ga-synth

Victor Gabaldon

Motivation

Challenges of conventional synthesisers

Sound synthesisers are revolutionary but convoluted machines that can make the production of a specific sound wave difficult for certain users.

The main reasons are:

- A **large set** of parameters that can affect the sound wave.
- A **lack of intuitive correlation** between the parameter values and the output, presenting a non-linear problem.
- Human **biases** and the need for an excellent **hearing acuity**.

Expanding from previous work

Evolutionary techniques applied to sound optimisation spaces have been studied before, however the existing literature only covers **very specific cases** and configurations.

It is often **difficult to replicate** these experiments due to the lack of detailed information about the algorithm and conditions.

There are **no general-purpose tools** publicly available to find the optimal parameters of sound waves using evolutionary techniques.

The project aimed at building **Rust library** from scratch to carry out further research on sound parameter optimisation and enable future applications to use GAs for this purpose.

Inspiration from biological evolution

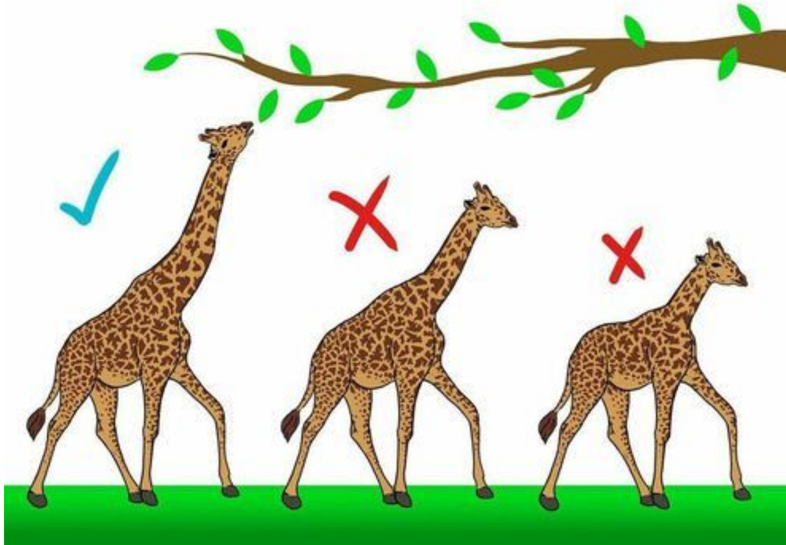
Genetic Algorithms (GAs)

Metaheuristic inspired by the process of **natural selection** that belongs to a larger class of evolutionary algorithms (EAs).

Use biologically inspired operators such as **mutation**, **crossover**, and **selection**.

We need:

- Genetic representation
- Fitness function



Applications in sound synthesis

Genetic algorithms may be a good method for finding the optimal configuration for a given sound wave.

Both **interactive** and **non-interactive GAs** have been explored before. They are generally successful but their configuration and fitness choices matter.

Solution

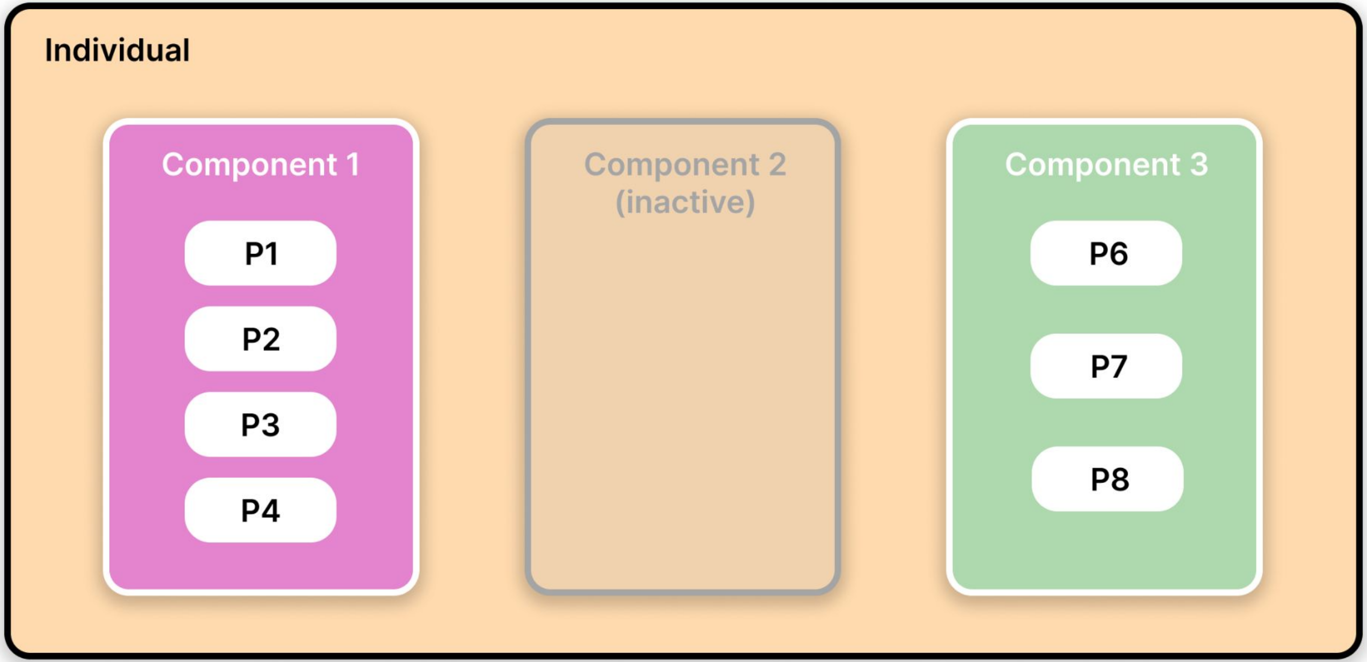
Overview

The solution consists of a **library** that can be used to **predict the optimal parameters** to generate a specific sound wave.

There are two main modules:

- Algorithm
- Sound synthesis

Individual representation



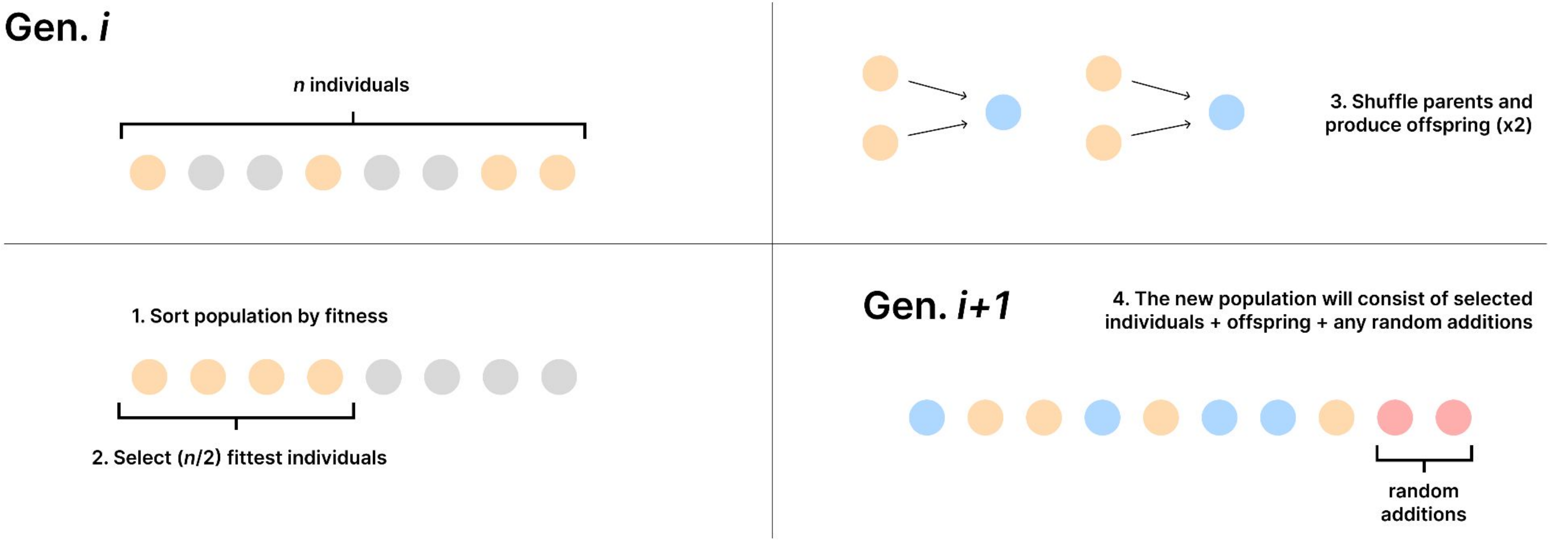
Component representation

```rust
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,
}
```

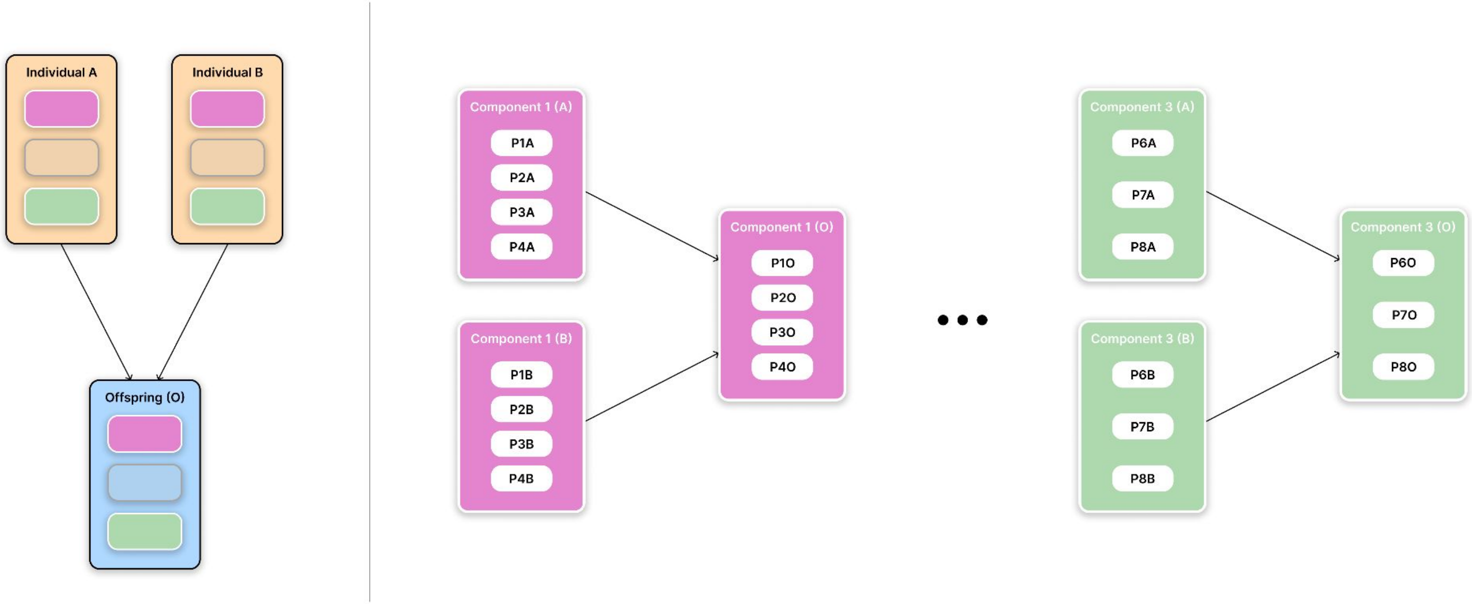
Component representation

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

Iteration


Gen. *i*
n individuals
3. Shuffle parents and produce offspring (x2)
1. Sort population by fitness
2. Select (*n*/2) fittest individuals
Gen. *i+1*
4. The new population will consist of selected individuals + offspring + any random additions
random additions

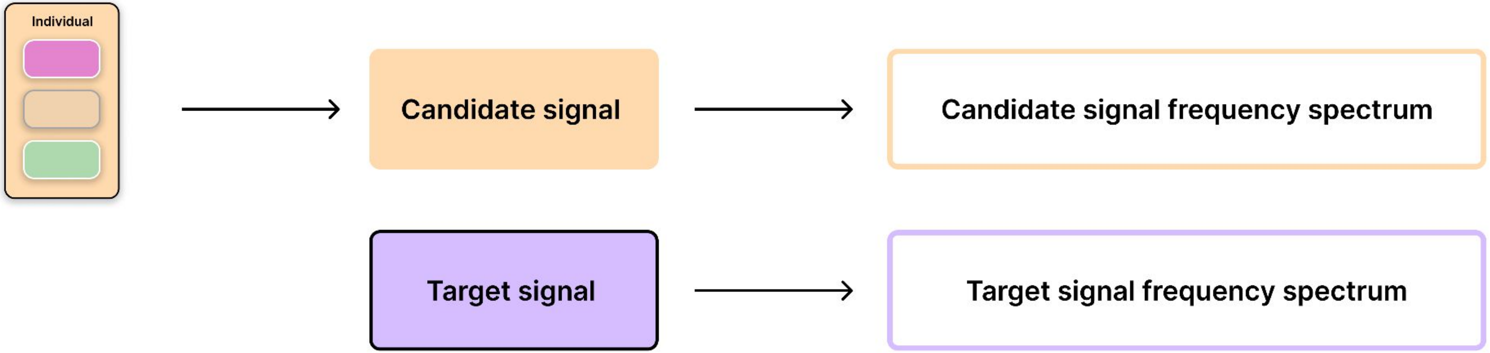
Crossover



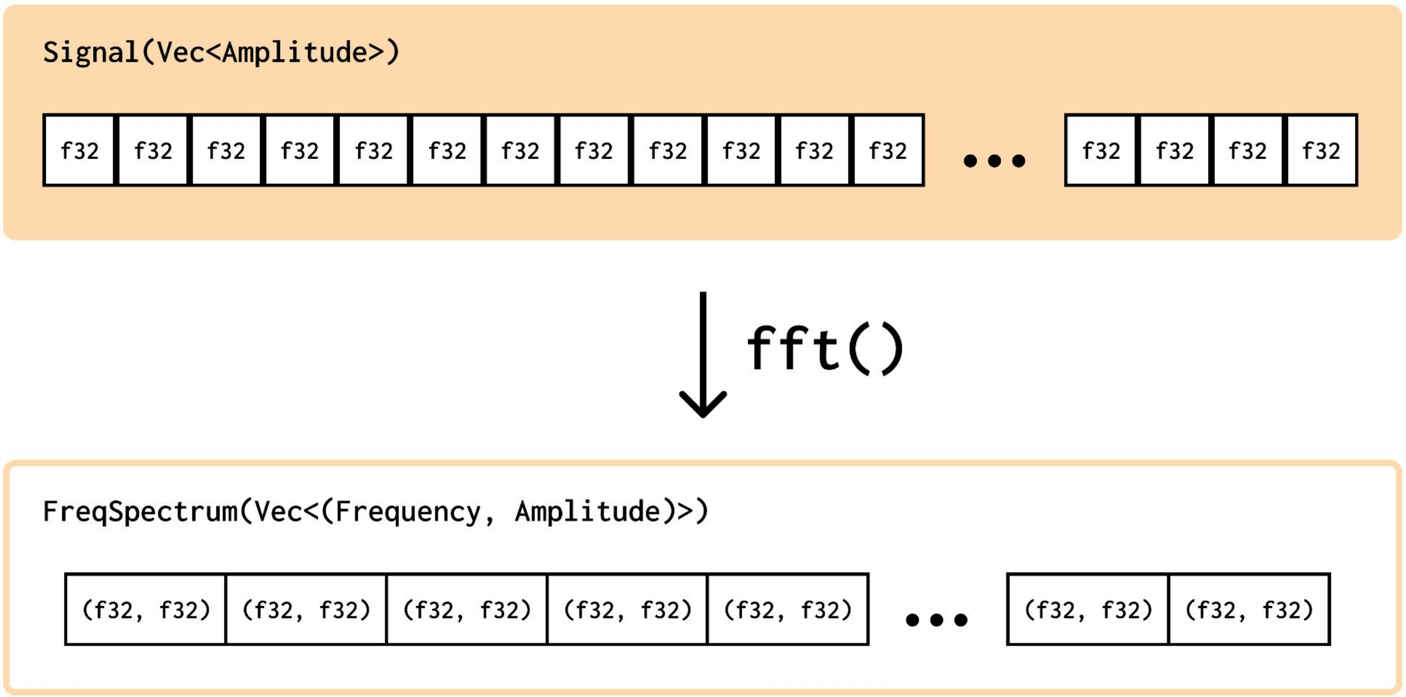
Individual A
Individual B
Offspring (O)
Component 1 (A)
P1A
P2A
P3A
P4A
Component 1 (B)
P1B
P2B
P3B
P4B
Component 1 (O)
P1O
P2O
P3O
P4O
Component 3 (A)
P6A
P7A
P8A
Component 3 (B)
P6B
P7B
P8B
Component 3 (O)
P6O
P7O
P8O

Fitness


f(
Individual
) = similarity(
C
,
T
)


Individual
Candidate signal
Candidate signal frequency spectrum
Target signal
Target signal frequency spectrum

Frequency decomposition



Synthesis methods

```rust
#[derive(Clone, Debug, PartialEq)]
pub struct SubtractiveIndividual {
    target: Arc<Signal>,
    fitness_type: FitnessType,
    fitness: Option<f32>,
    oscillator: Option<OscillatorComponent>,
    filter: Option<FilterComponent>
}
```

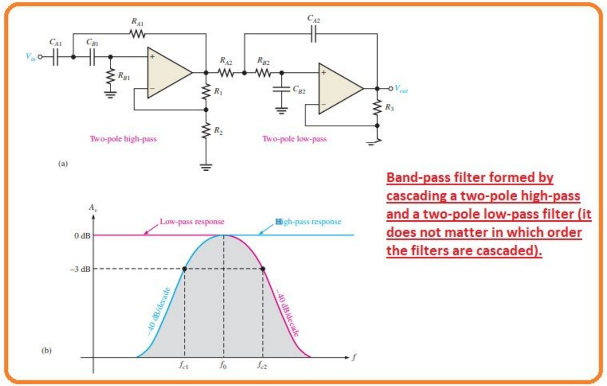
```rust
#[derive(Clone, Debug, PartialEq)]
pub struct AdditiveIndividual {
    target: Arc<Signal>,
    fitness_type: FitnessType,
    fitness: Option<f32>,
    harmonics: Option<HarmonicsComponent>
}
```

Sound synthesis

Starting from an empty signal, each synthesis component in the individual is applied following a top-down approach.

For instance, this is the function used to generate a band pass filter.

```
viktaur
fn band_pass_filter(low_freq: f32, high_freq: f32, band: f32) -> Vec<f32> {
    assert!(low_freq <= high_freq);
    let low_pass : Vec<f32> = Self::low_pass_filter(high_freq, band);
    let high_pass : Vec<f32> = Self::high_pass_filter(low_freq, band);
    utils::add(&high_pass, &low_pass)
}
```



Sound synthesis

$$y_1 = A\sin(2\pi ft + \varphi)$$

```
/// Produces a sine waveform with the specified parameters.
5 usages  viktaur *
pub fn sine_wave(freq: f32, length: f32, sample_rate: f32, amplitude: f32, phase_offset: f32) -> Signal {
    1 usage  viktaur
    const PI_2: f32 = core::f32::consts::PI * 2.0;

    let sample_period :f32 = 1.0 / sample_rate;
    let n :f32 = sample_rate * length;

    let mut samples :Vec<f32> = vec![];

    for i :u32 in 0..n as u32 {
        samples.push(
            value: amplitude * f32::sin( self: PI_2 * freq * i as f32 * sample_period + phase_offset),
        );
    }

    Signal(samples)
}
```

Running the simulation

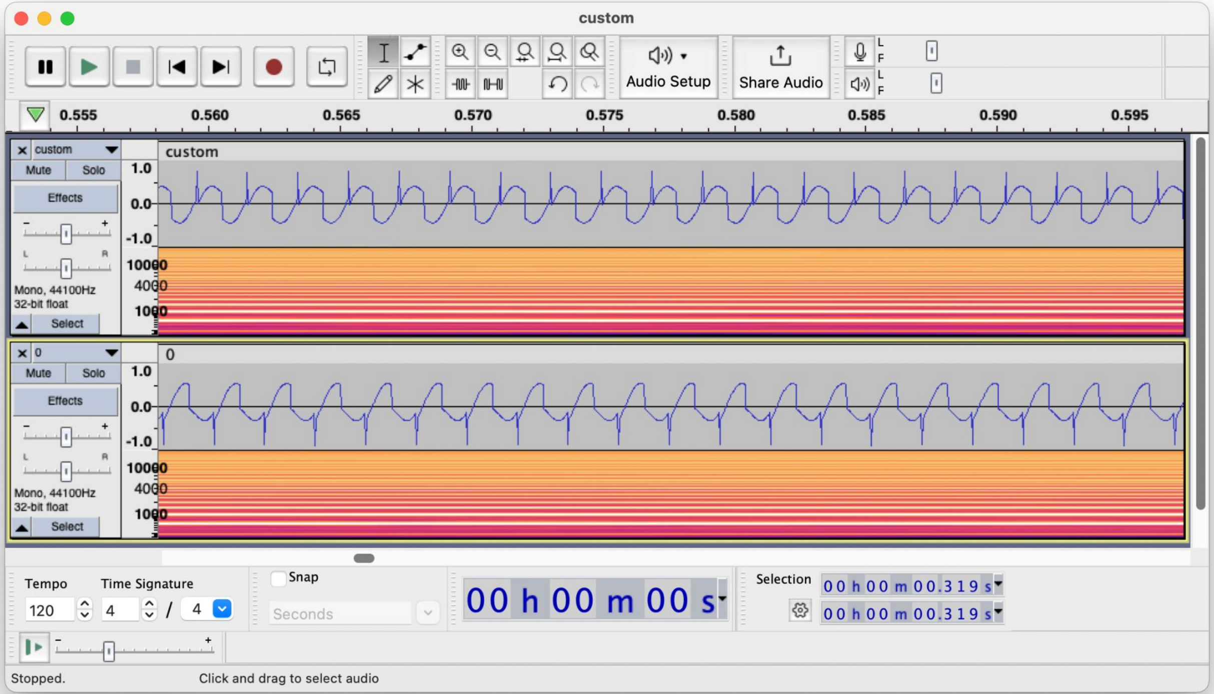
```rust
impl<T: Individual> GASimulation<T> {
    fn init_population(n: u32, generator: &T::Generator) -> Vec<T> {...}

    /// A step in the iteration of the algorithm. Given the current state of the simulation,
    /// calculates the next generation.
    fn next(&mut self) -> Result<(), GeneticSimulationError> {...}


    /// Runs a genetic algorithm simulation.
    pub fn run(&mut self) -> Result<T, GeneticSimulationError> {...}
}
```

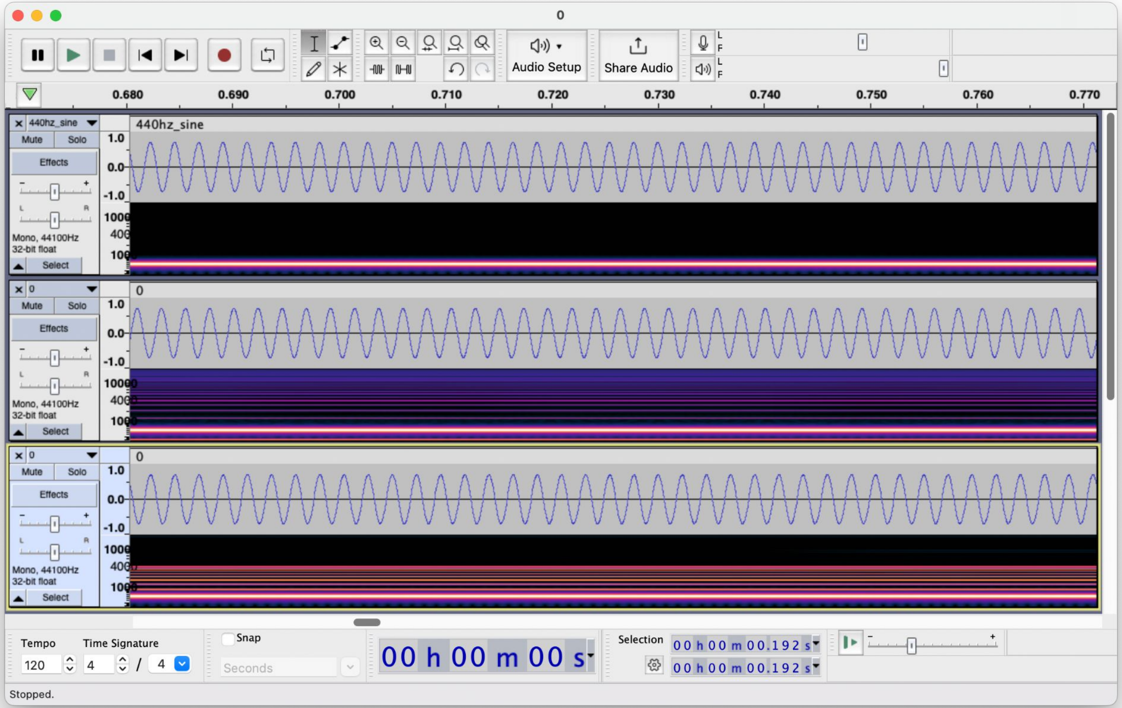
Testing and results

Example - *native* target wave



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

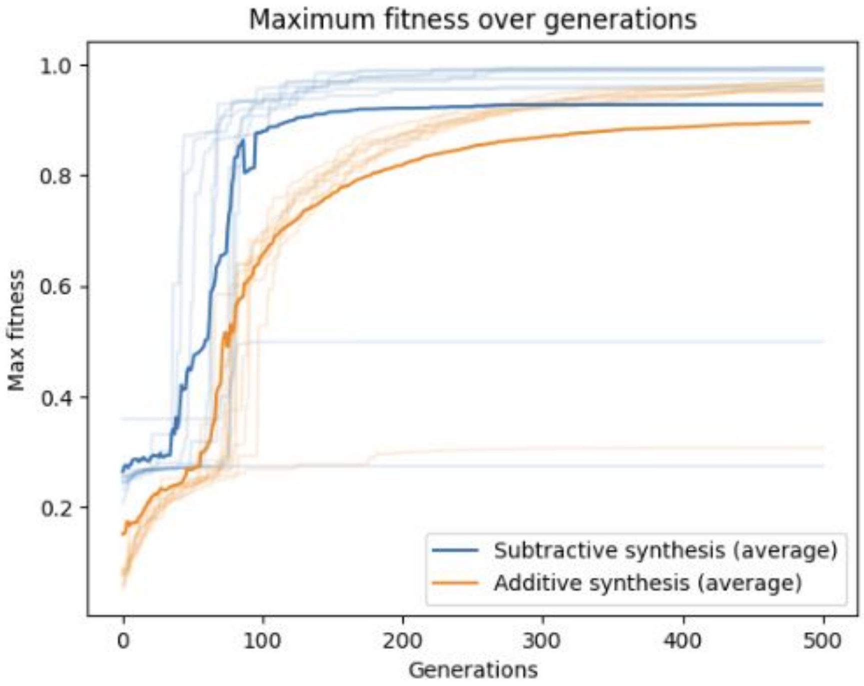
Example - sine wave (sub vs add)

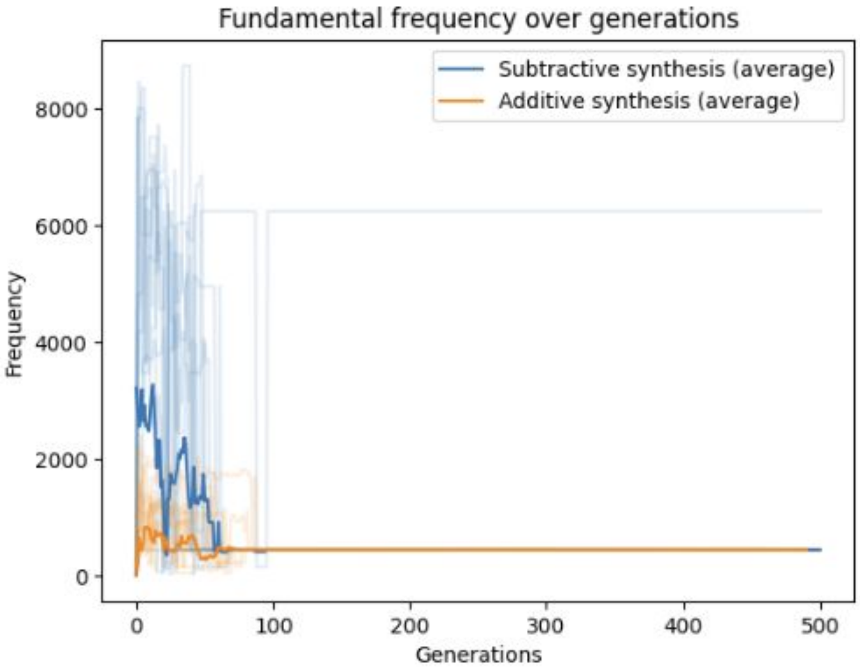


fitness	0.9962411
freq	439.99786
sine_amp	0.70819336
sine_phase	3.1442568
square_amp	0.0020760477
square_phase	4.739438
saw_amp	0.0031836072
saw_phase	3.3810263

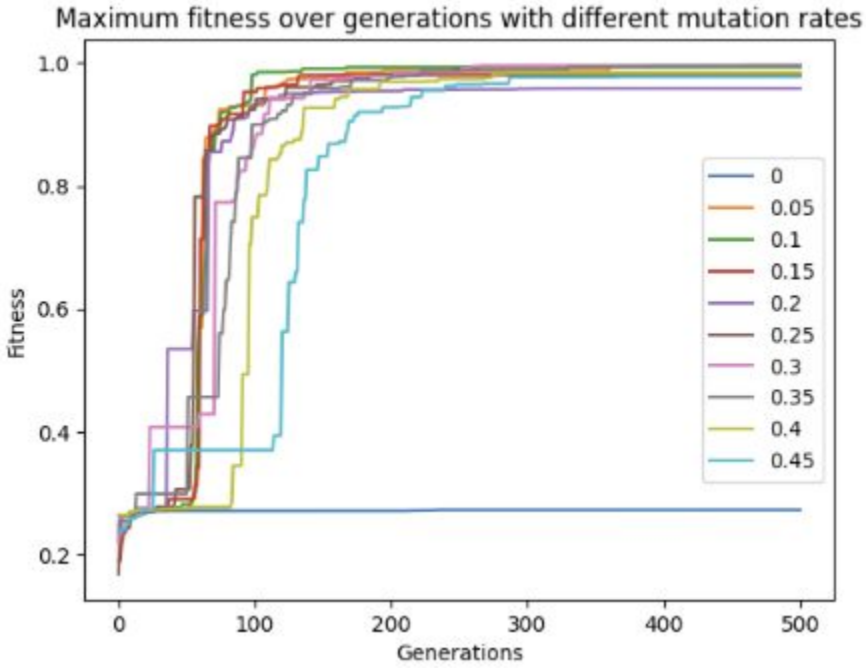
fitness	0.90962845
freq	439.90002
amp_1	0.703382
amp_2	0.004360318
amp_3	0.005996208
amp_4	0.003657665
amp_5	0.0038192347
amp_6	0.0033883916
amp_7	0.004195324
amp_8	7.981062e-5
amp_9	0.006465002

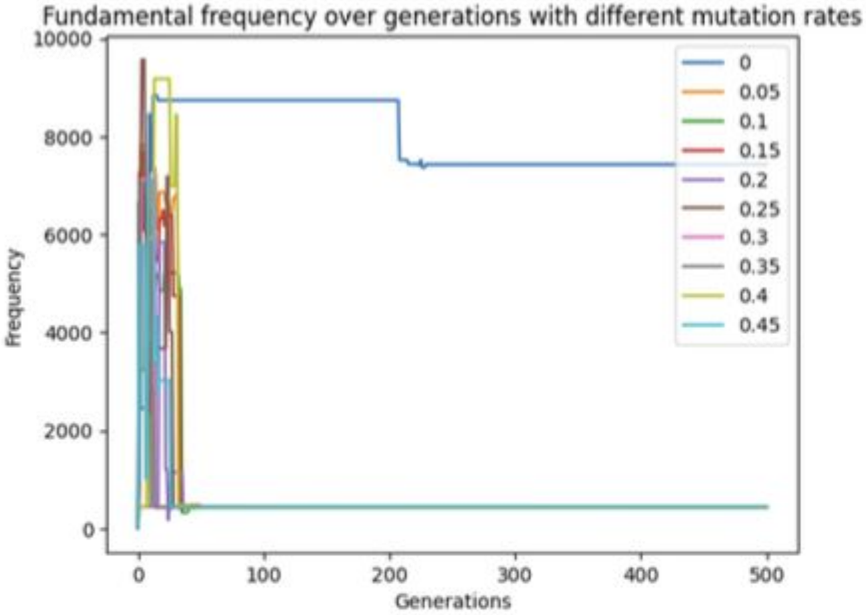
Example - sine wave (sub vs add)





Example - different mutation rates





Conclusion

- About 80% of the simulations reached a satisfactory value.
- Unusual approaches to reach a valid solution.
- GA performs much better than hill climbing.
- Populations as small as 16 individuals managed to reach good fitness.
- Performance tradeoffs with increasing populations.
- Random additions did not have a notable impact.
- Optimal mutation rate range is between 0.05 and 0.25.
- Obvious limitations in the current subtractive and additive synthesis individuals.

FAQs

What does the project consist of?

The design and implementation of a parametric optimisation **library** in Rust, and the **research** on multiple configurations of GAs applied to sound synthesis.

How can the library be used?

The library contains **API endpoints** that abstract away the complexity of the system and enable **researchers** and **third-party applications** to run parametric optimisation simulations with it.

```
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.");
}
```

What is it needed to run a simulation?

- Choosing a **synthesis method** and **components** that will be present.
- A file containing the **target** sound wave.
- Specifying **parameters** like fitness evaluation method, initial population, random additions, mutation rate, etc. Or rather leaving them as default.
- Optionally, the fittest signal can be **exported** to WAV file and the simulation data to a CSV file.

What was the testing strategy?

A total of 12 **unit tests** have been defined to test the evolution of a simulation and the correctness of the signal processing functions. This includes oscillators, harmonics, and signal analysis. CSV exporting has also been covered in the tests.

Simulations have been run as a form of **integrated testing**, to evaluate the success of the GA.

How many lines of code?

According to the *scc* tool, there are **23 files** and a total of **2,382 lines** of code.

Does it currently support IGAs?

No.

However, this could be easily incorporated by replacing the existing objective fitness functions with a new one considering user feedback.

Can you generate *any* sound?

No.

Synthesisers are incredibly complex machines with dozens of parameters. The implemented system is a proof-of-concept to study the performance of GAs, hence why only a limited range of sounds can be currently generated.

Thanks to the modular organisation of the library, new components could be easily created and integrated with the rest of the system.

How can the effectiveness of the GA be evaluated?

A **hill-climbing** algorithm, which usually performs best in problems with a single optimum, has also been included in the library for evaluation purposes.

Under the same conditions, both techniques can be **compared** in terms of fitness score or any other metric over the number of individuals generated.