Application of Genetic Programming to Creative Sound Synthesis in Web Audio

Student Name: B.A. Collins
Supervisor Name: S.P. Bradley
Submitted as part of the degree of MEng Computer Science to the
Board of Examiners in the Department of Computer Sciences, Durham University

Abstract—In modern music, sound synthesisers are everywhere. A sound synthesiser (henceforth 'synthesiser') is a physical or software device which generates a sound based on some set of input parameters. These devices have an extremely steep learning curve, and often require significant financial investment, so have very low accessibility. In this paper, we present a tool which allows users to creatively generate their own sounds (henceforth 'patches') with no prior knowledge required, and with no physical or digital dependencies other than a modern web browser. This is accomplished by applying genetic algorithms to automatically explore the parameter space of a simulated modular synthesiser, with the user selecting their favourite sound on a regular basis. Full control over the parameters is also provided, as well as the option to export each patch in the Faust DSP format, for more experienced users. The intended outcome is to produce a novel, easy-to-use, browser-based synthesiser generation tool that improves upon previous work on this field and allows users of all categories to take part in sound design.

Index Terms—Evolutionary computing and genetic algorithms (I.2.m.c); Program synthesis (I.2.2.c); Signal analysis, synthesis, and processing (H.5.5.c); Sound and music computing (H.5.5).

1 Introduction

Over the course of nearly a century of electric sound synthesis, artists have found myriad creative methods to generate novel and interesting sounds. This diversity makes synthesis a hugely exciting creative technique, but also can stop it being financially or intellectually accessible. Sound design features a steep learning curve, and industry-standard software and hardware often costs far more than many are willing to spend.

The aim of this research, then, was to allow users to generate their own sounds through an abstract process that allows them to focus on the sounds being generated, without needing to immerse themselves in the details of those sounds' construction.

There are various techniques to synthesise sounds, and a small number of the more popular options will be described below. The techniques of additive synthesis, subtractive synthesis, frequency modulation and modular synthesis will be described in detail, and wavetable, granular and physical modelling synthesis will be covered more briefly. For more information on sound synthesis and its history, see [1].

Perhaps the simplest technique is additive synthesis: the creation of timbres by adding sine waves together. This method can theoretically recreate any sound with enough sine signals, and Fourier transforms make it simple to find the necessary harmonics for some sound sample. In this technique, the sonic nuance comes from in the frequencies and relative amplitudes of these waves. Subtractive synthesis is the opposite of additive synthesis: it begins with a rich timbre, generated by an oscillator or set of oscillators and/or noise generators, and then subtracts harmonics using filters which block frequencies outside a given range.

Popularised in the 1980s, frequency modulation (FM) involves controlling the frequency of one 'carrier' oscillator with the output of another 'modulator' oscillator. Essentially, the carrier's frequency input is being adjusted rapidly in real-time. FM is capable of producing very complex sounds, and tends to sound 'messier' than other, 'cleaner' modes of synthesis.

Other methods include wavetable synthesis (the playback of pre-designed repeating waveforms, often involving realtime interpolation between several of these waveforms), granular synthesis (taking very short 'grains' of sound samples and playing these grains in combination), and physical modelling (simulation of real-world physics and sound transmission), but this is not an exhaustive review. Modern standalone synthesisers typically offer several of these techniques in a single box in order to facilitate a wide range of timbres.

The most flexible form of synthesis, modular synthesis, involves the physical interconnection of discrete 'modules' and tuning of physical knobs in order to find new and interesting sounds. Its flexibility in both topology and numerical parameter ranges means that most physical modular synthesisers can recreate additive, subtractive and FM synthesis by default, and with the inclusion of specialised modules can recreate other methods also.

An example of a physical modular synthesiser can be seen in figure 1 - note that the cable connections between modules and the tuning of knobs both affect the sound(s) produced. Similarly, a software modular synthesiser is visible in 2. Henceforth, the modules and the connections between them will be known as a synthesiser's 'topology', and the values chosen on those modules' controls will be called 'parameters'. The combination of topology and parameters will be known as a 'patch'; this is a common term in the synthesiser industry, derived from the fact that sounds

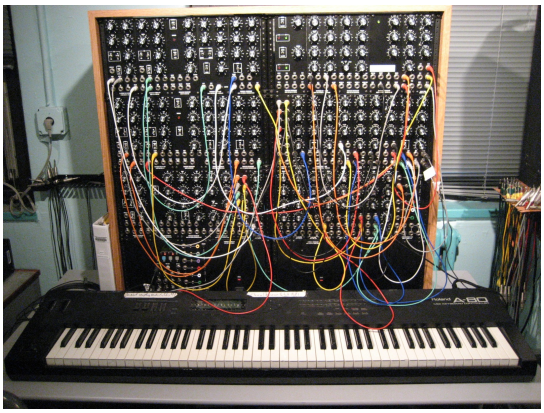


Fig. 1. An example of a modular synthesiser. Bennett, CC BY-SA 2.0, via Wikimedia Commons.

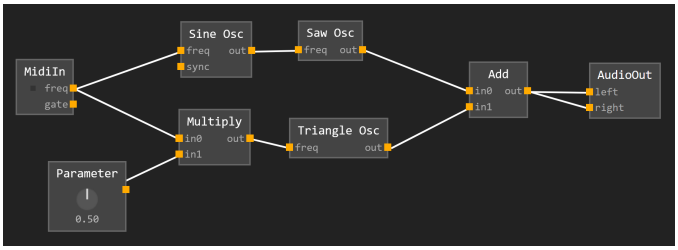


Fig. 2. An example of a simple patch on a software modular synthesiser. Generated using the Zupiter web synthesiser. This patch will be used in other examples later in the paper.

are constructed on a modular synthesiser by connecting modules with 'patch cables' as visible in figure 1.

By virtue of its flexibility, modular synthesis (albeit in the software domain) has been the technique of choice for some recent work in this area [2], and is also applied in this paper.

Much previous work applying genetic algorithms to sound synthesis (including but not limited to [2], [3], [4], [5], [6], [7], [8]) has looked at the generation of sounds that are similar to a given 'target' sound - which may be acoustically sampled from a physical instrument, as an example. The benefit of this strategy is that it provides a simple target: a single, provided sound. This previous research has looked at a range of different techniques to reproduce the target sound as accurately as possible, and settled on a number of robust genetic strategies.

A smaller minority of research ([9], [10], [11], [12]) has been dedicated to a more creative process: allowing the user to generate their own sounds, aided by a computer. In this strategy, known as 'Interactive Evolutionary Computation' (IEC) [13], the user guides a computer as it searches a synthesiser's parameter space.

This work was intended to combine these approaches - converging on a known 'target' sound, but allowing the user to choose the target, and allowing multiple repeated rounds of evolution with different targets. The focus on creativity rather than accuracy also allowed for run times to be reduced when evolving, as the end goal was only to find sounds similar to the target, rather than sounds that are the same.

The decision to model a modular synthesiser means that the problem is twofold: firstly, a topology must be chosen, and secondly, that topology's parameters must be tuned. This work absorbed parameters into the topology itself, and allowed them to mutate along with the synthesiser's structure, in order to simplify the applied process.

A number of other simplifications have been made: firstly, the modular topology has been limited to a tree in order to allow simpler recursive definitions, mutation and generation functions. Secondly, the focus has been placed on the sound's static timbre and temporal modulation has therefore been removed from the genetic process. That is, features which allow the sound to change over time like envelopes and low-frequency oscillators ('LFOs') are not modelled in the generated graphs. Finally, in order to increase stability of the generated results, the sets of nodes which can be connected together have been controlled in the recursive functions to generate and mutate the topology. These limitations are discussed in more detail in section 3.

It is also worth noting that this research was intended to focus on the niche of sound synthesiser timbres, and not on the related fields of automatic composition, synthesis of media other than sound, machine learning methods or human interface devices. These fields are relevant, but out of scope for this project.

To summarise: the ultimate aim of this project was to create an easy-to-use tool which enables the user to explore the parameter space of a modular synthesiser through interactive evolutionary computation. The user is presented with a number of pre-generated sounds and selects a favourite. The computer then generates a small population of new synthesisers, and uses genetic algorithms to converge on the user's chosen sound, before presenting its results to the user for evaluation. The tool also allows the user to export their patch in a portable format that can be compiled for a range of different use cases. The implementation runs in a standard web browser, meaning the user needs no dependencies in order to operate it.

In the following section, a brief overview of relevant literature will be presented. This will be followed by a discussion of the methodology applied, including detailed discussion of the design decisions that were involved in the project's implementation. Section 4 will examine the results achieved with the tool in its final form, before the report is concluded with a list of proposals for future study into this area.

2 Related Work

2.1 Search Strategy

Before embarking on the work of this project, the current state of the art was established by searching the ACM Digital Library, IEEE Xplore, SpringerLink and Google Scholar. Search terms used were:

genetic algorithms genetic programming evolutionary algorithms
sound synthesis sound synthesizer configuration sound matching parameter estimation

These were combined in pairs (one term from each box) to find relevant research on the above sources. Any research that cited particularly relevant papers ([2], [11], [12]) was also included. The results were then filtered to exclude duplicates and research into irrelevant areas as detailed in section 1.

Finally, papers were evaluated based on their abstract, then their conclusion and full text. Those with limited relevance or poor text quality were discarded. This research stage was completed on 23/10/2021, and gave us a final shortlist of high-quality, relevant and non-duplicate studies upon which to base our research.

2.2 Previous Work

In the application of genetic algorithms to the configuration of a synthesiser, there are a number of factors that need considering. The most important decision being made is how the synthesiser will be represented, since this model will be optimised by (and therefore restricts the choice of) the genetic algorithm. Ideally, the model would be easy to mutate, flexible in its representation of a synthesiser (so as to achieve a wider range of sounds and techniques) and computationally easy to use to generate a sound. A range of methods with various compromises on these goals will be discussed in section 2.2.1.

Moving onto the evolutionary algorithms themselves: a range of strategies have been trialled, including deep learning [14], but this project focusses upon the potential of genetic algorithms to iteratively converge upon solutions to the problem.

Section 2.2.2 will cover a selection of different fitness measures employed by previous work. The ideal fitness function would be computationally efficient and accurate at measuring similarity in both timbre and pitch. There is also the alternative strategy of using a human to judge fitness: clearly, this slows the evolution process considerably, but it allows the user to shape the direction that the algorithm takes and therefore could be said to be much more creative and useful than recreating pre-existing sounds.

Then, section 2.2.3 will discuss how the synthesiser model dictates how reproduction can occur: when modelled as a graph, it requires special graph modification algorithms, but simpler representations such as parameter bitstrings can be mutated and crossed-over in similar ways to the nucleic acids that genetic algorithms are based on. Typical reproduction methods include two-parent 'sexual' reproduction (involving some form of crossover), single-parent 'asexual' reproduction (carrying forward the genotype with some random mutation chance) and an additional step of random mutation applied to the entire population immediately before or after the formation of a new generation. The former is simple to implement with more basic synthesiser representations such as bitstrings, but becomes more complicated with graph topologies, while the remaining two can be implemented with all models. There are also less obvious choices to be made, such as the strategy used to select the individual(s) for reproduction and mutation, or the specific types of mutation that will be applied (eg, in a graph topology, there may be node type change, parameter change, node creation, node deletion, etc).

Finally, section 2.2.4 will briefly cover interesting ancillary technologies and methods that past research has employed.

2.2.1 Synthesiser Model

Many methods have been used to represent synthesiser topologies in a manner that can be evolved and evaluated in genetic algorithms. Since software synthesis is mostly used, there is the option to automatically evolve the synthesiser's topology as well as its parameters. This can be applied to many synthesis techniques, and even allows these techniques to be combined. These structures, with different modules connecting to one another with both sound and parameter signals, are most commonly represented as graphs.

Initial explorations took a range of routes here - with Takala *et al.* evolving topologies as 'timbre trees' [9], Horner *et al.* evolving parameters for a fixed FM topology [5], Jonsson *et al.* applying granular 'Fonction d'Onde Formantique' synthesis [10] and Garcia developing virtual modular synthesis topologies [6]. Trees were applied by Takala *et al.* [9], Chinen and Osaka [7] and Garcia [6], with the latter employing 'embryo trees' which described the steps required to construct a topology rather than explicitly describing the topology itself. This allowed them to use cycles in their topologies.

Later papers used more nuanced structures. Macret and Pasquier [2] applied Mixed-Type Cartesian Genetic Programming (MT-CGP), a method employing directed acyclic graphs instead of trees. This allows the generated signal paths to split and recombine, which by definition cannot be accommodated by a tree. These are co-evolved with a set of 3 input parameters - which are fixed 'inlet' values passed into each topology in the population. This model allows useful values to be retained, enduring generations which may not use them.

An interesting approach was taken by Jonsson *et al.* [11]: they evolved 'compositional pattern producing networks' (CPPNs): a type of artificial neural network which defines a function based on a given set of parameters and is not intended for conventional machine learning. These were applied to wavetable synthesis, with one parameter being varied to find the amplitude at different points in the table's period, and the additional provision of a series of sine waves to modulate.

Tatar *et al.* [15] focussed on the Teenage Engineering OP-1 synthesiser, with some success for this fixed topology. However, they admit that this gives them a much more limited search space (and therefore less timbral flexibility) compared to their previous work on coevolutionary MT-CGP. Later, Yee-King [12] developed EvoSynth - a web-based interactive synthesiser. This tool uses a unit generator graph model, and patches are constructed from oscillators and filters.

SpiegeLib [4] is a general library for the configuration of VST synthesisers using Python. In this project, the synthesiser VST is imported and treated as a 'black box' with its inputs being configured automatically by the library and outputs being evaluated without any understanding of how the two are linked. This essentially creates a fixed topology, with only parameters being tunable by the genetic

algorithm. Masuda and Saito used a similar method, also applying their genetic techniques to VST synthesisers using Python [8]. Both projects used the Dexed FM synthesiser, and the latter also used Diva, a virtual analogue synthesiser.

Regardless of the model being used, it is important that we can measure the 'quality' of a synthesiser in order to make progress towards the algorithm's goal. This is the job of a fitness measure, which will be discussed next.

2.2.2 Fitness Measure

A number of fitness metrics have been proposed for evaluating the sonic outputs of a synthesiser. The most obvious of these is human evaluation, used for 'Interactive Evolutionary Computation' (IEC), and indeed this method has been applied by [9], [10], [11] and [12]. This method brings the benefit of allowing the user to tailor the outputs towards a desired sound, or abstractly explore the space of possible sounds, rather than being constrained by the set of sounds that already exist, and the limited artistic potential of analytical methods. However, it may take the user several minutes to evaluate a population of sounds, so this does drastically impact the rate at which evolution can happen.

Several more analytical methods have been applied, but these usually converge upon a given target sound rather than aiming to create 'interesting' new sounds. The most obvious of these is performing a Fourier transform and comparing the spectra between the produced sound and target sound. Horner *et al.* used sum-squared difference between the two spectra, sampled at equally spaced intervals between the beginning of the sound and the peak of its amplitude [5]. Garcia also chose this method, applying it alongside simultaneous frequency masking (SFM) [6]. This strategy checks against a 'threshold of masking' in the amplitude domain - determining which frequency components will be heard and recognised by a human and which are less important and can therefore be allowed to deviate more.

Chinen *et al.* [7] chose to use a similar spectral method: like the previously discussed strategy, they analysed the spectra of sampled sounds and compared them - but they placed sound above a certain amplitude into 'bins', and combined adjacent bins in the frequency spectrum into wider bins - hence resulting in fewer bins to analyse.

A more recent trend has been the use of Mel-Frequency Cepstral Coefficients (MFCC). Initially developed for speech recognition, MFCC takes a Fourier transform of the sound, maps it onto the Mel scale (which models the human perception of the frequencies in the Fourier transform), takes logs of the powers in the Mel scale and finally performs a discrete cosine transform on the resulting log powers. This process returns 13 coefficients, which together represent the timbre of the analysed sound.

MFCC was applied by [3], [2] and [4] to some success. However, Masuda and Saito [8] noted that its design for voice results in an increased focus on timbre and reduced accuracy of pitch - less than ideal for musical applications. They instead compared standard genetic algorithms to two implementations using novelty as a fitness metric alongside using log-spectral distance for competition, and found that a combination of novelty and global competition performed best.

When applying genetic algorithms, particularly to more complex genotypes such as the graphs discussed above, one of the most important considerations is the method of reproduction. This choice decides how the algorithm traverses its search space, so it is crucial that it is chosen such that the population can gradually converge on good solutions.

2.2.3 Reproduction

When producing new generations, a number of strategies have been employed. The four main methods of generating a new individual in genetic algorithms are the creation of a new 'immigrant', the random mutation of an existing individual, the 'crossover' of two existing individuals and the 'carrying forward' of an individual into the next generation. These different methods can be combined, or a single method can be chosen: the decision often depends on the structures being evolved, as operations like crossover may be difficult on, say, a graph. The individuals selected for these operations are typically randomly chosen, but weighted towards the fitter individuals as defined by the functions above.

Takala *et al.* [9], with their tree-based topologies, employed a wide range of possible mutations - random node replacement, changing of a parameter or function for another one, or reordering of arguments. They also applied a crossover where nodes are swapped between two parent trees.

By contrast Horner *et al.*, with their fixed topology, encoded their genotypes as bitstrings [5]. These are much more similar to real-world genotypes so more conventional reproduction techniques were used. Random bitflip mutation affected all individuals, parents were selected in a binary tournament (where each individual traverses knockout rounds with fitness-weighted random results), and these parents performed one-point crossover (choosing a point on the gene and exchanging the sections after that point to create two new hybrid children). Jonsson *et al.* also used bitstrings and applied very similar methods, except a weighted roulette wheel was used instead of the binary tournament [10].

Chinen and Osaka used a method of mutation on their trees where each node is mutated with probability m [7]. If a node is mutated, the entire subtree beneath it is also mutated - numeric parameters are adjusted randomly, and other nodes are given the same probability m of moving in the tree. There is also the option of splitting up a 'NoiseBand' (oscillator) into multiple children, based on a coinflip. Nodes are given 'allele markers' which describe their evolutionary history, and crossover is performed by randomly swapping nodes, with preference given to nodes with similar allele markers (and therefore similarly-structured subtrees). During crossover, parameters are chosen by finding a value somewhere between the two parents' values in that location. Macret and Pasquier chose not to use crossover on their directed acyclic graphs, instead using a $1 + 4$ evolutionary strategy involving copying from the fittest member of the previous population with high rates (10%) of mutation [2]. The bitstring parameter sets are varied with random bitflip mutations and 2-point crossover.

Jonsson *et al.*, with their novel CPPN topologies, chose to use NeuroEvolution of Augmenting Topologies (NEAT - based on [16]) [11]. This evolution method begins with a simple network topology and gradually increases networks' complexity by adding and mutating nodes in the network.

For their 'EvoSynth' project, Yee-King [12] chose to use a range of mutation methods (point, grow, repeat and shrink mutation) as well as a crossover method (described as 'knitting together' two genomes) and a high rate of immigration.

Finally, recent papers appear to have gravitated towards variants of the non-dominated sorting genetic algorithm (NSGA). Notably, [15] and [8] chose to use NSGA II, and [4] used NSGA III. NSGA-II [17] measures both the fitness and the 'crowding distance' (how different an individual is to its peers) for each individual. A tournament process selects individuals preferentially for higher fitness and greater crowding distance.

2.2.4 Additional technologies

Alongside interesting techniques being used to directly represent and evolve synthesisers, there are other interesting nuances in the way that these methods are applied. For instance, [9] used their evolutionary methods to design visual feedback to go alongside the sound effects generated through synthesis - one example given was that of a ball bouncing, with each bounce being aligned with a synthesised sound effect.

Some researchers have also constructed their programs on the Web. This makes the product much more accessible, and has been facilitated by the relatively recent release of the Web Audio API. As JavaScript is quite slow, the Web is best suited to less time-dependent Interactive Evolutionary Computation (IEC) methods where processing latency is far outstripped by the time taken by the user to evaluate the populations. Examples of this include [11] and [12]. Fully automated systems would be held back by the longer runtimes offered by JavaScript, and while the modern WebAssembly pre-compiled format does offer a higher-performance option, it remains in relative infancy.

Finally, the application of genetic algorithms to existing standardised synthesis plugin formats has been recently explored by various groups. Both Shier *et al.* 's SpiegeLib [4] and Masuda and Saito's novelty-based methods [8] were designed to automatically optimise the output of standard VST software synthesis plugins. These plugins are common in music production, and in the above work are modelled as 'black boxes' with a set of MIDI inputs varying the generated sonic output.

One technology of interest in this project is the Faust (Functional AUdio STream) programming language: a functional programming language specifically designed for sound synthesis and processing, originally implemented by Orlarey *et al.* [18]. It can be compiled into a wide range of targets including web technologies like JavaScript, asm.js and WebAssembly and more audio-focussed environments like VCV Rack, Max and Pure Data. Its use in the web is also well-documented [19], and it even has its own web-based IDE. This makes it an attractive option for the generation of truly cross-platform synthesiser topologies, so we hope to make use of it in representing the patches we construct in our own project.

3 Methodology

In this section the specific techniques and design decisions employed in this project will be discussed. One major consideration when implementing the application has been to ensure that meaningful evolution happens on a reasonable timescale, as making the user wait too long for results will affect the usefulness of the tool.

The following user workflow was decided upon:

1) Generate and present a range of synthesiser configurations to the user
2) Allow the user to select a 'favourite' from this collection of sounds
3) Analyse this favourite topology, and designate its analysed sound as the 'target' sound
4) Perform a short period of evolution, aiming to create sounds as similar to the target as possible
5) Present the resulting sounds to the user again, and possibly allow them to repeat the process from step 2 indefinitely

Note that the user is able to guide the process, but that the fourth step aims to converge on a given target sound, meaning techniques from other works focussed on reproducing a target sound can be employed in this stage. In particular, the strategy put forward in [2] was used as a basis for the evolutionary strategies constructed in this project.

If the user desired to converge on a specific sound, previous target sounds could be factored into each new target, but the decision was made in this case to discard the target once each round of evolution is complete. The hope was that this would allow the user to be more exploratory, rather than becoming stuck in any particular niche.

In order for the project to be as accessible as possible, a static browser-based application was designed. This entirely eliminates the need for user dependencies, and avoids the additional complexity brought about by server-side processing, but requires that the software run entirely in the browser using JavaScript (JS). This limits the computational performance of the solution when compared with other solutions (such as [2] and [6]) which ran with more traditional local software implementations. Additionally, previous implementations would be left running for time on the order of hours [2] or even days [6] on state-of-the-art university-owned equipment. In this project's case, any evolution runtime higher than a few minutes on a consumer-grade system would inevitably result in the user losing interest.

These performance drawbacks are less significant, however, when the methods of this project are taken into consideration. Most of these previous approaches were aiming to perfectly recreate a sound, and therefore needed higher performance and runtimes in order to adequately explore the whole problem space, whereas this project's goal is to imperfectly locate similar sounds that the user may enjoy based on a selected favourite. As a result, the design goals of the project mean that it does not need to find an optimal solution - indeed, it is preferable that it finds similar sounds from a range of different topologies, as this will allow the user to more meaningfully explore the problem space and take part in a more creative sound design process.

Finally, TypeScript (TS), a typed extension to JS which allows the construction of more type-safe code, was selected for the main implementation. Although the application is static, TS does require compilation to JS before it can be run in-browser, and the compiled Sass language was also employed for styling. Therefore, Node.js was used to build the application, in conjunction with the lightweight Vite frontend build tool.

3.1 Sound Synthesis

The first problem to solve was how to represent the sound-generating components of a synthesiser in the browser. An ideal method would have the following characteristics, listed in descending order of importance:

1) Abstract enough to be easily generated and mutated programmatically
2) Flexible enough to allow for diverse signal chains such as FM as well as simpler additive or subtractive synthesis
3) High in performance, to allow for rapid evolution
4) Portable, such that resulting patches can be used in real-world sound design and music production workflows
5) Compatible with the flexible and modern Web Audio API

The Faust digital signal processing language was an obvious choice for synthesiser representation, as it instantly satisfies items 2, 4 and 5 in the above list, and boasts a reasonably fast WebAssembly-based compiler, faust2webaudio, which also lends it to item 3.

Finally, in order to satisfy item 1, an abstract module system was constructed, using JS classes. In this system, a patch is defined recursively, with properties and inputs passed into the constructor, and Faust strings generated with a recursive class function that is common to all node types. An example of a TS patch, and its equivalent Faust DSP code, can be found in figures 3 and 4 respectively. Faust topologies are compiled to a single Web Audio AudioWorkletNode by the faust2webaudio JS module.

Topologies are then contained within TypeScript-based 'SynthContext' constructs which provide the functions and data storage necessary to handle the generation, mutation and analysis of a single topology without causing interference between these topologies. Each SynthContext 'owns' a number of Web Audio nodes which remain constant: a GainNode used as a passthrough for topologies to be connected to, a second GainNode used to control whether the context's topology should be heard by the user, and an analysis node connected to the passthrough.

The passthrough node means that the rest of the analysis and output nodes only need to be initialised and connected once. In this system, when topologies need to be swapped, the old one is disconnected and the new one is connected to the passthrough. The rest of the SynthContext's Web Audio node structure can remain in place at all times.

A small collection of possible nodes is made available when generating or evolving topologies, shown in table 1. All evolved parameters, including frequencies, are mapped to the range [0,1] for consistency. Frequencies (parameters

```
const exampleTopology = new AudioOutput([new MathsNode(
    "+",
    new Oscillator(
        "saw",
        new Oscillator(
            "sine",
            new MIDIFreq()
        )
    ),
    new Oscillator(
        "triangle",
        new MathsNode(
            "*",
            new Parameter(0.5),
            new MIDIFreq()
        )
    )
)]);
```

Fig. 3. An example patch, written in the TypeScript class structure described above. This code describes the patch shown in figure 2, and produces the code shown in 4.

in the range [20,20000] - the range of human hearing) are mapped to [0,1] in the MIDIFreq node, and any frequency inputs on oscillators are then mapped from [0,1] back to [20,20000] in the Faust process code.

TABLE 1
The Types of Node Modelled in TypeScript.

Name	Description
MIDIFreq	Note frequency
Parameter	A numeric value, controllable with a UI slider
MathsNode	Performs mathematical operations on numerical values and/or signals
Oscillator	Creates audible waves with one of four basic waveforms
FrequencyModulator	A wrapper node, containing an Oscillator and two Parameters, for FM

Note that the MIDI nodes are simply Parameters and can therefore be controlled manually as well as manipulated using MIDI messages. The FrequencyModulator node is equivalent to an Oscillator being multiplied by one Parameter and added to another, in order to provide parametric control over both modulation strength and frequency offset.

With this system, the program is fully capable of modelling additive and FM synthesis. The presence of an envelope and a filter on the output also gives it some of the features of subtractive synthesis, but these processes fall outside the topologies that are generated by the evolutionary process, and it therefore is incapable of generating truly subtractive topologies.

3.2 Genetic Algorithms

With the synthesis methods in place, the next component to design was the genetic algorithm itself. This was broadly based on the work of Macret and Pasquier in 2014 [2].

```
import("stdfaust.lib");


// DEFINITIONS
midifreq = hslider("freq[unit:Hz]", 440, 20, 20000, 1);
bend = ba.semi2ratio(hslider("pitchBend[midi:pitchwheel]", 0, -2, 2, 0.01));

frequency = vgroup("Frequency", midifreq*bend);

p10 = hslider("[10]parameter10 (CC10)[midi:ctrl 10][scale:linear]", 0.5, 0, 1, 0.001) : si.smoo;


// PROCESS
process = ((os.sawtooth(19980*os.oscsin(19980*(frequency-20)/19980+20)+20)+os.triangle(19980*(p10*(frequency-20)/19980)+20))) : fi.dcblocker : fi.lowpass(1, 20000);
```

Fig. 4. An example patch, written in Faust DSP code. This is the code generated by the structure in figure 3, and also models the patch shown in figure 2.

In keeping with their strategy, a $1 + n$ elitist evolutionary strategy was employed, and Mel-Frequency Cepstral Coefficients were used in the fitness function.

However, as noted in the previous section, numerical genotypes were not employed to represent synthesiser topologies: the abstract recursive node system defined immediately above was used instead. This allowed more control over the generation and mutation of the synthesiser topologies.

During evolution, each new best score is stored.

3.2.1 Generation and Mutation

Both the generation and mutation functions are defined recursively.

The `generate()` function takes as an argument the type of the node that it should return, and randomly generates such a node, deciding the types of any children and then calling itself to generate these. Once it has reached the bottom of the tree, it returns the generated node and the tree is constructed through recursive returns all the way up to the root node.

For some node, a `carriesSound` property is true for any Oscillator, and any MathsNode for which at least one child has `carriesSound=true`. A `graphSize` value on every node stores the number of nodes beneath that node in the tree, including the node itself.

The type of the root node is decided by a wrapper function which chooses either a MathsNode or an Oscillator for the root (as all other node types will only return values rather than audible sound). This function also ensures that the root node has `carriesSound=true`, and `graphSize` $< g_{\max}$ (where $g_{\max}$ is the largest permitted graph size, set to maintain reasonable compilation times). If the root node does not satisfy either of these requirements, it is regenerated.

In a similar fashion, the `mutate()` function takes as an argument the node in the existing topology that it is mutating, and chooses a uniform random number n in the range [0,1]. If n is less than the mutation probability p_m it will 'mutate' that node. If n is also less than some replacement probability p_r ($\leq p_m$) that mutation will take the form of a complete replacement. Otherwise (for $p_r \leq n < p_m$), it will mutate that node without changing its type. These mutations are shown in table 2.

TABLE 2
The Mutations Available to Each Node Type

Node Type	Mutated version
MIDIFreq	A new MathsNode, multiplying a new Parameter by MIDIFreq (intended to produce harmonics through additive synthesis)
Parameter	A new Parameter node, with its value randomised
MathsNode	A new MathsNode, with the same children but a randomised operation
Oscillator	A new Oscillator, with the same children but a randomised waveform
FrequencyModulator	New MathsNodes, Parameters and an Oscillator that together recreate that FrequencyModulator's functionality but can be mutated more easily in future

3.2.2 Fitness

As mentioned previously, the same Mel-Frequency Cepstral Coefficients (MFCC) method as [2] was used for fitness. Specifically, the Meyda.js [20] audio analysis library was applied, as it is compatible with the same Web Audio API that is employed by faust2webaudio. The Euclidian distance between the coefficients returned by the MFCC process can be used as a measure of sonic distance between two sounds, as perceived by a human.

While much recent work has recommended and used MFCC for tone matching [3], Masuda and Saito reported in 2021 that it does less well at recognising pitch, resulting in sounds that are similar in timbre but are not accurate in pitch [8]. This is a bigger problem for those trying to match an externally-sampled target sound, but should not affect this work. The presence of the MIDIFreq node should mean that the genetic process creates sounds that are either in tune by default, or completely out of tune in the case that a parameter controls an oscillator's pitch.

In this research, we aim to characterise static timbres rather than temporal evolution, so we set the pitch of any MIDIFreq nodes in the topology, enable the gate and take a single MFCC on a 1024-sample window of the resulting sound (the window length being taken from [2]). This is repeated for three notes evenly spaced across the frequency spectrum - C2, A4 and F7 - yielding three sets of 13 coeffi-

cients for analysis. The average Euclidian distance between these three sets of coefficients for the candidate sound and the target sound is then computed, and finally the fitness is calculated from the resulting distance:

$$d_{note}(a,b) = \sqrt{\sum_{i=1}^{N_c}(a_j - b_j)^2}$$

$$d_{overall}(t,c) = \frac{\sum_{i\in[C2,A4,F7]} d_{note}(t_i,c_i)}{3}$$

$$fitness(c) = \frac{1}{1 + d_{overall}(t,c)}$$

In these calculations, t is the set of three note readings for the target sound, c is the corresponding set of readings for the candidate sound, and a_i is the ith member of the set of readings a.

3.3 Performance

The biggest potential issue with this project is its computational performance. JavaScript is not known for its performance [21], and the processes being taken on here are non-trivial, particularly in the case of compilation. There has also been a significant recent increase in the number of cores available to the average consumer with new devices [22]. Therefore, the intention was to try and maximise the parallelism of the system on these time-consuming tasks, so much discussion in this section will be devoted to parallelisation.

During the development of this project, asynchronous processing techniques such as JavaScript's Promise and Async/Await constructs were used regularly with the aim of parallelising computationally intensive tasks such as synthesiser compilation. While JavaScript in theory does not run in parallel, instead using context-switching to perform asynchronous processing [23], the intention was to ensure that time-consuming tasks would not block the main thread. It also would make retrofitting full parallelisation easier if that was to take place.

During evolution, CPU usages of greater than one core by a single tab have been observed, so it may be that the asynchronicity built into the system has enabled the Chromium engine to parallelise the intensive parts of the evolutionary process after all. This may also be due to background AudioWorklet processing, however, as Audio Worklets are essentially lightweight Web Workers and can therefore be parallelised [24].

It is also worth noting that significantly lesser performance was experienced when the browser was not in the foreground: this is likely due to power optimisations that limit JavaScript processing when the tab is not the user's focus.

3.3.1 Faust Compilation

The most significant time sink during evolution was generally found to be the compilation of Faust patches. The magnitude of its impact is discussed in more detail in section 4.2, but it was generally observed during development to take on the order of seconds for larger patches.

The option of using Web Workers for compilation was evaluated, but discarded as these workers currently do not support Web Audio [25]. The faust2webaudio module directly returns a Web Audio node, and takes the current Web Audio context as a parameter, so this made Web Workers impossible for this task.

One unfortunate downside discovered with the faust2webaudio compiler during implementation was that it appears to retain references to compiled nodes internally. The result of this is that, even after disconnecting them from the output, calling their `destroy()` function and deleting all reference to the Web Audio node, the nodes are still not garbage collected by the JavaScript engine. This was confirmed using a basic test case where clicking a button generated a new Faust node as well as a new Web Audio GainNode, and then deleted the reference to these items. The GainNodes were periodically removed, whereas the FaustAudioWorkletNodes remained in place. They do not seem to occupy too large a footprint in terms of system resources, but they still accumulate over larger evolution cycles, and therefore may cause the program to become unstable if used for an extended period. The test can be found at `gc-test.html` in the source code repository.

3.3.2 Analysis

In its current state, the implementation processes the MFCC analysis in real-time. While Web Audio does have support for offline (that is, faster-than-realtime) processing, this was unfortunately not prioritised for this project, as the main performance impact comes from the Faust compilation. It remains an interesting avenue for future exploration in this area, though realtime analysis does have one advantage: it allows the user to hear the evolution happening - a useful insight into the workings of the evolution process. A switch is present in the user interface to toggle this option.

The MFCC analysis is also performed sequentially. Even though there is the capacity for concurrent analysis thanks to the disconnection of SynthContexts' graphs, testing with identical sinewave patches (which should have returned fitnesses of at least 0.99) showed that parallelising analysis had a negative impact on the accuracy of the MFCC, causing it to be much more variable and reaching as low as 0.6. It is believed that the reason for this is the aforementioned non-parallelism of JavaScript's asynchronous processing.

3.4 Interface

A screenshot of the application can be seen in figure 5. Note that, as well as control over the value of every Parameter node in each of the four visible patches, the user is also presented with a number of simple controls to shape the produced sound. These include an amplitude envelope (which can also be triggered with an onscreen button), a low-pass resonant filter and gain control.

The user interfaces are constructed using the faust-ui module, which provides functionality to generate an HTML interface for a given Faust patch in an iframe. Messages are sent in both directions between the patch and the interface, allowing all changes made to be displayed on the interface.

Faust also supports MIDI by default, so the WEBMIDI.js module has been used to integrate MIDI messaging with

Fig. 5. Screenshot of the application in use.

the onscreen patches. This allows the user to audition the provided patches in realtime, playing notes and even adjusting parameter values with their physical hardware. The gate, gain and frequency controls can be controlled through MIDI key on/off messages, MIDI note velocity and MIDI note frequency respectively, and each patch features 4-voice polyphony to allow the user to play the synthesiser interactively with their physical MIDI hardware.

For each patch, there are also buttons to view the Faust process code of that patch (minus the boilerplate Faust syntax, definitions and output components like filter and envelope), and view the patch's full code (including the aforementioned). The first button can be used to more easily understand the interactions taking place within a patch, and the second can be used to export the patch for use elsewhere. The overlay displaying the code also includes buttons to copy it to the clipboard or save it to disk as a `.dsp` file.

Also considered was the option of displaying the patch as a graph (a function that is supported by faust2webaudio) but the results are quite verbose, dominated by the 'outer' user control elements like the envelope and filter, and quite large. They may be useful for a Faust developer, and indeed are shown in the Faust IDE, but would clutter the interface for the casual user.

The MIDI and evolution selection buttons respectively choose which patch should be controlled by incoming MIDI messages and select the target sound for a round of evolution. The 'Stop note(s)' button cancels any currently playing notes on that patch - this is useful in the case of, eg, switching to a different patch for MIDI whilst holding notes down on the MIDI controller.

There are also bar charts at the bottom which display the current MFCC values for their corresponding patches - this serves a dual purpose of allowing MFCC-based comparison of patches as well as providing the user with a visual cue to see which patch is currently making sound.

Once a user has chosen a favourite patch for evolution, they select it using the button corresponding to that patch and click the large 'EVOLVE' button in the top panel. The application will then allow evolution to take place in the background, leaving the existing UI elements accessible for further user experimentation. During evolution, each topology that generates a new best fitness is stored in an object alongside its fitness. Once evolution is completed, this object is returned and the highest-scoring patches are placed back into the interface for user experimentation.

3.5 Summary

To summarise, the application has been written in TypeScript, using Node.js to build the source code into a single static web-application with no server component. The faust2webaudio library enables high-performance synthesiser compilation within the browser using WebAssembly, and the Web Audio API is employed to facilitate both audio output and analysis.

The program involves two main intensive processes: compilation and MFCC analysis. Wherever possible, intensive but parallelisable sections of the program have been made asynchronous to ensure that the main thread is not blocked, to allow browsers to optimise performance, and to open the door to true parallelisation in the future.

4 Results & Evaluation

Before testing the quality of the solution, various parameters first had to be optimised. This process will be covered in section 4.3, before experimental results are discussed in section 4.4.

Tests were run in Windows 10 21H2 Microsoft Edge version 100 on a Dell XPS 15 9550 laptop computer, with a quad-core Intel Core i5-6300HQ (released in 2015) and 16GB of memory at 2133 MHz. The laptop was slightly elevated in order to ensure adequate ventilation and minimise heat build-up over the course of multiple tests. The hope was that this machine's processing performance would be roughly equivalent to a typical device used by the average consumer.

4.1 Fitness Testing

In order to verify its accuracy, the MFCC fitness measure described in the previous section was tested against human perception of sound similarity. Two sounds were randomly generated, and played to the user in the C2-A4-F7 method described above (albeit at a slower 8192-sample or 0.17-second note length). The user then ranked the similarity of the sounds on a scale of 1-10, and this was stored alongside the MFCC measurement of similarity between the two sounds. The second sound then became the first sound, and a new sound was generated to be played second.

This process was then repeated over 300 times, across several test sessions, and the program defining it can be found in `mfcc-test.ts` in the source repository. The results can be seen in figure 6.

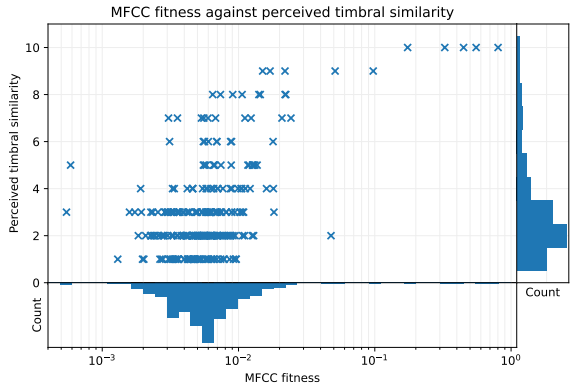


Fig. 6. MFCC fitness plotted against human-perceived similarity between randomly-generated pairs of sounds. The 'count' histograms show how many datapoints exist in bins spanning their respective axes.

As shown in the figure, a clear positive correlation between the MFCC fitness and the perceived similarity was observed. While the correlation was very much non-linear, this is of little consequence in the use case of this project as the fitness measure is only used to compare absolute values. A value indicating higher fitness only needs to be larger than other, less fit pairings - the ratio between them does not matter.

During testing, it was observed that the magnitudes of MFCC values appear to scale with the volume of the sound they represent. Therefore, it is possible that sounds with similar timbre but different volumes could be scored highly by the human but not by MFCC, and it is believed that this may explain the instances where the human assigned a score of 5-8 but MFCC did not score as highly. Normalising the MFCC results may help resolve this problem, as the relative values appear to remain similar, but this was not implemented in the current system. The downside of this approach would be that the volume effectively is removed from the fitness function, which is not desirable. The fitness function could be adjusted to slightly penalise differences in volume, but there was not time to implement this here so the normalisation was left for further exploration.

4.2 Compiler Performance

As mentioned in the previous section, Faust compilation was the main time sink during evolution; see figure 7 for a plot showing the time taken to compile synthesisers with a range of sizes. These measurements were taken with compilation taking place consecutively, and graph size was capped at 100 nodes. The 11 'user interface' nodes (filter, envelope etc) are also compiled but not included in the count. Note that the lower values are in the region of 140ms, whereas the highest time taken was 174 seconds for a patch with 54 nodes.

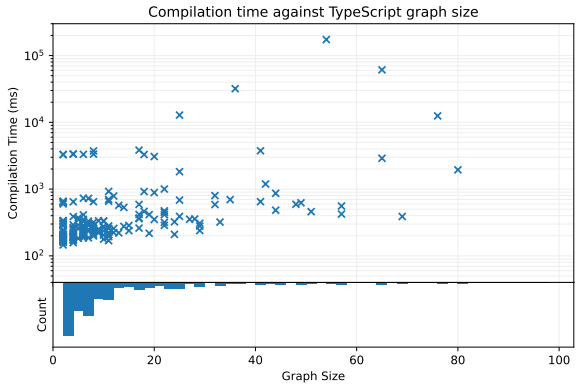


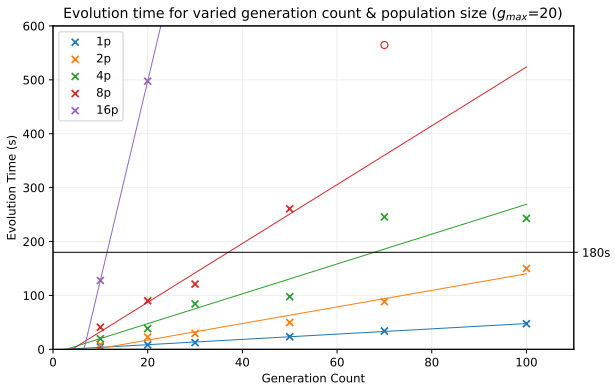
Fig. 7. Compilation times in faust2webaudio. The 'count' histogram shows how many datapoints exist in bins spanning the x-axis.

These results are still very acceptable for an in-browser compilation, thanks in part to the Web Assembly system powering the faust2webaudio module, and setting a maximum topology size of 30 nodes ensures that time taken remains under 3s for the vast majority of compilations.

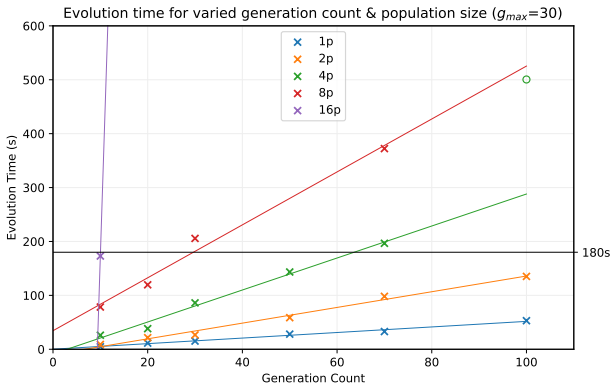
4.3 Time Optimisation

The first parameters to be optimised are those that affect the time taken to evolve. In all of the tests in this subsection, the mutation chance p_m was 0.3 and the replacement chance p_r was 0.1 (though these should not affect time results). The patches being chosen for evolution were those with process code of 100-200 characters - this is a medium complexity, as process codes can range from around 20 characters to nearly 400 with $g_{\max} <= 30$.

The target time is three minutes (180 seconds): any time longer than this would likely cause the user to lose interest. Evolution time was tested whilst varying population size



(a) Evolution time for $g_{\max} = 20$.



(b) Evolution time for $g_{\max} = 30$.

Fig. 8. Evolution time, plotted against generation count for a range of population sizes, across two different values of $g_{\max}$. Points that are believed to be anomalous are plotted with open circles, and not taken into account when calculating trendlines.

n_p, generation count n_g and the maximum graph size $g_{\max}$. Results are presented in figure 8.

Generally speaking, the tensions here are quality of results against time: higher population size and higher number of generations would increase runtimes but allow more mutations to occur, allowing the evolutionary process to explore the search space more.

Similarly, higher $g_{\max}$ would allow more complex synthesis graphs to exist and would theoretically increase average compilation time. Surprisingly, however, setting $g_{\max}$ to 30 appears to produce *more* consistent timings (for smaller timespans at least) than when $g_{\max} = 20$. From both these tests and experience using the system, it does tend to scale less reliably at higher generation counts and population sizes, however. Not visible on this graph are elapsed times of over 45 minutes for $n_p = 8, n_g = 100$ and $n_p = 16, n_g = 20$, and 10 minutes for $n_p = 8, n_g = 50$. The only anomalous results for $g_{\max} = 20$ were the visible 565-second time for $n_p = 8, n_g = 70$ and a time of 808 seconds for $n_p = 8, n_g = 100$.

Referring back to 7, it can be seen that graphs with size between 20 and 30 could take time up to the order of 20 seconds to compile in rare cases, whereas graphs under 20 are consistently under 5 seconds for maximum compilation time. Having larger populations and/or higher generation counts increases the probability that one or more of these rare cases occurs, which would explain the inconsistent time taken for these larger values.

From this experimentation, values of $g_{\max} = 30, n_p = 4, n_g = 50$ were chosen. As mentioned previously, the algorithm used here is loosely based upon the work of Macret and Pasquier in 2014 [2], which used values of $n_p = 4, n_g = 5000$. The difference in n_g can be explained by the difference in intended runtimes: Macret and Pasquier intended for their program to take on the order of hours and find perfect matches, whereas this algorithm should run on the order of minutes and find similar sounds.

They were then tested to see how they performed on two different systems: the aforementioned laptop system, and a desktop computer running Windows 11 21H2, Microsoft Edge version 100, an AMD Ryzen 7 2700X (released in 2018) and 16GB of memory at 2400 MHz. This machine has a more modern processor that runs at a higher clockspeed, so should perform better. The results are depicted in figure 9 and appear to be generally as expected. Runtimes rarely exceed the 180s target on either device.

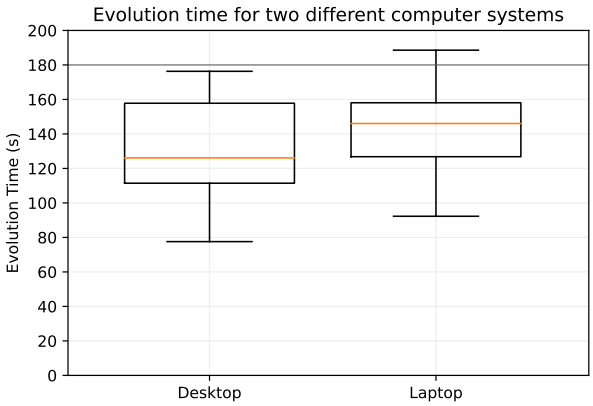


Fig. 9. Evolution time with $g_{\max} = 30, n_p = 4, n_g = 50$ on two different computer systems. Results taken over 8 repeats.

4.4 Convergence Optimisation

With n_p, n_g and $g_{\max}$ optimised, only the probabilities p_m and p_r remained to be tuned.

While the ratio of p_m/p_r determines what form mutations take, their magnitude decides how often these mutations occur in the first place and therefore will have a larger impact. Therefore, the priority was to optimise the magnitude first. The ratio $p_m/p_r \approx 3$ was chosen, meaning one in three mutations would replace the node instead of adjusting its properties. This ratio was employed for the remainder of this project, to keep the remaining settings consistent while the magnitude was varied.

The magnitude of p_m was varied across a wide range, and synthesiser patches of varying complexities were also trialled. The four categories of patch complexity are portrayed in table 3.

It is worth noting that patches with one oscillator were not considered complex enough to be 'small'. These patches

TABLE 3
Synthesiser Patch Complexity

Complexity	Criteria
Trivial	A single basic Oscillator, with its frequency controlled by MIDI
Small	Not trivial; process code length (in characters) ≤ 100
Medium	$100 <$ process code length ≤ 200
Large	$200 <$ process code length ≤ 400

are sonically just trivial patches being operated on by mathematical operations and parameters, so sound timbrally identical to the true trivial patches. A small patch was generally considered to contain at least two oscillators, then, combined with additive synthesis, FM synthesis or amplitude multiplication.

The results can be seen in figure 10. Twenty combinations of parameter value and synthesiser complexity were tested, and each combination was repeated four times. For each repeat, the minimum fitness, average fitness and maximum fitness in each generation was taken. These minimums, maximums and averages were then averaged across the four repeats in order to produce more consistent plots.

The global maximum fitness, over all 50 generations, was also measured for each repeat, yielding four 'best' values for each combination. These represent the fitness of the 'best' sound, which would have been presented to the user at the end of the process. The minimum, maximum and average of these values is depicted on each plot as the horizontal bands which span the widths of the plots. A tighter range reflects a more consistent result, and higher ranges reflect better results.

The ideal outcome, therefore, is a convergence plot that travels upwards over the course of the generations, with a tight band of global best values towards the top of the range.

The synthesiser complexity will vary with the user's selection, so the goal of this experiment is to choose a probability setting (column) which yields the best results for the widest range of complexities. It is worth noting that, towards the right hand side, the evolution is more chaotic: for higher mutation probabilities, it is closer to randomising the members of each new population. This essentially samples random points in the parameter space, and does not yield any opportunity to converge. For lower probabilities (on the left hand side), the opposite is true.

Looking at figure 10, a number of interesting trends can be seen. Firstly, as the probability of mutation grows, the graphs generally become less consistent and more erratic. This is as expected, and reflects the aforementioned effect where the search 'bounces' all over the parameter space. Similarly, for lower probability values, the search appears to generally remain in the space that it starts in.

Both of these behaviours are problematic: in the former case, the search will not remaining in promising niches for longer than a few rounds. In the latter, the algorithm will not manage to explore the space much, instead attempting to converge instantly.

The trivial patches, unsurprisingly, are responsible for all five of the best global results. This is likely because these very simple graphs are often generated randomly by the program. It only takes one Oscillator(MIDIFreq) graph to be generated before the evolutionary algorithm has all but solved the problem: from this point on, the only parameter requiring mutation is the waveform of the Oscillator. These trivial patches, being the most basic forms of sound synthesis, are also not particularly useful or interesting. For these reasons, they will not be prioritised in the optimisation process.

Of the five columns, the only one that appears to show a consistent controlled convergence is the second column, for which $p_m = 0.1, p_r = 0.03$. It performed extremely well for the medium patches in particular, but does not appear to converge as well for complex patches. This is no surprise given that these patches are always going to be far more timbrally complex than smaller examples. These more nuanced sounds will take more optimisation before similar sounds are generated, so it may be that more generations (and therefore more time) are needed for these sounds to converge. It also does not appear to perform particularly well for small patches, but these patches did not converge meaningfully with any combination of mutation probabilities. Indeed, it appeared to converge most in the second column, albeit rapidly and unfortunately without much success.

The $p_m = 0.1, p_r = 0.03$ combination was then evaluated for a second time, with more repeats to average over. It was also evaluated for longer periods, with the hope being to observe the evolution over longer periods than those possible in this project's current format.

With this combination of mutation probabilities, a further round of testing was conducted. 8 repeats were made for each patch size, and testing continued for 250 generations, rather than the previous 50, in an attempt to identify an upward trend in the more complex category of synthesiser. The results of this test are shown in figure 11.

Figure 11 appears to show that the parameters $p_m = 0.1, p_r = 0.03$ work well: all four graph sizes rapidly converge in the early stages of the algorithm, and show evidence of continuing to explore as the generations progress. In all plots, periods of improved and worsened performance are clearly visible, indicating that the algorithm continues to explore the space even after stabilising on a local maximum. Since strong performers are saved globally over all generations (rather than taking the best individual in the final generation), this is a useful trait: the algorithm has time to perfect strong candidates, but also will not linger on those candidates indefinitely, missing higher maxima elsewhere.

Unfortunately, not all graph sizes seem to reach a global maximum within 50 generations. In particular, the small and medium patches continue to improve past the 75-generation mark. This is in fact an encouraging result for the algorithm, as it shows that it is capable of gradual convergence, it may well be that parameters could be tuned slightly more to achieve these gains before 50 generations have passed. However, with the current implementation, it does not reach the maximum within the allotted 50 generation limit imposed due to time constraints, so perhaps this is an option that could be pursued in further work.

Examining fitnesses in terms of human perception

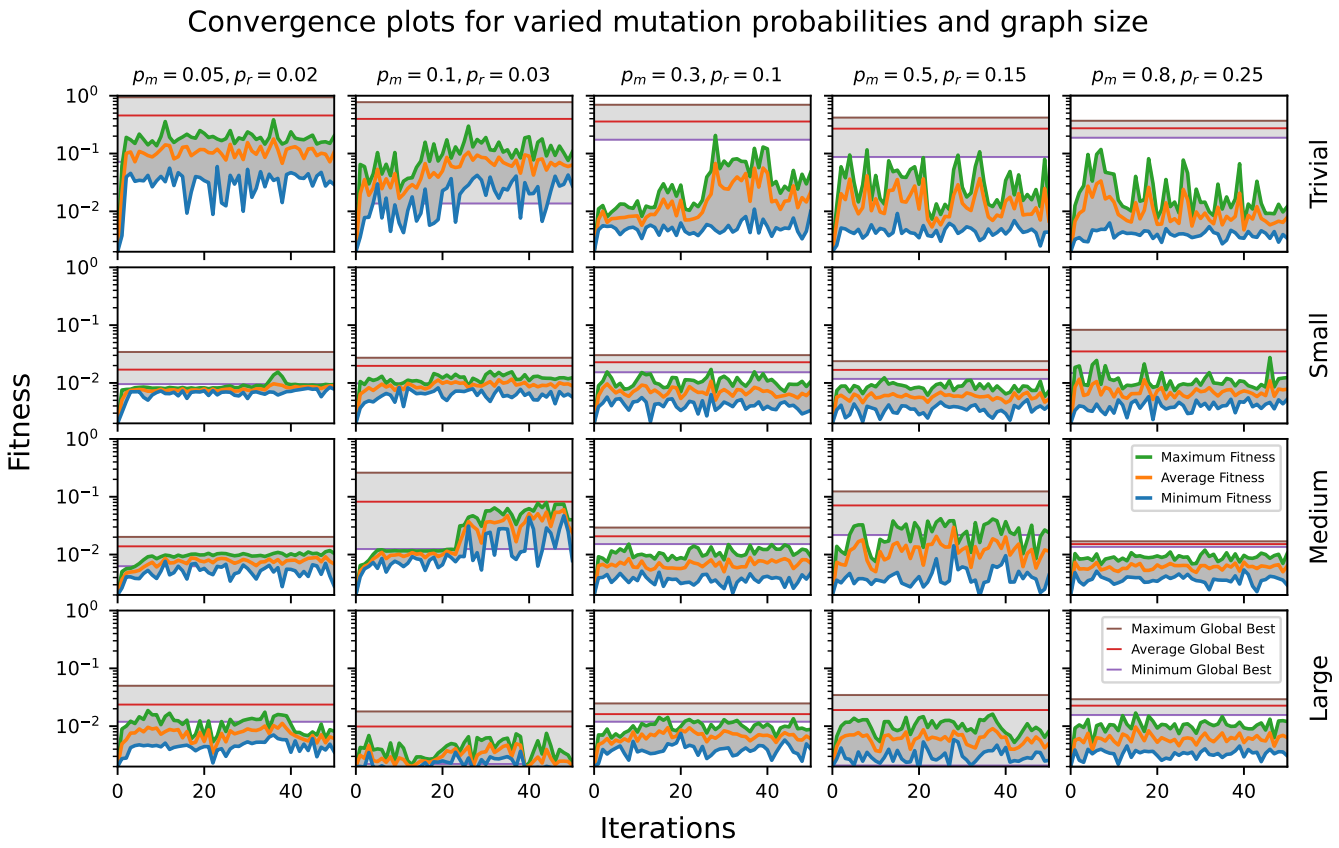


Fig. 10. Convergence plots for various parameter values and graph sizes. Each individual plot is averaged over four repeats. Each column represents a value of p_m, p_r as labelled at the top; each row represents a synthesiser patch complexity as labelled on the right. Horizontal bands depict the best fitness over all 50 generations, with the minimum, average and maximum taken across the four repeats.

(based on figure 6; henceforth 'HP'), it can be seen that the minimum global best fitness across all 32 runs shown here is 0.007, for large patches. This reflects a HP score of between 2 and 7. However, the average global best for all graph sizes is at least 0.03, or 7-9HP - a very strong result.

Finally, it should be noted that, while these probabilities clearly perform well, there is significant room for further optimisation. There may be even better-performing values within the range $0.05 < p_m < 0.3$, and the ratio p_m/p_r also remains untuned. While the chosen value of approximately 3 for this ratio is grounded in reason (see the second paragraph in 4.4 for explanation), it is likely not perfect. Unfortunately, further tuning falls outside the scope of this research, but it is a promising avenue for any future work on this particular system.

4.5 Convergence Results

With all values decided, the final task was to confirm the expected outcome of evolution in the real-world test case. With the final values $p_m = 0.1, p_r = 0.03, n_p = 4, n_g = 50, g_{\max} = 30$, 16 tests were run with each patch complexity. Box and whisker plots representing the global best fitnesses for each complexity are visible in figure 12.

Also shown in the figure is the plot from figure 6, indicating how the human perception of similarity correlates with the MFCC fitness value. Studying the figure indicates that all inter-quartile-ranges land above the region commonly scored below 4/10 for similarity by users. This is the area in which the vast majority of random patch pairings fall, so the evolutionary algorithm is clearly making significant progress away from random chance and towards recreating the original sound.

It is interesting to note that there is no obvious trend across topology complexities. The medium patches have consistently performed the best of the three non-trivial patch complexities with this probability choice through multiple tests. Surprisingly, the highly complex timbres of large patches also have performed better than the small patches, even though these small patches typically only comprise a single FM or additive sound. Large patches often contain complex combinations of many FM or additive patches, so theoretically contain more harmonic complexity. However, many of these details may not be contributing much to the overall timbre - it may well be that the MFCC fitness is simply optimising for a certain kind of timbral richness on these larger patches, rather than the specific tones being produced by smaller topologies.

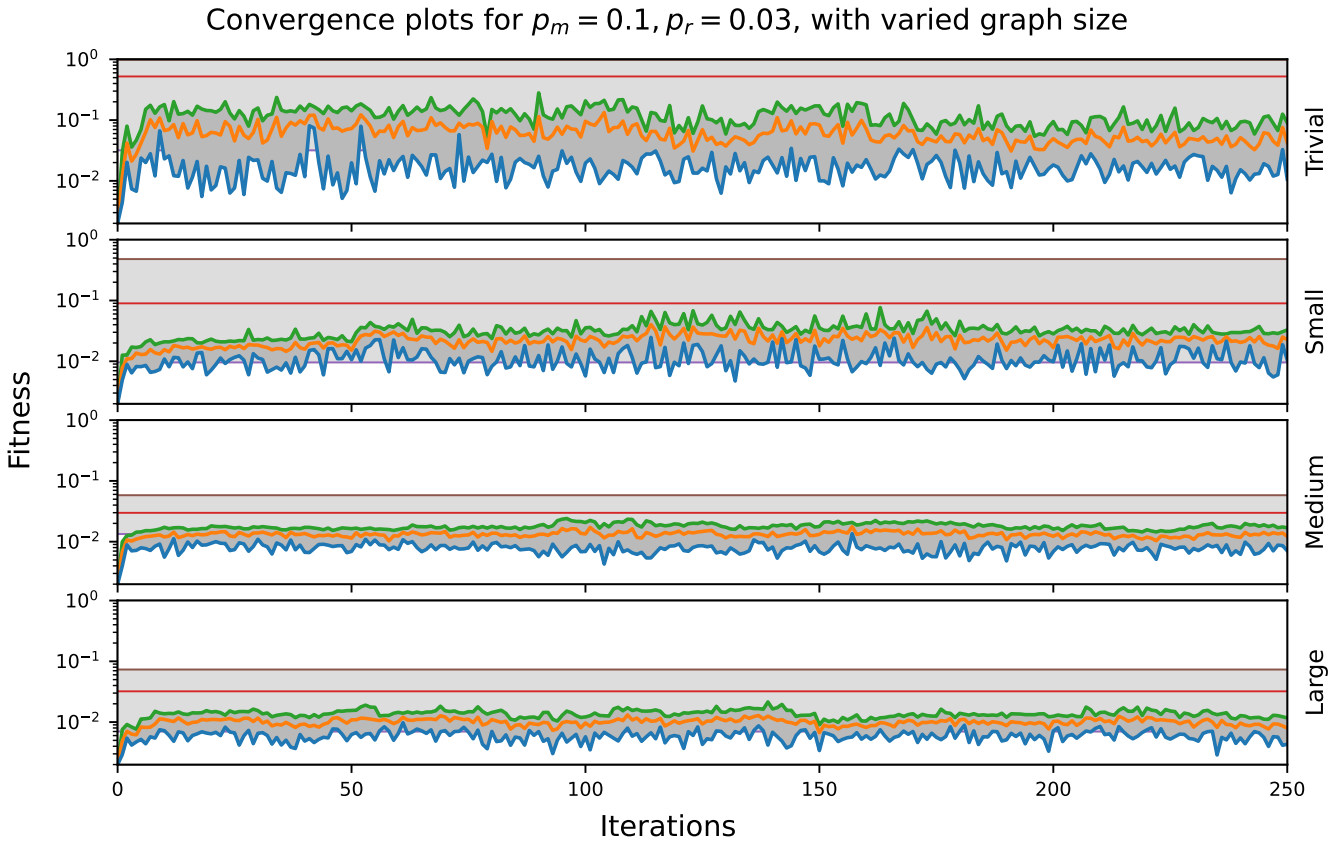


Fig. 11. Convergence plots for the chosen mutation probabilities $p_m = 0.1, p_r = 0.03$, shown for four different graph sizes. Each graph size was repeated 8 times and the results were averaged in the y-direction. The format otherwise remains the same as in figure 10.

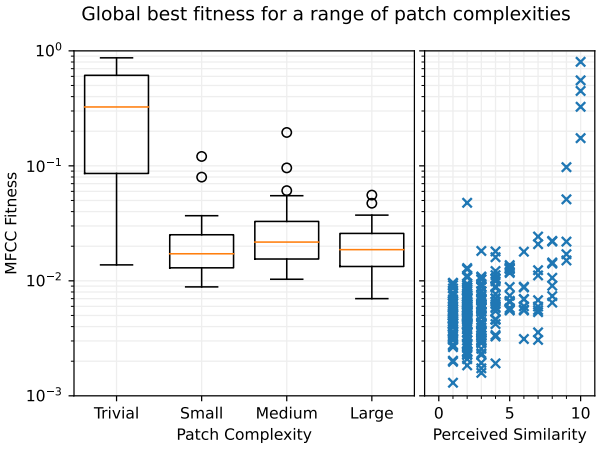


Fig. 12. Box plots indicating the best fitnesses achieved for a range of patch complexities with the final chosen settings. The plot of human-perceived similarity (from figure 6) is shown to the right for reference. Each box plot has a sample size of 16 repeats.

5 Conclusion

This paper has presented a novel approach to generating modular synthesiser patches through interactive evolutionary computation. This approach has been shown to reliably generate similar-sounding patches for synthesiser patches of arbitrary complexity in less than three minutes, with no local dependencies. These patches can then be easily downloaded in the Faust DSP format and from there compiled to a wide range of targets - including VST plugins, Pure Data patches and VCV Rack modules. As a result, any user with a computer and modern browser is able to explore the sound space of a synthesiser with no specific subject knowledge, and more experienced users are also able to employ it as a tool capable of integrating into a range of modern music production and sound design workflows.

During development, testing and evaluation, typically with the option to hear the evolution taking place enabled, it was observed that the algorithm does indeed gradually move towards the desired sound as one would hope. This is particularly obvious for simpler patches which are more recognisable, but could similarly be noticed with more complex patches also. Playing the patches using a MIDI keyboard also proved highly enjoyable: the ability to play a droning chord in one context and play above it in another was discovered early in development, and it is also possible to continue playing on the foreground contexts while evolution occurs in the background, meaning the user is able to entertain themselves while the evolutionary process occurs. Finally, the sounds generated both randomly and through mutation are stimulating and often land in the attractive area between basic patches and harsh noise, creating a rewarding process.

The author of this research is therefore of the opinion that the original aim stated at the beginning of this paper has been met. However, the end product is not without

compromise, and a number of attractive future avenues in this area will be enumerated next.

5.1 Future Work

Firstly, there is scope to develop the existing system further, in order to bring greater flexibility and usability.

A range of compromises has been made in order to keep the scope of this project reasonable, but future research could examine the possibility of working with more advanced synthesiser models by, for example, adding filters and noise generators to facilitate subtractive synthesis. Components such as envelopes and LFOs have also been ignored in this project; these could allow the user to generate evolving sounds as well as the static timbres generated here. Analysing patches using an offline Web Audio context would allow these evolving patches to be evaluated faster than real-time, alleviating the problem of needing a longer analysis window to capture evolving sounds. The problem mentioned in 3.3.1 of old WorkletNodes being retained in memory also remains, and may cause performance problems or even crashing when the program is used for extended periods of time.

The evolutionary algorithm also stands to be optimised further: while the values of p_m and p_r chosen here perform well, they have not been finely tuned and are therefore not likely to be optimal. Synthesiser topologies could also be categorised further in this analysis, in order to more accurately examine the effect of these parameters on different kinds of patch, and more repeats could be used to make more precise conclusions. The characteristics of evolved patches compared to the original patches could also be examined further. Finally, a useful analytical technique used in previous papers ([2], [4]) would be to use spectrograms as well as fitness plots to quantify sound similarity. Spectrograms covering a full evolutionary session could be a powerful method of tracking the sonic evolution of the program.

Remaining in the area of evolution, a number of other specific changes could be made: firstly, adjusting parameters does not require the topology to be recompiled, so the option of tuning the parameters before creating a new generation of topologies may be worth exploring. Also, as mentioned in 4.1, the MFCC fitness could be normalised to lessen the impact of differing sound volumes on the fitness function.

A useful evolutionary technique used elsewhere [26] is to start with high mutation probabilities and reduce them as generations progress, to allow wide initial explorations that rapidly converge in order to find a local maximum. This could be combined with options such as reintroducing the global best as an 'immigrant' to the population later in the evolutionary cycle to be optimised further if necessary.

Moving on from evolution, there are a number of interface improvements that could be made to give the user even more flexibility and feedback. For instance, the interface currently only supports large landscape screens, and has not been tested on mobile devices. A responsive interface would make the tool even more accessible, though the performance considerations of running evolution on a mobile device would need to be evaluated.

Another issue not mentioned elsewhere is that, upon commencing evolution of a target sound, the topology of the chosen synthesiser is copied, and parameter values are not retained in this process. The result of this is that, if a user adjusts the parameters on a patch to their liking, these adjustments will not be included in the target sound measurement. This is currently not a trivial fix due to architectural issues such as the parameter numbering system mentioned below.

There is also a range of additional feedback that could be provided to the user: for instance, they currently have no visualisation of the sound being produced other than the MFCC bar charts at the bottom. A spectrograph and/or oscilloscope visualisation showing the sound being produced by the page could be added to the top toolbar easily, and would give the user some visual insight into the properties of the sounds they are hearing.

The user could also be given the option to decide the parameters of the algorithm, such as how long to evolve patches for. Parameters also currently have their indices set on a global scope rather than, say, on a per-patch or per-SynthContext basis. This proved to be a difficult problem to solve due to choices made early in the design process, but could be fixed with some adjustments to the synthesiser node constructs. This would allow the user to control arbitrary properties over MIDI CC, as currently only the first 127 parameters are CC-enabled before the range of possible MIDI CC controls is exhausted. It would also stop unreasonably high parameter indices being set in exported patches. Finally, the option to import a sound into the system as a Faust patch or a TypeScript structure could be added, to enable longer-term sessions of evolution without the need to keep a browser tab open.

6 Acknowledgements

Firstly, I would like to express my considerable gratitude to Professor Steven Bradley, my project supervisor, whose consistent support, interest, patience and advice were absolutely invaluable in the success of this project.

Secondly, I would like to thank Stéphane Letz and Fr0stbyteR of Grame (France's National Centre for Musical Creation) for creating and maintaining the faust2webaudio module and online Faust IDE, both of which were huge helps in developing the synthesis components of the final product. Similarly, Macret and Pasquier's 2014 paper on MT-CGP served as a fantastic basis upon which to build the evolutionary components of my implementation.

Finally, thanks to my friends for keeping me sane and nodding appreciatively whenever I pointed at another figure that I'd spent several hours generating and/or agonising over.

REFERENCES

[1] P. Manning, *Electronic & Computer Music*, 2nd ed. Oxford: Oxford University Press, 1993. [Online]. Available: https://global.oup.com/academic/product/electronic-and-computer-music-9780199746392

[2] M. Macret and P. Pasquier, "Automatic design of sound synthesizers as pure data patches using coevolutionary mixed-typed cartesian genetic programming," in *GECCO 2014 - Proceedings of the 2014 Genetic and Evolutionary Computation Conference*, 2014, pp. 309–316. [Online]. Available: https://dl.acm.org/doi/10.1145/2576768.2598303

[3] M. Yee-King and M. Roth, "A comparison of parametric optimisation techniques for musical instrument tone matching," *130th Audio Engineering Society Convention 2011*, vol. 2, pp. 972–979, 2011. [Online]. Available: https://www.aes.org/e-lib/browse.cfm?elib=15885

[4] J. Shier, G. Tzanetakis, and K. McNally, "SpiegeLib: An automatic synthesizer programming library," in *148th Audio Engineering Society International Convention*, 2020. [Online]. Available: https://www.aes.org/e-lib/browse.cfm?elib=20794

[5] A. Horner, J. Beauchamp, and L. Haken, "Machine tongues XVI. Genetic algorithms and their application to FM matching synthesis," *Computer Music Journal*, vol. 17, no. 4, pp. 17–29, 1993. [Online]. Available: https://www.jstor.org/stable/3680541

[6] R. A. Garcia, "Automatic Generation of Sound Synthesis Techniques," Ph.D. dissertation, Massachusetts Institute of Technology, 2001. [Online]. Available: https://dspace.mit.edu/handle/1721.1/61542

[7] M. Chinen and N. Osaka, "Genesynth: Noise band-based genetic algorithm analysis/synthesis framework," in *International Computer Music Conference 2007*, Copenhagen, 2007. [Online]. Available: https://michaelchinen.com/GenesynthICMCChinen.pdf

[8] N. Masuda and D. Saito, "Quality diversity for synthesizer sound matching," in *The 24th International Conference on Digital Audio Effects (DAFx20in21)*, Vienna, Austria, 2021, pp. 300–307. [Online]. Available: https://www.dafx.de/paper-archive/2021/proceedings/papers/DAFx20in21_paper_46.pdf

[9] T. Takala, J. Hahn, L. Gritz, J. Geigel, and J. W. Lee, "Using Physically-Based Models and Genetic Algorithms for Functional Composition of Sound Signals, Synchronized to Animated Motion," in *Proc. Int. Computer Music Conf. (ICMC-93)*, no. September, 1993, pp. 180–184. [Online]. Available: https://apps.dtic.mil/sti/citations/ADA456431

[10] C. G. Johnson, "Exploring the sound-space of synthesis algorithms using interactive genetic algorithms," *Interface*, 1999. [Online]. Available: https://kar.kent.ac.uk/21844/

[11] B. Jónsson, A. K. Hoover, and S. Risi, "Interactively evolving compositional sound synthesis networks," in *GECCO 2015 - Proceedings of the 2015 Genetic and Evolutionary Computation Conference*, 2015, pp. 321–328. [Online]. Available: https://dl.acm.org/doi/10.1145/2739480.2754796

[12] M. J. Yee-King, "The Use of Interactive Genetic Algorithms in Sound Design : A Comparison Study," *ACM Computers in Entertainment*, vol. 14, no. 3, pp. 1–14, 12 2016. [Online]. Available: https://research.gold.ac.uk/id/eprint/20164/

[13] H. Takagi, "Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation," *Proceedings of the IEEE*, vol. 89, no. 9, pp. 1275–1296, 2001. [Online]. Available: https://ieeexplore.ieee.org/document/949485

[14] O. Barkan, D. Tsiris, O. Katz, and N. Koenigstein, "InverSynth: Deep Estimation of Synthesizer Parameter Configurations from Audio Signals," *IEEE/ACM Transactions on Audio Speech and Language Processing*, vol. 27, no. 12, pp. 2385–2396, 12 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8854832

[15] K. Tatar, M. Macret, and P. Pasquier, "Automatic Synthesizer Preset Generation with PresetGen," *Journal of New Music Research*, vol. 45, no. 2, pp. 124–144, 4 2016. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/09298215.2016.1175481

[16] K. O. Stanley, "Compositional pattern producing networks: A novel abstraction of development," *Genetic Programming and Evolvable Machines*, vol. 8, no. 2, pp. 131–162, 2007. [Online]. Available: https://link.springer.com/article/10.1007/s10710-007-9028-8

[17] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, apr 2002. [Online]. Available: https://ieeexplore.ieee.org/document/996017

[18] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust," pp. 623–632, sep 2004. [Online]. Available: https://dl.acm.org/doi/10.1007/s00500-004-0388-1

[19] S. Letz, S. Denoux, Y. Orlarey, and D. Fober, "Faust audio DSP language in the Web," *Linux Audio Conference*, pp. 29–36, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02159002

[20] J. Fiala, N. Segal, and H. A. Rawlinson, "Meyda: an audio feature extraction library for the Web Audio API," in *1st Web Audio Conference (WAC), January 2015, Paris, France.*, 2015. [Online]. Available: https://webaudioconf.com/_data/papers/pdf/2015/2015_17.pdf

[21] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas, "Improving JavaScript performance by deconstructing the type system," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 496–507, jun 2014. [Online]. Available: https://dl.acm.org/doidoi//10.1145/2666356.2594332

[22] K. Rupp, "Microprocessor Trend Data," 2022. [Online]. Available: https://github.com/karlrupp/microprocessor-trend-data

[23] U. Hiwarale, "Parallel programming in JavaScript using Web Workers," 2018. [Online]. Available: https://medium.com/jspoint/achieving-parallelism-in-javascript-using-web-workers-8f921f2d26db

[24] MDN Authors, "Using Web Workers," 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers#other_types_of_worker

[25] "Worker support for BaseAudioContext · Issue #2423 · WebAudio/web-audio-api." [Online]. Available: https://github.com/WebAudio/web-audio-api/issues/2423

[26] M. Srinivas and L. M. Patnaik, "Genetic Algorithms: A Survey," *Computer*, vol. 27, no. 6, pp. 17–26, 1994. [Online]. Available: https://ieeexplore.ieee.org/document/294849