Genetic Algorithms for Interactive Music Creation

Garrick Fernandez
Department of Computer Science
Stanford University
Stanford, CA
garrick@cs.stanford.edu

***Abstract*—Genetic algorithms have been applied to a number of problems, including the composition and generation of music. More broadly, computers are a cornerstone of the music-making process, often utilized for the arrangement and synthesis of new sounds. In this paper, we apply genetic algorithms to the task of music creation, with an eye towards live feedback and interactive performance. Our main contributions are (1) a formalization of the application of genetic algorithms to sound synthesis and rhythmic sequencing, (2) an exploration of human input and methods for fitness approximation, and (3) the production and interpretation of generated music.**

***Index Terms*—genetic algorithms, evolutionary music, sound synthesis, fitness approximation**

I. Introduction

Computers have been applied to the generation and composition of music since the 1950's, and as such, the field of computer music is vast. Genetic algorithms have seen use in a number of problem domains. This project seeks to apply genetic algorithms to the problem of music creation. In addition, we wish to incorporate human feedback as a mechanism by which the genetic algorithm proceeds. We first formalize the problem and describe the method and implementation used to optimize it. We then present results on the music generation and progression of our algorithm.

II. Problem

There are many creative aspects to music we may wish to augment with a genetic algorithm. To formalize our problem, we consider a simplification of the music generation task, taking inspiration from step sequencers.[1]

A sequencer [1] records, stores, and plays back music; it is operated or programmed by a musician. All sequencers must store some form of note and performance information; they may produce sounds with one or more voices, or unique timbres (think of the difference between a trumpet and a tuba).

In lieu of exact timing, a step sequencer rounds note triggers to the nearest whole beat. Usually a grid of 16 buttons is provided to allow toggling a note to play on a certain beat. At each step, or beat, a note is either triggered, or it is not. This 16-beat measure loops indefinitely.

This project was done for AA222/CS361: Engineering Design Optimization. The code is available on GitHub at garrickf/genalg-sequencer. Thanks to Prof. Mykel Kochenderfer and the rest of the teaching team for their feedback on the project and a great class!

[1]Of particular inspiration to the author were Pocket Operators, handheld drum synthesizers and sequencers designed by Teenage Engineering.

The tradeoff in the precision and beat-accurate timing of step sequencers is that timing variation due to expressive performance is non-existent. As such, this style of sequencing lends itself well to drum machines and electronic music.

With this perspective, we can model music as a collection of chosen voices ("parts") performing different notes at discretized timings. A single part in our sequencer can be thought of as an individual design point which encodes information about its voice and trigger time information. A musical collection of parts is a collection of design points. The formal goal of our genetic algorithm is to optimize these design points and produce better parts, with "better" being dependent on the user/listener's tastes.

An important aspect of our task is integrating the genetic algorithm into a human-centered music production process. When creating music, one can imagine starting a song with a single track, then introducing and layering more tracks as time goes on. Some tracks may be better or worse than others, creating selection pressure on the track. As such, we focus on methods which optimize with respect to a population of chromosomes, and allow the user to pick those individuals they like for inclusion in their pieces.

Another perspective on our task is that we seek to incorporate genetic algorithms into the exploration process for sounds and patterns in a synthesizer/sequencer. The human operator is not removed from this process; rather, they close a loop between the genetic algorithm and the music, providing feedback to the genetic algorithm and thereby manipulating the outputted sound.

III. Method

Genetic algorithms fall under a larger class of optimization techniques, population methods, which involve optimization using a collection of design points. In particular, genetic algorithms borrow inspiration from biological evolution [2], [3]. In order to apply genetic algorithms to our task, we must specify how to encode our music as genetic material, as well as the mechanisms with which it evolves over time.

A. Chromosome Structure and Representation

To represent points associated with individuals in the design space, we used real-valued chromosomes (n-dimensional vectors in $\mathbb{R}^n$) with some components constrained to be integers. Each chromosome is interpreted as having three main portions:

1) A single positive integer $k \in \{1, \dots K\}$, the "instrument," encodes which voice is used, where K is the total number of voices available to the sequencer.
2) A $\mathbb{R}^{n_{\text{emb}}}$ chunk of the chromosome, the "expression," encompasses all points in the design space for instrument k. Client code in an audio synthesis program can ingest these numbers and map them to properties of the instrument (such as the frequency, filter cutoff, reverb, attack, etc.). For us, $n_{\text{emb}} = 16$.
3) An $\{0, 1\}^{n_{\text{tim}}}$ chunk of the chromosome, the "timing," encodes trigger time information (1 if the note should play on time-step $t \in \{1, \dots, n_{\text{tim}}\}$, 0 otherwise). For us, $n_{\text{tim}} = 16$, corresponding to the typical grid of sixteen steps seen in step sequencers.

It follows that the dimension of our chromosomes is

$$1 + n_{\text{emb}} + n_{\text{tim}} = 33 \tag{1}$$

The inclusion of an expression component allows the genetic algorithm to perform exploratory sound synthesis in addition to sequencing notes. The parameters and bounds of this exploration are set by client (instrument) code. Because of this, a programmer or musician retains some control over the expression of their instruments through the way they interpret incoming design points. However, the genetic algorithm remains the process by which these design points are evaluated and improved. This decoupling grants flexibility in terms of being able to create new instruments, or a new suite of instruments, which react to expression parameters in different ways.

B. Evolutionary Dynamics

All genetic algorithms require instructions for how to produce the initial population of chromosomes, how to perform selection, how to perform crossover of genetic material between two parents, and how to perform mutation of a chromosome. We refer to these jointly as the evolutionary dynamics of the genetic algorithm.

For each individual in the initial population, the instrument is drawn from a categorical distribution assigning equal probability mass to each instrument $k \in \{1, \dots, K\}$. The expression information is sampled from a uniform distribution between 0 and 1 for each coordinate. Since trigger times can only be 0 or 1, we sample from a Bernoulli distribution for each of the sixteen time steps. The distribution is weighted more towards sounding a note ($p = 0.75$), so that the algorithm starts off with denser rhythms, although this can be tuned.

Selection determines which chromosomes will be parents for the next generation. We considered two approaches for selection.

The first selection method we considered was truncation selection, where we sample from the top-k chromosomes in the population.

The second selection method we considered was roulette wheel selection, also known as fitness proportionate selection. In this method, each parent is chosen with a probability proportional to its performance relative to the rest of the population. Compared to truncation selection, roulette wheel gives a small chance of less fit individuals being selected, which can help maintain genetic variability in future generations.

Our method for implementing roulette wheel selection involves selecting the maximum objective value $y_{\max}$ among objective values $y_1, \dots, y_{N_{\text{pop}}}$, where N_{pop} is the population size. Then, the selection likelihood for an individual with objective value y_i would be:

$$\frac{y_{\max} - y_i}{\sum_{j=1}^{N_{\text{pop}}} y_{\max} - y_j} \tag{2}$$

Since smaller objective values are more optimal (we use the convention of minimization), the smaller y_i is, the larger the difference between it and $y_{\max}$. Thus the smallest y_i values are associated with the highest chances for selection. In this scheme, the worst-performing individual will have zero probability mass assigned to it, as the numerator in (2) would be zero.

In both selection methods, the sampling process is repeated multiple times to select two parents for each of the N_{pop} offspring.

Crossover is the process by which two parents combine their genetic material to form children. There are multiple approaches to crossover; for our application, we use a modified version of uniform crossover that behaves differently for each portion of the parent chromosomes. In this method, instrument and expression information are selected from either parent with equal probability. For timing information, however, a four-beat bar is selected from either parent with equal probability. This choice was to attempt to preserve an element of phrasing a listener may experience when hearing music—some beats are perceptually grouped together into a unit with its own musical sense, and a user may want to preserve or explore these traits more. We leave it to the mutation operation to produce trigger timing variations at the more fine-grained, beat-by-beat level.

Mutation allows new traits to appear in the population, driving the exploration of the genetic algorithm. For our application, we mutate each component in the chromosome (resample it) with some fixed probability. In our implementation, the probabilities are specific to the the instrument, expression, and timing regions of the chromosome, and they be modified.

C. Fitness Approximation Based on User Feedback

Selection in a genetic algorithm happens with respect to a fitness function over the individuals in a population. Our fitness function is dependent on the user's tastes, which are not known prior to the start of the algorithm. One idea is to collect observations of the user's tastes (like or dislike), and fit an objective function to those observations.

The methodology introduced in [3] for fitting surrogate models of an objective function from data can also be applied to this fitness function. Such methods are often referred to as fitness approximation.

This is a natural fit for a fitness function which depends on the the preferences of the user; function evaluations are expensive in the sense that they would take up a lot of the user's time, which can result in poor user experience (imagine having to listen to and evaluate tens or hundreds of individuals).

Instead, we bootstrap the fitness function and capture he user's preferences by fitting a surrogate model, which steers the genetic algorithm towards sounds and patterns the user likes more.

Assuming that we get the data from some upstream source (see Section IV), we use logistic regression with L2 regularization to fit the points. This model requires at least one point of each class (like and dislike) of each class in order to train, so we skip training and use the same random population initialization for new generations until enough data has been collected. We provide a way to pass weights on the data points through to the solver. This enables a form of recency-weighting, where points the user liked or disliked more recently are weighted more than previous data points. We use an exponential-decay scheme where, upon the addition of a new point or the evolution of a new generation, all points observed up to this point are weighted by a multiplicative discount factor γ. This can capture changing tastes or the progression of the algorithm into new regions of the design space, and it allows the regression to handle changing relationships over time.

IV. Implementation

Our implementation of the methods discussed in the previous section span sound generation and the genetic algorithm itself.

In order to generate music, a separate process must run audio synthesis code and generate audio samples comprising the sound we hear. To do this, we use SuperCollider [4], an audio synthesis platform comprised of two main components:

1) `scsynth` is a real-time audio server that produces the audio samples themselves, based on sets of instructions.
2) `sclang` is an interpreted programming language that is used to talk to `scsynth`. Within `sclang`, we can define unique instruments (called `SynthDefs`), configure them on-the-fly, and schedule them to play at regular intervals.

At the time of writing, we developed $K = 6$ configurable voices in SuperCollider.

Our genetic algorithm, the controlling process, needs to talk with `sclang` to coordinate the creation and scheduling of instruments for our genetic algorithm synthesizer/sequencer. To do this, the process talks to `sclang` using the OpenSoundControl (OSC) data transport specification [5] for real-time message communication. OSC messages are sent from the controlling process to SuperCollider via loopback (`localhost`, i.e. `127.0.0.1`) and look like the following:

$$\texttt{/addressName}\,[\texttt{param1}\ \texttt{param2}\ \texttt{param3...}] \quad (3)$$

where `/addressName` is a user-specified string specifying the type and purpose of the message. The address is followed by a variable-length sequence of parameters of user-specified types (for example, the individual values of our chromosome). The audio synthesis process in Supercollider is set up to respond to various OSC addresses, comprising a callable API into the synthesizer/sequencer. One may use the API to listen to what a chromosome sounds like, "push" a track onto the mix to layer sounds, etc.

A simple REPL (read-eval-print loop) client (see Fig. 1), written in Python, was implemented to allow a user to orchestrate the entire genetic algorithm and music creation process from start to finish. As the user types commands, the client issues OSC messages to a running SuperCollider instance and manages the genetic algorithm. The client supports operations such as liking/disliking an individual (used in fitness approximation), triggering a new generation of individuals, recording, and more.

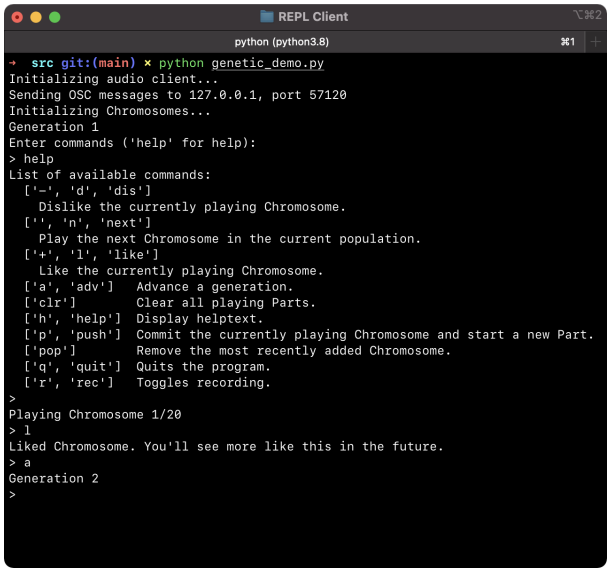

Fig. 1. Example run of the REPL client.

The genetic algorithm itself was developed in Python. While some Python packages exist for performing optimization using a genetic algorithm, the genetic algorithm here was implemented from scratch, for the author's edification. The external package scikit-learn [6] was used for fitting the approximate fitness functions.

V. Results and Interpretation

A. Optimizing Test Functions

To test the veracity and versatility of our genetic algorithm infrastructure, we first direct the algorithm to optimize some more concrete objective functions: Rosenbrock's and Booth's function. We alter the evolutionary dynamics with prior knowledge of the objective function: the initial chromosomes are points in $\mathbb{R}^2$, we use truncation selection to more aggressively exclude less fit individuals, and we employ a mutation strategy that frequently adds a small amount of noise to the individuals and occasionally adds a large amount of noise.

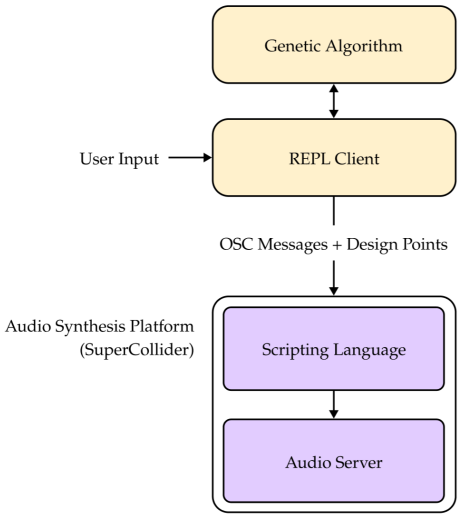

Fig. 2. Diagram capturing the interaction between the genetic algorithm, user input, and the sound-generating process.

See Fig. 3 for scatter plots of the population across multiple generations.

B. Interpreting the User-Based Fitness Approximation

We also interpret the efficacy and development of the fitness approximation trained on observed user preferences. While this is a somewhat subjective thing to evaluate, we can plot some results over the course of a run and verify the development of the fitness approximation towards exploring points of interest.

To plot, we use t-distributed Stochastic Neighbor Embedding (t-SNE) [7] to project the chromosomes down to a two dimensional representation, and we use a Voronoi tessellation [8] to approximate the high-dimensional decision boundary in 2D space. We make the disclaimer that the decision boundary must lie somewhere between the positively and negatively classified points, but in regions with few data points, the true boundary can deviate from this. In particular, the true distances between points, and therefore the distances between points and a decision boundary, are warped upon projection onto a lower-dimensional space.

With a high number of dimensions, the logistic regression classifier is often able to completely linearly separate the data (Fig. 4). We find that the next points in the design space are clustered in regions where the user has previously liked points (Fig. 4 and Fig. 5).

C. Generating Music

Of course, we must make an evaluation of the music produced by the genetic algorithm, with the usual qualifications regarding personal taste.[2]

We find the results to be musically pleasing and rhythmically interesting, capturing elements of the original step

[2]The demos are hosted at https://garrickf.github.io/genalg-sequencer/, check them out!

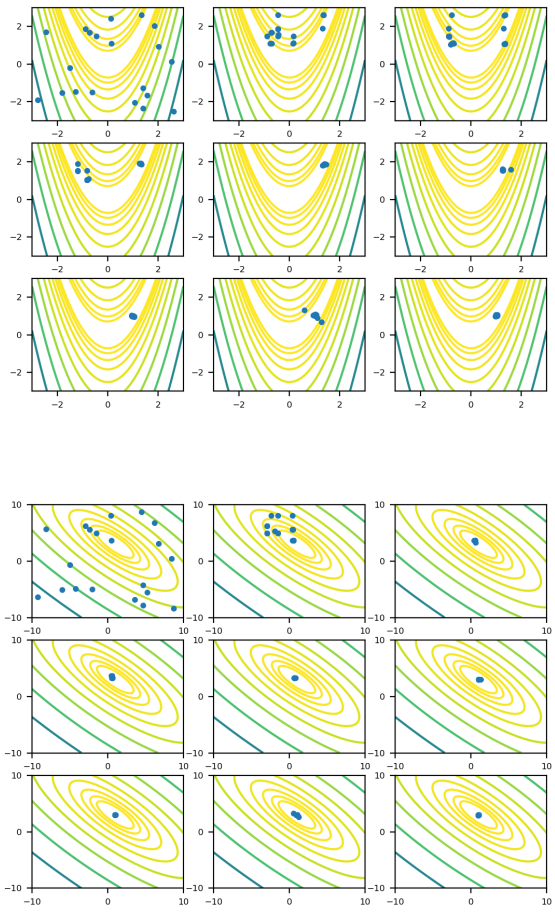

Fig. 3. Running genetic algorithm optimization for 999 iterations for Rosenbrock's function (top) and Booth's function (bottom). Plots, in reading order, are of the individuals at iteration 1 (the initial population), 2, 3, 5, 10, 100, 500, 900, and 1000.

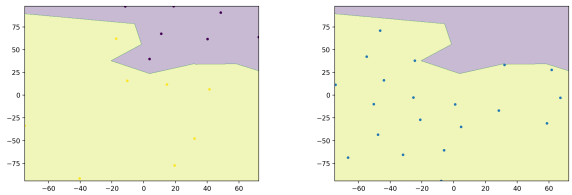

Fig. 4. An example run where the surrogate model can linearly separate the data. Points are colored according to their true label (yellow for positive, purple for negative). Suggested points (blue) are exclusively in the positive region of the classifier.

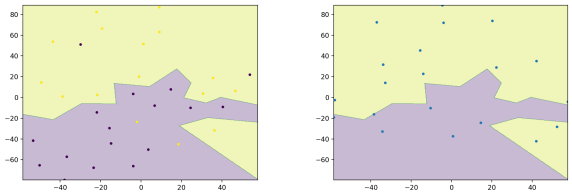

Fig. 5. As points accumulate, the model becomes no longer able to linearly separate the data. Suggested points reflect the classifier's uncertainty and place some points in negatively classified regions.

sequencer inspiration we sought to reproduce. The exploration induced by the genetic algorithm allows the discovery and experimentation with patterns and sounds one may not necessarily find through fiddling with multiple knobs and settings on a synthesizer or digital audio workstation (DAW). We view the genetic algorithm as an interface to the timing and expression parameters of a synthesizer and sequencer, one that reduces the friction for trying new things.

VI. Related Work

Our project seeks to augment music creation through applying genetic algorithms to the model of a synthesizer and sequencer. Horner and Goldberg also apply genetic algorithms to music creation, focusing on the problem model of "thematic" bridging, which seeks to interpolate from a starting musical pattern to a final one [9]. A solution's fitness is a function of how closely the ending pattern matches the starting one. In contrast, our fitness function is interactive, or dependent on user input. Whereas the iterative process for music generation in their work involves repeated runs of the algorithm until the results are acceptable, our work introduces human input into the loop for steering the genetic algorithm.

Matíc also applies genetic algorithms to music composition, modulating rhythms and pitches with a predefined set of mutation operations [10]. Similarly to Horner's and Goldberg's work, automated criteria are used to determine the quality of a composition, related to the evaluation of the intervals between successive notes, deviations from reference values, and counts of implementation-defined "bad" tones. In contrast, our model does not work with pitches or varied durations of notes, simplifying our design space to that of a sequencer.

Some systems for drum loop evolution have been produced (e.g. MuSing [11]). Our algorithm works in a similar way. Whereas some drum machines rely on fixed samples, we have the ability to evolve the synthesis of sounds though the expression component of a chromosome.

Many musical artists use elements of genetic algorithms or computation in their performance, and their tools to do so are often highly specialized. George Lewis, one of the pioneers of computer music, created the Voyager system, which provides live accompaniment to an instrumentalist [12]. Al Biles applied genetic algorithms to producing improvised jazz solos [13].

VII. Conclusion and Future Work

In conclusion, we show an application of genetic algorithms to the creation of music, through the lens of a sequencer/synthesizer.

More broadly, methods like these may be used for algorithmic-assisted exploration of a large design space where function evaluations are relatively cheap, and expert knowledge is useful. One can imagine algorithms being applied to help designers or artists find and rapidly prototype new ideas.

It would be exciting to see and extension with more instruments and more values and ranges for expression. For example, one such extension could add melodic expression to the chromosome, whose coordinates can encode the pitch of each note.

References

[1] "Music sequencer," https://en.wikipedia.org/wiki/Music_sequencer (accessed May 2021).

[2] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[3] M.J. Kochenderfer and T.A. Wheeler, *Algorithms for Optimization*, MIT Press, 2018.

[4] "Supercollider: supercollider," https://supercollider.github.io/ (accessed May 2021).

[5] M. Wright and A. Freed, "Open sound control: a new protocol for communicating with sound synthesizers," in International Computer Music Conference (ICMC), 1997, pp. 101-104.

[6] "Scikit-learn: machine learning in python," https://scikit-learn.org/stable/ (accessed May 2021).

[7] L.J.P. van der Maaten and G.E. Hinton, "Visualizing high-dimensional data using t-sne," in Journal of Machine Learning Research vol. 9, 2008, pp. 2579-2605.

[8] M.A. Migut, M. Worring, and C.J. Veenman, "Visualizing multi-dimensional decision boundaries in 2d," in Data Mining and Knowledge Discovery, vol. 29, 2015, pp. 273-295.

[9] A. Horner and D. E. Goldberg, "Genetic algorithms and computer-assisted music composition," January 1991.

[10] D. Matíc, "A genetic algorithm for composing music," Yugoslav Journal of Operations Research, vol. 20, pp. 157-177, April 2010.

[11] "MuSing," http://www.geneffects.com/musing/ (accessed May 2021).

[12] G.E. Lewis, "Too many notes: computers, complexity and culture in voyager," Leonardo Music Journal, vol. 10, pp. 33-39, 2000.

[13] J.A. Biles, "GenJam: a genetic algorithm for generating jazz solos," International Computer Music Conference, pp. 131-137, July 1994.