Audio Approximation

Yonatan (Yoni) Berg
AI class, school of CS
Hebrew University
Jerusalem, Israel
yonatan.berg1@mail.huji.ac.il
yoniberg1@gmail.com

Alon Soffer
AI class, school of CS
Hebrew University
Jerusalem, Israel
alon.soffer@mail.huji.ac.il
alonsoffer@gmail.com

Shir Molina
AI class, school of CS
Hebrew University
Jerusalem, Israel
shir.molina@mail.huji.ac.il

Hagai Yehoshafat
AI class, school of CS
Hebrew University
Jerusalem, Israel
hagai.yehoshafat@mail.huji.ac.il

***Abstract* — we developed a genetic algorithm that, given a short audio file, uses pre-recorded building blocks (e.g. piano notes) to construct the most similarly-sounding tune. In addition, we fit a polynomial to the input audio and sample it to generate a different similarly-sounding tune.**

Keywords—genetic algorithm, curve fitting, generative, audio

I. Introduction

Audio generation is an exciting, challenging field, today mostly done manually by playing instruments. The main computational alternative methods used today for audio generation are neural networks of different sorts. One such challenge is approximating a given melody - be it using a different instrument than the original, constructing the tune by overlaying pure sound frequencies, or some other method of 'approximation'.

One can think of the task of approximating/mimicking an existing audio excerpt using pre-recorded building blocks (e.g. piano notes) as 'learning to play' this particular melody. For this generative-natured task we present a classic generative AI approach - a genetic algorithm; it cultivates a population of tunes, each made up of pre-recorded building blocks with varying intervals between them, aiming for a high similarity score (which is computable) with the input audio. We demonstrate the effects of tweaking the genetic algorithm's parameters as well as its core functions' design on the method's success, providing numerical comparisons as well as a person's opinion on the sound. We believe the final results are convincing evidence of the solution's appropriateness for the problem.

We also present polynomial fitting as a way of estimating/learning the audio signal's behaviour, as common visualization methods often show audio in macro as wavy functions. Sampling said polynomial to obtain audio based on it, we produce an approximation of the input audio that sounds remarkably similar to the original. We visualize the process, compare the results to those of the genetic algorithm and suggest directions for further investigation.

II. Methods - Genetic Algorithm

A. Tune Representation

Each individual in the population is a 'tune' that (potentially) approximates the input audio. A tune is made up of a varying number of instances of the building blocks, which are in turn characterized by the block type [e.g. the musical note *C4* ('do') versus *D4* ('re')] and the offset in the melody at which the block is started.
A tune can be 'compiled' to audio data.

The variation between tunes consists of the number of blocks, the block types and the block offsets.

A newly initialized tune has a random number of blocks, yet it's correlated to the audio length. Block types and offsets are random.

B. Genetic Algorithm

The population size is predefined and constant. The initial population is generated as described in '*Tune Representation'*.

Each generation reproduces amongst itself, parents selected using a weighted random based on their fitness. Offsprings are randomly selected for mutation. The next generation is chosen from the combined group of parents and offsprings, with some percentage of the best being selected with certainty and the rest selected using a weighted random based on the softmax of their fitness.

The mutation rate decays over generations. The algorithm halts after a predefined number of generations.

C. Reproduce

We developed two methods of reproduction:

- 'Split' reproduce - a point in time in the tune is randomly chosen, and all building blocks starting

before it are taken from one parent while all blocks starting after it are taken from the other.

- 'Random selection' reproduce - random building blocks are selected from each of the parents to form the offspring.

D. Mutate

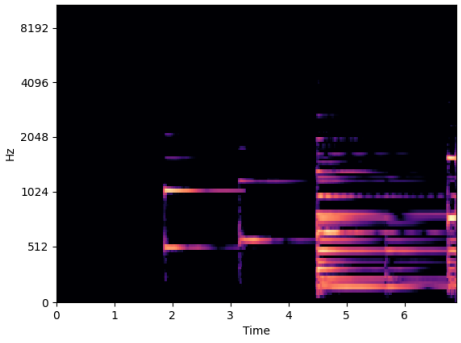
Blocks for mutation are selected at random. A mutated block might be removed, or have a random new block type and start time. New blocks might be added randomly.

Audio processing - brief, simplified basics

A signal can be deconstructed to a sum of sin/cos components with different frequencies using a *Fourier transform*. The frequency description vector of long music/speech audio is usually very convoluted as it tries to simultaneously analyze many different kinds of small pieces of audio stringed together, for example a sequence of syllables each composed of several frequencies. *Short time Fourier transform (STFT)* divides the audio into short time windows and performs a Fourier transform on each of them separately, making the frequency description of every window relatively simple.

Audio frequencies are interpreted by the human ear approximately in log scale, and a transformation from hertz to the *mel scale* is appropriate for that: $m = 1127 \cdot log(1 + \frac{h}{700})$. Amplitudes are also interpreted in log scale, so a transformation to *decibels* is useful.

A *spectrogram* is a common visualization method for such data, and is used throughout the report. The *X* axis represents *time* (in seconds), the *Y* axis represents the frequency (in mel scale) and the color the amplitude (in decibels).



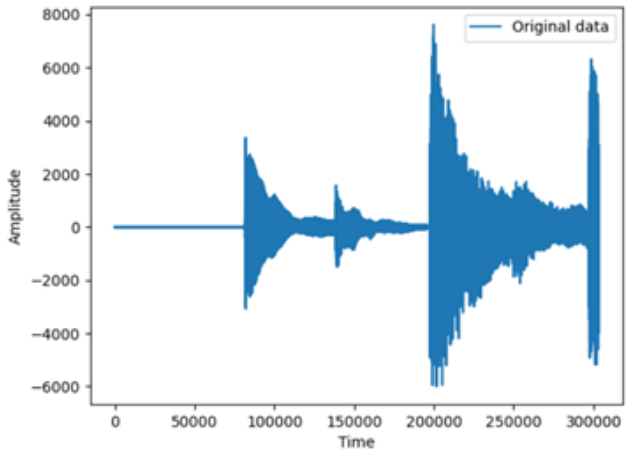
Example spectrogram

E. Fitness Functions

We developed several fitness functions that measure similarity between the input audio and a 'compiled' proposed tune:

- Mean mel Euclidean distance - STFT is performed on both input and proposed audio data, and a transformation to mel scale and decibels is applied on each window. Per window, the Euclidean distance between the (mel, decibel) vectors of the input audio and proposed tune is computed, and the inverse of the mean between all distances is returned as the similarity score.
- Signal Euclidean distance - the Euclidean distance between the signal vectors of the input audio and proposed tune is computed. Its inverse is returned as the similarity score.
- ZCR RMS distance - a vector combining the *zero crossing rate (ZCR)* and *root mean square (RMS)* (term clarification is not supplied here) is created for both the input audio and the proposed tune. The inverse of the Euclidean distance between the resulting vectors is returned as a similarity score.

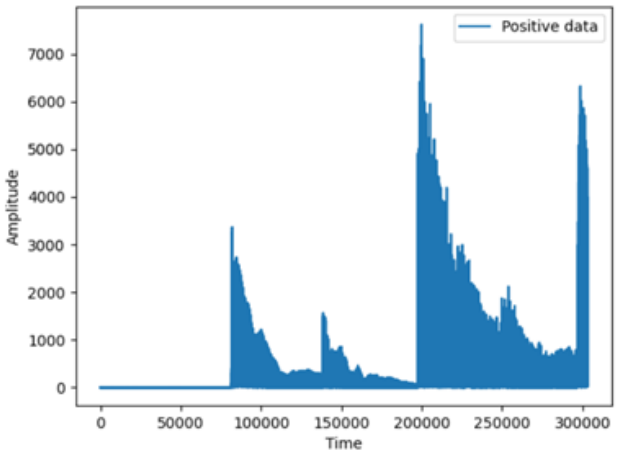
III. Methods - Polynomial Fitting



Input audio signal visualization

As we assume that this signal resembles a polynomial function well enough, we fit a 50-degree polynomial to the data set and then sample it at the same sampling rate as the input to obtain audio data.

Due to the seemingly symmetric form of the graph with respect to the x-axis (though not truly symmetric), straight forward fitting returns the zero function. In order to break the symmetry, we take the absolute value of the signal, fit the polynomial and restore negativity to the previously negative indices.



Signal after absolute value

Most of the signal data is concentrated in lower values, preventing the fitted polynomial from reaching the high values that are important for audio similarity. To overcome this we multiply the function by a constant value.

Finally, we returned the obtained data to the original symmetrical nature of sound signals by multiplying the values at the indices that were negative in the original signal (that we saved earlier), by -1.

IV. Results - Genetic Algorithm

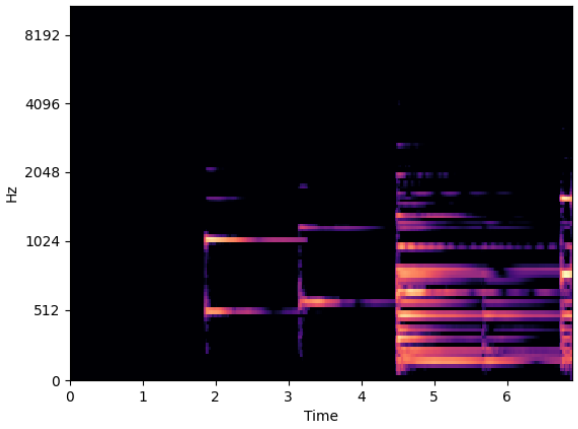
In the process of developing our genetic algorithm, in order to achieve the best results, we performed a few experiments. We tested various parameters and configurations of the algorithm, a few fitness functions, 2 methods of reproducing a new individual from 2 existing ones, and a couple of ways of choosing which individuals will continue to the next generation. The main way to verify the success in this challenge (other than simply listening to the audio output) is to compare the spectrograms (see '*Audio processing - brief, simplified basics'*) of the input and the algorithm's output.

All results created use the same example input audio, a 6 second piano track beginning silently, then 2 single notes are played consecutively, and finally a chord (a few notes together) is played. The building blocks are always the notes of 7 piano octaves; they're not the same recording as of the input audio recording, making our task more challenging.

For each test we generally ran the algorithm several times and chose the runs with the highest final fitness scores. This is done to obtain relatively meaningful and consistent results despite the random nature of our algorithm. Using the average fitness across runs would have been possible, but as we wanted to also show the generation progression plots we preferred selecting a single run and not just a fitness value.

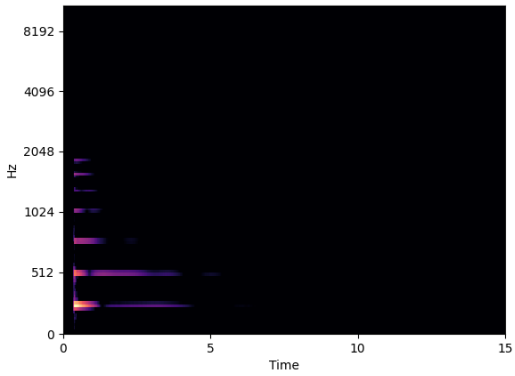
Note that color is always in decibels, and the precise values aren't important for our report's purposes. Fitness absolute values are practically meaningless, while their relative values do hold meaning.

The following is the spectrogram of the input audio:

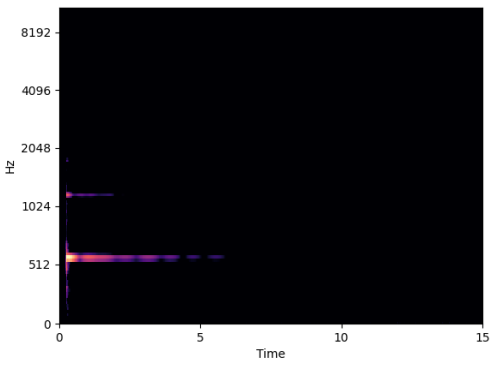


Input track spectrogram

And these are spectrograms of two of the building block notes we used:

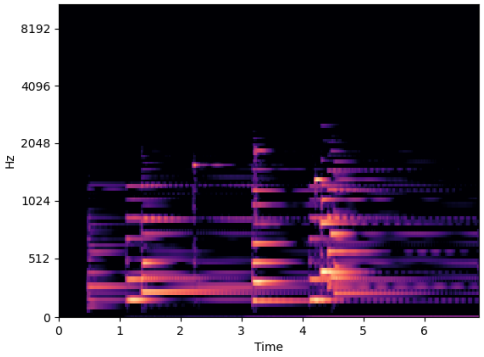


C4 note spectrogram

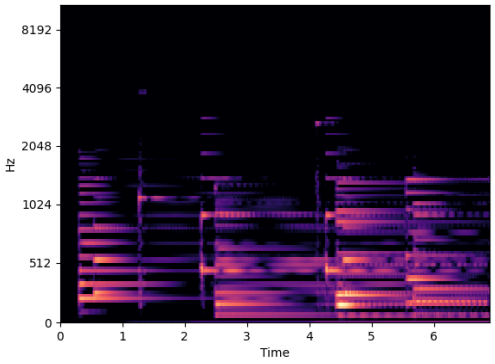


D5 note spectrogram

1. Final Results

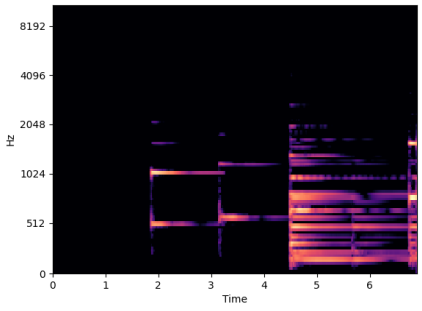


Random tune 1 spectrogram

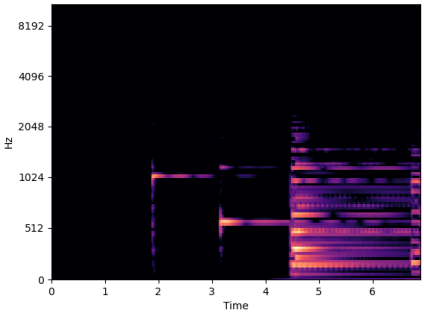


Random tune 2 spectrogram

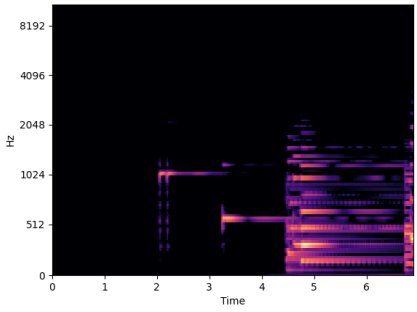
Above are spectrograms of randomly created tunes generated for the first generation of the algorithm.



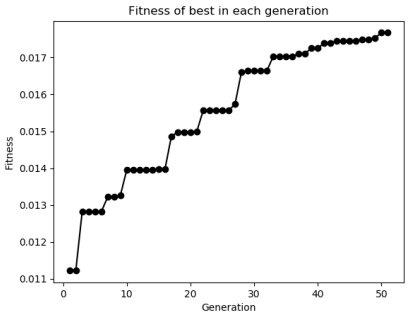
Original (goal)



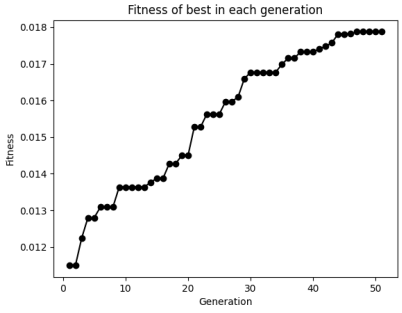
Results 1



Result 2



Fitness plot result 1



Fitness plot result 2

Above are rinal results demonstrating some of the best runs of our algorithm.

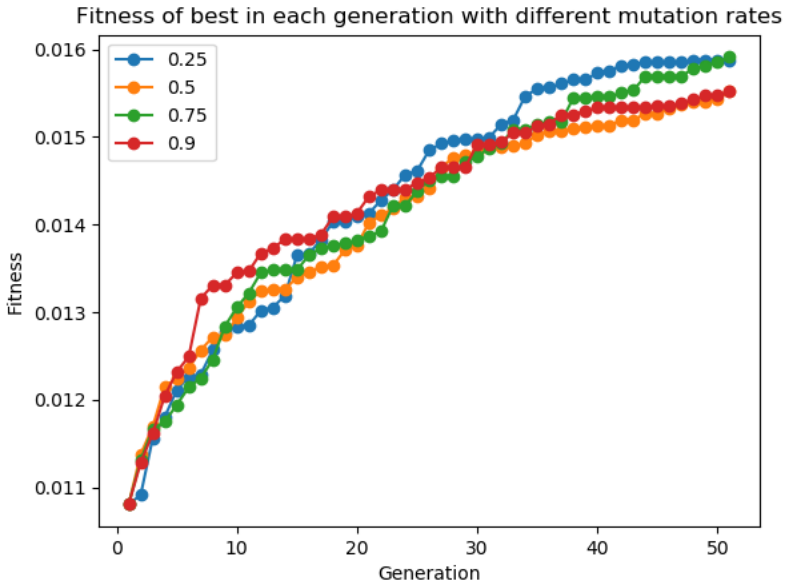
The spectrograms suggest a nice fit to the goal audio and a vast improvement over the random tunes. The fitness curves indicate a healthy improvement process through generation progression. Listening to the produced audio, it is clear that

our algorithm indeed came very close to the goal - silence is kept nicely, the single notes and chord are well positioned in time, the single notes are quite correctly chosen, and the chords bear a clear resemblance in sound to the original (we are not musicians so it's hard to tell).

1. Experiments

A. Mutation Rate

One of the key elements of a genetic algorithm is individuals going through mutations, helping the algorithm avoid local maxima. Without mutations, the algorithm would be only reproducing with individuals with characteristics derived from the original random population it began with. The parameter of mutation rate defines the probability of a new individual undergoing a mutation, causing it to change some of its 'genes' (the note building blocks in our case). Lower mutation rate - less mutations, and vice versa.

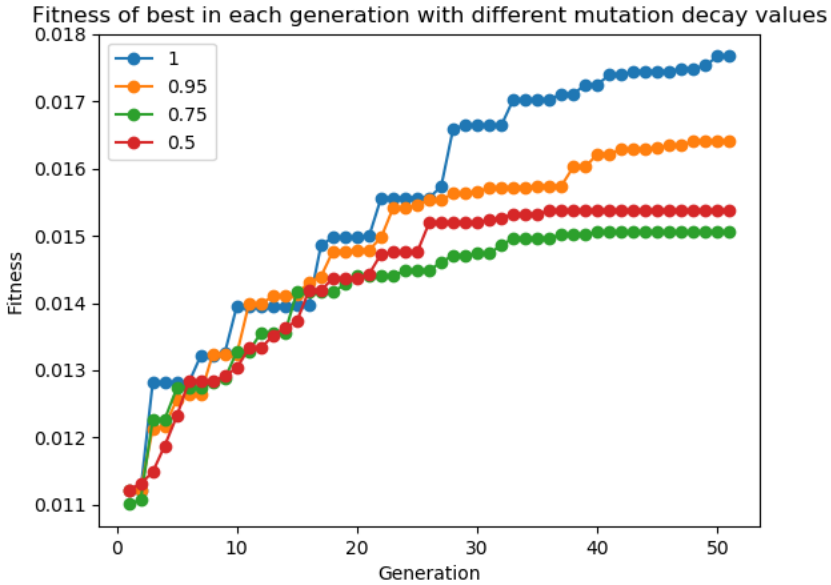


Results of various mutation rates we tested

These results suggest that all tested mutation rates are sufficient for the production of equivalently good outcomes and the avoidance of blatantly lesser local optima. We did not test lower mutation rates.

B. Mutation Decay

Another factor regulating mutation is the decay rate. As the algorithm proceeds through its iterations, having better and better populations by making many turns in new directions, it might be a good idea to gradually cause less mutations and only try to poke around looking for small changes. For this we have the mutation decay factor. At the end of each iteration, the mutation rate is multiplied by mutation decay factor (mutation_decay $\in [0,1]$), causing the mutation rate to decrease. Lower mutation decay factor - faster mutation rate decrease (usually causing stagnation of fitness of the best individual of each generation), and vice versa.

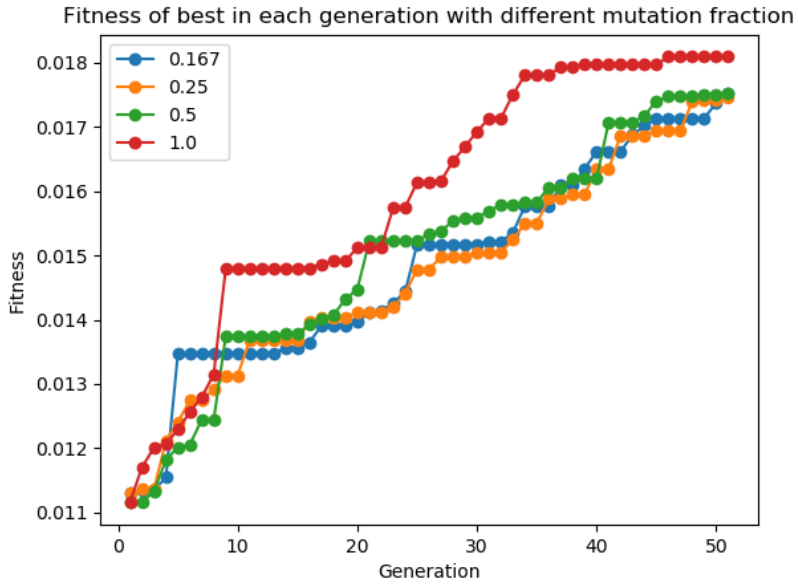


Results of various mutation decay rates we tested

From this experiment we got some expected and some unexpected results. As expected, lower mutation decay caused early stagnation (flattening of the curve). No mutation → reproducing within a population with limited genes, stopping at some local maximum, and specifically in our case - tunes with some given notes without being able to add/remove/change those notes. Unexpectedly, we see that using a mutation decay of 0.5 got better results than results obtained when using a factor of 0.75 despite reaching stagnation first. Our explanation to this is that thanks to the random nature of the mutations sometimes there is simply a great individual(s) created causing a big increase in the fitness of populations. Using larger population sizes will increase the probability of this phenomenon. On the other hand, according to the law of large numbers, repeating this experiment many more times (more than we did), should correct this observation. In any event, our results don't suggest an advantage associated with the use of a mutation decay rate, as keeping it set to 1 provided by far the best results.

C. *Mutation* Intensity

The last factor of mutation we tested was the mutation intensity, i.e what fraction of the individual's genes (building blocks) were mutated. Our mutation function works by randomly selecting some of the genes of the individual and changing them as discussed above. The amount of genes that are selected is on average AMOUT_OF_GENES * MUTATION_FRACTION. Intense mutation will change the individual substantially, and many substantial mutations may completely change the population. The larger the fraction - the more intense the mutation, and vice versa. Here are results of various mutation rates we tested:

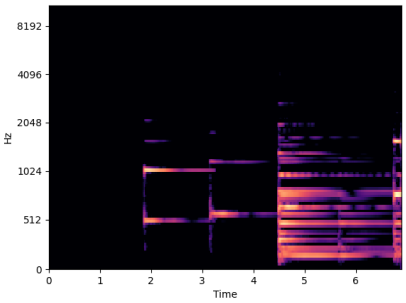


With the mutation fraction set to 1.0 (red curve, very intense mutation) we catch an extreme increase in fitness between some couples of consecutive generations. This might be due to a terrific mutation, but it can also be explained by a successful reproduction of two complementary existing individuals.

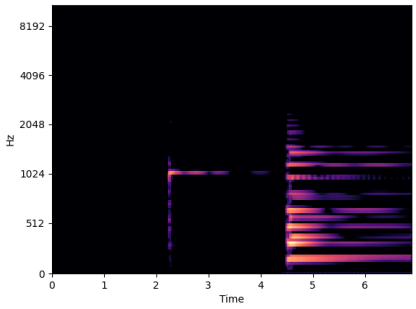
Moreover, there are also some giant leaps with the lowest mutation fraction we tested (the blue curve) which interferes with our suggested explanation.

D. Population Size

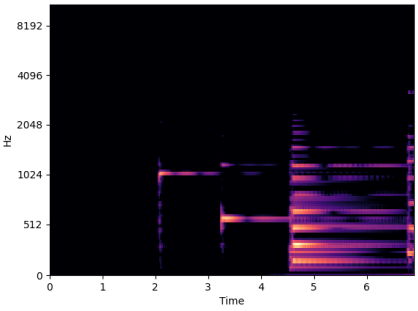
A key factor for the success of the genetic algorithm is the size of the population. Larger populations' potential for success is greater (more individuals to choose from), but it on the other hand significantly increases the run time. We tested various population sizes and compared final results and runtimes:



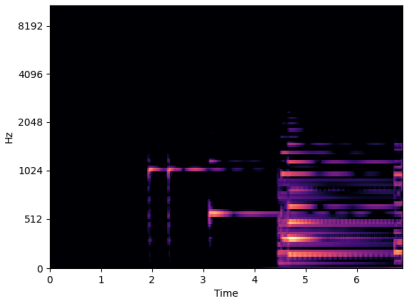
Original (goal)



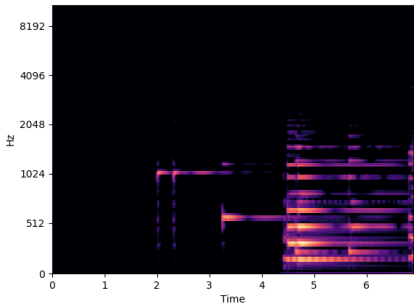
Population size 20



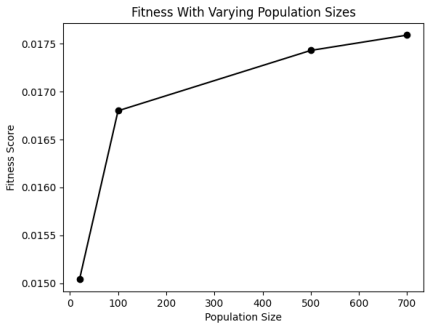
Population size 100



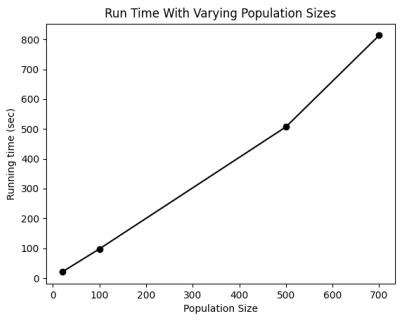
Population size 500



Population size 700



Fitness

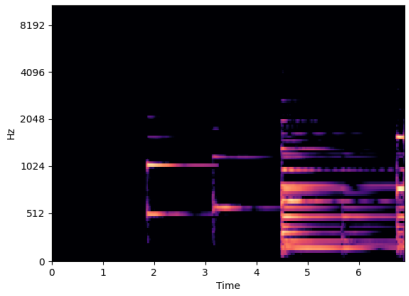


Running times

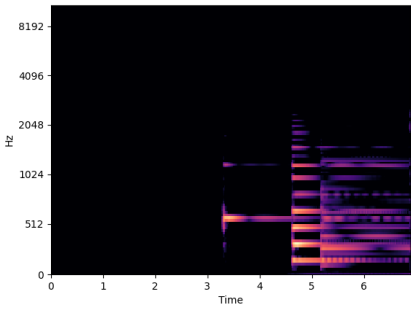
As expected, higher population sizes produced better results. Even with smaller sizes (~150-250), some runs had decent results, but for more stable and consistent results we must raise the population size to about 500. As expected, large populations require longer runtimes, but the difference between results with population sizes of 500 and 700 were not significant while the runtime with 500 was 30% faster. Note that the reported running times do not include the loading of the audio files.

E. Methods for Selecting the Next Generation

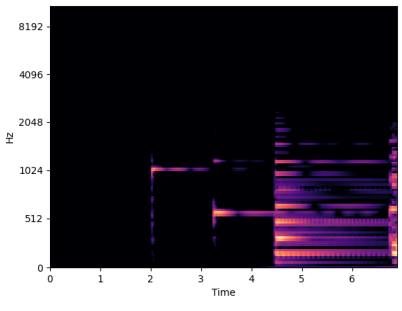
In each iteration, the existing population of size *N* creates *N* new offsprings. We tested 2 strategies for deciding which individuals will be in the population for the next iteration. Strategy 1 is to simply take only the newly created offsprings, and to discard the individuals from previous generations. We thought this method might lose good individuals, maybe even ones that are better than all the new offsprings created. So we implemented a second strategy in which we combine the parents and offspring populations in one pool, reserve a percentage of the best individuals with certainty, and use a weighted random to select the remaining.



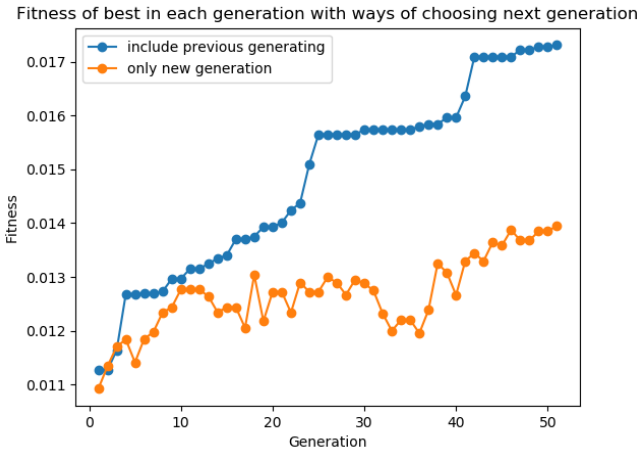
Original (goal)



Only new



Best from old and new



The difference here was significant. As seen in the plot on the right, using only newly created tunes had unstable results and ended with much worse results.

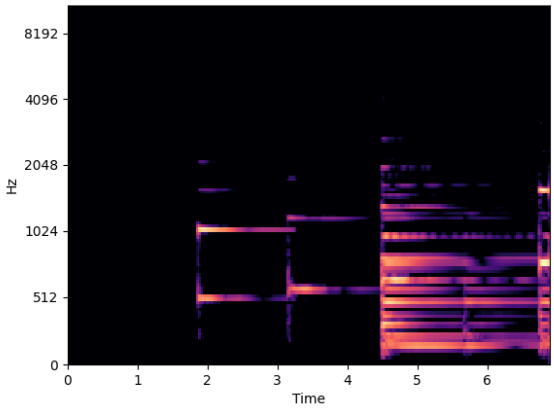
F. Reproduce Function

We compared the two reproduction functions ‘random selection’ and ‘split’ (implementation details in the ‘methods’ section).

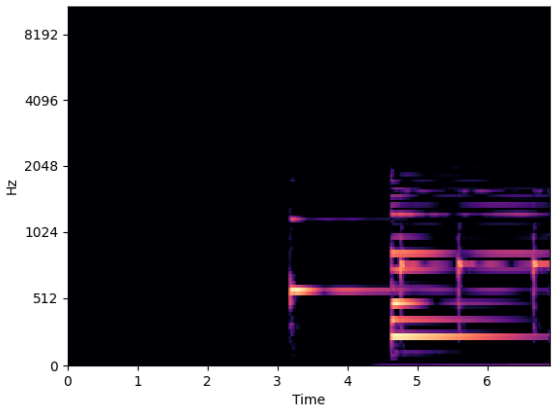
Testing the algorithm multiple times, there wasn’t a method that clearly outperformed the other. Maybe with different settings we might reach a different conclusion, but in the current situation choosing either of them is good.

G. Fitness Function

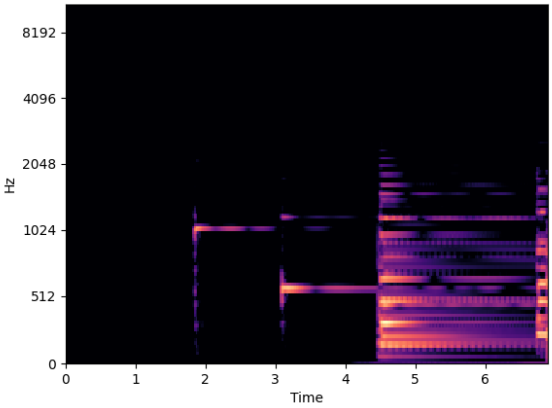
As described in the ‘methods’ section, we created several fitness functions. Here we compare between the results they achieve (fitness values are obviously meaningless here):



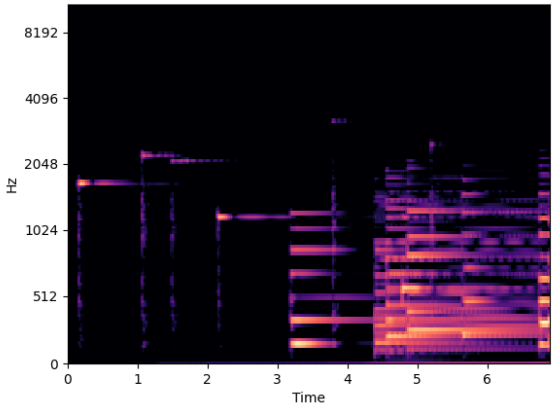
Original (goal)



Signal Euclidean distance



Mean mel Euclidean distance



ZCR RMS

As seen above, the mel spectrogram fitness function was proven to achieve best results. It’s run time is significantly higher than the others, but the results were in a totally different ballpark, especially when listening to the audio.

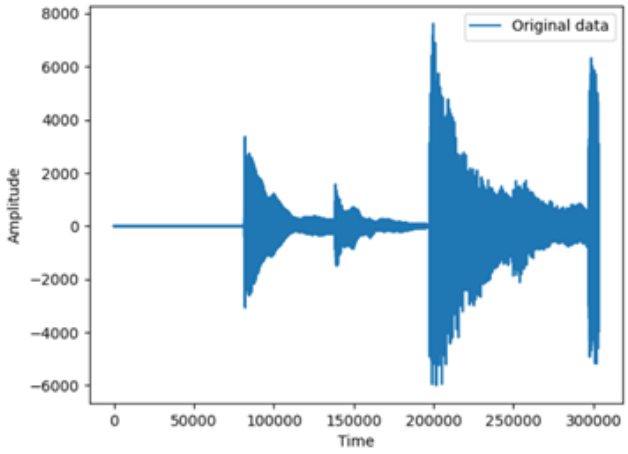
H. Additional Attempted Tests

We tried having Google’s song recognition guess our result’s melody name, but it failed repeatedly. Using more well known melodies such as Fur Elise (Mozart) didn’t help. Analyzing longer audio excerpts resulted in quite bad outputs which didn’t stand a chance of working.

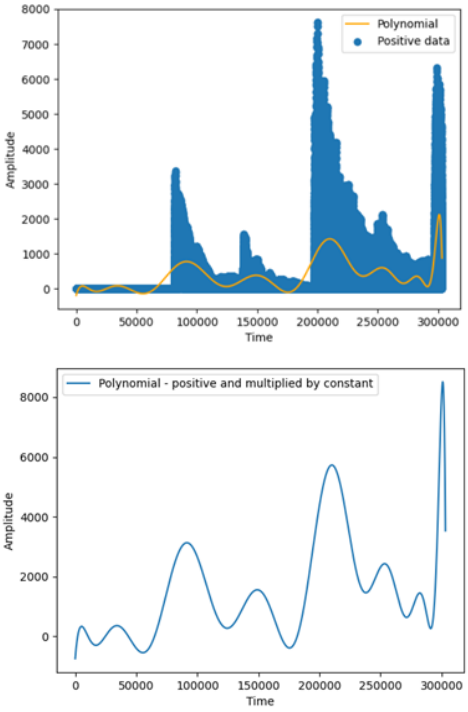
We tried using input audio that contains instruments other than a piano while still using the same piano building blocks. Conceptually it should work, but we didn’t get good enough results. The same goes for longer audio (6 seconds of a simple melody is not much) - the outputs weren’t satisfactory even though it could work. Perhaps using a lot more computational resources could help, and optimizing the fitness function might improve the results greatly.

V. Results - Polynomial Fitting

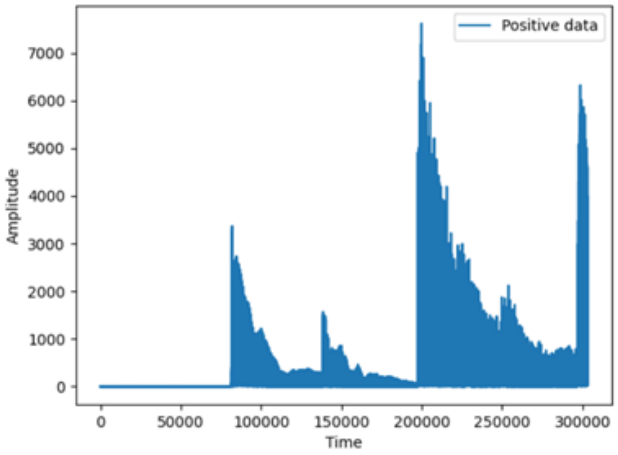
A. Fitting process



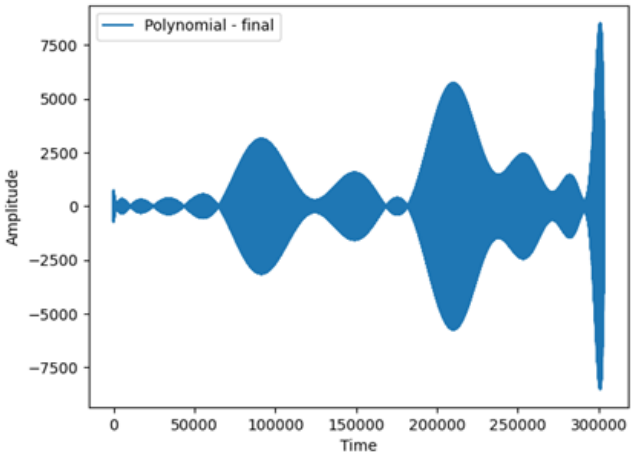
Input audio signal



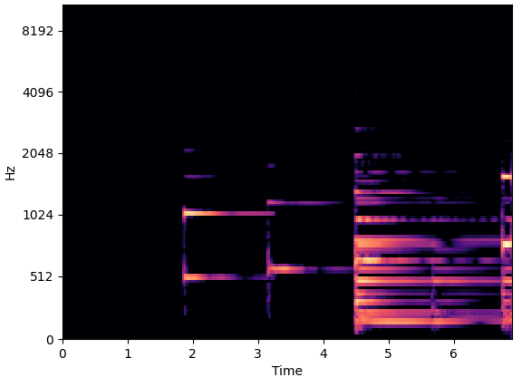
Polynomial fitting without & with amplification



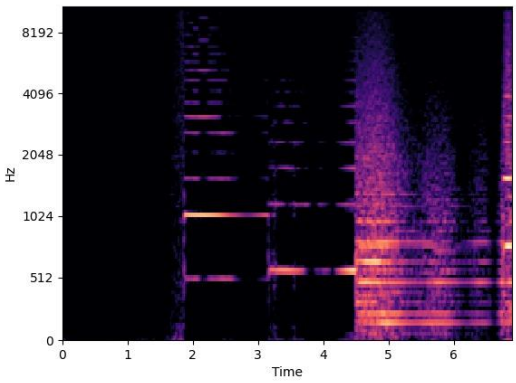
Absolute signal



Final result after restoring negativity



Original (goal)



Result spectrogram

Timing is nicely aligned with the original audio spectrogram, and the prominent frequencies are present. There is a lot more going on than in the genetic algorithm results, which makes sense given this technique isn't limited to existing building blocks.
We will refrain from making bold statements about the specific details seen in the spectrogram (the pyramid-like structures, patterns rising through the frequencies) as it demands in depth knowledge in audio which we lack.

Listening to the resulting audio, it sounds just like the input audio with added noise and with an electronic vibe to it. The longer the audio, the stronger the noise.

Here too, we tried having Google's song recognition guess our result's melody name, but although we had higher hopes for success using polynomial fitting it failed as well.

VI. Extra Discussion

A. *Our Improvement Process - Genetic Algorithm*

- Natural selection - at first we discarded the parent population and only used the offspring as the next generation. We figured this often doesn't preserve the good individuals, so we pooled parents and offsprings together, selected the best quarter with certainty, and used a weighted random to select the rest.
  We realized that as the differences between fitness scores may seem slim when used as probabilities, we might be able to select better results while allowing down-hill movement by using the softmax function as weights.
- Reproduction - our initial reproduction function was 'random selection' (in the Methods section), but at some point we realized it's more 'natural' for the genetic algorithm to divide the tune based on some time stamp and take each part from a different parent.
- Fitness function - the first fitness function we used was the 'signal Euclidean distance'. This is a naive attempt, not taking into consideration the most common way of approaching audio - frequencies. We made the transition into frequency space, and when we found out about the mel scale we decided it would be beneficial to use it.

B. *Our Improvement Process - Polynomial Fitting*

- Absolution and negation restoration - we quickly came to the understanding that using both positive and negative parts of the signal for the polynomial fitting wouldn't get us anywhere, as there is too much crossing of the x axis.
  Deciding to use absolute values, we knew we had to restore negativity. We tried making every other value negative, but that produced sharp squeaking noises rather than a proper melody. Eventually, restoring the original sign to each value proved to be a decent solution.
- Amplification - the result was at first too weak, and it was difficult to distinguish between notes. A simple multiplication by a constant improved the results nicely.

C. *Thoughts About Our Method - Genetic Algorithm*

- We believe having each note in both the input audio and the building blocks be lengthy instead of momentary was beneficial. If it were otherwise, finding the right offset for a building block to fit some note played in the input audio would be much more difficult: there would be no indication whether the block is 'close' (in time) or not.

D. Ideas for Future Improvements - Genetic Algorithm

- Try a different instrument for the input audio and the building blocks, say, a guitar.
- Try building blocks created by a different instrument from the input audio, or even building blocks made up of chords or recordings of human speech. Our code supports this feature as it is!
- Allow the genetic algorithm to also mutate the amplitude. We actually have code written for it but didn't get a chance to use it.
- Systematically run the algorithm several times and choose the best as the output. This would require additional running time, but would produce more consistent results.
- Create new reproduction methods - e.g. creating several offspring and immediately choose the best.
- Improve the fitness function - take into consideration frequency proximity (currently if two frequencies don't match then it doesn't matter how close they are).

E. Ideas for Future Improvements - Polynomial Fitting

- Use a higher degree polynomial.
- Fit polynomials to the positive signal and to the negative signal separately, and combine the results for sampling.
- Fit a different kind of function, for example a sequence of sines/cosines.

F. Note on the Two Solutions

As we were working on the project, we got the feeling that the genetic algorithm solution allows us to emphasize more the concepts we learned in class, more so than curve fitting. This feeling was strongly supported by staff member Reshef's input on the matter. Therefore we consciously decided to invest more in the genetic algorithm, both in implementation and in result analysis.

Acknowledgment

We thank staff member Reshef for being available to us for consults and for approving our project.

References

[1] The source of our test input audio: https://freemusicarchive.org/

[2] The source of our individual piano notes: http://theremin.music.uiowa.edu/MISpiano.html

[3] Tool for converting AIFF to WAV: https://www.freeconvert.com/aiff-to-wav

[4] Blog about mel spectrograms: https://towardsdatascience.com/learning-from-audio-the-mel-scale-mel-spectrograms-and-mel-frequency-cepstral-coefficients-f5752b6324a8

[5] Blog about mel spectrograms: https://towardsdatascience.com/getting-to-know-the-mel-spectrogram-31bca3e2d9d0

[6] Article about the relevance of the mel spectrogram to music analysis (and not just speech): https://ismir2000.ismir.net/papers/logan_paper.pdf

[7] Tool for quickly visualizing frequencies: https://academo.org/demos/spectrum-analyzer/

[8] Google's tool that recognizes the identity of a hummed/played song: https://blog.google/products/search/hum-to-search/