Modular Music Generation Using NEAT
IT University of Copenhagen
Bachelor Thesis

Nikolaj Daugaard Olsen
Hans Emil Almegaard Hansen
Supervisor: Marco Scirea

May 15, 2017

Abstract

In this thesis, we experiment with NEAT and music generation. We examine how a system with a number of modules that accompany a melody and each other can be created. Moreover, we examine which fitness rules can be formulated.

We construct these fitness rules with different properties and purpose, where some of them encourage following basic music theory and others concern themselves more with how the modules should interact with each other in a larger scope. We evaluate different ways the modules can relate and listen to each other, as well as what it means for the result, when we go from two to an arbitrary number of modules.

We conduct a survey where we present the participants to clips generated using a combination of different fitness rules to see what effect these rules have in combination in four different categories - *pleasing*, *harmonious*, *random* and *interesting*.

Finally, we discuss our results and that we found out that it is possible to create an arbitrary number of modules that can relate to each other, but that it gets slower and increasingly more difficult to get a high fitness score the more modules you create. We also found out that the combination of rules and the weighting of them have different effects in the four categories. For more *pleasing* and *harmonious* results, the modules should stay within scale, avoid dissonance, avoid uniqueness and distribute playing time. *Random* results can be obtained better by varying often in pitches. We found it a lot harder to deem which of the rules that made the clips more *interesting*, but there seems to be a correlation between *pleasing*, *harmonious* and *interesting*.

Contents

1 Introduction

This project is built from an idea by our supervisor Marco Scirea, where an arbitrary number of robots would accompany a musician playing an instrument, through monophonic melodies that fit the music. It leads to them not only having to listen to the person but also each other, which ultimately would create a coherent piece of music. In this thesis we will simulate this process in the computer. Additionally we will not focus on the real-time aspect, but limit ourselves to a given melody that will act as something a person has played. The abstraction for each individual robot will throughout this paper be called *a module*.

The modules will need to learn from each other and respond to different situations and for this we will use neural networks. The neural networks will be trained with the genetic algorithm NEAT through a simulation of evolution where they increasingly adapt to some fitness rules that we create. Genetic algorithms are chosen for this problem, since they are able to create non-deterministic and varied solutions, that fit the somewhat unpredictable nature of music. This way the modules will hopefully adapt and create interesting, fitting results.

The first part of the thesis will cover the technical aspects of creating the arbitrary number of modules. The second part of this thesis will deal with fitness rules, and how we can bring out certain characteristics in the generated music.

To summarize we will in this paper try to answer the following research questions:

- *Is it possible to create an arbitrary number of music modules that are able to listen to a given melody as well as each other to create a coherent piece of music?*
- *Which fitness rules can we formulate to generate music, and how do these affect the output?*

To answer these questions we will begin in the following section 2, *Background*, by shortly talking about other computer generated music solutions that are out there. Later in that chapter we will mention the core concepts and theory that is the foundation behind our project and how music is represented in this thesis. In section 3, *Modules*, we will talk about the implementation of modules, the challenges faced and the solutions found. In section 4, *Fitness Function*, we will discuss which different kinds of fitness rules we have created to train the modules. In section 5, *Analysis*, we will analyze the general performance of our program through a series of experiments. In section 6, *User Survey*, we will conduct a survey with different generated music clips that follow different fitness rules to measure what effect people think it has. Finally, in section 7, *Discussion*, we will discuss our results and how they corresponds to what we set out to accomplish.

Some parts of the report are accompanied by music examples. These can be found at `http://goo.gl/Rkr9vg`, and will be referred to when relevant.

2 Background

In this section we discuss the knowledge required to understand the rest of the report. This includes information about the algorithms used, the choices made in regards to representation of music and the state of the art that this report leans upon. While music theory is a part of the justifications behind choices made regarding fitness functions, it will not be discussed in this section, since the rules depended on are fairly basic. Instead, they will be discussed briefly when describing rules in section 4, Fitness Function. Reading the thesis without prior knowledge of music theory is still possible.

2.1 State of the Art

For almost as long as computers have been around, people have tried to use them to generate computer music, also called algorithmic composition. Over time, many different techniques have been developed and used to try to achieve this endeavor ranging from statistical models, knowledge-based systems, grammar-based models, evolutionary methods, learning systems or hybrid approaches. [1]

In the category of genetic algorithms, there is a project originally created by Amy Hoover called MaestroGenesis. It uses an algorithm called Functional Scaffolding for Musical Composition (FSMC), which is built upon on an approach called NEAT Drummer, which uses NEAT. MaestroGenesis is capable of taking an input MIDI melody and generate different modules of accompanying music to it, where the user can selectively breed further the modules of his/her choice [2]. Our project will be heavily based on her project.

Another approach in algorithmic compositions is learning-systems where Google quite recently, in June 2016, released Magenta, with the main purpose of creating compelling art and music [3]. Magenta uses Tensorflow, a machine learning library also made by Google, that allows them train their models with a large dataset of sampled songs in order to learn general patterns that occur. An application of Magenta is Google's experiment A.I. Duet.[4] It allows you to play a duet with a Magenta model, where you in real time can play on a piano and have the model respond with accompanying music. Not quite unlike what we hope this project in the long run also will be able to do.

These are only some of the many different algorithmic composition projects that you can find that continues to grow and improve as technology evolves.

2.2 Artificial Neural Networks

An artificial neural network (or ANN) is a network which tries to simulate how the neuron network of the brain functions. It consists of a number of input nodes (or neurons) that connect to a number of output nodes. Between the input and output nodes there might be a number of hidden nodes (See figure 1) depending of the complexity of the network.

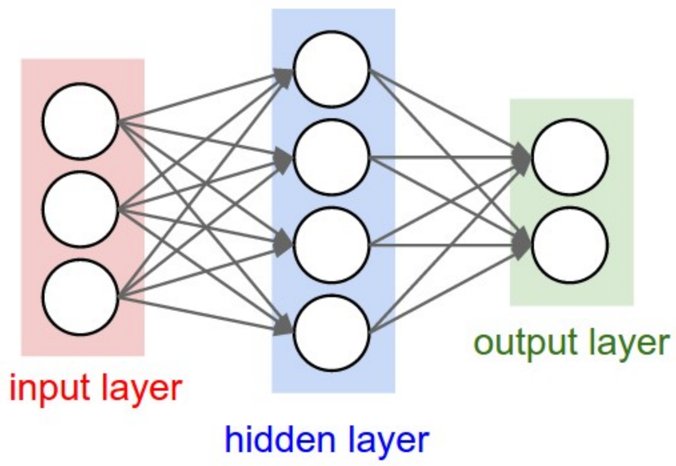


Figure 1: Model of simple a simple artificial neural network with three input nodes, two output nodes and four hidden notes [6].

The main purpose of the ANN is to learn to recognize patterns, so that it later knows how to behave when it has to make a decision. This is done by using weighted connections between the nodes. With a given input, the ANN will add the weights together to decide what the resulting output should be using an activation function. In order to get closer to a desired output, the weight of the connections are adjusted iteratively through a training process [6]. NEAT has its own way of creating the network which will be described in the following section.

2.3 NEAT

NEAT stands for Neuroevolution of Augmenting Topologies and is a method for creating and evolving artificial neural networks using genetic algorithms. It aims to simulate evolution as we know it from nature where, it begins with a simple network that evolves and becomes evermore complex over multiple generations. In this project we have used SharpNeat which is a complete implementation of NEAT written in C# [8].

NEAT generates and improves its own ANN, and is often quicker at arriving at effective networks than other neuro-evolutionary techniques [9]. It adjusts the weight of the connections by using a fitness function. Every generation NEAT will create a population of children (the genomes) that each, through small random variations, presents a solution to how the weights of the ANN might be distributed. We then decode the children into phenomes so that their properties can be analyzed, and then decide which one of them has evolved closest to a desired output. This is done in a fitness function, and the final score is called the fitness score. The best children will form the base for the next generation of children that is created. Through this process we will hopefully get closer and closer to the best possible trained ANN.

Another approach to the problem of evaluation phenomes, would be having a trained model based on a dataset containing desired outcome. The model would then be able to evaluate the output. A challenge of going this route is the availability of big datasets of music in appropriate formats.

In our case, what we give as the input is a music abstraction, which will be described in the following section.

2.4 Music Representation

A central issue when creating a program that deals with music is the representation. Ours is based on the one used by Amy Hoover, detailed in the paper *"Functional Scafolding for Musical Composition"* (Amy Hoover, 2012) [10]. The program is comprised of modules, which relate to the given input melody and each other. Every module consists of two disjointed ANNs - one for pitch and one for rhythm. Evolution is also separated into two separate algorithms, that do not interfere with each other. When creating the output, the pitch algorithm then controls which note is played, while the rhythm output controls when it is played.

Pitch is represented as integers in the range from 0 to 24, each representing its own semitone. 0 is defined as being the tone C4, and each semitone is laid out in consecutive order to 24, which therefore is the tone C6. This allows the modules two octaves of range. Figure 2 shows the an illustration of a piano with the pitches respective integer representation. A new pitch is played for every tone that the input melody plays.

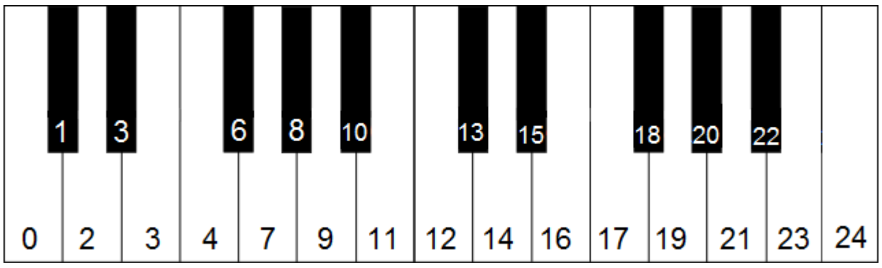


Figure 2: The available pitches and their respective integer value.

The rhythm representation is divided in a unit of time called ticks. How long a tick lasts is defined by the user. For a MIDI file this value is stated as *PPQN* (or *Pulse Per Quarter Note*), with a minimal possible value of 24. If the project were to run in real-time, this would be a desirable value. For our project, however, we have mainly been running on 2 *PPQN*, which corresponds to an eighth note.

Rhythm is represented as decimals between 1 and 0, where 1 represents the beginning of a new note. The decimals then linearly decay for each tick until it reaches 0. A positive peak then represents a new note. Similarly, a negative peak represents a rest. This creates a sawtooth representation, where a new note is a peak and the sustain is the steady decay. Figure 3 shows an example of how that might look.

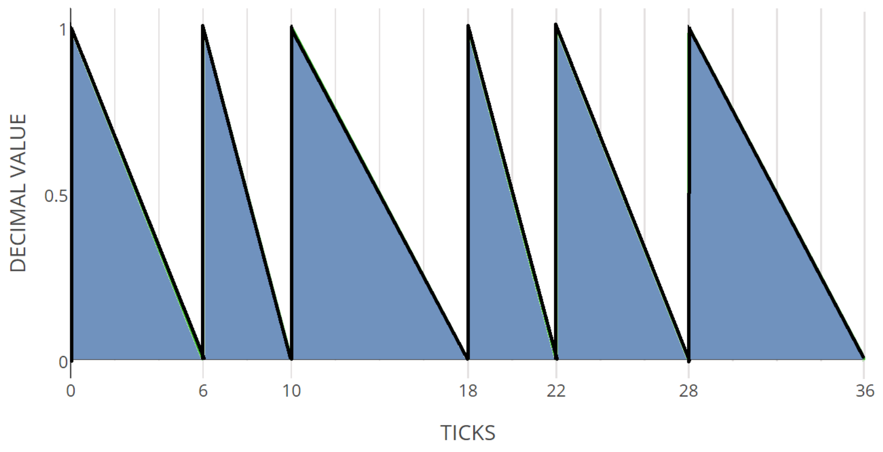


Figure 3: Note length representation where a note starts at 1.0 and then decays until a new note starts.

3 Modules

A central part of the project is the abstraction of modules. One challenge has been expanding the given code from handling two to N (an arbitrary number) modules. It created a number of technical problems, which lead to a couple of choices. Among these were the challenge of how the modules should relate to each other, and how each individual genome should be evaluated. These will be discussed in the following section.

3.1 Relations Between Modules

When generating the output for a tick, we want to listen to the input and the other modules at the corresponding tick. Ideally, each module knows what each other plays, so it has a wider picture of the output to adapt to. However, this creates a circular dependency, since each module would need all other modules to finish before it can start, which would leave all modules waiting for each other. Therefore we need to find some other model for relations between the modules.

During the project, we have tried two different configurations, which are illustrated in figure 4 and 5.

In the first, each module listens to the modules that come before it, i.e. *Module 2* listens to *Module 1*, *Module 3* listens to *Module 2* and *Module 1*, and so on. Additionally, each module listens to the input melody. This configuration will hereafter be referred to as *cascading*. Figure 4 shows how the input to the neural network of each module is the input melody and the output of the module that came before itself. This is the case with both the neural network of the rhythm as well as the pitch. The number of inputs to the neural networks across the modules must stay the same. Therefore, the available slots in *Module 1* and *Module 2* will be filled with a never changing, static value that hopefully over time will have very little influence on the weights in the final generated ANN's.

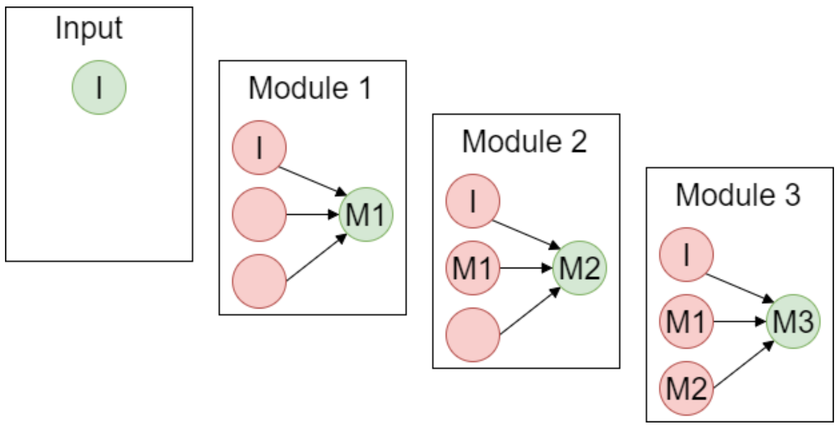


Figure 4: Simplified illustration how cascading works. Red nodes symbolize input and green nodes symbolize output. Every module can see the input melody but can only see the output of the module before itself.

In the second configuration, each module listens to what all other modules generated in the tick before. This creates a bit of delay, but allows all modules to relate to all the others. The difference is less when generating the rhythm than for the pitch, since the decay between ticks is usually smaller, while the pitches always have a discrete difference. This configuration with be referred to as *delayed*. Figure 5 shows how the pitch tick output for *Module 3* is calculated by looking at the current tick from the given input melody, but the previous tick from *Module 1* and *Module 2*. Instead of their integer representation the pitches are here showed with their actual name. While this example only demonstrates pitches, the same principles applies to the calculation of the rhythm output as well as to more than three modules.

It might seem like this configuration would cause the modules to always be one step behind, and therefore have a risk of being dissonant or offbeat. However, since the evaluation of the genomes happens after the complete output has been generated, and since the input melody remains the same, the output is still evaluated at the right time. This means that the selected responses are always ones that generate output with the correct timing, and therefore the delay is not noticeable in the final generated clip.

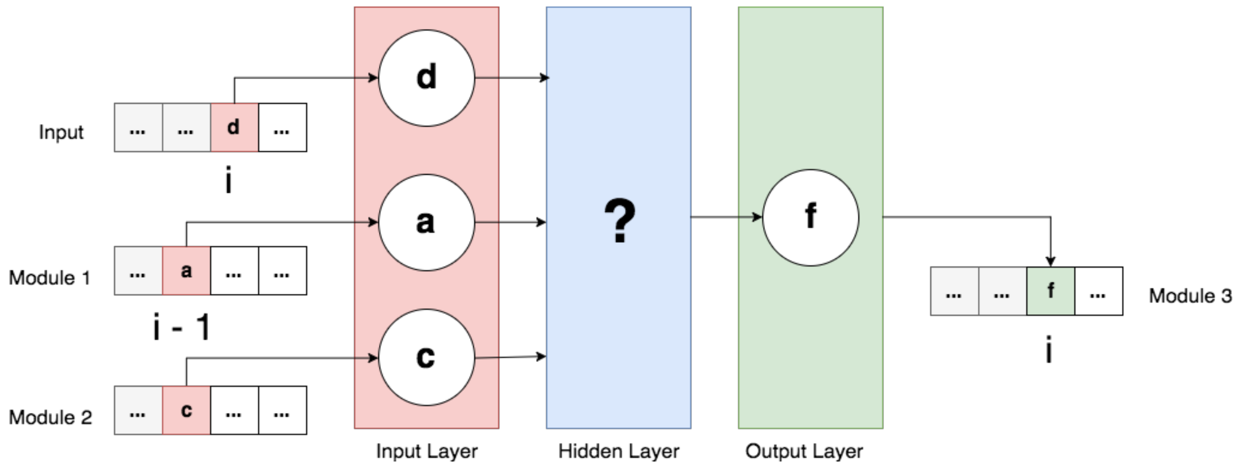


Figure 5: Simplified illustration of the pitch neural network of *Module 3* and how the input is chosen from the input melody and *Module 1* and *2* with a *delayed* configuration.

Both configurations have pros and cons. In the cascading configuration, the modules listen to what is played at the same time that it is generating for. However, some only get a smaller part of the overall output. In the delayed configuration, all modules gets input from all other modules, but with a one-tick delay. For rhythm this is not as critical, since the difference between rhythm-ticks are typically small. However, for pitch, where the output is discrete, it is a bigger disadvantage.

In practice, we experienced that the delayed configuration produced the most interesting results. This is probably due to the modules having a more full picture, and more variation in its input. Additionally, the modules sometimes generated nice little interludes in melody breaks, where they play off of each other instead of just the input melody, which does not appear in the same manner in the cascading configuration.

3.2 Selection of Genomes for Evaluation

In each generation, you want to evaluate all genomes from one ANN against some from the other modules to find the genomes that have evolved the best. Ideally, each candidate should be evaluated against all candidates from other modules. In practice, however, this would be too many calculations to perform.

Instead, each candidate is compared to a user-defined amount (typically between 3 and 5) sets of other candidates. These consists of the best of the previous generation, with each set containing one genome for each module. The selected genomes are chosen between the best of each specie. Since the chosen candidates are among the best, they are some of the most important to evaluate against, since that genetic material is what is bred upon for the next generation. Compared to evaluating with just one set of parasites, this solution provides a more robust evaluation. Figure 6 illustrates how *Module 1* in this example compares its first genome with the two best genomes from *Module 2* and *Module 3*. *Module 1* will iteratively compare all of its genomes, and the genome that obtains the highest fitness score will be used for future comparisons. This process happens for both the pitch genomes and the rhythm genomes of each module.

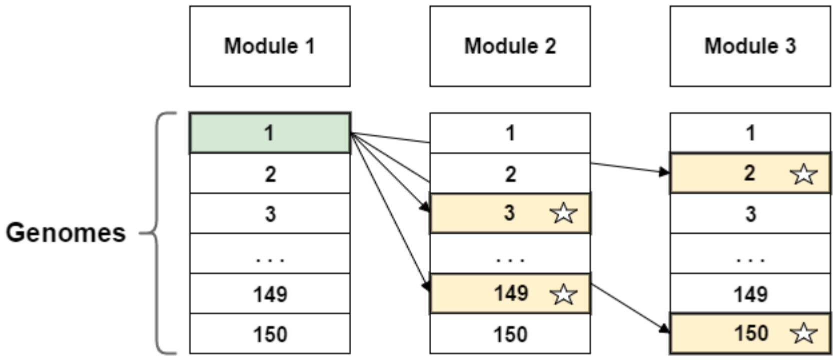


Figure 6: Illustration of how the modules iteratively compare their genomes with N number of candidates from the other modules. In this case it is comparing with two from each module.

Additionally, our program also contains a *Hall of Fame* which saves the best genome from each generation. This is done so that the current population doesn't only have to compete with each other but also a selection of previous winners. Since we have a small population of only 150, old gains can quickly be forgotten and using a *Hall of Fame* structure might help ensuring that old working strategies can keep being built upon [7].

Now that we have talked about how modules relate to each other, we will now in the following section describe how each individual genome is evaluated using a fitness function.

4 Fitness Function

In this section we will talk about the fitness function, which is used when evaluating how well a genome has evolved and will as a consequence shape how the music should develop further. The fitness function is comprised of a set of rules, which each evaluate the output on some criteria. Each time a genome is evaluated, it begins with a perfect fitness score, from which points are deducted for each rule it breaks. The following section describes which rules we have created. Additionally, it contains a description of the harmonization function, which supports the rule for playing chords.

The rules we create are a mixture between general music rules each module on its own should follow, but also rules describing how the modules should relate to each other. Since the main focus of this project is not directly about music, but more about genetic algorithms, music theory will not be covered extensively. We will however, as we in the following sections go through the different fitness rules, describe their correlation with music theory and the underlying ideas.

4.1 Pitch Fitness Rules

The following section describes the rules that we have formulated for evaluating the pitch.

- **PitchDifference** Every time a pitch is generated that is not the same as input melody a penalty point is deducted. This is a way to check whether the neural network is able to mimic the input, and therefore if evolution is working.
- **OutOfKey** Every time a pitch is generated that doesn't appear in the given song key, a penalty point is added. Even though it is not uncommon in western music to expand to pitches beyond the given key, we will in this program strive for pitches only within the given key for simplicity.
- **SameNote** Every time a pitch repeats itself two times in a row, a penalty point is added. The purpose of this is to create varying pitch sequences so that the music does not become too boring.
- **IdentityInput** Every time a module plays the same pitch as the main melody, a penalty point is added with the purpose of creating sequences that don't just copy the main melody.
- **IdentityModules** Every time a module plays the same pitch as the other modules, a penalty point is added. This is done so that the modules get their own identity and don't just play the same. Of course with a high enough amount of modules it becomes impossible for each module to play a distinct pitch.
- **DissonanceInput** Each time a module plays a pitch that is one or two steps away from the main melody, a penalty point is added. Even though music can and often has pitches bordering each other, like in seventh chords, this simple rule is avoiding all kind of dissonance. A more advanced function might be able to separate bad dissonance from good dissonance, which would help create tension in the music [11].
- **DissonanceModules** Each time a module plays a pitch that is one or two steps away from the other modules, a penalty point is added. Like the rule before the idea was to try and

avoid dissonance. However with more than two modules this will conflict with *OutOfKey*, as there is only a limited amount of pitches.

- **TooBigGap** Every time a gap larger than 7 semitones is present between two pitches in a generated sequence, a penalty point is deducted. The idea is here to keep the modules from jumping back and forth between two octaves which might sound weird.
- **Chord** If a tone is not in the currently desired chord, the genome is penalized. The chord sequence is generated by the harmonization function, which is described in section *Harmonization*. This rule encourages the modules to create triads, which helps the output in sounding harmonious.
- **BelowInput** Every time a pitch higher than the corresponding input pitch is chosen, the module is penalized. The idea is to have the melody be the leading pitches, which is a good general rule for supporting a melody.

4.2 Rhythm Fitness Rules

Now that we have showed the rules we have set up for pitch, we will in this section show the rules for rhythm.

- **RhythmDifference** Every time a duration is generated that is not the same as the input duration, a penalty point is added. This is a way to check whether the neural network is able to mimic the input, and therefore if evolution is working.
- **DistanceFromTotal** If a modules has not created the same amount of durations as the input melody, a penalty by the sum of missing durations is added. This is done to make the modules more interesting, compared to only having one duration.
- **SameDurationsInput** Every time a module plays the same duration as the main melody, deduct a point from the total score. This is done to discourage the modules from copying the given input.
- **SameDurationsModules** Every time a module plays the same duration as the other modules, deduct a point from the total score. This is done to discourage the modules from playing the same as each other.
- **TooFew** If the module has fewer durations than a predetermined threshold (which is 25% of the input melody), a point is deducted. The idea is to discourage modules with very few durations.
- **TooMany** Like the one before but discourage the modules from having too many durations with a threshold of 75% of the input melody to help stop the modules from playing all the time making the music messy.
- **TooShort** A module is penalized for each duration that is less than 0.125, which is equal to one eight note. This is done to discourage short durations when generating with a high amount of ticks per quarter node.

- **PlayingAtDifferentTimes** If half of the modules are not "playing" a rest, deduct a point from the score. The idea is here to try to get the modules to play at different times so that the output does not get too cluttered.
- **RestCheck** This rule controls that rests are constructed nicely. Firstly, it checks if the amount of rests for the module is less than five. Any number less than five is penalized by the difference times 10. Secondly, it checks whether the total duration of the rests is below a limit, which is dependant on the number of modules. This rule ensures that all modules have rests, and that no module is resting for an excessive amount of time.
- **StandardDeviation** If the standard deviation of the generated output is too low, the module is penalized 100 points. This is to ensure that the peaks in the output are clear, and to discourage solutions that have a flat output.
- **OnBeat** This rule discourages the module from playing offbeat. For each duration, if the total duration including the current duration is not divisible by 0.0625, the module is penalized. The duration of 0.0625 is equivalent to a sixteenth note.

When combining these rules, it is inevitable for them not to contradict each other occasionally. A situation where we reward a certain feature but later gives a penalty for it might confuse the training of the neural network. A solution for this is to put weights on the rules, where the penalty for breaking one rule is insignificant compared to the penalty of breaking a much more heavily weighted, contradicting rule.

4.3 Harmonization

A central part of music is the concept of chord progressions. They provide tension and a musical framework for the instruments to work off of. It would therefore also be interesting to examine how the modules might be able to construct harmonies. To test this, we have created a naive method for harmonizing a melody.[1]

Algorithmic harmonization of melodies is a complex area, and an ongoing research problem. To explore it fully would require another project of similar or larger size compared to this. Since the object of this experiment is to examine how the modules behave, we have chosen to implement a fairly naive solution.

For each bar, the algorithm picks a chord from the seven triads in the given key. This means that all chords are either major or minor, except the chord on the seventh step, which is diminished. The chord is chosen by looking at the tones played during the bar, and selecting the chord that contains the most tones. If multiple chords contain the same amount of tones, one of those are chosen randomly. Figure 7 shows two outputs stacked on top of each other of the function for the Forrest Gump theme. Notice how some chords stay the same, while others change between potential chords.

[1]Method *Harmonize* in the *Music* class, taking the current song as an argument

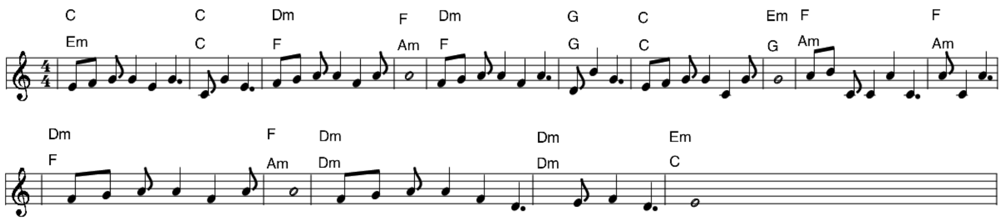


Figure 7: Two suggested harmonizations for the Forrest Gump theme melody

The function could be improved. For example, it currently assumes that the melody starts at the first beat of a bar, which is not often the case. Moreover, the function assumes that the melody is known beforehand, which makes it not work in full time. However, the chord relation to their pitches are usually harmonic and therefore provides a satisfactory output for testing.

5 Analysis

This section contains further analysis of the program and its produced output. We will create a series of tests to see whether our program acts as expected, and whether it is able to fulfill the problems we mentioned in the problem statement. Since we are dealing with genetic algorithms, the outcome is non-deterministic and while they sometimes act as expected, there is no certainty it will do so the next time.

5.1 Evolution Evaluation

To determine whether the modules are actually able to evolve and learn from the input melody, we create an experiment where the only fitness rule is to mimic the given melody both in terms of pitches and rhythm. In figure 8 we see a snippet of the generated melodies, where the first bar is the given melody and the three below are the modules 1, 2 and 3 respectively, all with an almost perfect fitness score.

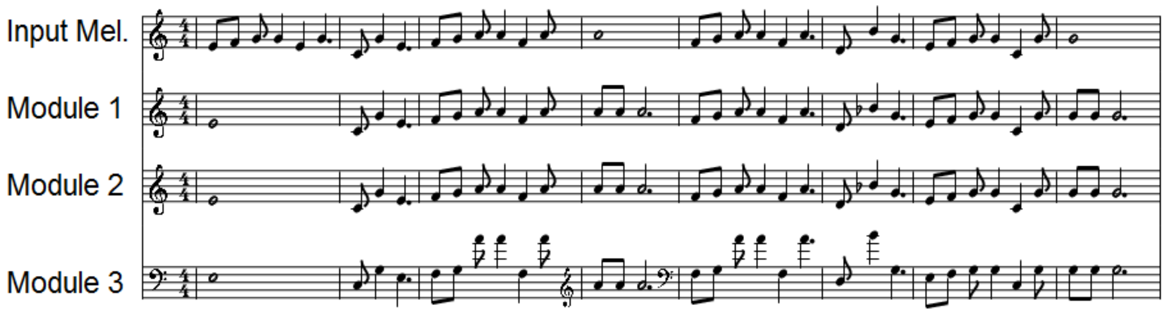


Figure 8: Generated output with tree modules where they tried to mimic the input melody

As can be seen in the figure, the modules are somewhat successfully mimicking the given melody with a few exceptions with the rhythm, especially in the beginning and when the given melody is playing a long note in bar four. Since the ANN's have been trained with a song in C Major, it will also be able to mimic other songs in the same key. While this shows that the modules can relate to the given input melody, it does not show that they can relate to each other. In theory maybe module 1 might mimic the input melody, while module 2 and 3 just mimic module 1 - which would be an acceptable solutions to our fitness rules. However to test the module relations to each other, we will in the next section try some different experiments where they are forced to relate to each other

5.2 Relations Between Modules

A central part of the project is modules relating their output to each other, and correcting themselves to create one whole piece when combined. In this following section, we will examine their ability to do just that. Firstly, we will present an experiment examining if the modules are capable of relating to each other. Secondly, we will look at how different amounts of modules affect the output.

Each module contains two ANN's which allow us to create isolated experiments for pitch and rhythm respectively. To check whether the modules can relate to each other, we will use *SameDura-*

tionInput and *SameDurationOutput* to make the modules play something different from each other. Figure 9 below shows a snippet of the created output. The pitch is not relevant for this experiment, so it just uses *PitchDifference* to mimic to input melody.

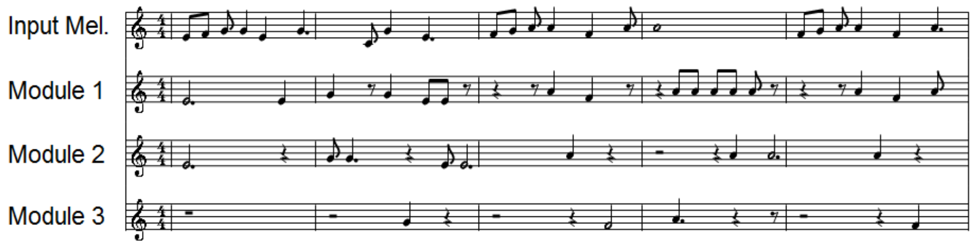


Figure 9: Module relations example with rhythm

As can be seen in the figure, the modules are somewhat successful at playing at different times than both the given input melody, but also each other. This particular run was about 250 generations, where the fitness scores stagnated at about 50-25 points from a theoretical perfect fitness score, which might also be why we see places where they play at time.

A similar experiment for pitch can be seen in figure 10 below, where the rules were *IdentityInputs* and *IdentityModules*, meaning that the modules would be given penalties for playing the same pitches at the same time. Here we try to mimic the rhythm, so that we can see which pitches are played at the same time.

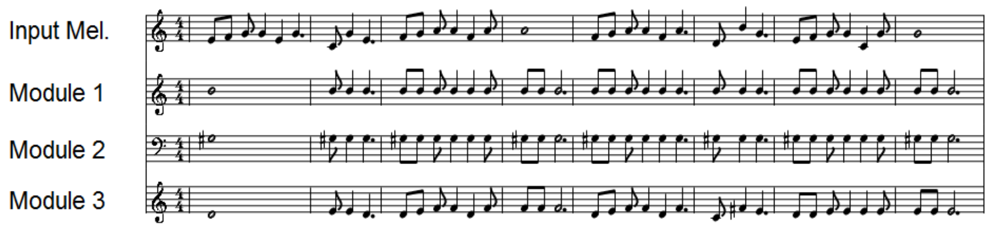


Figure 10: Module relations example with pitch

This snippet offers some results, which we can interpret into what patterns the modules display. *Module 2* simply found out that by playing a g-sharp pitch, it would never collide with either the other modules or the main melody, giving a perfect fitness score. *Module 1* decided to go with the same strategy and play the same 'b' pitch, resulting in a not perfect score since 'b' occurs in the given melody. What is more interesting is how *Module 3* somewhat mimics the input melody but changes the values slightly to make unique pitches, which also gave a perfect fitness score.

While these two experiments have very simple rules they do provide some insight in that the modules are capable of adapting to each other when evolving.

Let us now take a look at how different amounts of modules affect the output. Figure 11 and figure 12 shows output from a test run with two and four modules respectively, while a figure showing how the output is affected at six modules can be seen in the Appendix section 10.3, Module Relations Output. The examples have been generated over the *Forrest Gump* theme using the fitness function configuration *Deep*, which is described later in section 6.2. The examples can be found as audio at http://goo.gl/Rkr9vg, with the file names "output2modules.mp3", "output4modules.mp3" and "output6modules.mp3".

Firstly, let us analyze the output from the experiment with two modules, shown in figure 11. Here, the modules are good at alternating between taking rests and playing notes, leaving plenty of room for both the input melody and the other module. The notes picked are centered around the C-major chord, and support the melody well.

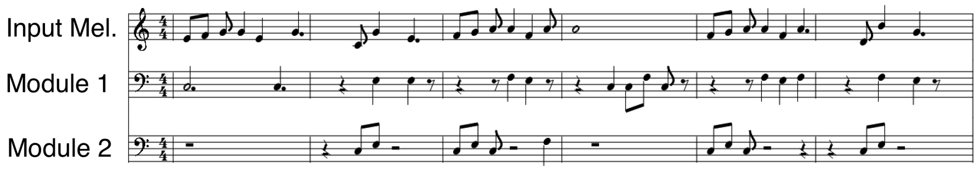


Figure 11: Snippet of output from evolution with two modules

When looking at figure 12, and what the system has produced with four modules, it becomes a bit more muddy. The tones are still largely centered around the C-Major chord, which causes some unavoidable clashes between the modules, since four modules sharing three tones is bound to have some repetition. Another tendency is some modules degenerating to sticking to one pitch, which can be seen in *Module 3*, or taking long rests, which can be seen in *Module 1.*

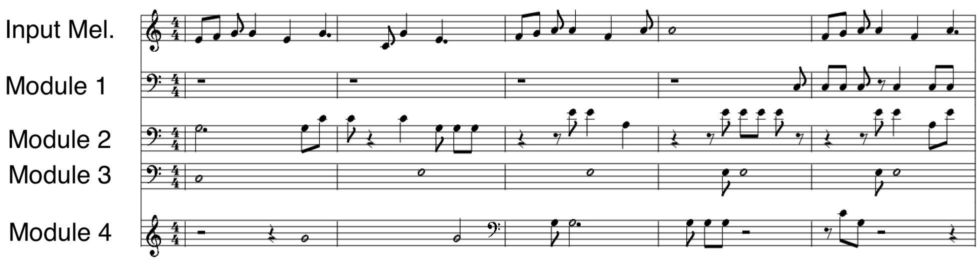


Figure 12: Snippet of output from evolution with four modules

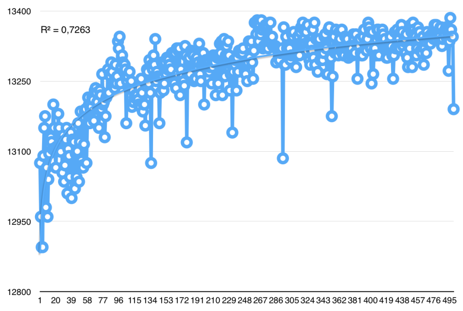
Six modules follows the same trend, which is visible in figure 35 in the Appendix, section 10.3. It should be noted that this clip is less developed, since the system needs a significantly larger amount of time to develop, will will be covered in the next section. Multiple modules now stick to one note, the output seems more jumbled and it has a higher degree of dissonance. There are, however, still signs of the modules preferring the C-Major chord.

5.3 Evolution of Fitness Scores

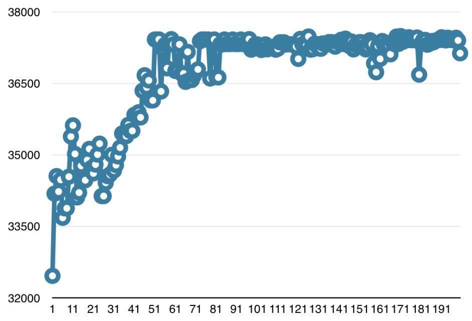
This section contains an experiment to see how fitness scores evolve with different module sizes. All parts of the setup, except the number of modules, are kept constant. The values presented are sums of the fitness values given to each module for each generation. We are hoping to prove that the fitness value is increasing with the generation count. Additionally, we want to analyze how the evolution varies between the pitch and rhythm algorithms, as well as how it behaves for different amounts of modules. While the figures presented in this section are small, larger figures are available to look at in the Appendix section 10.2, Fitness Evolution Charts. The Appendix also contains charts for pitch evolution with 2 modules and 500 generations and both rhythm and pitch evolution with 4 modules and 1000 generations.

First of is the evolution when using two modules. Figure 13a and 13b shows how rhythm and pitch evolve. Rhythm shows 500 generations of fitness score evolution, while pitch shows 200 generations. It is clear to see how rhythm and pitch evolves differently. Rhythm evolves logarithmic with relatively big fluctuations. It slows down and reaches something akin to an optimal point after roughly 300 generations. On the other hand, pitch climbs quickly and steadily to a plateau, which is reached after roughly 70 generations. Figure 13b shows 200 generations, however it continues in the same manner for all 500 generations, which can be seen in section 10.2.

Figure 13: Fitness evolution with 2 modules



(a) Rhythm fitness evolution over 500 generations



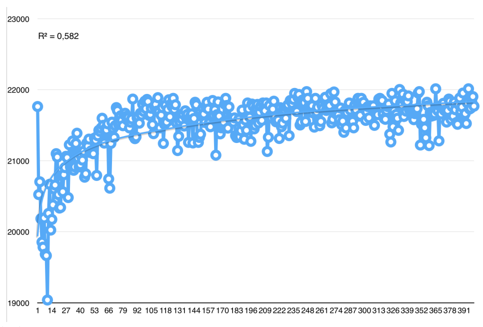
(b) Pitch fitness evolution over 200 generations

For four modules, the fluctuations become bigger and the evolution slower. Figure 14a and 14b shows how the modules evolve. The rhythm evolution is roughly the same as it was for two modules. It still has logarithmic growth, and evolves at around the same speed. For pitch, however, the difference is more pronounced. The fluctuations are bigger, and the evolution happens slower. It also does not reach an optimum, but seems to be able to continue. In addition to the test with 400 generations shown in the figures, we also conducted one with 1000 generations, which can be seen in the Appendix section 10.2. It shows that the pitch keeps growing linearly. The reason for this slowdown is that a lot of pitch fitness rules rely on relations to other modules. Since all modules evolve at the same time, it naturally causes the modules to need some more generations to adjust to each other. Additionally, the optimal solution might be discarded because of changes done by the other modules, so even through a better solution is presented, it is not necessarily chosen.

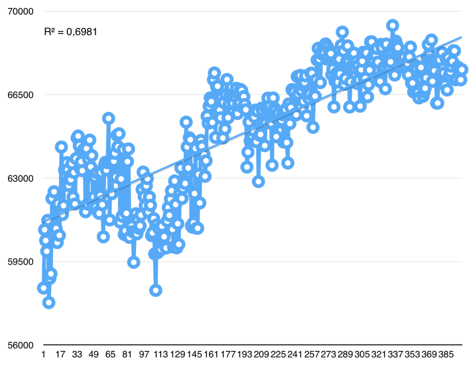
When evolving with six modules, the same patterns seen when generating with four modules are seen, but amplified. The charts can be seen in figure 15a and 15b, where they evolved for 270 generations. Especially pitch evolution becomes slowed down. However, a linear progression is still visible.

These experiments show, that rhythm evolves in logarithmic manner, while pitch evolves linearly with big fluctuations. Increasing the amount of modules decreases the rate of evolution and increases the size of fluctuations, since modules have to listen to adapt to each other.

Figure 14: Fitness evolution with 4 modules

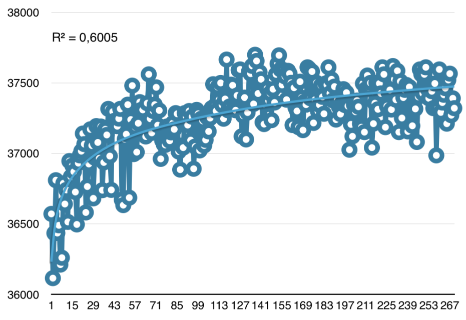


(a) Rhythm fitness evolution over 400 generations

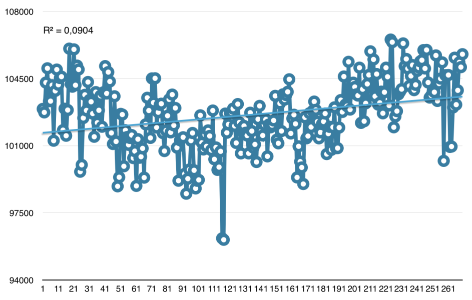


(b) Pitch fitness evolution over 400 generations

Figure 15: Fitness evolution with 6 modules



(a) Rhythm fitness evolution over 270 generations



(b) Pitch fitness evolution over 270 generations

6 User Survey

To test how the output of our program was perceived, we decided to conduct a user survey. Since music is subjective, it is difficult to predict how a person might react to hearing a piece of music. When making a music program such as this, you have to make a couple of assumptions. Therefore, it is useful to get them verified. In the following chapter we will discuss how the survey was constructed, what pieces the users were presented to and what the results of the survey are.

6.1 Data Collection

The survey is based on a framework created by our supervisor Marco Scirea, and is hosted on a website.[2] In it, the user is asked to listen to two clips and compare them to each other based on four criteria. These are:

- Pleasantness
- Harmoniousness
- Randomness
- Interestingness

These four descriptors were chosen since they might cover different areas in music. For each parameter, the user can choose "Clip A", "Clip B", "Both equally" or "Neither" as the answer.

To test our program, we created four configurations of fitness functions, that would hopefully accentuate different aspects of the capabilities of the program. These will be detailed later, in section 6.2. Additionally, to test the fitness functions in varying scenarios, we included four songs. These songs expose our fitness functions to different pitch- and duration scenarios. Due to the non-deterministic nature of the program, each iteration will produce something new, which might be a desired outcome, but could also be the opposite. Therefore we created four instances of the same song with the same fitness configuration, to get a better selection of what the program produces.

Figure 16 shows an illustration of how, for each of our four fitness configurations (labeled F1-4), we used four songs (labeled S1-4), that each has produced four instances of that song (the yellow blocks in the bottom) totalling 64 individual clips. To get a true selection of what the program produces, these clips are all created in a sequence, with no filtering or selection before it is presented to the respondent, as that would amount to cherry picking and would not be representative of the real performance of the system. Furthermore, each clip was created with three modules, to keep this element constant.

The clips presented to the user were selected by first finding one clip, *"Clip A"*, randomly, and afterwards finding another clip, *"Clip B"*, that fit to it. *Clip B* was restricted in that it had to use another fitness configuration and had to use the same melody. In practice this was done by having each song be represented by a four-digit number, with the fitness being signified by the 1000's (e.g. 1000, 2000, 3000 and 4000), the melody in the 10's (e.g. 10, 20, 30 and 40) and the variation in the 1's (e.g. 1, 2, 3 and 4). If either of the songs had been played before we started the whole process again. While this technique might be a little naive, we considered it acceptable for a survey

[2]The survey is hosted on `www.hansalmegaard.dk`

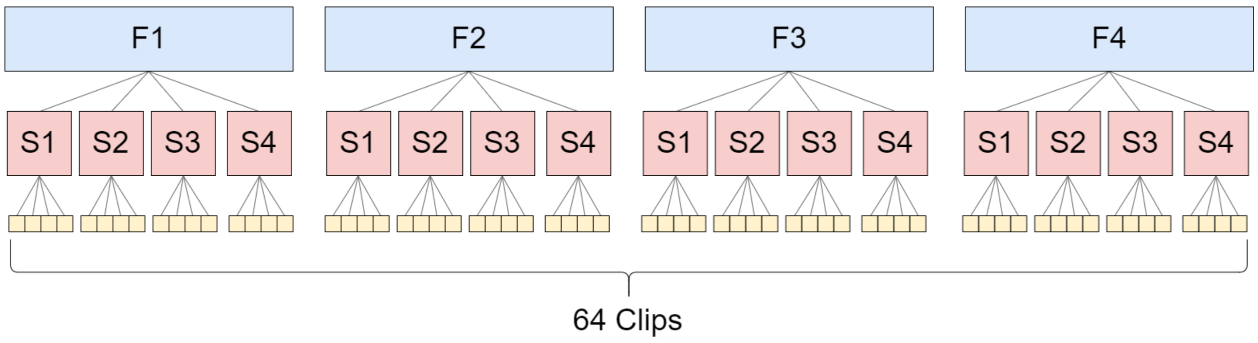


Figure 16: Model showing how many clips we have and their categorical fitness placement.

of this size. When presenting the two songs to the user, it was important that they both used the same melody, since we did not want a certain melody to influence whether the produced clip scored higher in any of the four categories.

All generated clips, sorted based on the fitness configuration, can be heard at `http://goo.gl/Rkr9vg`.

6.2 Fitness Function Configurations

The following section describes each of the four fitness configurations. The configurations are based on a set of smaller functions, referred to as *rules*, which are described in section 4. Table 1 and 2 shows weighting of the different rules for pitch and rhythm respectively. The values for each fitness function configuration were decided upon through experimentation. Note that the function return values are not normalized, which means the weighting is not directly proportional with their influence on the output. For instance, in the configuration *Deep*, the fitness uses the function *BelowInput* with a weighting of 5, but even though the weighting is small it still influences the output a lot. The weightings can, however, be used to compare the different configurations to each other.

Each configuration is accompanied by a suggestion for representative clips. These can be found at `http://goo.gl/Rkr9vg`.

Table 1: Pitch fitness configurations

Pitch	Chords	Rhythm	SameNote	Deep
OutOfKey	10	10	10	20
BadPitch	1	5	1	1
SameNote	50	25	100	25
IdentityInput	15	15	15	10
IdentityModules	10	10	10	5
LowerThanPrevious		1		
TooBigGap	10	10	10	10
DissonanceInput		5		5
DissonanceModules		5		5
Chord	100	5		10
BelowInput				5

Table 2: Rhythm fitness configurations

Rhythm	Chords	Rhythm	SameNote	Deep
RhythmDifference	1			
TooFew		10	1	10
TooMany		1	1	1
SameDurationsInput		20	1	10
SameDurationsModules		10	1	5
RestCheck		1	1	1
PlayingAtDifferentTimes			1	5
StandardDeviation		1		1
OnBeat		10		10

- **Chords**

  This configuration places a strong emphasis on generating chords. To have the chords be very pronounced, the rhythm is imitating the one given by the input melody. Overall this should create harmonious compositions, since the chords are hopefully well formed. Representative clips are *1002*, *1012* and *1030*.

- **Rhythm**

  This configuration has an emphasis on rhythm. It is supposed to be dynamic and play a lot of tones. The configuration of the rhythmic part discourages playing few tones, and playing the same note durations as the others. The pitch part is relatively simple, encouraging note changes and discouraging disharmonious notes. This is supposed to create an interesting output, since it is highly dynamic. Representative clips are *2012*, *2022* and *2032*.

- **SameNote**

  This configuration is heavily discouraged from playing the same note twice. It is done through weighting the *SameNote* parameter very high in the pitch configuration. The rhythm configuration is fairly bland. The output is supposed to be random, but might also create some interesting patterns. Representative clips are *3002*, *3010* and *3020*.

- **Deep**

  This configuration is based on accompanying by playing notes lower than the input melody, which often creates some deep pitches. For the pitch, the *BelowInput* parameter has a big impact. Other than that, it is similar to the other configurations, but with a bit more emphasis on creating output which fits within the scale. The rhythm output is similar to that of the rhythm configuration, but with less penalization for playing the same. Supporting the melody, and having the generated music be more akin to backup, should have this configuration create some pleasant pieces. Representative clips are *4000*, *4022* and *4030*.

To test the fitness function configurations on different input, we chose four different melodies. Each have a couple of interesting features. The chosen melodies are:

- **Forrest Gump theme** Theme from the movie Forrest Gump, composed by Alan Silvestri.
- **Vi Maler Byen Rød** Song by danish *"dansktop"* singer Birthe Kjær.
- **Auld Lang Syne** Scottish folk song, known from New Years Eve.
- **Flim** Song by Aphex Twin.

All songs are in shortened versions, comprised of the most important part of the main melody. This is done to minimize the amount of time a person has to spend to take the survey.

6.3 Data Analysis

In the end we had 67 people answering the survey, with a total of 432 performed comparisons, which is 6.4 comparison per person on average. Out of the 432 comparisons, the distribution of the configurations were *Chords* (24%), *Rhythm* (26%), *SameNote* (25%) and *Deep* (25%), giving a pretty fair distribution of playing time. A little less equal distribution was found when looking at the combinations of configurations, totalling six different combinations with 16.6% as the theoretical best distribution between them. In practice though, as table 3 below shows, the combinations ranged from 13% to 21%, with the *Chords-Rhythm* combination as the most used and *Chords-SameNote* as the least. Figure 17 shows the distribution in a pie chart.

Table 3: Statistics of performed configuration comparisons

	Amount	Percentage	Binomial test
Chords-Rhythm	89	21%	0.0329
Chords-SameNote	56	13%	0.0387
Chords-Deep	66	15%	0.4777
Rhythm-SameNote	65	15%	0.4013
Rhythm-Deep	71	16%	0.9486
SameNote-Deep	85	20%	0.0935

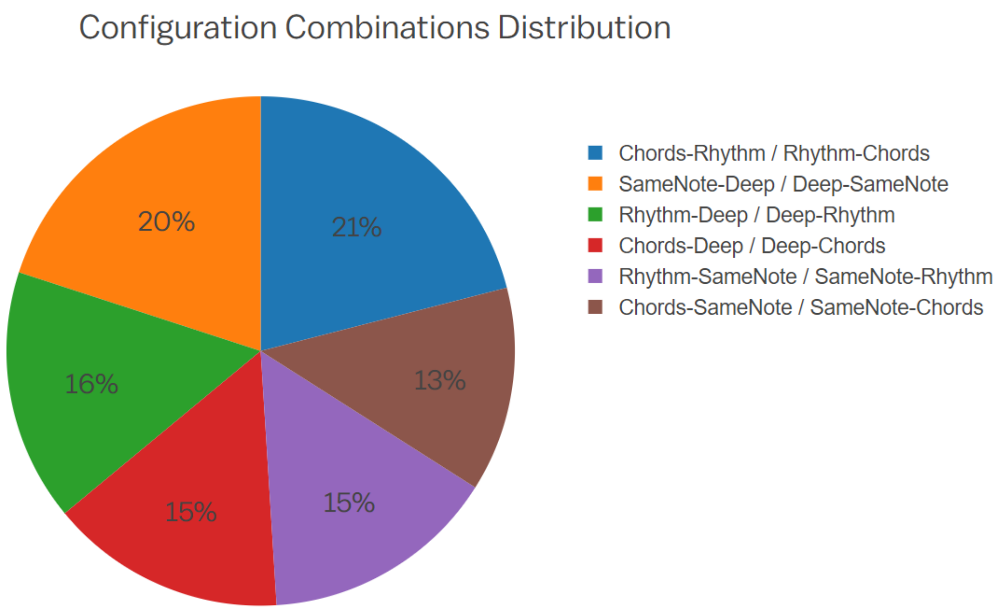


Figure 17: Distribution of the presented configuration combinations

Looking at the binomial test, presented in table 3, it is clear that the results are not statistically significant, judged by a reasonable metric. The comparisons between *Chords* and *Rhythm* and *Chords* and *SameNote* have been performed respectively 89 and 59 times, which gives us binomial test values of 0.0329 and 0.0387. This discrepancy is probably caused by our selection process. Even though we tried to get a fair distribution, these numbers point to that our code does not actually produce random results. Therefore, we have chosen to display the results as percentages, instead of in absolute numbers. Hopefully it can still show a tendency when looking at the data as a whole.

Out of the 67 people attending, 46 were male, 25 were female and 2 would not say. Table 4 shows a collection of meta-data about the survey.

Table 4: Meta-data about the survey

Participants	Comparisons	A or B	Both Equally	Neither
67	432	1302	161	270

6.3.1 Configuration Wins

To get an idea of when the respondents found a configuration to be superior in one of the categories against the other, let us first start by looking at the data were there was a winner in the comparisons. Figure 18 shows the wins for each different fitness configuration, distributed in the four different criteria. In this chart we do not distinguish between which configuration beat which configuration, but only the total accumulated wins a configuration got. Answers of equal or neither are not accounted for in this chart.

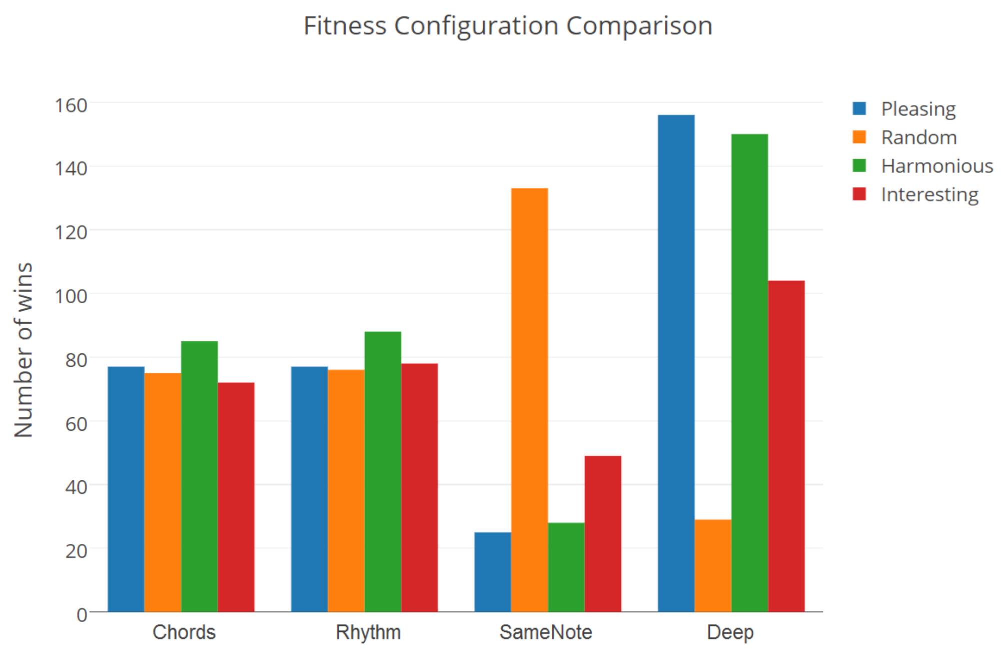
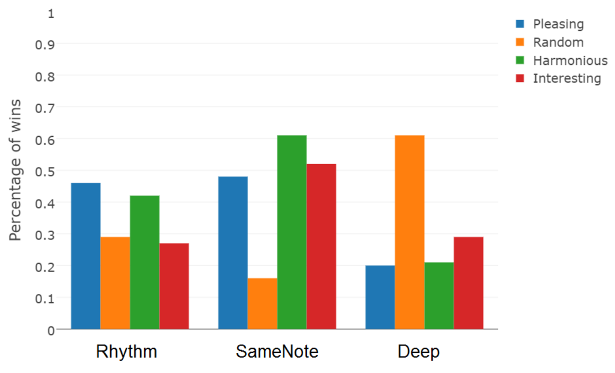


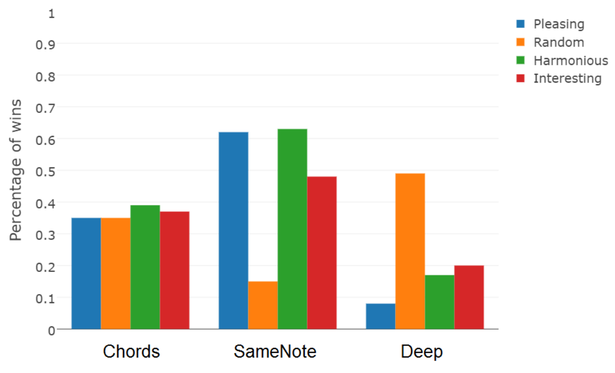
Figure 18: The amount of wins for each fitness configurations in the different criteria

From this chart, it is clear to see big fluctuations when looking at the *Deep* and *SameNote* configurations. *Deep* was considered the most pleasing, harmonious and interesting, while at the same time being considered least random. On the other side, *SameNote* got most points in random while getting very few in pleasing, harmonious and interesting. The graph also shows that as a whole *Chords* and *Rhythm* are almost indistinguishable from one another, with an almost even distribution in all of the four categories.

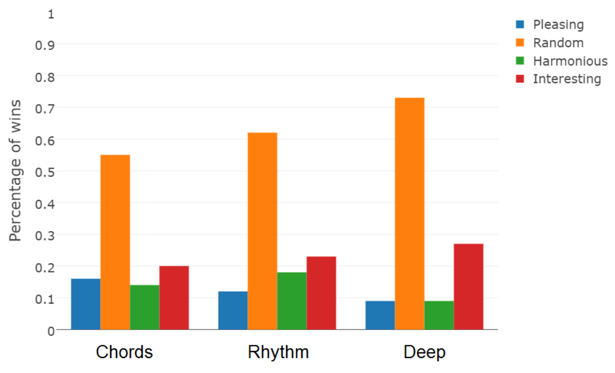
To get a better idea of how they compare with each other individually, figure 19 shows all four fitness configurations, and their total amount of wins for each criteria when being compared to the other configurations. *Chords* is in the upper left corner, *Rhythm* in the upper right corner, *SameNote* in the lower left corner and *Deep* in the lower right corner. The appendix section 10.1 contains enlarged versions of the figures, along with a table containing the associated data.



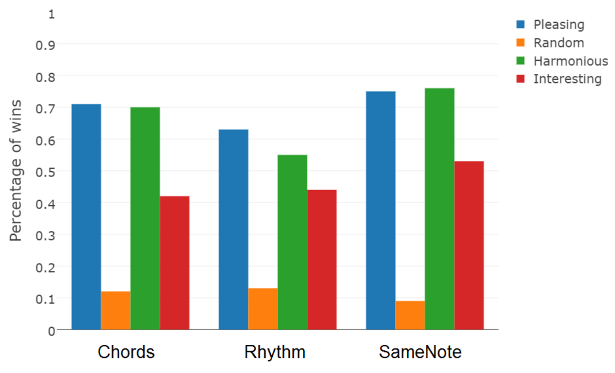
(a) Chords



(b) Rhythm



(c) SameNote



(d) Deep

Figure 19: All four fitness configurations and the total wins they got in each of the four categories when comparing themselves with each other

If we look in figure 19a and 19b and compare how *Chords* and *Rhythm* scored against each other, we can see that the total *Pleasing* votes of 35% *Rhythm* got, versus the 46% votes *Chords* got against it, might indicate that *Rhythm* is considered a little less pleasing than *Chords*. However, *Rhythm* got 37% wins in *Interesting* against *Chords* which got 27% wins, indicating that the participants found the varied rhythm fitness rules more interesting than the mimicking rhythmic fitness rules of *Chords*.

Every configuration that measured itself with *SameNote* was found to be much more pleasing, harmonious and interesting, while at the same time being the least random. At the same time whenever *SameNote* was compared to the others it was found to be much more random, while it scored really low when measuring pleasing, harmonious and interesting. On the opposite side, every time a configuration was compared to *Deep*, they were found to be more random while scoring low in the other categories.

6.3.2 Equal or Neither

Even though there sometimes was a distinct winner between the configurations, the respondents did not always find a favorite in the different categories. Figure 20 below shows the percentage of the different kinds of answers the respondent gave to each configuration combination. It shows the combinations where the users thought there were many similarities, and to find those were there was a clear distinction between the configurations. Since the different combinations did not appear an equal amount of times, the graph shows the ratio between the different available options for each combination.

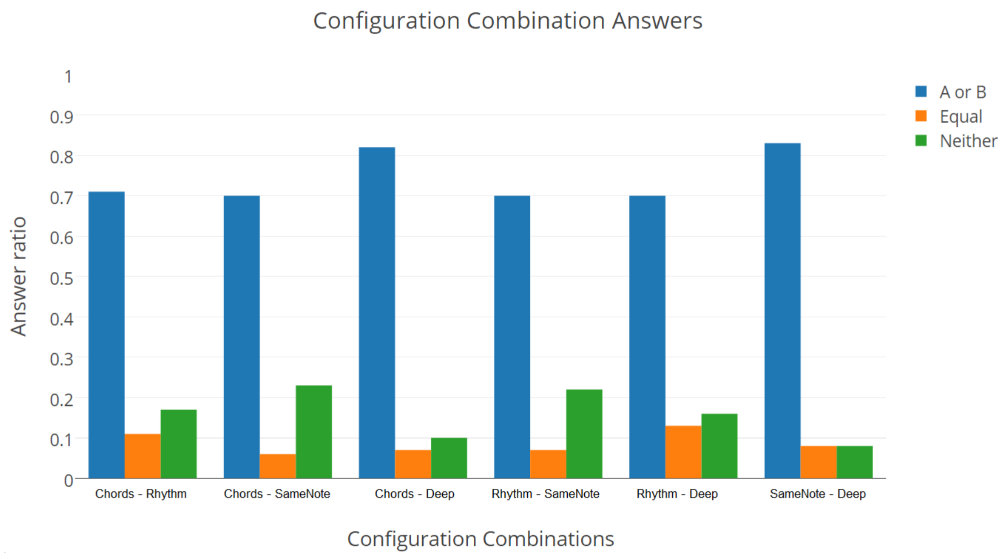


Figure 20: Answers for comparisons between two fitness configurations

As can be seen in figure 20 the two combinations where people answered equal or neither the least are in column group three and six where *Deep* was present, with a win percentage of 82-83%. In addition to what we saw in figure 18, this hints that respondents were quite certain when comparing with the *Deep* configuration, and found the biggest distinction between the configurations in these groups. The other four combinations had a pretty similar distribution of wins, with a percentage around 70%. The main difference between these are the distribution of *Neither* and *Equal*. Column group two and four, where *SameNote* is present, has the biggest gap between *Neither* and *Equal*. A theory of ours is that people have treated *Neither* as a dislike for both clips while *Equal* as an equal like for both clips. While this graph can give us a little insight into how they clip relate to each other an a larger scale, it does not really tell us anything about which categories they scored in. Figure 21 shows the total percentage of the answers where people did not find a specific winner in either of the configurations, divided on configurations being compared and criteria.

Looking at the *Chords/Rhythm* combination in column group one, we can see that the respondents often found either or neither of the clips more *Random* or *Interesting* than the other. They did however find a larger distinction when comparing them in pleasantness and harmoniousness. In the *Rhythm/Deep* combination in column group five, *Rhythm* and *Deep* got an interesting score of almost 40% indicating that people could often not tell them apart in this category. In column

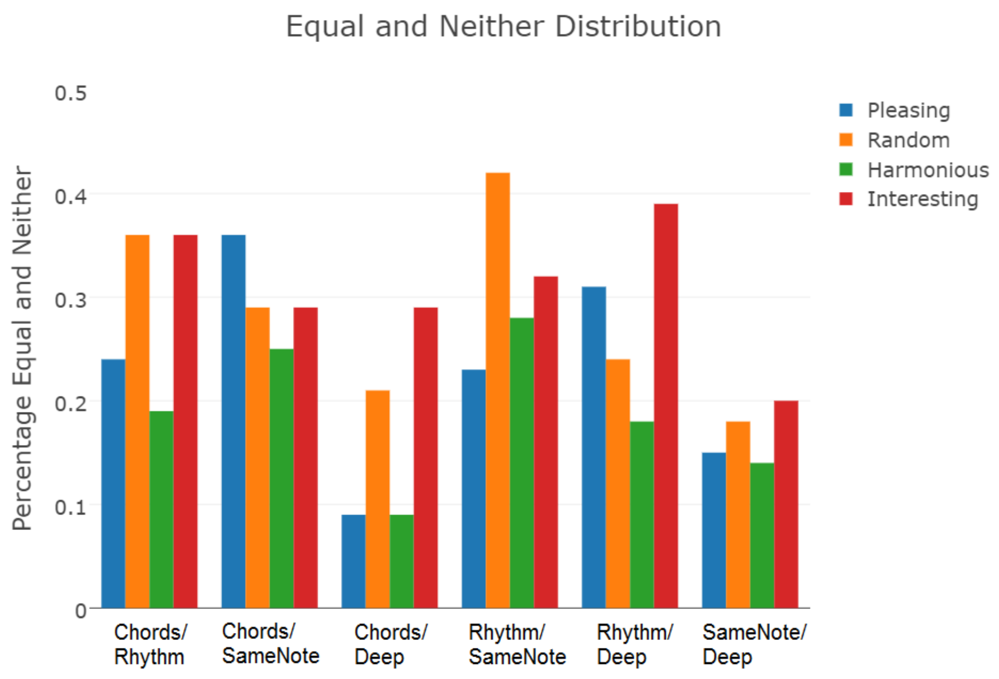


Figure 21: Total accumulated answer distribution of *Equal* or *Neither* for each configuration combination in each descriptor category

group two (*Chords/SameNote*) and four (*Rhythm/SameNote*), the respondents showed the most equal distribution of uncertainty concerning which of the clips they thought superior in one of the categories. Especially in the later they had over 40% uncertainty about which of the two clips were more *Random*.

6.4 Discussion of Results

Of course the four descriptors pleasant, harmonious, random and interesting can mean very different things depending on how a certain person interprets them. We also had our own interpretation of what they meant, that was largely based our knowledge of evolutionary music. Our expectation was that respondents would find the *SameNote* configuration to be somewhat interesting, when in reality it was found to be the least. Here our idea of interesting was that *SameNote* would produce more special results, while *Deep* would be producing more harmonious results, but maybe also a bit more boring. There is always the uncertainty whether people always answer truthfully or just rush through the experiment. Of course without more data the uncertainty in these results are higher and it might be hard to draw any certain conclusions.

One theory for the clear results regarding *SameNote* and *Deep*, is that the respondents answered in terms of likes and dislikes. This can be seen since *Deep* is assigned all the words with positive connotations (pleasing, harmonious and interesting), while *SameNote* is assigned the word with a negative connotation (random). One explanation might be that the respondents were unable to judge the pieces based on different parameters, since they overall had a low proficiency in music theory, cf. Table 5. Therefore the answers might have been more based gut-feeling to questions like *"do I like or dislike this piece?"*.

Table 5: Responses to the question "How well do you know music theory?"

	Not at all	Slightly	Moderately	Very well	Exceptionally well
Amount	36	17	11	8	1

However, even through the former might be true, we are still able to form conclusions on the fitness function configurations, based on the data presented in the prior section. *Deep* is the preferred configuration, *SameNote* the least preferred, while *Rhythm* and *Chords* are somewhere in between. Chords appeared to be a little more pleasing than *Rhythm*, however *Rhythm* was deemed more interesting than *Chords*.

7 Discussion

This section will contain discussion upon the results from the previous sections. First, we will discuss the results of the user survey. Second, we will address whether we were able fulfill our problem statement concerning the modules and what consequences multiple modules might bring. Lastly, we will make a couple of suggestions on where the system could be extended.

7.1 Fitness Rules

Now that we got the results from the user survey, we will take a look at which of the fitness rules that might have caused the particular results. In the following section, we will try to answer which rules that brings out certain qualities in the music. Of course, since the configurations are a mixture of multiple rules, it might be difficult to say which rule that caused a certain effect. Ideally we would be able to take each to take each fitness rule and compare them with each other to get a clear picture of how they individually compare. However, since that would be infeasible, we will analyze the broad lines through the fitness configurations from section 6.2. Table 1 and 2 from section 6.2 shows how different rules are weighted in each fitness configuration, which is what we will use to see how they differ from each other.

7.1.1 Pleasant and harmonious

Let us first look at what might make a piece pleasant and harmonious. From the results of our study, these two criteria are highly correlated, which is why these two are discussed together. Seeing as *Deep* was deemed the most pleasing and harmonious, we will analyze it to see what makes it different from the other configurations.

It has a higher weight in *OutOfKey*, which makes it discourage out of key pitches, that can cause dissonance. It does not focus as much in pitch variety through the *SameNote* rule, which might make it easier for it to satisfy some of the other rules better. Additionally, it has a lower weighting of the *IdentityInput* and *IdentityModules* rules, which allows it to not go too much out of its way to make something unique.

Looking at how *Deep* differed from the other configurations in terms of rhythm, we can see that it, like *Rhythm*, aspired to not play too few notes by weighting the *TooFew* rule heavily. Additionally, *Rhythm* and *Deep* shared a high weighting in the *SameDurationInput* and *SameDurationModules* categories, whose aim was to force the modules to play more often. Where *Deep* especially stood out in rhythm was its weight in *PlayingAtDifferentTimes*, which tried to spread out the resting and playing times of the modules, which might have given a more pleasing and uncluttered coherent piece of music.

Another set of rules that are used by *Deep*, in addition to *Rhythm*, is the *DissonanceInput* and *DissonanceModules* rules. Looking at how the *Deep* and *Rhythm* was often deemed equal or neither in the *Pleasing* and *Harmonious* criteria in figure 21, we might assume that these rules had an impact on this. This would also make sense, since these particular rules were created to diminish dissonance, which would create less harmony. However, the *Chords* configuration, that instead of these rules included the *Chord* rule with a heavy weighting, still get more wins in *Pleasing* and *Harmonious* against *Rhythm. Deep* also utilized the *Chord* rule, but with a much lower weighting than *Chords*, and since it scored higher in *Pleasing* and *Harmonious* than *Chords* did, we can conclude that a strict following of the chords might produce some pleasant and harmonious results, but it might not be the most important rule.

Where *Deep* is unique compared to the other configuration is it utilizing the *BelowInput* rule, which aims to let the modules have lower pitches than the input, allowing the main melody to be in focus. This may have also been a contributor to why people found *Deep* superior in most of the categories.

To summarize, a pleasing and harmonious result is reached by having rules that encourages the modules to play within the given scale, avoiding dissonance and taking a supportive role in regards to the input melody. In regards to rhythm, it is good to distribute the times at which the modules play to have more room for each individual module, as well as not having the uniqueness of each module be the main priority. Additionally, we can see that configurations that have a good all-round configuration (*Deep* and *Rhythm*), are more pleasing and harmonious that those that are centered around one rule (*SameNote* and *Chords*).

7.1.2 Random

According to figure 18, *SameNote* is by far the most random. The most characterizing rule of the configuration is it heavy weighting of the *SameNote* rule, which it is also named after. This rule discourages the modules from playing the same note two times in a row, and therefore makes them play a lot of different pitches.

In regards to the rhythm, *SameNote* is not that interesting. It utilizes a few basic rules, all with a weighting of one.

To sum up, randomness can be achieved by having the modules play a lot of different tones.

7.1.3 Interesting

Saying which configuration that produces the most interesting result is hard to conclude from the results, which are not a clear-cut as the ones from the other criteria. However, as discussed in section 6.4, some statistics point to the fact that respondents often judged the clips as positive or negative, where words with positive conjunctions (pleasing, harmonious and interesting) are assigned to one clip, and the word with a negative conjunction (random) is assigned to the other.

If this is the case, we can see some interesting results from the *Deep* configuration in figure 19d. From it, we can see that *Deep* has less wins in interesting than in pleasing and harmonious, which might signify that, although it is the configuration that the respondents prefer, it is also not all that interesting. The biggest difference is seen when compared with *Chords*, whose biggest feature is the strong weighting of the *Chord* rule, whereas *Deep* only has a weak weight. This means that *Chords* follows the harmonization described in section 4.3. It has a bigger tendency to change chords throughout the song, while *Deep* usually sticks to one chord. This signifies that changes in terms of harmonies cause the generated clip to be perceived as more interesting. However, in terms of the rhythm, it is important to note that the *Chords* configuration often loses when compared to the *Rhythm* configuration (see figure 19a), which also signifies that the rhythmic pattern of *Rhythm* is more interesting than the one from *Chords*, where the rhythm is mimicked from the input melody.

However, this argument only holds true if we accept the notion that respondents largely decide based on personal preference. Without that assumption, *Deep* is still deemed the most interesting. This would signify that the factors that make a clip interesting are the same that make it pleasing and harmonious - e.g. playing within the given scale, having a supporting role and giving each module enough sound space.

7.2 Modules

We were able to extend the program to support an arbitrary number of modules. These modules were able to listen to the input melody as well as each other to create a coherent piece of music. However, as we discussed in section 5.2, the larger the amount of modules, the more the modules stuck to one note, the output seemed more jumbled and there were a higher degree of dissonance present. Additionally, as discussed in section 5.3, the program becomes much slower with an increase in modules as well as a higher fluctuation in the fitness score, as the modules have more factors to take into account. So while it is be possible to create as many modules as the user wants, it would require a high amount of patience to wait for the modules to evolve and the fitness functions would have to heavily adjust to downplay the playing time of the modules in order for them to be given individual wriggle room.

In the end though, since it is not possible to distinguish the modules from one another audibly, it does not actually matter that much, whether we use two modules which play regularly, or use ten modules that play rarely. In practice, they might produce very similar results. A solution to this might be to give the modules personalities, where they each focus on filling a specific role in terms of the output. This would make it easier to distinguish the modules from one another, as well as give motivation to select a higher amount of modules.

7.3 Future Work

In this section we will discuss how we could expand on our project given more time. Of course it is always possible to make more improvements and try new fitness configurations to see whether they could yield better results. We will however here cover other directions that the project could progress in.

7.3.1 Roles and Genres

As mentioned in the previous section, a feature that might be interesting to add to the project is different personalities for the modules. Right now they all just follow the same rules in the fitness function, but it is easy to imagine them taking different roles when playing together. An example could be a bass player module which would play deep pitches and a rhythm that emphasizes a beat rhythm on one and three. Another role could be a guitar role, that might try and hit the off beats and add a second voice to the melody. In the same category, and with a code structure that enables substitution and interfaces, it might also be possible to change the fitness evaluation so that it rewards a particular genre like rock or classical music.

7.3.2 Musical Complexity

If more work was to be done on this project, one might want to increase the musical complexity. One thing is to change the structure so that the music can be viewed in a larger context. Right now the modules are compared by looking at the music tick by tick, but by increasing the abstraction level and looking for patterns and sequences, we might be able to create even more interesting results.

Another possibility for extension is adding song structure to the music, where changes between e.g. verse and chorus might result in changes in tension, volume, breaks and so on.

7.3.3 Real Time

Of course one final thing that might be done on this project is to make it more real time compatible, as was the idea for this project. It would require an overhaul though since a lot of the functionality is revolved around the given static melody. Nevertheless, the fitness principles and the module comparison methods would be transferable.

8 Conclusion

In this project we set out to find a way to create a system with an arbitrary number of modules that using NEAT could evolve and adapt to play accompanying music, fitting to a melody as well as each other. From this, we formulated the following research questions:

- *Is it possible to create an arbitrary number of music modules that are able to listen to a given melody as well as each other to create a coherent piece of music?*
- *Which fitness rules can we formulate to generate music, and how do these affect the output?*

We found out that it was possible to create an arbitrary number of music modules that were able to listen to a given melody as well as each other to create a coherent piece of music. This was achieved by using a delayed configuration where the input to the modules ANN's would take what the main melody and the last modules played in the last tick. By creating fitness rules that followed some basic music theory as well as guidelines for how the modules should relate to each other, the modules were able to create a coherent piece of music. However, the larger the amount of modules, the more difficult it would be for them to get high fitness scores, as they would all need to take each other into consideration.

We formulated different fitness rules for our program in section 4. To evaluate these, and figure out which qualities they brought out in the input, we conducted a survey, where users were asked to judge a set of clips based on the criteria *pleasing*, *random*, *harmonious* and *interesting.* From this, we deduced which elements cause which effects. Harmonious and pleasing results were highly correlated, and can be achieved by using rules that enforces playing within the given scale, avoiding dissonance, staying below the input melody and dividing the times at which modules play. A random result can be achieved by playing a lot of different tones. What makes a melody interesting is less conclusive, but might be influenced by having different harmonies and a dynamic rhythm.

9 Bibliography

References

[1] António Oliveira, *Algorithmic Composition – State of The Art*, `https://apspoliveira.wordpress.com/2016/04/12/algorithmic-composition-state-of-the-art/`, 10/05-2017

[2] Evolutionary Complexity Research Group at the University of Central Florida, *Maestro Genesis*, `http://maestrogenesis.org/`, 13/05-2017

[3] Douglas Eck, Welcome to Magenta!, `https://magenta.tensorflow.org/welcome-to-magenta`, 10/05-2017

[4] Yotam Mann, A.I Duet, `https://aiexperiments.withgoogle.com/ai-duet`, 11/05-2017

[5] Jonathan Harnum, *Basic Music Theory, 4th ed.: How to Read, Write, and Understand Written Music*, 4.0, (2013), CreateSpace Independent Publishing Platform

[6] *Convolutional Neural Networks for Visual Recognition*, `http://cs231n.github.io/neural-networks-1/`, 11/04-2017.

[7] Andersen, Timothy, Kenneth O. Stanley, and Risto Miikkulainen, *Neuro-evolution Through Augmenting Topologies Applied to Evolving Neural Networks to Play Othello*, Computer Science Department, University of Texas at Austin, 2002, page 7.

[8] Colin Green, *SharpNeat - Evolution of Neural Networks*, `http://sharpneat.sourceforge.net/`, 11/04-2017

[9] Stanley, Kenneth O and Miikkulainen, Risto, *"Evolving neural networks through augmenting topologies."* Evolutionary computation 10.2 (2002): 99-127. MIT Press

[10] Amy K. Hoover, Paul A. Szerlip, Marie E. Norton, Trevor A. Brindle,Zachary Merritt, and Kenneth O. Stanley, *"Generating a Complete Multipart Musical Composition from a Single Monophonic Melody with Functional Scafolding"*, Proceedings of the Third International Conference on Computational Creativity (ICCC-2012, Dublin, Ireland).

[11] Benjamin Truitt, *Dissonance in Music: Definition & Examples*, `http://study.com/academy/lesson/dissonance-in-music-definition-examples.html`, 20/04-2017

10 Appendix

10.1 User Survey Data

10.1.1 Gender Table

Male	Female	Other	Wouldn't say
46	25	0	2

Table 6: Table of the gender distribution in our survey

10.1.2 Fitness Configuration Total Appearance

Chords	Rhythm	SameNote	Deep
211 (24%)	225 (26%)	206 (25%)	222 (25%)

Table 7: Table showing how many time each configuration has appeared in the survey

10.1.3 Fitness Configuration Combinations Distribution

	Chords	Rhythm	SameNote	Deep
Chords		89 (21%)	56 (13%)	66 (15%)
Rhythm	89 (21%)		65 (15%)	71 (16%)
SameNote	56 (13%)	65 (15%)		85 (20%)
Deep	66 (15%)	71 (16%)	85 (20%)	

Table 8: Table showing how many times a certain configuration combination has occurred where the total number of combinations is 432

10.1.4 Fitness Configuration Scores General

	Pleasing	Random	Harmonious	Interesting
Chords	77	75	85	72
Rhythm	77	76	88	78
SameNote	25	133	28	49
Deep	156	29	150	104

Table 9: Table of general scores between the different fitness configurations where each number is incremented when it is one of the four descriptors more than the other configurations

10.1.5 Configuration Combination Scores Wins

	Pleasing	Random	Harmonious	Interesting
1000-2000	37 (46%)	26 (29%)	37 (42%)	24 (27%)
1000-3000	27 (48%)	9 (16%)	34 (61%)	29 (52%)
1000-4000	13 (20%)	40 (61%)	14 (21%)	19 (29%)
2000-1000	31 (35%)	31 (35%)	35 (39%)	33 (37%)
2000-3000	40 (62%)	10 (15%)	41 (63%)	31 (48%)
2000-4000	6 (8%)	35 (49%)	12 (17%)	14 (20%)
3000-1000	9 (16%)	31 (55%)	8 (14%)	11 (20%)
3000-2000	8 (12%)	40 (62%)	12 (18%)	15 (23%)
3000-4000	8 (9%)	62 (73%)	8 (9%)	23 (27%)
4000-1000	47 (71%)	12 (12%)	46 (70%)	28 (42%)
4000-2000	45 (63%)	9 (13%)	39 (55%)	31 (44%)
4000-3000	64 (75%)	8 (9%)	65 (76%)	45 (53%)

Table 10: Individual pairwise scores between the different fitness configurations where each number is incremented when a configuration wins in one of the four descriptors over the other configuration. 1000 = Chords, 2000 = Rhythm, 3000 = SameNote, 4000 = Deep

10.1.6 Configuration Combination Scores Equal

EQUALS	Pleasing	Random	Harmonious	Interesting
Chords - Rhythm	7 (8%)	15 (17%)	4 (4%)	15 (17%)
Chords - SameNote	1 (2%)	11 (20%)	0 (0%)	2 (4%)
Chords - Deep	3 (5%)	4 (6%)	4 (6%)	8 (12%)
Rhythm - SameNote	1 (2%)	10 (15%)	5 (8%)	4 (6%)
Rhythm - Deep	6 (8%)	12 (17%)	6 (8%)	13 (18%)
SameNote - Deep	4 (4%)	9 (11%)	6 (7%)	11 (13%)

Table 11: Table of scores between the different fitness configurations where each number is incremented when the answer to a descriptor has been Equal

10.1.7 Configuration Combination Scores Neither

NEITHER	Pleasing	Random	Harmonious	Interesting
Chords - Rhythm	14 (16%)	17 (19%)	13 (15%)	17 (16%)
Chords - SameNote	19 (34%)	5 (9%)	14 (25%)	14 (16%)
Chords - Deep	3 (5%)	10 (15%)	2 (3%)	11 (17%)
Rhythm - SameNote	14 (22%)	17 (26%)	13 (20%)	17 (26%)
Rhythm - Deep	16 (23%)	5 (7%)	7 (10%)	15 (21%)
SameNote - Deep	9 (11%)	6 (7%)	6 (7%)	6 (7%)

Table 12: Table of scores between the different fitness configurations where each number is incremented when the answer to a descriptor has been Neither

10.1.8 Chords Category Wins

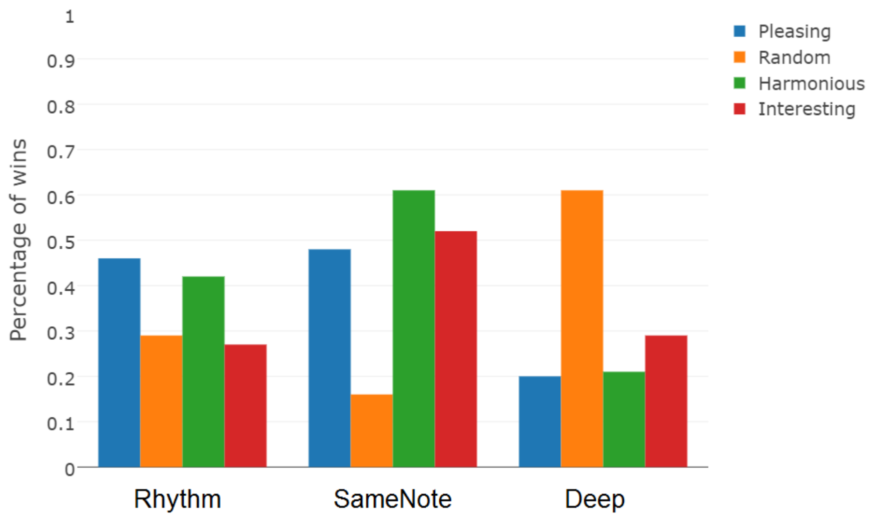


Figure 22: Chords win percentage in the four descriptor categories against the other configurations

10.1.9 Rhythm Category Wins

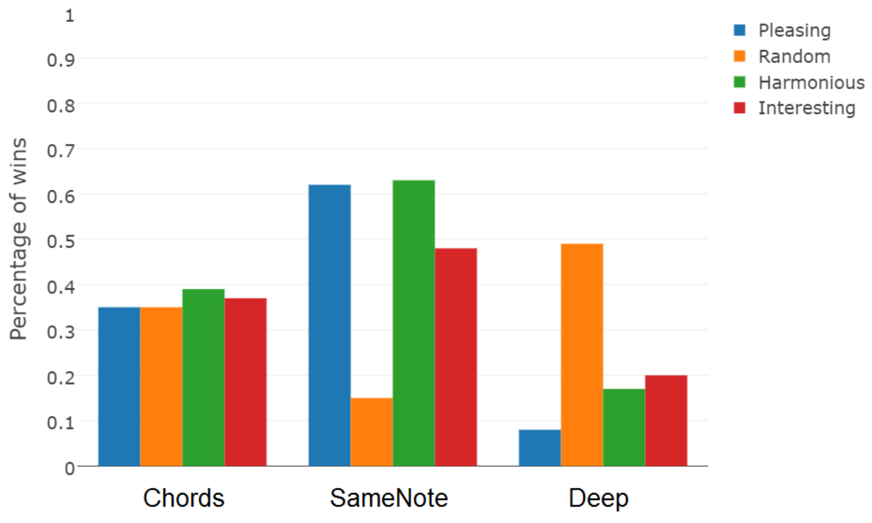


Figure 23: Rhythm win percentage in the four descriptor categories against the other configurations

10.1.10 SameNote Category Wins

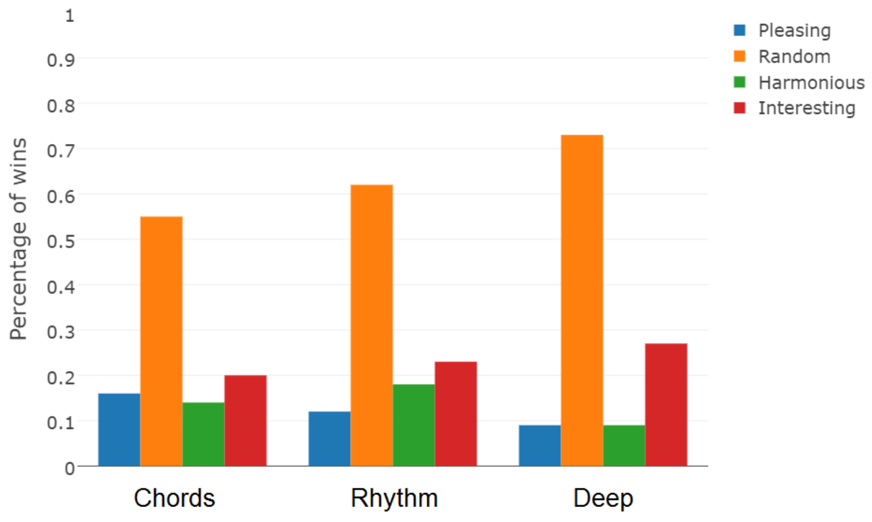


Figure 24: SameNote win percentage in the four descriptor categories against the other configurations

10.1.11 Deep Category Wins

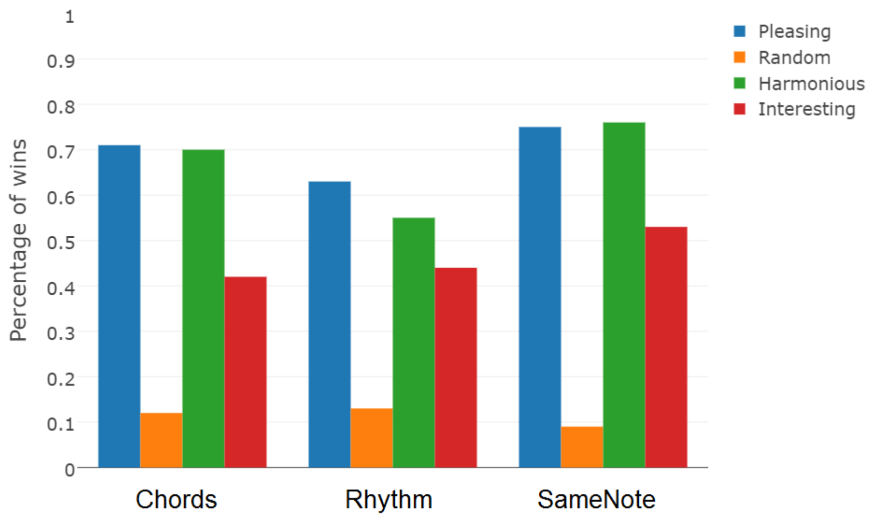


Figure 25: Deep win percentage in the four descriptor categories against the other configurations

10.2 Fitness Evolution Charts

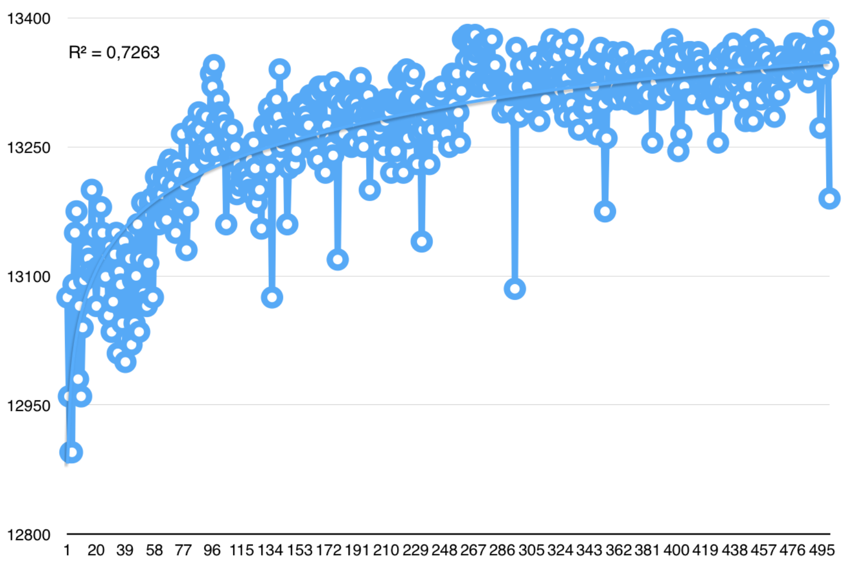


Figure 26: Rhythm fitness evolution with 2 modules over 500 generations

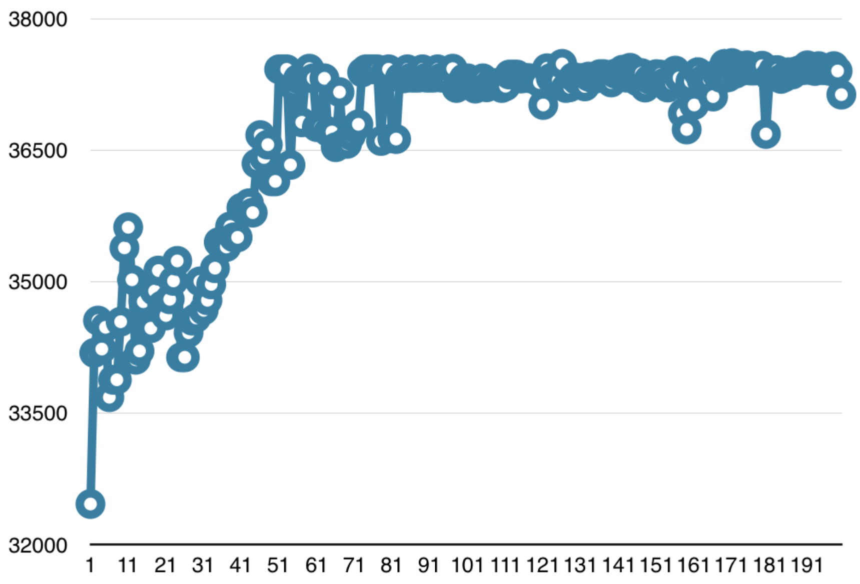


Figure 27: Pitch fitness evolution with 2 modules over 200 generations

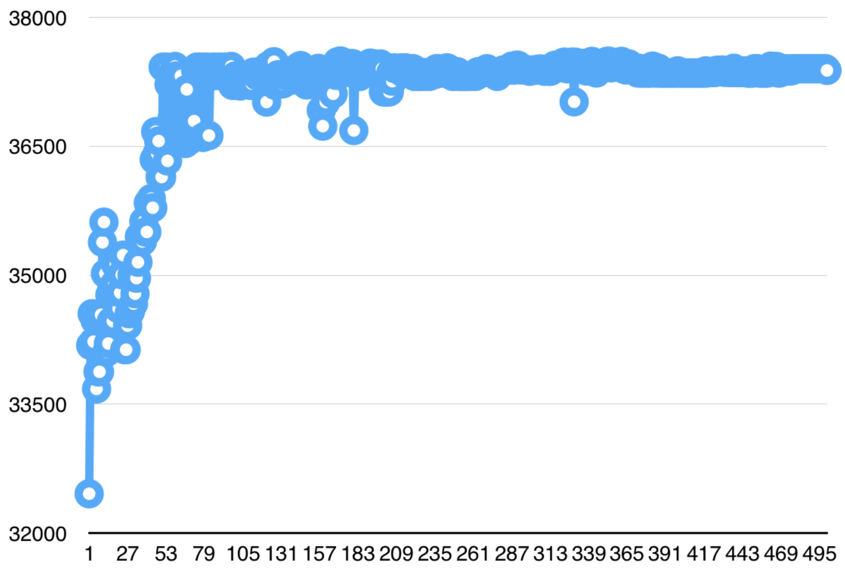


Figure 28: Pitch fitness evolution with 2 modules over 500 generations

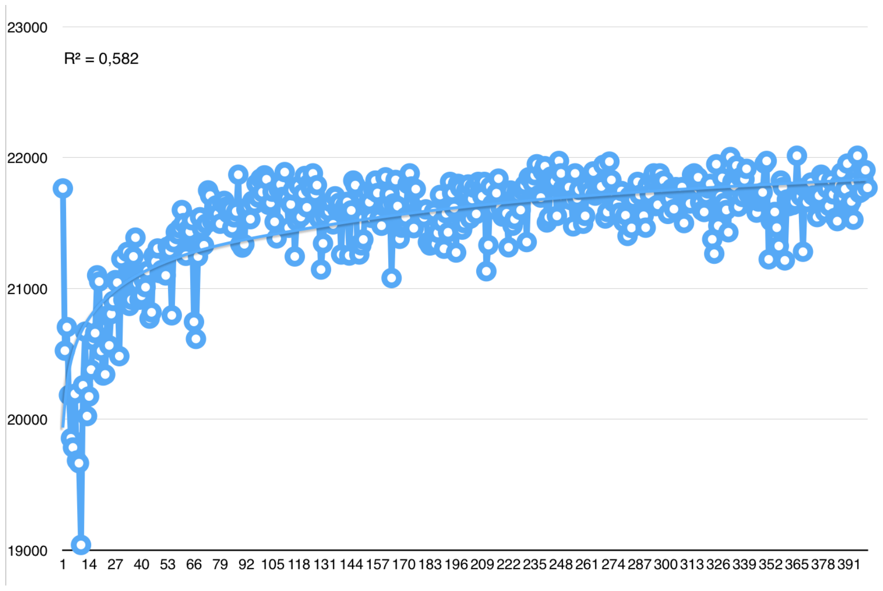


Figure 29: Rhythm fitness evolution with 4 modules over 400 generations

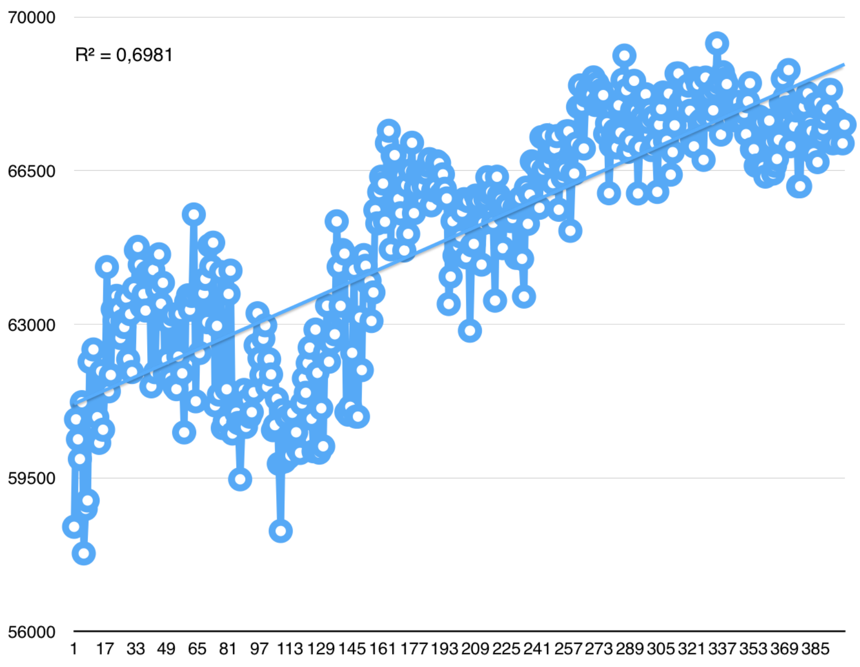


Figure 30: Pitch fitness evolution with 4 modules over 400 generations

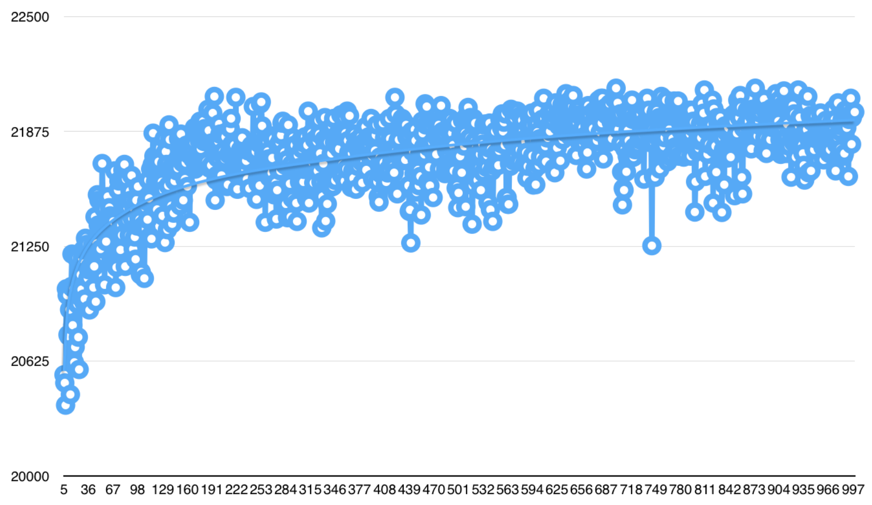


Figure 31: Rhythm fitness evolution with 4 modules over 1000 generations

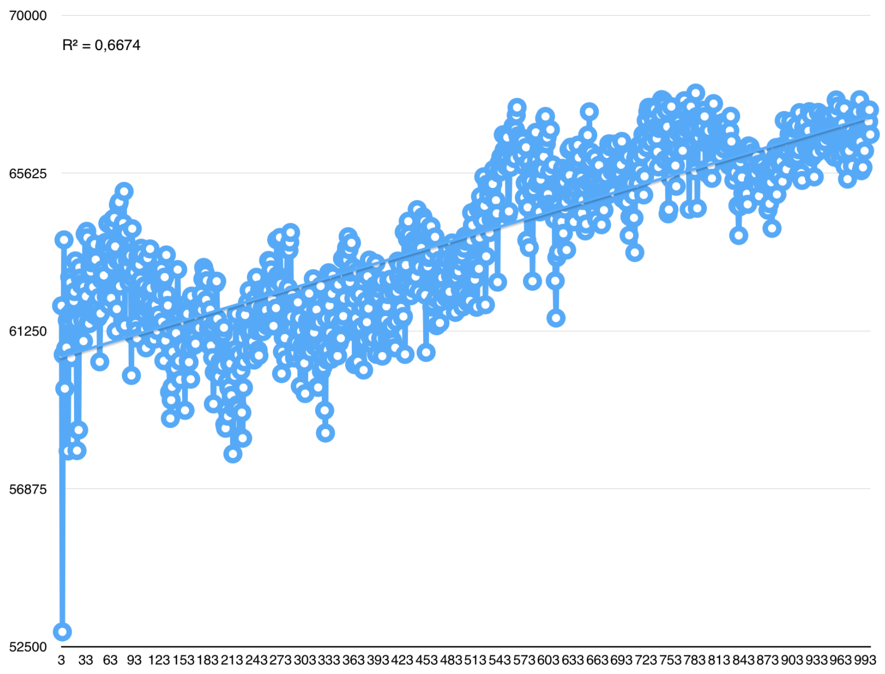


Figure 32: Pitch fitness evolution with 4 modules over 1000 generations

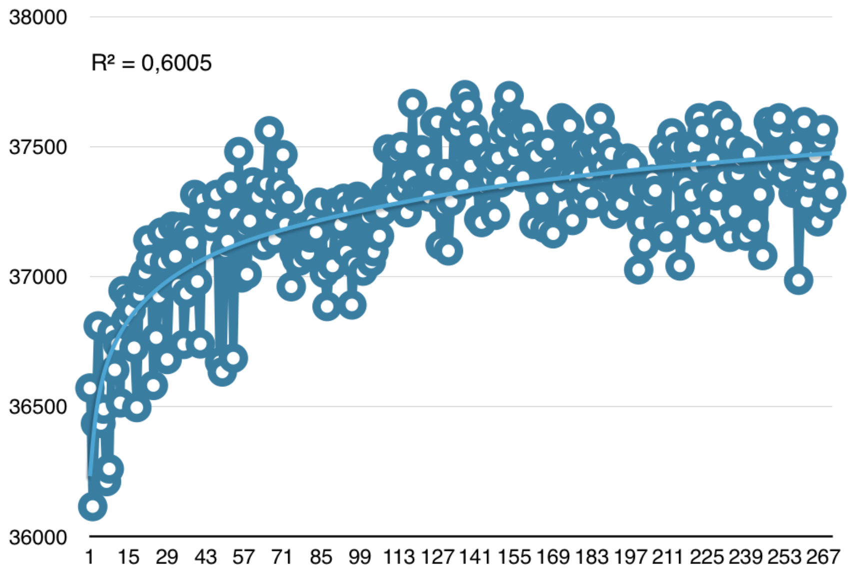


Figure 33: Rhythm fitness evolution with 6 modules over 270 generations

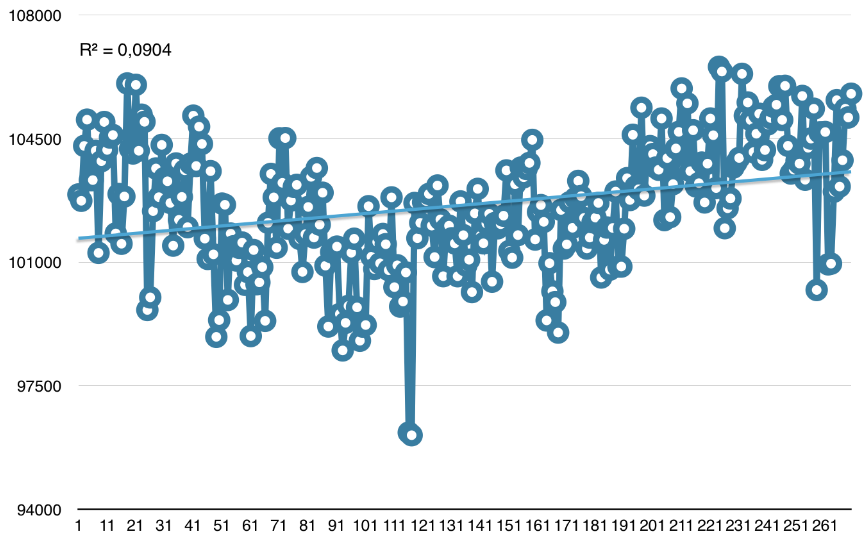


Figure 34: Pitch fitness evolution with 6 modules over 270 generations

10.3 Module Relations Output

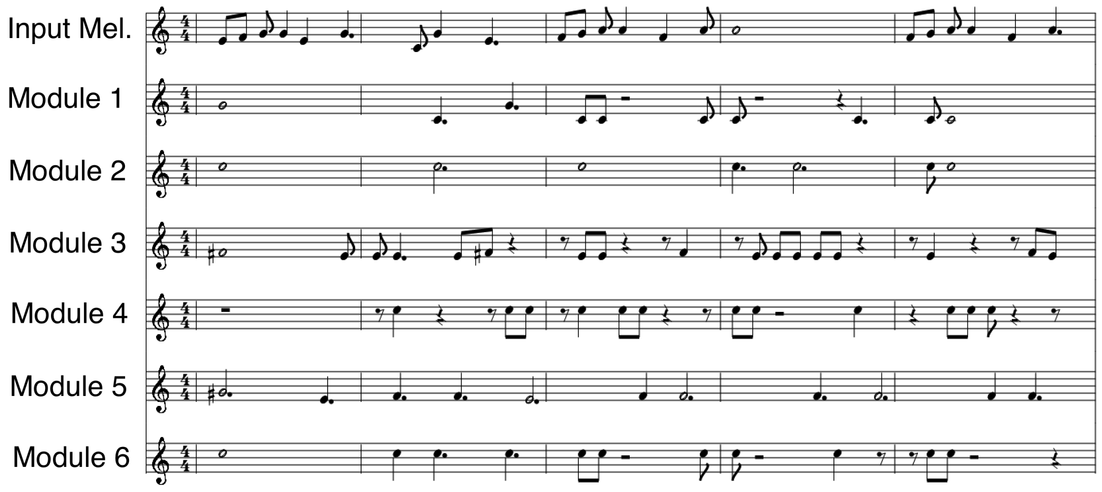


Figure 35: Output from evolution with six modules