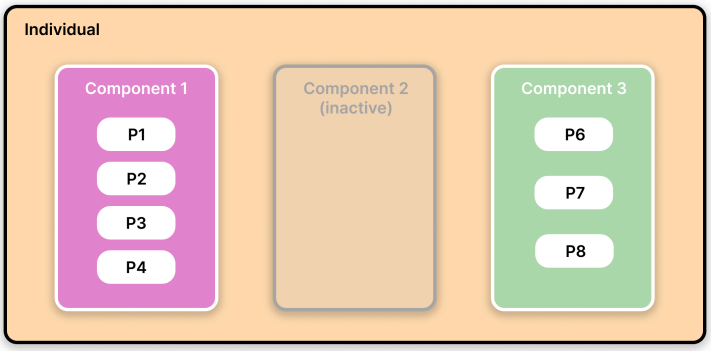


Figure 7: A representation of an individual's layout, where parameters (P1, P2, etc.) are floating-point values.

4.2. Selection, Mating and Breeding

The selection procedure will consist of selecting the $\frac{n}{2}$ fittest individuals from a population of n. This new sample will then be shuffled and pairs of individuals formed, producing as many as the population size allows. This process is then repeated once to maintain the population growth steady. The selected individuals and their offspring will constitute the next generation.

Additionally, each generation may include randomly generated individuals to preserve diversity and potentially increase the population size. Users should have the flexibility to specify whether n should remain fixed or vary over time, determining whether the population size remains constant or increases as the algorithm progresses.

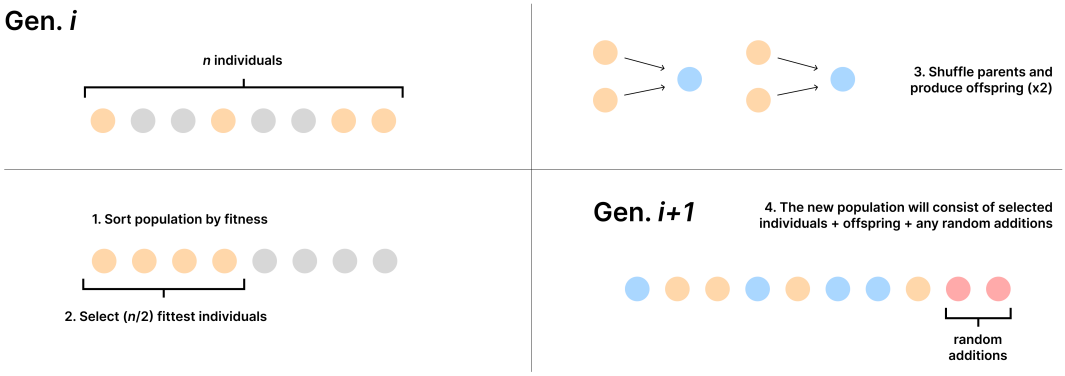


Figure 8: Multiple stages of a GA iteration.

4.3. Crossover Operation

In order to combine two individuals and produce offspring, intermediate recombination will be used, as previously described. This operation takes two individuals of the same type and structure and will be handled by their components.

For each parameter, a function performing a random weighted average will be called accepting the other individual's corresponding parameter, a mutation rate, and a random value within the parameter range in case mutation happens. For our purpose, the β parameter will be a random value between 0 and 1.

The mutation rate will be defined as the probability of the weighted average being replaced by the provided value. This factor is necessary to increase diversity within a population, affecting the quality of the produced individual.

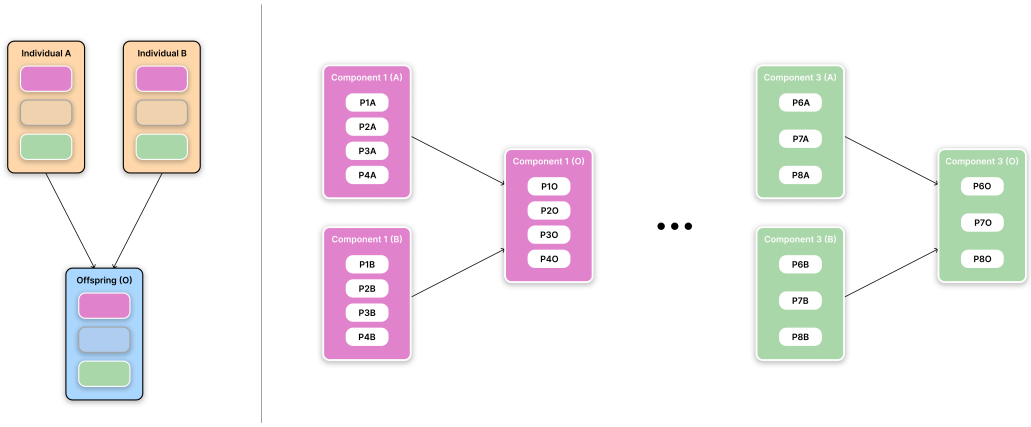


Figure 9: When combining two individuals, each component operates independently.

4.4. Fitness Evaluation

An individual's fitness will be evaluated on the signal it produces. To determine how *fit* a candidate is, it is necessary to calculate the proximity of its corresponding signal to a target sound. Signal comparison methods such as the MSE over the frequency domain or Euclidean distance on the time domain will be used, returning a fitness value between 0 and 1. This can be achieved by using a sigmoid function σ and a constant k to scale the function to convenient bounds.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.13}$$

$$f(x) = 2\sigma\left(-e^{\log(\frac{x}{k})}\right) \tag{4.14}$$

Fitness values on their own are somewhat arbitrary and thus are only meaningful when compared to other values calculated with the same metric.

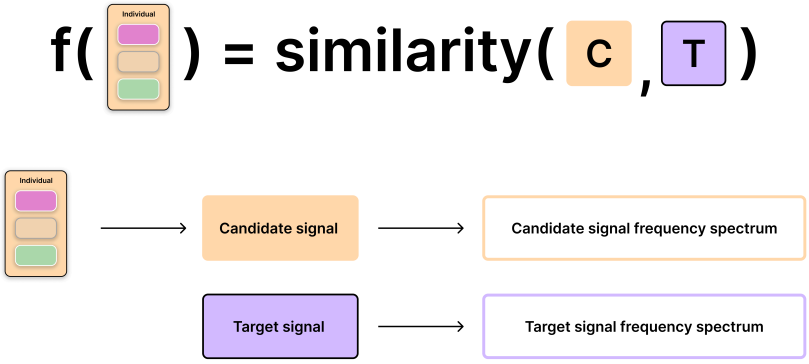


Figure 10: A simplified overview of the fitness evaluation definition f. The similarity of the two signals can be defined in terms of the time or frequency domain, for which the frequency spectrum would be necessary.

4.5. Signal Processing and Representation

Signals should be represented as an array of floating-point numbers representing the amplitude at each sample taken. If the sampling rate is 44,100 Hz, that means 44,100 of these samples will represent one second of the signal. Signals should support operations such as combining multiple oscillators or applying filters, which can be reduced to addition, subtraction, and scaling operations.

A Fourier transformation should be defined to enable the conversion of an array of samples representing a signal into (f, A) pairs, providing the frequency information. Since these will be used for fitness evaluation, there should be enough frequencies to accurately represent their frequency components.

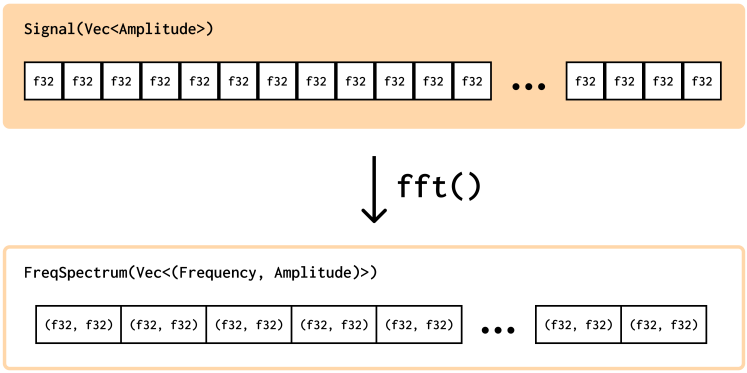


Figure 11: Memory representation of a signal and its corresponding frequency spectrum.

5. Implementation

The implementation phase has been the most time-consuming part of this project but key to enabling experimentation and analysis. A robust and efficient system was built from scratch following the specifications previously described. This process required a profound understanding of both GAs and sound synthesis as well as previous experience with the Rust programming language.

5.1. Prototyping

Prototyping is an important step in software development as it allows developers to visualise, test, and validate an early iteration of a system before incorporating more features and functionality.

One of the fundamental features of this project was to be able to run a simulation successfully and export the fittest sound wave to an audio file so that it could be listened to. After setting up a new Rust project, a simple struct was created, storing the current generation, maximum number of generations, population, and target sound wave.

```
pub struct GASimulation<T: Individual> {
  generation: u32,
  max_generations: u32,
  population: Vec<T>,
  target: SynthSignal
}
```

The `population` field is an array of a type implementing the `Individual` trait. Traits in Rust are similar to interfaces in Java, where the shared behaviour of a type is defined.

```
pub trait Individual {

  fn fitness(&self, target: &SynthSignal) -> u8;

  fn crossover(&self, other: &Self, r: f32) -> Self;

  fn to_signal(&self) -> SynthSignal;
}
```

Given that subtractive synthesis is one of the simplest and most popular methods, it was the one chosen for the prototype. To represent an individual using this

method, another struct was defined containing an oscillator component that implemented the `Individual` trait.

```
pub struct SubtractiveIndividual {
  oscillator: Option<OscillatorComponent>,
}

impl Individual for SubtractiveIndividual { ... }
```

To facilitate customisation, components in an individual were made optional, hence the `Option<T>` type. Each of the component structs contains the parameters being optimised. For instance, this is how the oscillator component struct was defined.

```
pub struct OscillatorComponent {
  pub freq: f32,
  pub sine_amp: f32,
  pub sine_phase: f32,
  pub square_amp: f32,
  pub square_phase: f32,
  pub saw_amp: f32,
  pub saw_phase: f32,
}
```

The `crossover` function takes two individuals of the same kind and calls the `combine` function, defined for each component to produce a new individual. Recombination is then performed by taking a random weighted average of each parameter from both parents, where the mutation rate specifies the probability of assigning a completely random value to the new instance within its domain. As previously explained, the reason why this technique was used instead of splitting the chromosomes at k crossover points is due to parameter values being real numbers [27].

The subsequent step involved defining a method to assess the proximity of each individual to the target, thereby determining its fitness. To accomplish this, the system needed to support the conversion of any individual into its corresponding signal. This was achieved by implementing the to_signal function within the Individual trait. The method generates a default `Signal` and then applies the specified components serially. Auxiliary methods for generating waveforms such as `sine_wave` were also written following their mathematical definition.

As previously seen, there are multiple approaches to signal comparison, among which frequency spectrum analysis seemed the best suited in general and thus was implemented first. A function was defined performing the mean squared error (MSE) over the array carrying the frequency spectrum information. This method is then called from the `fitness` function when working with frequency-domain evaluation.

```
fn fitness(&self) -> f32 {
  let mse = self
            .to_signal()
            .freq_spectrum_mse(&self.get_target())
            .expect("MSE over the frequency spectrum should be valid");
  let cost = (mse / 1000.0).log10().exp();

  // the higher the cost, the lower the fitness
  2.0 * sigmoid(-cost)
}
```

Finally, the `run` method was defined to execute the simulation, which calls the `next` method for each generation. Initially, the population was being stored as a `BinaryHeap<T>`, which was later changed into a `Vec<T>` (contiguous growable array) as it turned out to be more efficient. Once the simulation concludes, the fittest individual is returned, which can be then converted into an array of samples and exported to a WAV file. The full implementation of these methods can be found in the appendix Section A.

5.2. Expansion

5.2.1. Additive Synthesis

Once the structs and functionality related to subtractive synthesis had been implemented and a few successful simulations performed, the focus was set on additive synthesis. Having a functioning system with two synthesis methods would enable their comparison from an early stage of the implementation.

Given that additive synthesis involves combining certain harmonics into a single wave, a component was defined to represent a fundamental frequency alongside an array of amplitudes for each harmonic. The current implementation supports up to nine of them.

```
pub struct AdditiveIndividual {
    target: Arc<Signal>,
    fitness: Option<f32>,
    harmonics: Option<HarmonicsComponent>,
    ...
}

pub struct HarmonicsComponent {
  pub freq: f32,
  pub amplitudes: Vec<f32>
}
```

5.2.2. Introducing Filters

Filters are ubiquitous components in synthesisers, enabling the removal of certain frequencies. The representation of this component was defined as the following.

```
pub(crate) enum FilterComponent {
  LowPass {
    cutoff_freq: f32,
    band: f32,
  },
  HighPass {
    cutoff_freq: f32,
    band: f32,
  },
  BandPass {
    low_freq: f32,
    high_freq: f32,
    band: f32
  },
  BandReject {
    low_freq: f32,
    high_freq: f32,
    band: f32
  },
}
```

Based on the `synthrs` library, a function was written for each type of filter, which may be then applied to the signal. The `low_pass_filter` function takes a cutoff frequency and band as parameters, then passes a sinc^3 wave through a Blackman window, and finally returns a filter vector. The other filter types can be constructed from this definition.

5.2.3. Separating Components from Methods

Initially, components were written as part of the `synthesis_methods` module, however, this went against the principles of decoupling and dependency inversion. The filter component, for instance, belonged to the subtractive synthesis module but was at the same time being called from the additive synthesis module. To make components independent and reusable, the entire codebase was restructured, including them in a separate module.

```
├── signal_processing
│   ├── components
│   └── signal_analysis
├── simulation
│   ├── algorithms
│   ├── components
│   └── synthesis_methods
```

[3] https://en.wikipedia.org/wiki/Sinc_function

5.2.4. Hill Climbing

In order to have something to compare the performance of the GA with, hill climbing was introduced. Its implementation reuses some of the components originally written for the GA, such as the types holding the genetic representation or the functions used for fitness evaluation, but with a single individual and a different evolution mechanism.

The algorithm continuously calls an `evolve` method that stochastically looks for a better neighbour within a specified range and moves to it if the fitness is higher than the current. It terminates when the step size is too small or the number of unsuccessful attempts to find a better neighbour is too high.

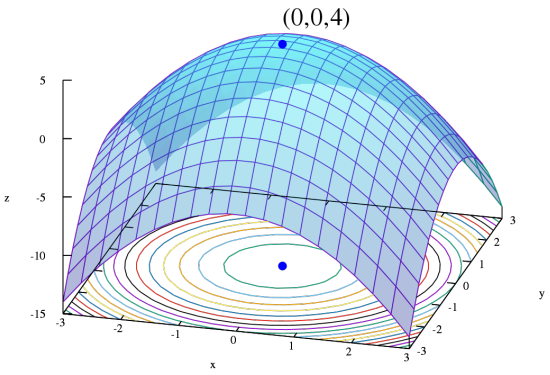


Figure 12: Hill climbing algorithms perform well on problems with a single local maximum. *By IkamusumeFan - Wikimedia Commons, CC BY-SA 4.0, https://commons.wikimedia.org/wiki/File:Max_paraboloid.svg*

5.3. Optimisations

One of the most notable performance optimisations was to parallelise the production of offspring using `par_iter` from the library `rayon`[4]. The previous process was particularly expensive as the fitness value needed to be calculated for each individual, blocking the execution thread for a significant amount of time. Representing the population as `Vec<T>` instead of a `BinaryHeap<T>` also brought significant performance improvements.

Rust offers multiple build configuration choices depending on the developer's needs. By default, a *dev build* is produced, where several runtime performance optimisations are disabled to make the debugging experience more straightforward. However, at the expense of longer compile times, the performance of a program can be drastically improved by running it in release mode (`cargo`

[4]https://docs.rs/rayon/latest/rayon/index.html

`run --release`). If compile times were too high, settings like *codegen units* could help decrease them through parallelisation, although this comes with some code optimisation trade-offs[5].

5.4. Challenges

The strict type system and borrow checker that Rust offers constrained the implementation of certain structures and the relationships between them. This forced the appearance of complex generics and associated types in traits like `Individual` and `IndividualGenerator`. Moreover, a considerable amount of time was spent fixing these compiler errors, in exchange for type correctness and memory safety guarantees.

The complexity of some components and signal analysis techniques such as FFT presented difficulties in the implementation phase which were eventually overcome. A deep theoretical understanding was necessary to implement them effectively, delaying their development.

[5]https://nnethercote.github.io/perf-book/build-configuration.html

6. Testing and Results

6.1. Testing Strategy

We can differentiate between two main testing approaches: unit tests, which work with small and isolated pieces of code, and integrated tests, which evaluate how multiple units work with each other in a single block.

Because GAs are inherently probabilistic, the testing approach during the development phase primarily involved conducting full simulations followed by manual evaluation of their performance. This approach was subsequently formalised during the research phase by conducting simulations across a broad range of scenarios.

6.2. Unit Testing

To ensure the correct behaviour of important library components, multiple unit tests have been written in functions marked with the `#[test]` attribute inside a `tests` module[6]. There are currently 12 unit tests covering oscillators, harmonics, signal analysis, CSV exporting, and population evolution.

```
running 12 tests
test signal_processing::components::harmonics::tests::test_generate_harmonics ... ok
test signal_processing::components::oscillator::tests::test_saw ... ok
test signal_processing::signal_analysis::tests::test_euclidean_distance ... ok
test signal_processing::components::oscillator::tests::test_square ... ok
test signal_processing::components::oscillator::tests::test_sine ... ok
test analytics::tests::test_csv_export ... ok
test signal_processing::signal_analysis::tests::test_normalise ... ok
test signal_processing::signal_analysis::tests::test_freq_spectrum ... ok
test signal_processing::signal_analysis::tests::test_extend_pow_two ... ok
test simulation::algorithms::genetic::tests::test_increasing_population_odd ... ok
test simulation::algorithms::genetic::tests::test_constant_population ... ok
test simulation::algorithms::genetic::tests::test_increasing_population_even ... ok

test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 3.58s
```

Figure 13: List of tests currently defined and their results.

[6]https://doc.rust-lang.org/rust-by-example/testing/unit_testing.html

The following is an example of a simple unit test for the sine wave generator.

```
#[test]
fn test_sine() {
  let mut signal = sine_wave(1.0, 1.0, 4.0, 1.0, 0.0).into_iter();
  assert_eq!(signal.next(), Some(0.0));
  assert_eq!(signal.next(), Some(1.0));
  signal.next();
  assert_eq!(signal.next(), Some(-1.0));
}
```

6.3. Simulations

Several simulations have been run in order to assess the effectiveness of GAs, comparing their performance against alternative algorithms and settings. Before running a simulation, it is necessary to construct an individual generator which will be then passed to a simulation builder.

```
fn main() {
    let generator = SubtractiveIndividual::new_generator()
        .target_file("audio_samples/sample.wav")
        .fitness_type(FitnessType::FreqDomainMSE)
        .oscillator();

      let mut simulation: GASimulation<SubtractiveIndividual> =
GASimulationBuilder::new()
        .generator(generator)
        .population_evolution(PopulationEvolution::Constant)
        .initial_population(100)
        .n_random_additions(4)
        .mutation_rate(0.05)
        .max_generations(500)
        .signal_export("out.wav")
        .csv_export("out.csv")
        .build();

    simulation.run().expect("Simulation should have completed.");
}
```

Since these are stochastic and non-deterministic algorithms, in some cases, multiple instances have been executed at a time to be able to work with statistical averages, even though it is an expensive process.

6.3.1. Replicating a Generated Wave

As a first experiment, a target signal was constructed from specific `OscillatorComponent` parameters to assess how accurately the GA would be able to replicate them. After simulating a population of 1,000 individuals over 500 generations based on frequency-domain evaluation, the fittest individual reached a fitness value of 0.972. However, it can be observed from the following table how some of the estimated parameters were very different from the target ones.

Parameter	Target	Estimated
freq	520.0	519.9991
sine_amp	0.3	0.3308869
sine_phase	0.2	3.1442568
square_amp	0.3	0.0020760477
square_phase	0.1	4.739438
saw_amp	0.4	0.0031836072
saw_phase	0.0	3.3810263

Table 1: Parameters of the constructed target wave vs the estimated ones after running the simulation.

The interesting reason behind this anomaly is that the GA had inverted the wave's amplitude as the frequency composition between two signals that cancel each other is essentially the same, hence also sounding alike. This can be observed in Figure 14.

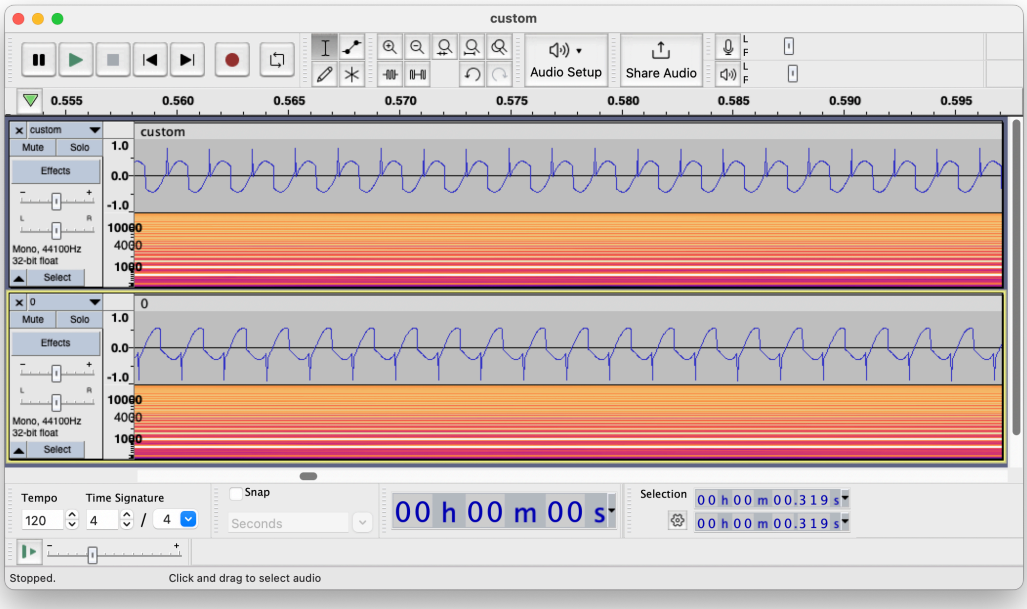


Figure 14: Waveform and spectrogram of the target vs generated waves.

6.3.2. Synthesis Methods

To compare the performance of subtractive vs additive synthesis, two simulations were set up with a sine wave as the target. The fitness evaluation method was set to MSE on the frequency domain and the population size, to 100. One of the simulations ran for 200 generations and the other for 500.

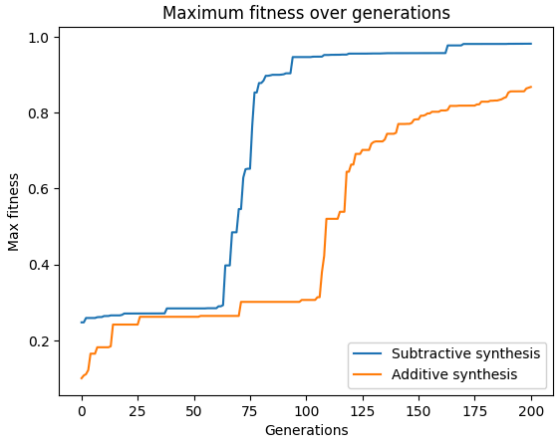


Figure 15: Highest fitness achieved with subtractive vs additive synthesis over the number of generations. Configuration: `population=100, n_generations=200, mutation_rate=0.05, n_random_additions=4, evolution='constant'`.

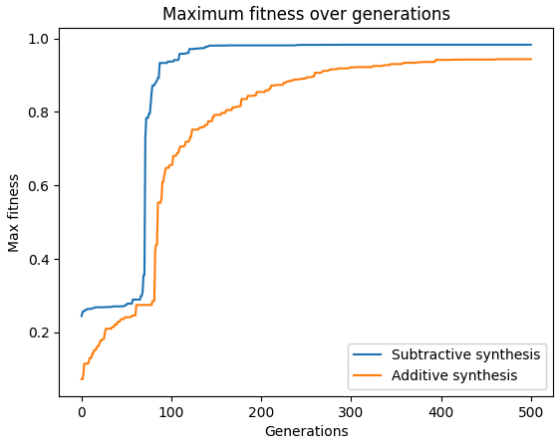


Figure 16: Highest fitness achieved with subtractive vs additive synthesis over the number of generations. Configuration: `population=100, n_generations=500, mutation_rate=0.05, n_random_additions=4, evolution='constant'`.

As it can be observed, in both cases, subtractive synthesis slightly outperformed additive synthesis and had a faster convergence, although both methods still achieved a high fitness score.

One of the most fascinating findings about the additive simulation was how instead of setting the fundamental frequency to 440 Hz and the rest of the harmonics to zero, the GA was choosing unusually high-frequency values and favouring one of the harmonics over the others. Despite the abnormal parameter values reached, the generated file still looked and sounded like a perfect 440 Hz sine wave.

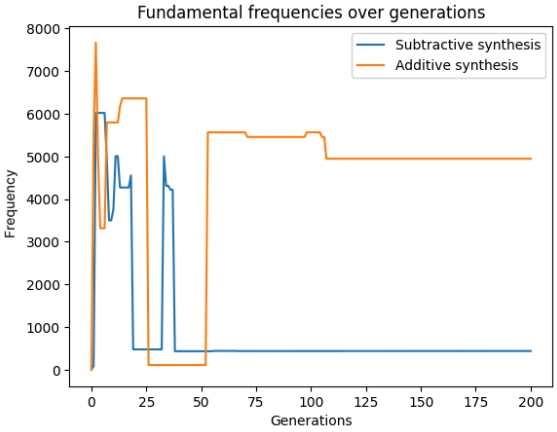


Figure 17: Fundamental frequency of the fittest individuals over generations using subtractive vs additive synthesis. Configuration: `population=100, n_generations=200, mutation_rate=0.05, n_random_additions=4, evolution='constant'`.

At first, it was thought this issue was caused by the implementation of the `harmonics` component. However, after closer inspection, the problem was found to be related to sampling. In one of the experiments, the fundamental frequency of the fittest individual resulted in 4,851.11 Hz and the ninth harmonic, at 43,660 Hz, was remarkably strong. Interestingly, the subtraction of the sampling rate, 44,100 Hz, from that harmonic was exactly −440 Hz, the negative target frequency.

This happened because harmonic frequencies were allowed to go over the expected Nyquist frequency, in this case, 22,050 Hz, leading to aliasing. A good analogy to this phenomenon is when a car is being recorded on a highway and its wheels seem to be slowly moving backwards, known as the *wagon-wheel effect*[7]. In signal processing, this may occur when there are less than two samples per cycle, which is the minimum needed to correctly represent a signal.

To fix this, all individuals containing any harmonics above the Nyquist frequency were invalidated, by setting their fitness value to zero.

```
fn harmonics_are_valid(&self) -> bool {
  match self.harmonics.as_ref() {
    Some(harmonics) => {
      let fund = harmonics.freq;
      let niquist_freq = SAMPLE_RATE as f32 / 2f32;
      // Ensure all frequencies are below the Nyquist frequency.
      (1..=harmonics.amplitudes.len())
        .all(|i| (fund * i as f32) < niquist_freq)
    },
    _ => true // This doesn't apply if there's no harmonics component.
  }
}

fn fitness(&self) -> f32 {
  self.fitness.unwrap_or_else(|| {
    if self.harmonics_are_valid() {
```

[7] https://en.wikipedia.org/wiki/Wagon-wheel_effect

```
      self.calculate_fitness()
    } else {
      0.0 // Invalidate the individual if the harmonics are too high.
    }
  })
}
```

After this addition, the fundamental frequency consistently resulted in a value close to 440 Hz, albeit the fitness value was not significantly improved (even lower than before in some cases).

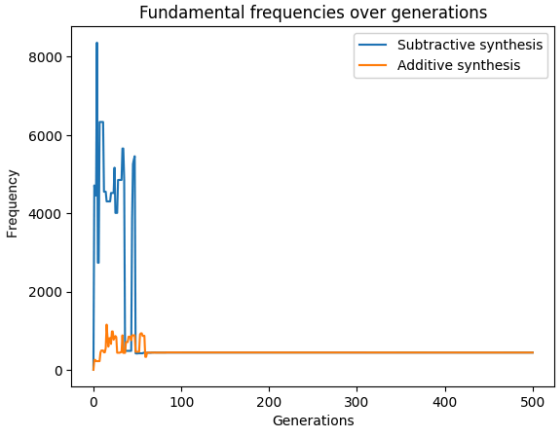


Figure 18: Fundamental frequency of the fittest individuals over generations using subtractive vs additive synthesis (after frequency restrictions). Configuration: `population=100, n_generations=500, mutation_rate=0.05, n_random_additions=4, evolution='constant'`.

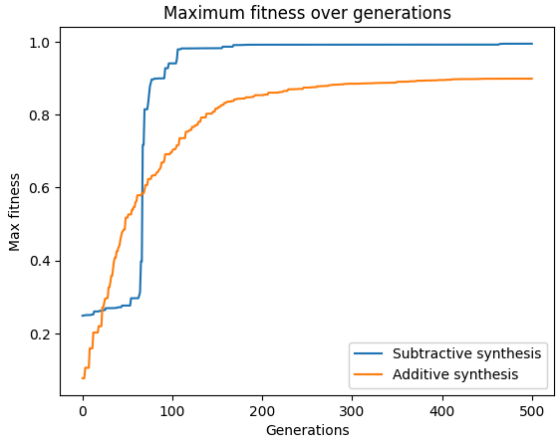


Figure 19: Highest fitness achieved with subtractive vs additive synthesis over the number of generations (after frequency restrictions). Configuration: `population=100, n_generations=500, mutation_rate=0.05, n_random_additions=4, evolution='constant'`.

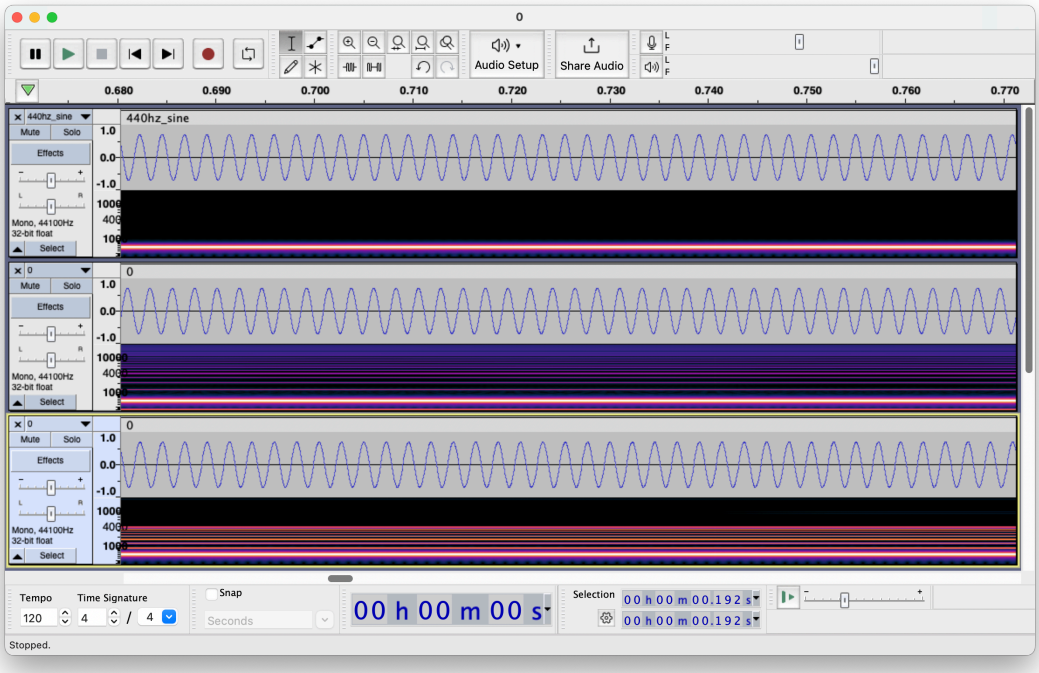


Figure 20: Waveforms and spectrogram of the sine target followed by the generated ones by subtractive and additive synthesis.

fitness	**0.9962411**
freq	439.99786
sine_amp	0.70819336
sine_phase	3.1442568
square_amp	0.0020760477
square_phase	4.739438
saw_amp	0.0031836072
saw_phase	3.3810263

fitness	**0.90962845**
freq	439.90002
amp_1	0.703382
amp_2	0.004360318
amp_3	0.005996208
amp_4	0.003657665
amp_5	0.0038192347
amp_6	0.0033883916
amp_7	0.004195324
amp_8	7.981062e-5
amp_9	0.006465002

Table 2: Oscillator (left) and harmonic (right) components of the fittest individuals by subtractive and additive synthesis respectively. Both reached a very similar frequency and amplitude on the parameters `sine_amp` and `amp_1`.

In order to have a better assessment, the testing methods were redesigned to run multiple instances of a simulation in parallel, studying the average fitness and fundamental frequency on each generation. To improve statistical significance, outliers over $\sigma = 2$ have been excluded from the estimation of the mean.

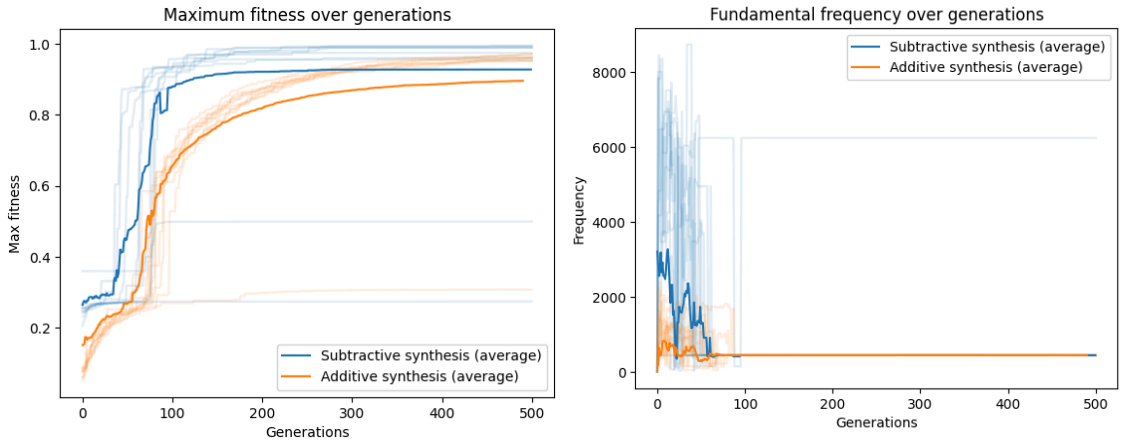


Figure 21: Evolution of the fitness and fundamental frequency for each generation run on 10 parallel GA subtractive and additive simulations. Configuration: `population=100, n_generations=500, mutation_rate=0.05, n_random_additions=4, evolution='constant'`

Subsequently, the experiments were repeated with two additional target waves; a sawtooth and a harmonically-rich wave to which we will refer as *shimmer*. Subtractive synthesis performed better than additive synthesis in terms of fitness with the sawtooth wave, reaching a score of 0.95 vs 0.67 within 1,000 generations. Both methods eventually reached the expected fundamental frequency.

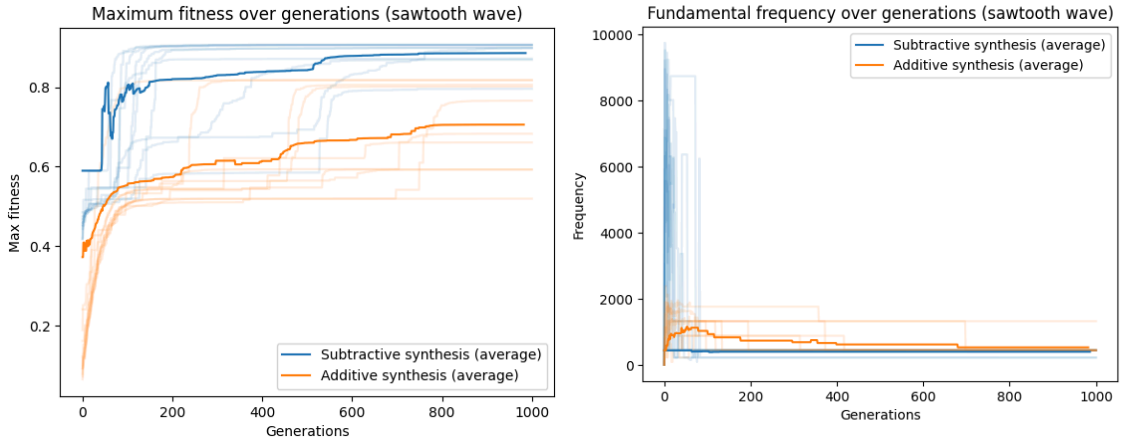


Figure 22: Evolution of the fitness and fundamental frequency for each generation run on 10 parallel GA subtractive and additive simulations with a sawtooth target wave. Configuration: `population=150, n_generations=1000, mutation_rate=0.1, n_random_additions=4, evolution='constant'`

Subtractive synthesis was able to successfully replicate the target wave while additive synthesis, for some reason, produced a sawtooth wave with a negative ramp.

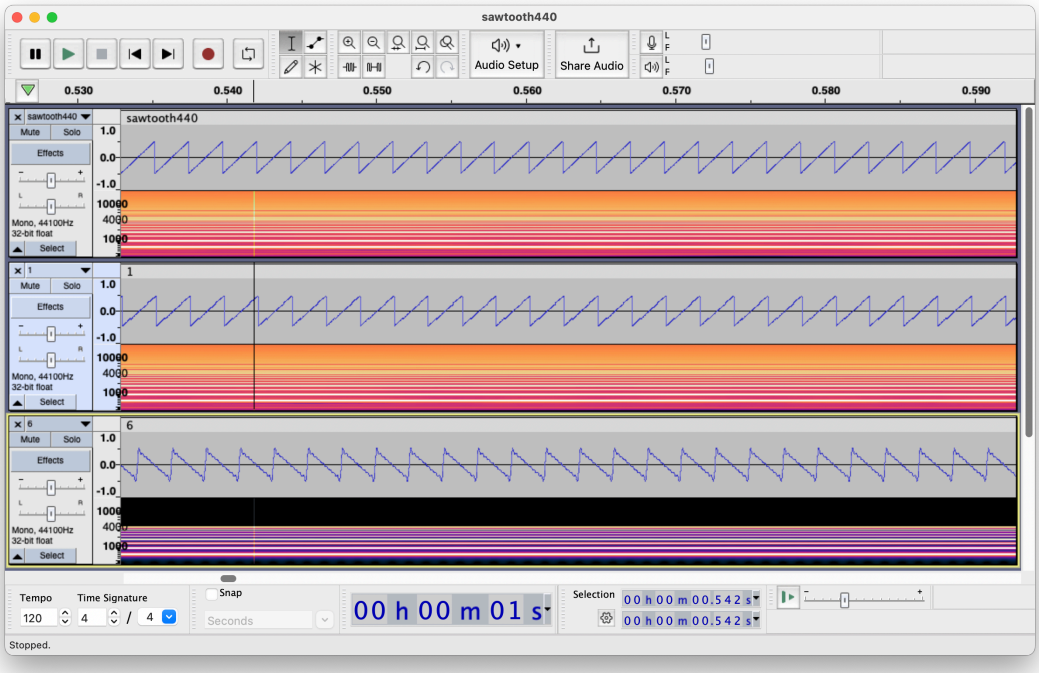


Figure 23: Waveforms and spectrogram of the saw target followed by the generated ones by subtractive and additive synthesis.

Subtractive and additive synthesis produced very similar fitness values when repeating the previous experiment with the shimmer target wave, although not as good compared to the previous targets. This is likely due to the harmonic complexity of the signal; the oscillator component is not able to model signals beyond a composition of sine, square, and saw waves, and the harmonic component is somewhat constrained by the number and ratio of harmonics modelled.

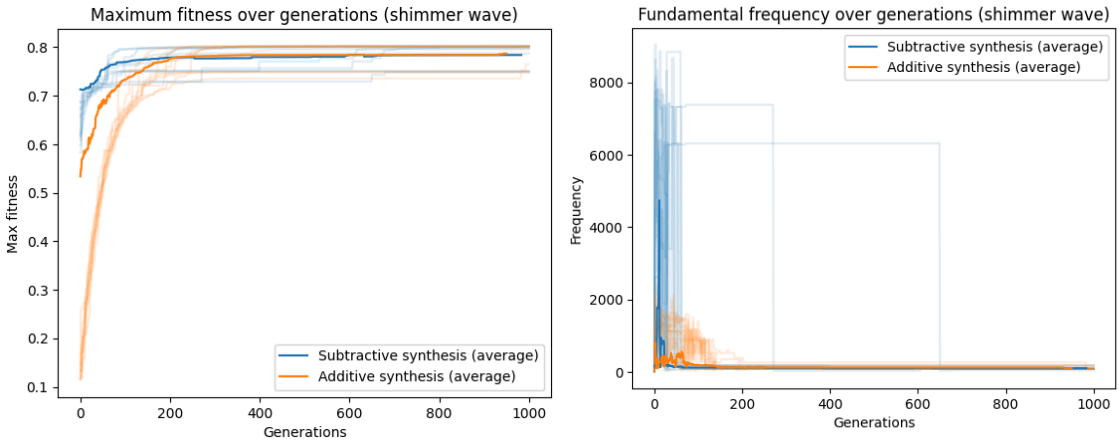


Figure 24: Evolution of the fitness and fundamental frequency for each generation run on 10 parallel GA subtractive and additive simulations with a complex harmonic wave. Configuration: `population=150, n_generations=1000, mutation_rate=0.05, n_random_additions=4, evolution='constant'`

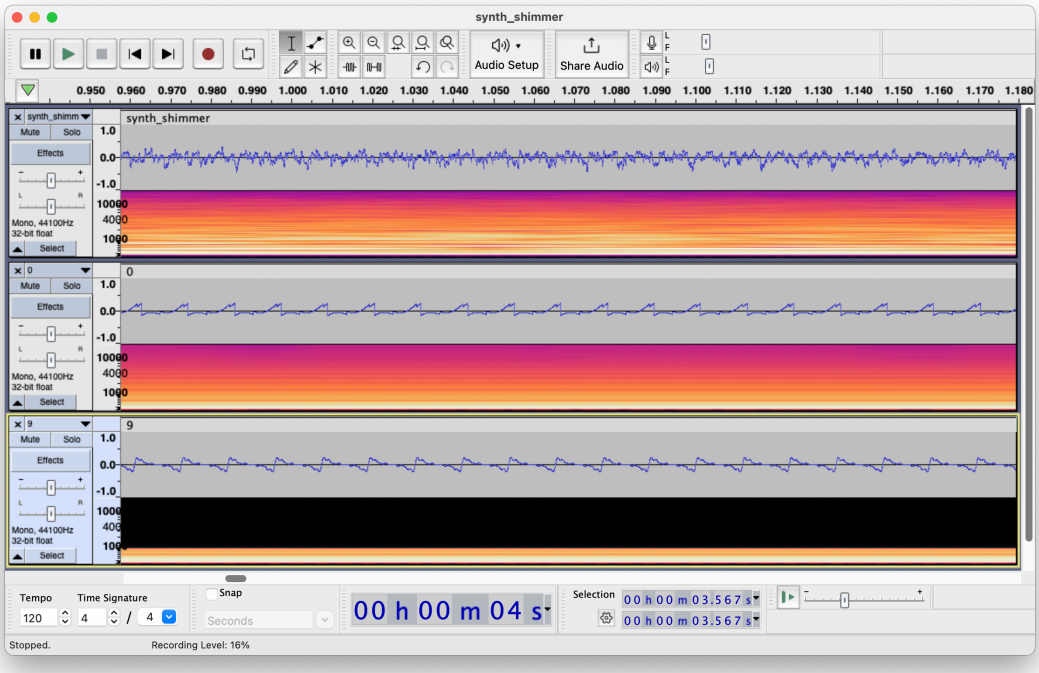


Figure 25: Waveforms and spectrogram of the shimmer target followed by the generated ones by subtractive and additive synthesis.

6.3.3. Genetic Algorithm vs Hill Climbing

To compare the performance of the GA with hill climbing, 10 simulations of each type were run with the number of individuals generated as a common metric. As shown below, hill climbing encountered a local maximum in every instance, being unable to achieve a fitness score over 0.5, although it did eventually manage to find the right fundamental frequency.

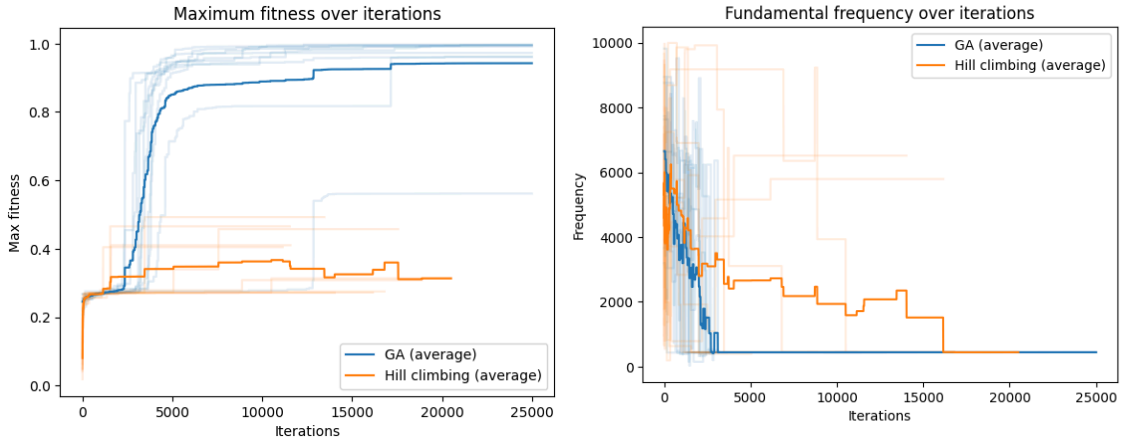


Figure 26: Performance of 10 GA vs hill climbing simulations based on fitness and fundamental frequency. Configuration for the GA: `population=100, mutation_rate=0.05, n_random_additions=4, evolution='constant'`. Configuration for hill climbing: `mutation_rate=0.05, init_step_size: 1.0, min_step_size: 0.0001`.

6.3.4. Fitness Method

There are currently two fitness methods supported: MSE over the frequency domain and Euclidean distance on the time domain. It was already intuited that frequency-domain evaluation was the more accurate of the two, and that hypothesis is backed by the following simulation results.

We can observe that frequency-domain evaluation achieved a high fitness value with a very steep rate of convergence. Interestingly, time-domain evaluation was almost constant and identical in every simulation, at around 0.7.

In terms of fundamental frequency, the variance in time-domain simulations was very high, and the average simulation did not converge to the target fundamental frequency within 500 generations. Frequency-domain evaluation, on the other hand, reached convergence within 150 generations.

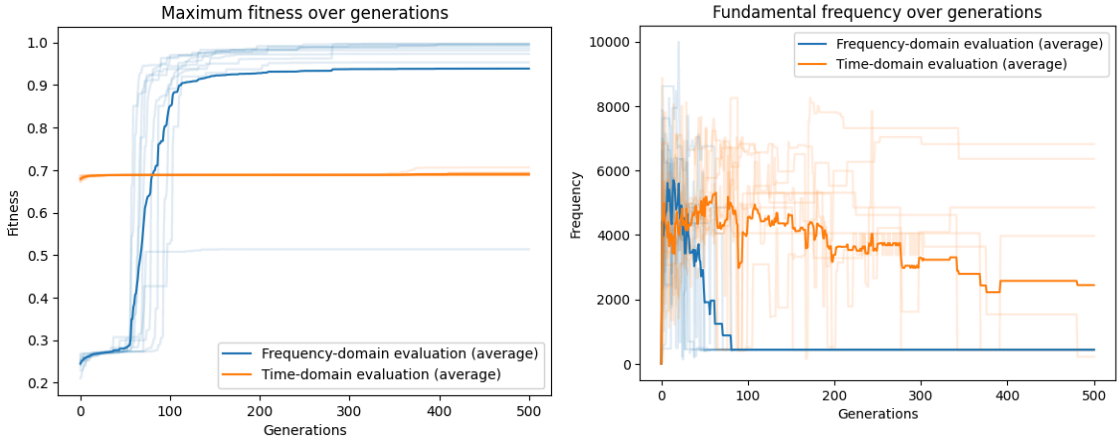


Figure 27: Performance in terms of fitness and fundamental frequency of 10 subtractive synthesis simulations with frequency and time-domain fitness evaluation. Configuration for the GA: `population=100, mutation_rate=0.05, n_random_additions=4, evolution='constant'`.

6.3.5. Constant vs Incremental Population

To compare the performance of the algorithm with constant or incremental population, ten simulations of each type were run. A population may increase or stay constant depending on whether the number of individuals selected increases with random additions. As expected, the increasing population simulations converged faster than the constant ones, albeit this comes with the caveat of a progressively higher execution cost.

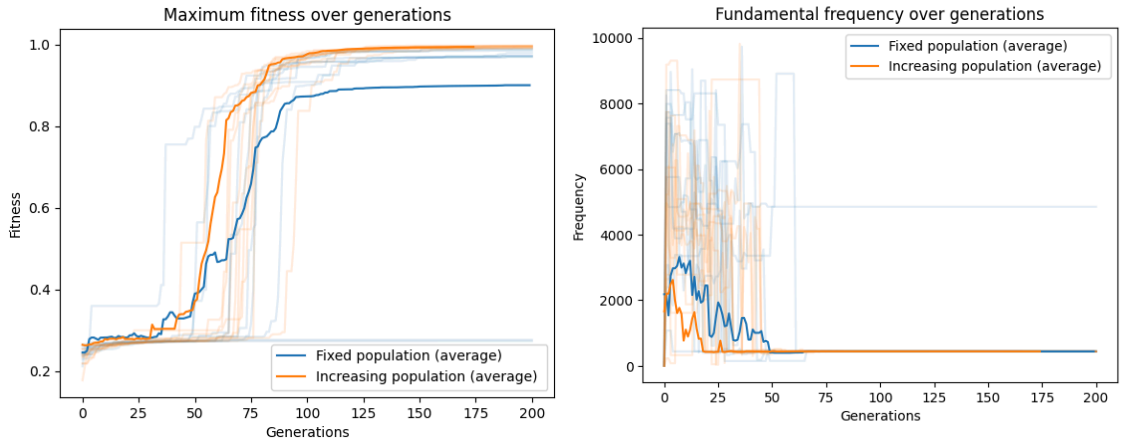


Figure 28: Performance in terms of fitness and fundamental frequency of 10 subtractive synthesis simulations with fixed and evolving populations. Configuration for the GA: `population=100, mutation_rate=0.05, n_random_additions=4`

6.3.6. Population Sizes

Multiple simulations were run with exponentially incremental population sizes, from 8 to 512. The hypothesis was that larger population sizes would result in more accurate simulations, at the expense of longer execution times. As expected, the simulation with 512 individuals performed best while the one with only 8 individuals did not achieve higher than a fitness score of 0.3.

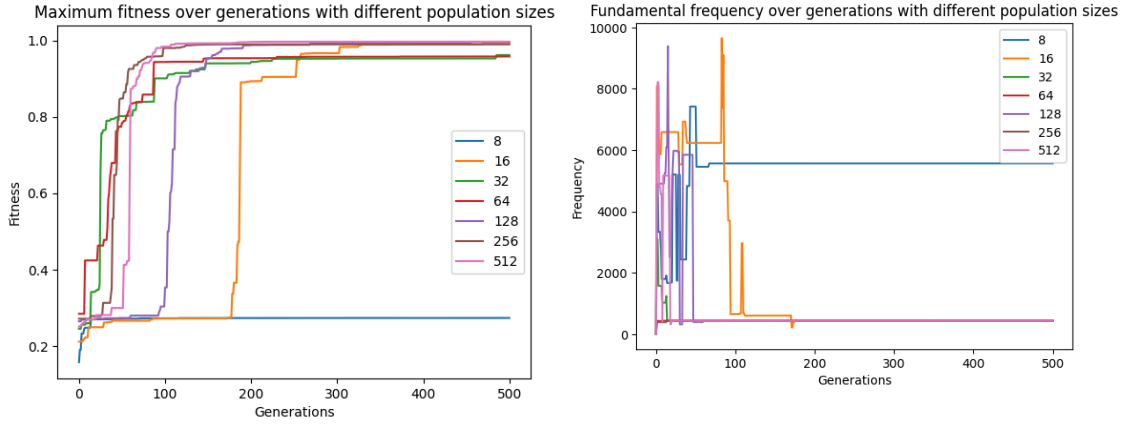


Figure 29: Results of running 7 simulations with different population sizes in terms of fitness and fundamental frequency. Configuration for the GA: `mutation_rate=0.05, n_random_additions=0, evolution='constant'`.

6.3.7. Mutation Rates

To study the role of mutation rates in GAs, ten simulations were run with values ranging from 0 to 0.45. The simulation with no mutation whatsoever did not manage to achieve a fitness value higher than 0.3, while the other simulations all scored over 0.9. Each mutation rate over 0.3 took longer than the previous to reach an acceptable fitness value, although there was little difference in terms of fundamental frequency.

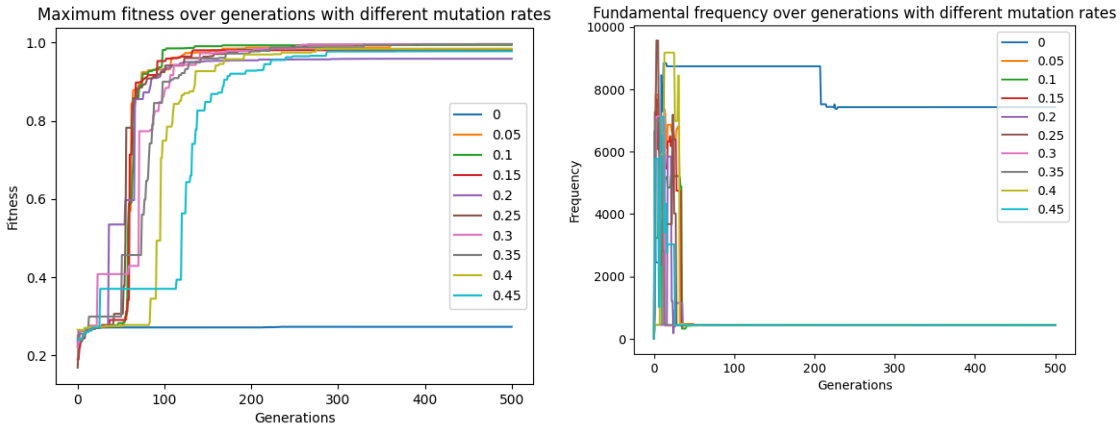


Figure 30: Results of running 10 simulations with different mutation rates in terms of fitness and fundamental frequency. Configuration for the GA: `population=100, n_random_additions=4, evolution='constant'`.

6.3.8. Number of Random Additions

The incorporation of n randomly generated individuals into each generation can help preserve diversity among the population. Eight simulations were run with multiple values for n, ranging from 0 to 35, to test the way they affected the performance of the simulation. Simulations with $n = 20$ and $n = 25$ performed worse than the rest, but these could merely be statistical outliers. Unlike the previous two test cases, there seems to be no logical correlation between the value of n and fitness, as most simulations have followed a very similar path with no remarkable deviation. No correlation between n and the fundamental frequency reached was found either.

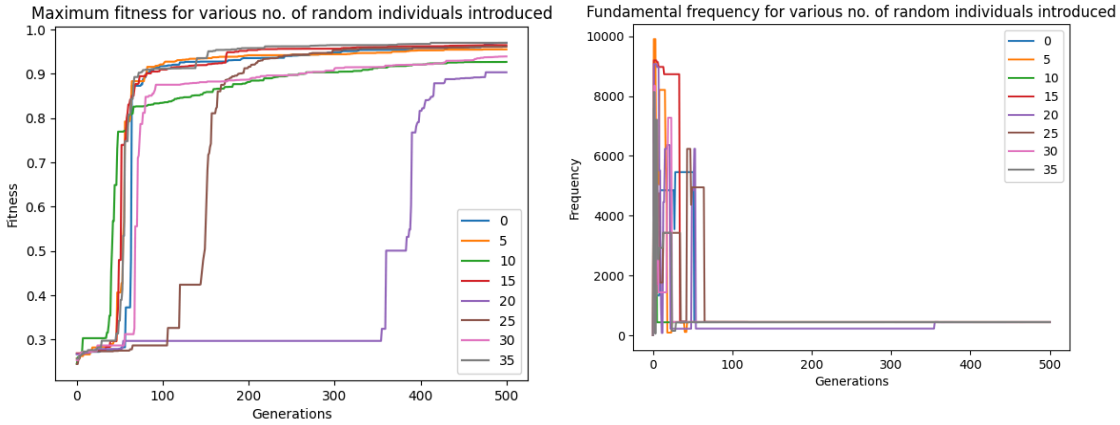


Figure 31: Results of running 8 simulations in terms of fitness and fundamental frequency with different numbers of individuals randomly introduced. Configuration for the GA: `population=100, mutation_rate=0.05, evolution='constant'`.

7. Evaluation

7.1. Design and Implementation

A library has been written that enables running custom genetic algorithm simulations to optimise the parameters of synthesiser components. Multiple configurations have been supported including two synthesis methods, three components, and two fitness methods. This is enough to perform research simulations within the scope of this project but probably not for general-purpose parameter estimation.

An API has been provided to define and set up a configurable simulation. It has also been documented to facilitate its utilisation by third parties for various purposes, such as research, industry applications, or further development. Good design practices have been followed to ensure the library remains modular, extensible, and upgradeable.

Even though some of the original requirements have not been implemented, overall the library contains the necessary functionality to meet the research goals. All the *must* requirements and most of the *shoulds* have been completed. New features could be easily incorporated without affecting the existing codebase.

Nevertheless, some aspects of the current implementation could still be improved. For instance, the state of a simulation should be stored entirely in a struct rather than across multiple local variables. The `run` and `next` methods could be separated into multiple independent functions, increasing readability and modularity. The fitness evaluation logic should still be optimised and simplified. Each variant of the `FilterComponent` enum could have been broken down into multiple structs, reducing the length of the `combine` method.

7.2. Research

Genetic algorithms are interesting global optimisation techniques which often take unusual approaches to problem solving. This could be observed in phenomena like amplitude inversion or aliasing, where the algorithm produced an unexpected yet still technically valid solution. Sometimes, it may be necessary to correct undesired behaviour by reducing or invalidating the fitness of an individual, like was done with the harmonic component.

GAs have been found to perform much better than hill climbing, in terms of the number of individuals evaluated, in this global optimisation problem. Hill climbing is a simple and efficient optimisation technique that is best suited for problems with a single local optimum due to its greedy nature. This has been demonstrated during the tests, where the algorithm was unable to progress further from a suboptimal solution.

Another noteworthy aspect of GAs is their pronounced rise from a low fitness value to a high value within just a few generations if it surpasses a certain threshold. Otherwise, the algorithm will struggle to achieve a higher fitness and thus the simulation will not yield a useful solution. The reasons for this progress barrier are still unknown. Based on the test data, it can be concluded that 8 in 10 simulations are normally successful, indicating the advantage of running multiple instances to ensure high-quality results.

The fitness evaluation method is probably the most influential aspect of a GA. Frequency-domain evaluation was found to be far superior to time-domain, which was not able to make any effective progress.

The population size is also an important factor to consider as the quality of candidate solutions tends to improve with the number of individuals represented. Nonetheless, a simulation containing as few as 16 individuals can already reach a satisfactory fitness value within 400 generations, which is a much lower number than expected. This implies that simulations could be effectively run at a lower computational cost for a higher number of generations, achieving more accurate results.

In genetic algorithms, an increasing population will have more individuals per generation and therefore reach higher fitness values than a constant one. However, this improvement may not outweigh the continuous increase in the number of fitness evaluations and recombinations required, which is something that would need to be formally studied and benchmarked.

Random additions did not seem to have a notable impact on the performance of the algorithm, even though the introduction of randomness should theoretically enhance the collective genotype. Furthermore, increasing the number of random additions significantly slowed down the simulation, which could be caused by inefficiencies in the implementation. The effect of random additions on a simulation is therefore inconclusive and would require further study.

Mutations are another way of bringing randomness into the simulation. Preserving diversity in the population while retaining enough genetic information from the previous generation is an important tradeoff to consider. Based on the tests, we can conclude that the optimal mutation rate in this context is between 0.05 and 0.25.

Currently, subtractive synthesis generally performs better than additive synthesis, despite some obvious limitations with the oscillator component. It would have

been interesting to introduce an additional component which constructed a signal from multiple independent sine waves and was not restricted by the mathematical relations between them, thus being able to generate any periodic signal, in theory. Nevertheless, there would still be a fundamental limitation in the number of sine waves that could be represented in the genotype, augmenting the search space dimensionality as this number increases.

In summary, it can be concluded that genetic algorithms are effective global optimisation techniques that can be applied to sound synthesis parameter discovery. Synthesisers are highly complex devices that require a sophisticated architecture to produce certain sounds beyond basic component capabilities. This is why most limitations and inaccuracies encountered are not believed to be directly caused by the GA itself, but rather by the correspondence between the genetic representation and the synthesiser parameters, the fitness method, the synthesis techniques chosen, or their implementations.

8. Project Management

8.1. Time Management

At the beginning of the project, a Gantt chart was designed to efficiently distribute time for each phase and task. Gantt charts are useful tools to visualise project timelines and deadlines, manage task dependencies, and track progress. The majority of the preliminary research was scheduled to be completed by the progress report deadline, with the design, implementation, simulation, evaluation, and report writing planned for afterwards. This strategy would ensure having sufficient knowledge and resources to effectively proceed with the next stages.

In practice, the plan was closely followed with some overlaps between the research, implementation, and evaluation phases. The codebase was set up earlier than planned and most of the time and effort was spent on the implementation. There were no major delays at any stage of the project and the number of hours invested per week has been consistent, albeit gradually increasing over time.

The original and updated Gantt charts have been included in the appendix Section B.

8.2. Risk Assessment

Every project carries potential risks that may hinder its progress. Identifying and addressing these risks from an early stage can help reduce their chances of occurring and impact. A table containing risk assessment, mitigation, and evaluation can be found in the appendix Section D.

8.3. Reflection

Throughout a project of this magnitude, it is important to have a clear sense of direction and follow a strong plan. Even though there were no major project management issues, there are still some potential improvements.

For instance, it was unwise to start part of the implementation so early without studying the project's feasibility and performing extensive research first. Specifically, a basic GUI was started following a top-down approach which later

had to be discarded due to user feedback no longer being relevant in the system. Despite still being a learning experience, this time would have been better dedicated to more research instead.

Finding the right purpose and type of project was not something trivial, as the project carries a strong research and engineering component. Meeting the research goals was not feasible without first developing a robust system, just as it was impossible to validate the implementation without conducting thorough testing and research on its effectiveness. For that reason, both aspects were incorporated into the project's aims, making it both a design-and-build and research work.

On many occasions, it was also difficult to adhere to the original plan in the Gantt chart, as the design, implementation and testing phases often overlapped each other. Sometimes, it was necessary to ensure a certain feature had been correctly implemented, by running unit tests or evaluating simulations, before implementing another component.

9. Conclusion and Future Work

This project has successfully developed a system that optimises synthesis parameter values for a target sound wave and evaluated its performance across multiple configurations. This has been accomplished through the design and development of a library in Rust that can execute complex genetic algorithms and return accurate results within seconds.

In the future, the implementation could be expanded to support a larger number of synthesis methods and components, which may perform better with richer and non-periodic signals. Envelopes and LFOs for instance would enable amplitude modulation, for which new fitness evaluation techniques would need to be explored.

Some design changes could also be considered, such as rearranging the structs representing individuals to contain a `Vec<dyn Component>` instead. This would allow for more flexibility in the order and number of components applied, including repetition. A function or macro could also be implemented to validate the component application sequence at compile time to ensure a correct processing pipeline. New API endpoints could be designed to enable more flexibility in the GA specification, such as selection, mating, or recombination. Error handling could be improved by only panicking on irrecoverable errors and building custom error types that provide helpful information about failures.

At the moment, a limited set of information and statistics can be retrieved from the simulation, such as maximum fitness, average fitness, standard deviation, fundamental frequency, or parameter values. Being able to visualise and inspect each individual and generation in detail may be helpful for future researchers. A GUI could be developed to display the state of the simulation in real time.

A. Simulation Source Code

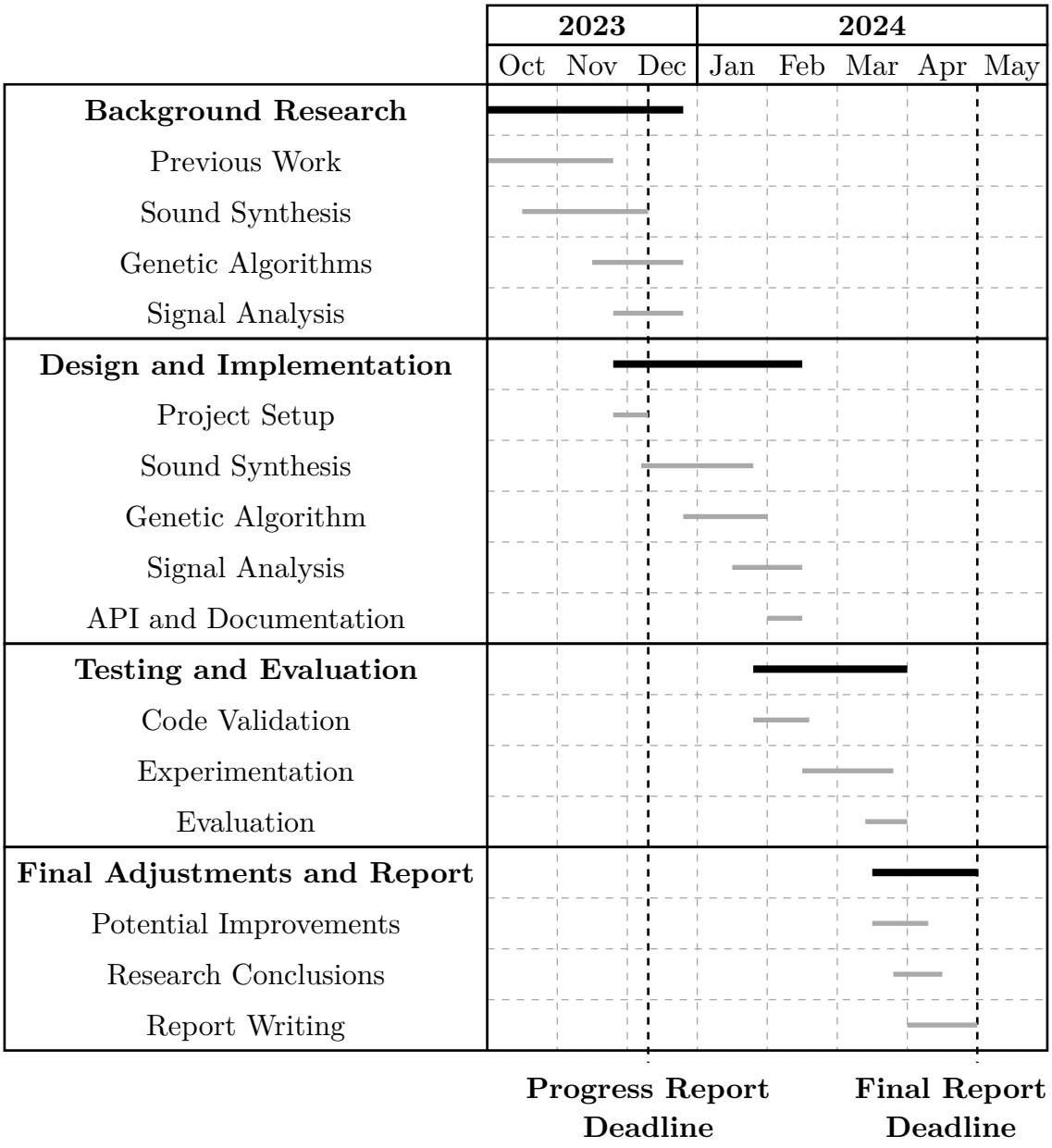
The following is the core implementation of the genetic algorithm simulation, including the `run` and `next` methods.

```
impl<T: Individual> GASimulation<T> {

  pub fn run(&mut self) -> Result<T, GeneticSimulationError> {
    // let mut generation = 0;
    let mut recorder: Recorder<GenerationRow> = Recorder::new();

    if self.csv_export.is_some() {
        recorder.add_record(self.into());
    }

    while self.generation < self.max_generations {
      // print current population
      let fittest: &T = self.population
        .first()
        .expect("There should be a fittest individual in the population");
      println!("Gen: {}, - {:?}", self.generation, fittest.dbg());

      // calculate the next generation and update state
      self.next()?;

      // increase generation count
      self.generation += 1;

      // update the number of evaluations
      self.evaluation += self.population.len() as u32;

      // update the record
      if self.csv_export.is_some() {
        recorder.add_record(self.into());
      }
    }

    if let Some(file_name) = &self.csv_export {
      recorder.to_csv(file_name).unwrap();
    }

    // Once the iteration is finished, we select the fittest in the final population
    let fittest: T = self.population
      .first()
      .expect("There should be a fittest individual in the population").to_owned();
    println!("{:?}", fittest.dbg());

    if let Some(file_name) = &self.signal_export {
      fittest.to_signal().to_wav(file_name).unwrap()
    }

    Ok(fittest)
  }
```

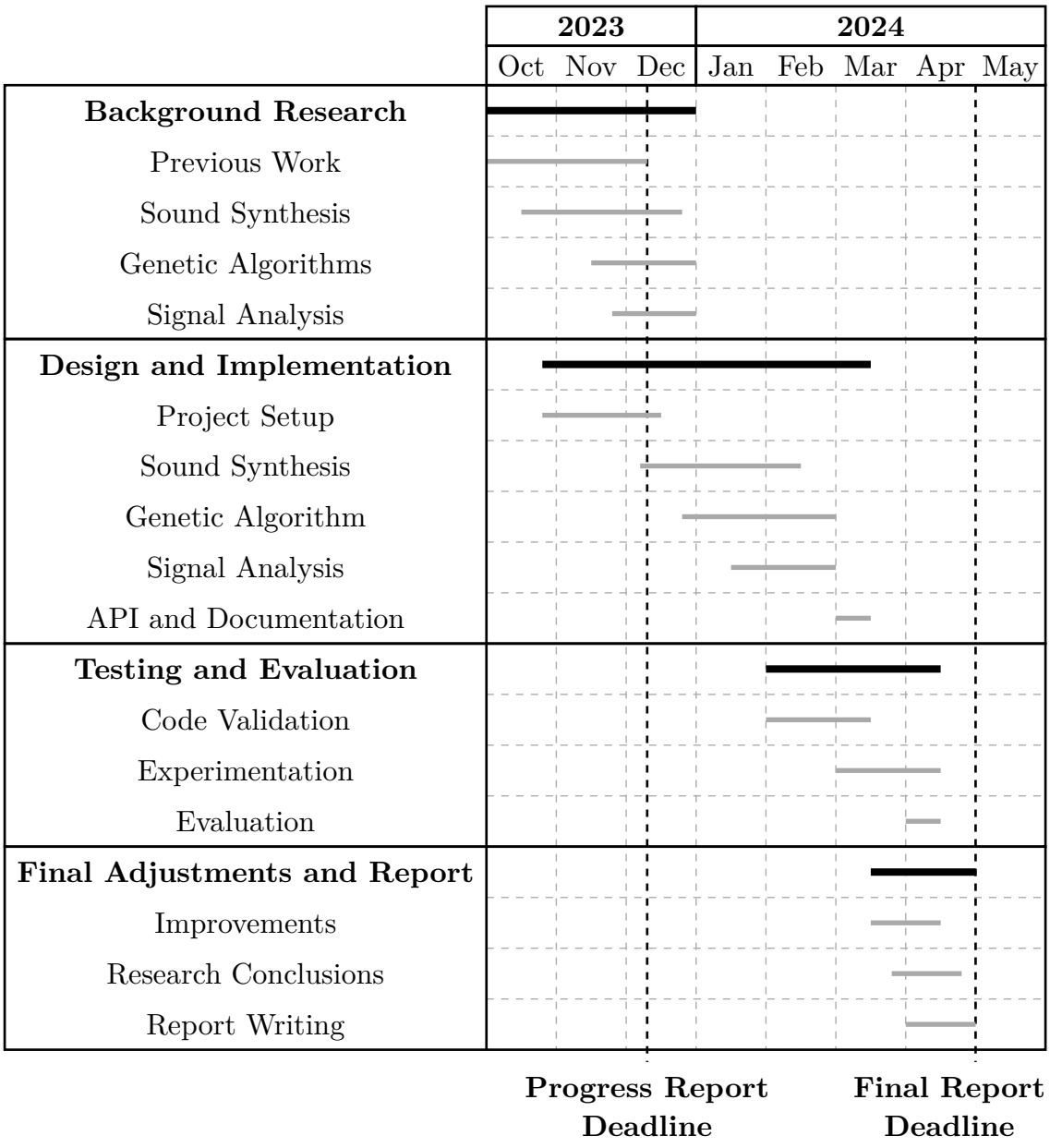
```
  fn next(&mut self) -> Result<(), GeneticSimulationError> {
    // Add n randomly generated individuals to the current population and sort it.
    let mut current_population = self.population.clone();
    let mut random_additions = vec![];
    for _ in 0..self.n_random_additions {
      random_additions.push(
        self.generator
          .generate()
          .map_err(|_| GeneticSimulationError::RandomIndividualNotGenerated)?
      );
    }
    current_population.extend(random_additions);
    current_population.sort_by(|a, b| b.cmp(a));

    // Number of selected individuals for the next generation.
    let n_selected = match self.population_evolution {
      PopulationEvolution::Constant => {
        self.initial_population as usize / 2
      }
      PopulationEvolution::Increasing => { current_population.len() / 2 }
    };

    // Construct a new population vec from the n selected individuals.
    let mut new_population: Vec<T> = Vec::from(
      &current_population[0..n_selected]
    );
    let offspring_mutex: Arc<Mutex<Vec<T>>> = Arc::new(Mutex::new(vec![]));

    let mut rng = thread_rng();

    // Perform crossover by randomly mating selected individuals.
    for _ in 0..2 {
      new_population.shuffle(&mut rng);
      new_population.par_iter().chunks(2).for_each(|p| {
        if p.len() == 2 {
          if let Some(c) = p[0].crossover(p[1], self.mutation_rate) {
            let mut guard = offspring_mutex.lock().unwrap();
            guard.push(c);
          }
        }
      });
    }

    let offspring = offspring_mutex.lock().unwrap();

    // Join the new population and offspring vecs, then sort it.
    new_population.extend(offspring.to_vec());
    new_population.sort_by(|a, b| b.cmp(a));

    // Update the generation population with the new one.
    self.population = new_population;

    Ok(())
  }
}
```

B. Gantt Chart

B.1. Estimated



B.2. Actual


2023
2024
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Background Research
Previous Work
Sound Synthesis
Genetic Algorithms
Signal Analysis
Design and Implementation
Project Setup
Sound Synthesis
Genetic Algorithm
Signal Analysis
API and Documentation
Testing and Evaluation
Code Validation
Experimentation
Evaluation
Final Adjustments and Report
Improvements
Research Conclusions
Report Writing
Progress Report Deadline
Final Report Deadline

C. Original Requirements

No.	Requirement	Priority	Status
1	Successfully run a GA simulation	MUST	Completed
2	Customise the configuration of the GA	MUST	Completed. The properties that can be currently customised are target function, synthesis method, filter component, number of generations, population size, population evolution, mutation rate, and diversity rate.
3	Evaluate individuals based on spectral analysis	MUST	Completed
4	Import the target wave from any WAV file	MUST	Completed
5	Support subtractive synthesis	MUST	Completed
6	Have implemented oscillators	MUST	Completed
7	Provide an API for research and external apps	SHOULD	Completed
8	Evaluate individuals based on time-domain analysis	SHOULD	Completed
9	Have implemented envelopes, unison, and LFOs	SHOULD	Not completed. Envelopes and LFOs added considerable complexity to the system. The unison component was excluded in the end due to time constraints and the difficulty of reconstructing a continuous signal from its samples.

No.	Requirement	Priority	Status
10	Support FM and additive synthesis	SHOULD	Partially completed. Only additive synthesis has been incorporated.
11	Provide comprehensive API documentation	COULD	Completed
12	Support wavetable and granular synthesis	COULD	Not started.
13	Import the target wave from other file formats	COULD	Not started. Unlike WAV, formats like MP3 use lossy compression to encode data, which adds unnecessary complexity and inaccuracy to its conversion into samples.
14	Support IGAs through human-based evaluation	COULD	Not started. Many components of the library could be reused to design an interactive simulation. However, given the large number of fitness evaluations that would need to be manually and subjectively done, it would be counter-productive in its current form.
15	Provide a GUI for feedback-based sound synthesis	WON'T	Not completed. A user interface was started at the early stage of the project, but has not been further developed in favour of more important components.

Table 3: List of library requirements sorted by MoSCoW prioritisation.

D. Risk Assessment

Risk	I.	L.	Mitigation	Comments
Underestimating the time needed to complete the essential requirements.	5	3	Write a detailed plan in advance of each task that needs to be done and how long it will take; modularise the GA, synthesis and signal analysis components so that they can be developed independently.	Even though more features could have been included, the essential requirements were all completed and it has been possible to use the system for research.
Being unable to find reliable evaluation methods for certain synthesis components.	3	3	Build a solid understanding of alternative methods to spectral analysis that could detect changes in amplitude over the time domain.	Two fitness methods have been implemented, based on frequency and time analysis. Other techniques like cross-correlation could have been completed, but are not necessary for any of the current components.
Not being able to effectively compare the GA results with other optimisation methods.	2	4	Provide charts to graphically compare the generated sound wave to the target, as well as use perceptual evaluation.	Charts displaying the result of multiple simulations covering a wide range of cases have been provided.

Table 4: List of risks initially estimated and a brief reflection on them, where *I.* indicates impact and *L.*, likelihood.

E. Technologies Used

Tool	Purpose
Rust	Building the library, writing tests, and running simulations
Python	Importing the simulation data with `pandas` and plotting charts with `matplotlib`
CSV	Storing statistical data about the simulation
Typst	Report writing
Figma	Designing charts

Table 5: List of tools used and their application.

F. Word Count

Using the Typst `wordometer` package, the total word count of the document body is **9537 words**.

G. Project Brief

Problem Statement

When working with synthesizers in Digital Audio Workstations (DAWs), finding a precise timbre can be a challenging task, mainly due to the complexities of categorising, describing, and mapping sounds to specific waveforms. Music producers often need to familiarise themselves with multiple synthesizers, each with its particular architecture, parameters, and set of possible outputs, which can be an inconvenience. Besides, synthesizers producing sharper sounds are often expensive and yield results not usually found in more accessible ones. These challenges can sometimes lead to producers settling for a configuration which approximates their original intent, even when there still is room for improvement.

Goals

The project aims to provide a way for electronic music producers and audio engineers to stochastically generate a synthetic sound purely based on their feedback. This will be accomplished through the use of genetic algorithms, which will allow an initial population of soundwaves to progressively evolve into soundwaves closer to the user's aim. Aesthetic selection will allow the user to rate each sound for each generation, determining its chances of reproducing and passing its configuration onto the next generation. However, an objective, well-defined fitness function will be used to evaluate the quality and success of the synthesis process, by comparing the generated sounds to preset samples. Furthermore, this project also aims to examine the advantages and disadvantages of stochastic sound synthesis compared to more traditional means.

Scope

Genetic algorithms will be used in a modular way during the different stages of soundwave synthesis: generating the basic periodic waveform, unison, filters, ADSR envelopes, LFOs, FM, EQs, and other effects. Due to the complexity of digital audio signal processing (DASP) and time constraints, only some of the methods and parameters mentioned will be within the scope of this project. On the other hand, the software should allow users to export the final soundwave at a specific frequency, in order to be imported into a DAW as a virtual instrument (VST). Potentially, MIDI input could also be supported, enabling the user to evaluate the synthesis directly within the software. Any additional DASP methods

as well as other features related to the integration and compatibility between the program and DAWs will most likely not be covered.

Bibliography

[1] S. Liu and D. Manocha, "Sound Synthesis, Propagation, and Rendering: A Survey." Accessed: Dec. 11, 2023. [Online]. Available: http://arxiv.org/abs/2011.05538

[2] N. Bernardini and G. de Poli, "The Sound and Music Computing Field: Present and Future," *Journal of New Music Research*, vol. 36, no. 3, pp. 143–148, Sep. 2007, doi: 10.1080/09298210701862432.

[3] C. G. Johnson, "Exploring Sound-Space with Interactive Genetic Algorithms," *Leonardo*, vol. 36, no. 1, pp. 51–54, 2003, Accessed: Oct. 04, 2023. [Online]. Available: https://www.jstor.org/stable/1577281

[4] Y. Lai, S.-K. Jeng, D.-T. Liu, and Y.-C. Liu, "AUTOMATED OPTIMIZATION OF PARAMETERS FOR FM SOUND SYNTHESIS WITH GENETIC ALGORITHMS," 2006.

[5] M. Miki, H. Orita, S. H. Wake, and T. Hiroyasu, "Design of Sign Sounds using an Interactive Genetic Algorithm," in *2006 IEEE International Conference on Systems, Man and Cybernetics*, Oct. 2006, pp. 3486–3490. doi: 10.1109/ICSMC.2006.384659.

[6] R. Priemer, *Introductory Signal Processing*. World Scientific, 1991.

[7] A. Agarwal and J. Lang, *Foundations of Analog and Digital Electronic Circuits*. Elsevier, 2005.

[8] A. Oppenheim and J. Lim, "The importance of phase in signals," *Proceedings of the IEEE*, vol. 69, no. 5, pp. 529–541, May 1981, doi: 10.1109/PROC.1981.12022.

[9] H. L. Kennedy, "A New Statistical Measure of Signal Similarity," in *2007 Information, Decision and Control*, Feb. 2007, pp. 112–117. doi: 10.1109/IDC.2007.374535.

[10] P. Senin, "Dynamic Time Warping Algorithm Review," Jan. 2009.

[11] O. Özhan, "The Fourier Transform," *Basic Transforms for Electrical Engineering*. Springer International Publishing, Cham, pp. 335–439, 2022. doi: 10.1007/978-3-030-98846-3_6.

[12] D. J. Benson, *Music: A Mathematical Offering*. Cambridge University Press, 2007.

[13] J. McDermott, M. O'Neill, and N. J. L. Griffith, "Target-driven genetic algorithms for synthesizer control."

[14] M. Macret, P. Pasquier, and T. Smyth, "Automatic Calibration of Modified FM Synthesis to Harmonic Sounds using Genetic Algorithms," *Proceedings of the 9th Sound and Music Computing conference (SMC 2012)*, pp. 387–394, Jul. 2012.

[15] B. Ostertag, "Human Bodies, Computer Music," *Leonardo Music Journal*, vol. 12, pp. 11–14, 2002, Accessed: Nov. 26, 2023. [Online]. Available: https://www.jstor.org/stable/1513343

[16] M. Cobussen, V. Meelberg, and B. Truax, *The Routledge Companion to Sounding Art*. Routledge, 2016.

[17] J. Pekonen and V. Välimäki, "The Brief History of Virtual Analog Synthesis," 2011.

[18] D. Hosken, "Review of Computer Music: Synthesis, Composition, and Performance," *Computer Music Journal*, vol. 23, no. 4, pp. 92–95, 1999, Accessed: Apr. 17, 2024. [Online]. Available: https://www.jstor.org/stable/3680689

[19] J. Pekonen, "Computationally Efficient Music Synthesis – Methods and Sound Design."

[20] C. Roads, "Introduction to Granular Synthesis," *Computer Music Journal*, vol. 12, no. 2, pp. 11–13, 1988, doi: 10.2307/3679937.

[21] T. Bartz-Beielstein, J. Branke, J. Mehnen, and O. Mersmann, "Evolutionary Algorithms," *WIREs Data Mining and Knowledge Discovery*, vol. 4, no. 3, pp. 178–195, 2014, doi: 10.1002/widm.1124.

[22] A. Shukla, H. M. Pandey, and D. Mehrotra, "Comparative review of selection techniques in genetic algorithm," in *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, Feb. 2015, pp. 515–519. doi: 10.1109/ABLAZE.2015.7154916.

[23] G. Syswerda, "Uniform Crossover in Genetic Algorithms," presented at the Proc. 3rd Intl Conference on Genetic Algorithms 1989, Jan. 1989.

[24] A. E. Eiben and J. E. Smith, "Fitness, Selection, and Population Management," *Introduction to Evolutionary Computing*. in Natural Computing Series. Springer, Berlin, Heidelberg, pp. 79–98, 2015. doi: 10.1007/978-3-662-44874-8_5.

[25] P. Ross and D. Corne, "Applications of genetic algorithms."

[26] J. McDermott, N. J. L. Griffith, and M. O'Neill, "Evolutionary Computation Applied to Sound Synthesis," *The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music*. Springer, Berlin, Heidelberg, pp. 81–101, 2008. doi: 10.1007/978-3-540-72877-1_4.

[27] K. Rasheed, H. Hirsh, and A. Gelsey, "A genetic algorithm for continuous design space search," *Artificial Intelligence in Engineering*, vol. 11, no. 3, pp. 295–305, Jul. 1997, doi: 10.1016/S0954-1810(96)00050-7.