School of Electronic Engineering and Computer Science

MSc in Digital Music Processing
Project Report 2015

Evolutionary Music Composition

Author
Csaba Sulyok

Supervisor
Dr. Andrew McPherson
Dr. Christopher Harte

August 2015

Disclaimer

This report, with any accompanying documentation and/or implementation, is submitted as part requirement for the degree of MSc in Digital Music Processing at the Queen Mary University of London.

It is the product of my own labour except where indicated in the text.

The report may be freely copied and distributed provided the source is acknowledged.

Acknowledgements

I would like to start by thanking my initial supervisor Dr. Christopher Harte for allowing me to contribute to his vision of the current project. I know that he holds dear the concepts of this thesis ever since their inception, therefore I am thankful to see the progress our collaboration could bring. I believe that without his passionate drive and great attention to detail, the current work could be nowhere near its level of quality.

I would also like to thank Dr. Andrew McPherson, who has helped us through some rough transitional periods in the lifecycle of this project. His helpful comments, encouragement and availability played an important role in keeping my faith in distance learning as a viable educational option.

My special thanks are extended to everyone involved in the 13th European Conference on Artificial Life (ECAL2015) for accepting the first version of the current work with open arms. Their encouragement and level of interest were a proof that our pursuits are worthwhile.

Lastly, I am grateful to my colleagues at Accenture Industrial Software Solutions, who have provided me their understanding with regards to my absence in the pursuit of this project.

Abstract

In this project we present a music composition system that uses a corpus-based multi-objective evolutionary algorithm. We model the composition process using a Turing-complete virtual register machine to render musical models. These are evaluated using a series of tests, which judge the statistical similarity of the model against a corpus of real music. Exploring the space of possible parameters, we demonstrate that the methodology succeeds in creating pieces of music that converge towards the properties of the chosen corpus. These pieces exhibit certain musical qualities (repetition and variation) not specifically targeted by our fitness tests; they emerge solely based on the similarities.

Contents

1 Introduction **6**

1.1 Document structure 7

2 Background **9**

2.1 Genetic programming 9

2.2 Literature review 10

3 Concept & design **13**

3.1 System overview 15

4 Genetic programming framework **16**

4.1 Population 16

4.2 Phenotype renderer 16

4.3 Fitness tests 17

4.4 Evolution strategy 18

4.4.1 Survival of the fittest 18

4.4.2 Natural selection 18

4.4.3 Crossover & mutation 18

4.5 Framework example 20

5 Virtual composer **22**

5.1 The genetic string 22

5.2 Evolving genetic strings 23

5.3 Interpreting genetic strings 23

5.4 Opcode interpretation 24

5.4.1 Arboreal opcode interpretation 25

5.5 Generating the opcode interpreter 26

5.5.1 DSL model 26

5.5.2 DSL templates 28

6 The musical model **30**

6.1 The model builder 31

6.2 MIDI support 31

6.2.1 Importing MIDI files 33

6.2.2 Exporting MIDI files 33

7 Corpus-based Similarity Tests **35**

7.1 Chosen corpus 35

7.2 Normal distribution tests 36

7.3 Model descriptors . . . 37
7.3.1 Histogram . . . 38
7.3.2 Histogram of first differential . . . 39
7.3.3 Spectrum . . . 39
7.3.4 Spectrum of first differential . . . 41
7.4 Descriptor correlation tests . . . 41
7.5 Corpus clustering . . . 42
7.6 Using a multivoice corpus . . . 44

8 Experiments & results **47**
8.1 Tested configurations . . . 47
8.2 Overall results . . . 48
8.3 Using different population sizes . . . 48
8.4 Single vs. dual track corpus . . . 50
8.5 Impact of survival mechanism . . . 51
8.6 Listening to the output . . . 52

9 Conclusion **53**
9.1 Future work . . . 53

A Supporting material **55**

B Tools and build management **56**

C Opcode interpreter instruction sets **58**
C.1 The immediate opcode interpreter . . . 58
C.2 The indirect opcode interpreter . . . 62

D Resulting MIDI files **65**

1 Introduction

The process of creating and assessing music is fundamentally subjective and hard to define. Thywissen (1999) describes it as 'an aesthetic search through the space of possible structures that satisfy the requirements of that process: in our case, creating interesting music'. Composers spend years perfecting and evolving their technique; to create new music successfully, both experience of the composition process and knowledge of existing 'good' music is required. The composer's judgement as to whether a new musical idea is good or bad will be a subjective decision based on their knowledge and memory of previous pieces (see Figure 1). As their creative process evolves, so too should the quality of their compositions.

The knowledge of external music is indirect: a listener only hears the output of the composition process, without intimate knowledge of the creation steps. This indirect involvement is the seed a starting composer may use. After the initial phase, later compositions are more directly inspired by own previous works. The composer is intimately familiar with the process used in his/her previous attempts; this process may be refined over time.

Since the development of a composer's skill can be viewed as an evolving process, it seems intuitive that evolutionary computation techniques should find utility in algorithmic music composition.

In this paper we model the composer as a Turing-complete virtual machine. His/her brain is viewed as a computer with a predefined set of instructions. These instructions mimic the real-world atomic steps taken to output a piece of music. The *program* running on this machine is the subject of evolution. While the instruction set is fixed, the order and parameters with which they are executed evolve over time.

This virtual machine is not limited to music creation. Its output is merely a set of bytes which receives context only once we view it as a musical piece. The machine and the evolu-

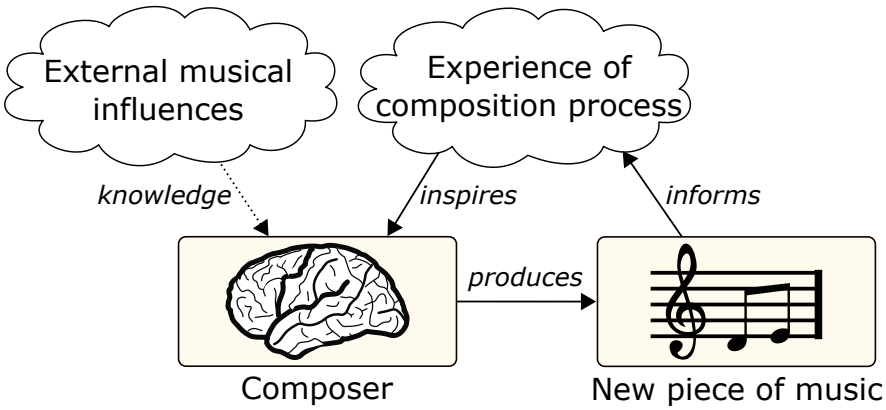


Figure 1: The creative process of a composer is informed by both experience of music in general (external influence of others' music) and experience gained through practice of the composition process itself.

tionary strategy of its programs could virtually mimic any creative process.

In our context, the output of the virtual machine is interpreted as a musical piece which is assessed by judging statistical similarity to a corpus of real music. The corpus represents the set of our virtual composer's external inspirations. We have no knowledge about the composition process leading to the music within the corpus, just as in real life a composer only sees the product of his/her idols' labour, not their steps in creating it.

The statistics we calculate to judge the similarity to the corpus are intuitively chosen based on musical traits they may represent. On the other hand, the comparison process is once again applicable to any corpus-based evolutionary process.

This research builds on the foundation set in our previous research (Sulyok et al., 2015) by exploring the space of different parameters. We aim to answer the question "What are the set of parameters that help the evolutionary algorithm most in its guided search for music similar to a corpus?".

Our results show musical pieces with a tendency for reoccurring patterns. While repetition-based musical traits emerge, other musical properties (such as harmony) are lacking and signal the need for tests more directly inspired by music theory.

1.1 Document structure

The remainder of the current document is structured as follows:

- Section 2 presents the concepts of genetic algorithms, genetic programming and how they have been used in the past for similar research.
- Section 3 outlines the intuition behind the choice of evolutionary algorithms and the design overview of the algorithm.
- Section 4 presents the design and implementation of an abstract genetic programming framework, which is extended in further sections for our purposes.
- Section 5 details the virtual machine as the representation of a composer's process. It introduces the genetic string (a condition of a virtual machine - our genotype), its evolution strategy and the details of execution. It furthermore presents how we generate code to interpret a genetic string using custom instruction sets presented in Appendix C.
- Section 6 shows how we represent music (our phenotype) and how a genotype is rendered into a musical model.
- Section 7 presents the assessment methods used to evaluate a musical model. It introduces the chosen corpus along with preprocessing steps we apply to it as well as the statistical transforms used for quality assessment.
- Section 8 presents the different configurations used in our experiments and discusses the results.

- Section 9 draws conclusions and presents possibilities for future extensions.
- Appendix A describes the supporting material attached to this document.
- Appendix B details the programming tools used to implement, build and run the algorithm, as well as possible command-line arguments.
- Appendix C details the instruction sets used by the virtual machine in the current tests.
- Appendix D shows a few randomly selected result models.

2 Background

2.1 Genetic programming

A *genetic algorithm (GA)* (Goldberg, 1989) is a computational solution search method inspired by Darwinian evolution theory. It hinges on evolving potential solutions by modelling natural selection on a population. The change and natural progression of this population is influenced by an arbitrarily defined *fitness* of each individual which determines the probability of reproduction and/or survival. Computational models based around the principles of evolution have been proposed since Turing (1950). Since then, genetic algorithms have found use in countless fields including computer science, mathematics, economics, bioinformatics etc.

Genetic programming (GP) (Cramer, 1985; Koza, 1992) is an extension of genetic algorithms. It evolves programs which produce solutions instead of evolving the solutions themselves. The intuition behind genetic programming lies in the separation of the *genotype* and *phenotype*. A genotype is the genetic composition of a unit which is modelled by the evolving program. The phenotype represents the observable traits of a rendered unit, i.e. the output of the evolving program. This separation implies the genotypes as the subject of evolution (they are spliced and mutated to create new individuals) and the phenotype as the subject of assessment.

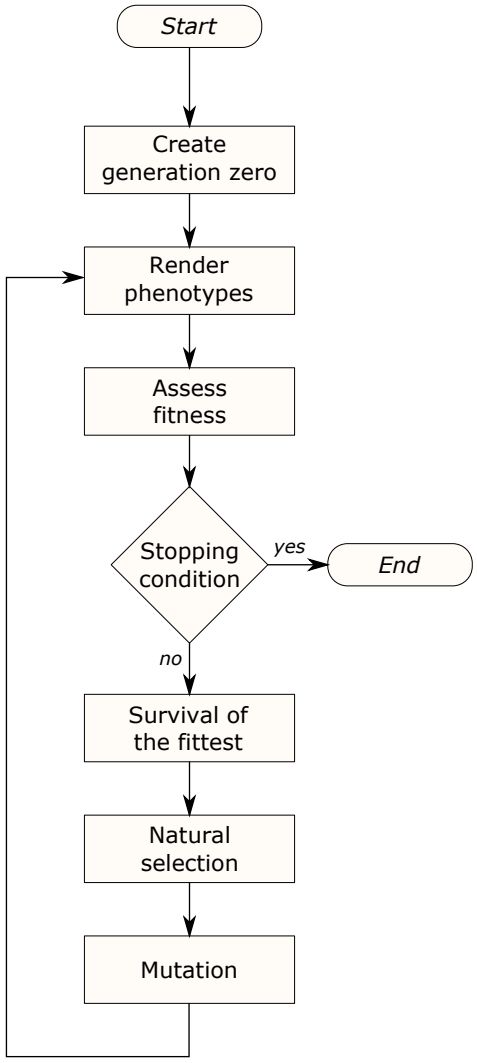


Figure 2: Typical workflow of a genetic programming algorithm

The generic steps of an algorithm using genetic programming are as follows (see Figure 2):

1. We start with an initial population of random genotypes (programs), which we name *generation zero*.
2. Phenotypes are *rendered* from the genotypes (the programs are run).
3. Phenotypes are *assessed* based on how good they solve the given problem(s): numeric grades are assigned to them.

4. A new generation is built using the following steps:

   (a) Some of the best units survive to compete in a new generation (*survival of the fittest*).
   (b) The process of *natural selection* selects the next generation's parents based on their fitness.
   (c) The *crossover* process creates new genotypes by splicing the genetic code of the chosen parents.
   (d) The new units are subjected to *mutation*.

5. Steps 2-4 are repeated until an arbitrary stopping condition is reached.

Early works on genetic programming (e.g. Cramer (1985)) used programming languages organized in a tree structure. *Linear genetic programming* (Brameier and Banzhaf, 2010) uses an alternative representation: an arbitrary sequential language. The current work uses linear genetic programming since we view the composition process as a program running on a Turing-complete virtual machine (details in Section 5). Although the code execution is linear, this does not exclude the emergence of arboreal structures in the musical output; branching instructions may push a program to output repeating notes, motifs or entire sections, creating a tree-like structure. This design choice does not exclude exploring a tree-representation in future work (see Section 9.1).

A solution as found by genetic programming may apply to multiple problems, not just one. *Multi-objective genetic algorithms (MOGAs)* (Fonseca and Fleming, 1993; Murata and Ishibuchi, 1995) extend conventional GAs by assessing fitness from multiple aspects. The different tests impact the overall fitness in different degrees, therefore we can represent the overall fitness as a weighted sum of the results of the independent tests. Multi-objectivity may be exploited to simulate the *"opposites attract"* effect (Dolin et al., 2002): favouring the mating of parents who excel at different tests (see Section 4.4).

A more recent modelling technique for evolutionary systems is *grammatical evolution (GEs)* (O'Neill and Ryan, 2001). This approach allows the incorporation of domain knowledge by using a problem-specific formal grammar.

2.2 Literature review

Using evolutionary models for composing music is nothing new; examples date back to the early 1990s. Most examples use genetic algorithms and not genetic programming for evolving music, either partially or in its entirety. However, the subjective nature of music quality makes defining appropriate machine fitness tests difficult (Hartmann, 1990; McIntyre, 1994; Miranda and Biles, 2007; Waschka, 2007). As a result, much previous research in this area attempts to narrow the search space.

One of the earliest applications of evolutionary algorithms in a musical context was performed by Hartmann (1990). His program *LUDWIG* evolved 'musical identities': an event-based representation of musical properties. He viewed all properties as different dimensions which exponentially increased the search space; he concluded that this resulting space was 'very difficult to handle'.

Gibson and Byrne (1991) proposed segmenting the composition process and encapsulating the resulting model in a neural network. This allowed the evolution of the network, resulting in a prototype able to produce short musical pieces (4 bars). Similarly, Chen and Miikkulainen (2001) have evolved neural networks to create melodies obeying Bartók's rules.

Horner and Goldberg (1991) used GAs to connect 2 musical motifs (an initial and final pattern). The algorithm searched for a route between the 2 patterns by applying small variations at each step. They have dubbed this method 'thematic bridging'. While their research purely used GAs to produce longer pieces of music, it still required the input of the 2 patterns. Similarly, Towsey et al. (2001) used GAs for 'melodic extension': generating musically appropriate succeeding segments for input melodies. Besides generating the extensions, their algorithm's assessment method was also aimed towards evaluation/assistance of music students.

To reduce the otherwise vast solution space of all possible music, many previous approaches have focused on evolving one particular musical property. Early examples include McIntyre (1994) and Phon-Amnuaisuk et al. (1999) who both used a GA to harmonise input pieces to obey preset harmony-related rules.

In more recent years, De Prisco et al. (2011) used multi-objective differential evolution to solve 'unfigured bass harmonization': the composition of new tracks around a given bass track. Martins and Miranda (2007) have attempted to evolve rhythmic patterns by modelling music 'as a cultural phenomenon whereby social pressure steers the development of musical conventions (in this case, repertoires of rhythmic sequences)'. Dostál (2012) explored autonomous fitness tests to evolve rhythm accompaniments by comparing with cues given by other instruments, e.g. bass. The novelty of Dostál's work also includes mutation operators inspired by music-theory, e.g. adding syncopation or changing stroke types.

Alfonseca et al. (2007) have used melody distancing as an assessment method to evolve pieces reminiscent of a certain author. Similarly, Nuanáin et al. (2015) attempted to evolve rhythm patterns by evaluating distances between patterns in the population and a preset 'target pattern'.

Another approach to help the guided search for good music is inputting the initial population rather than generating it randomly. Waschka's *GenDash* (Waschka, 2007) employs this technique to 'help compose' pieces rather than composing them entirely. Waschka has released a large number of pieces created with the aid of this program. Similarly, Eigenfeldt's *Kinetic Engine* (Eigenfeldt, 2009, 2012) evolves rhythmic and melodic patterns; its 'initial population is derived from an offline analysis of a corpus'.

Other previous approaches use *interactive genetic algorithms (IGAs)*: GAs relying on human feedback for fitness evaluation. For example, Horowitz (1994) and Tokui and Iba (2000) applied IGAs to evolve rhythm patterns, while Jacob (1995) used it for multiple musical properties. One of the more well-known early music-related IGAs is *GenJam* by Biles (1994): a program capable of evolving improvisatory passages. It used an arboreal IGA to model the evolution of an improvising jazz soloist. Other examples of IGAs include the MIDI-based *GeNotator* (Thywissen, 1999) and *SBEAT3* (Unemi, 2002). In 2012, MacCallum et al. (2012) created the online *DarwinTunes* community to crowd-source human feedback for choosing which music pieces will be selected for breeding.

Donnelly and Sheppard (2011) have presented a GA evolving entire four-part musical pieces only given a chord at initialisation. Their work also hinges on adaptive crossover and mutation rules during the run of the algorithm.

Some of the previous work has also focused on grammatical evolution. For example, Reddin et al. (2009) used GE to create short musical pieces which were evaluated with listening tests. A recent GE-based work by Loughran et al. (2015) composed piano pieces and accompaniments by incorporating Zipf's distribution to derive a suitable search space.

The work seen thus far has focused on evolving sheet music with no tempo changes or musical expression. Dahlstedt (2007) breaks this cycle by also evolving performance elements along with the pieces. He does this by modelling music composition as a binary tree with notes in the leaf nodes.

An exhaustive review of the different AI-based algorithmic composition methods has recently been published by Rodriguez and Vico (2013).

The current research differs from previous literature by incorporating linear GP elements via the virtual machine, effectively evolving the process rather than musical pieces. Our approach also differs in the way we utilise the corpus; we use it to assess fitness instead of deriving the initial population from it. We also attempt to model all musical properties exhibited by our corpus (this excludes velocity, see Section 7.1).

3 Concept & design

When designing an evolutionary system, we must first answer the question: "What is this system supposed to evolve?". Previous composition systems have generally attempted to evolve musical pieces but in this paper we propose evolving the composition *process* instead. We cast the action of composing a piece of music as a process running on a Turing-complete virtual computing machine. The virtual machine has a set of instructions that will be executed in a given order depending on the initial state of its memory (i.e. its program) and some way of writing notes onto a musical 'score' whenever an output instruction is encountered.

The development of the composer's skill then becomes a genetic programming problem; the genotype is the program string presented to the virtual machine and the phenotype its musical output. In such a system, the executing process on the virtual machine can hold internal structuring rules and information that are not visible in the final musical phenotype. Whorley et al. (2013) address this point in the context of melody harmonisation, the exact steps of which cannot be known just by listening to the piece of music. Decoupling the genetic program and the resulting musical phenotype in this way allows for the emergence of complex structures not possible in more constrained musical models. For example, a conditional branch statement in the program can cause the repetition of a single note or section, or perhaps even the entire piece depending on where it occurs. While the musical possibilities of this approach increase in number, the resulting space of possible outcomes is very large and hard to analyse, so careful design of fitness evaluation is necessary.

As mentioned in Section 1, the composer's creative process cannot operate in a vacuum; it is necessarily dependent on the influence of works from previous composers. In the same way, our evaluation process employs fitness tests that judge the similarity between statistical

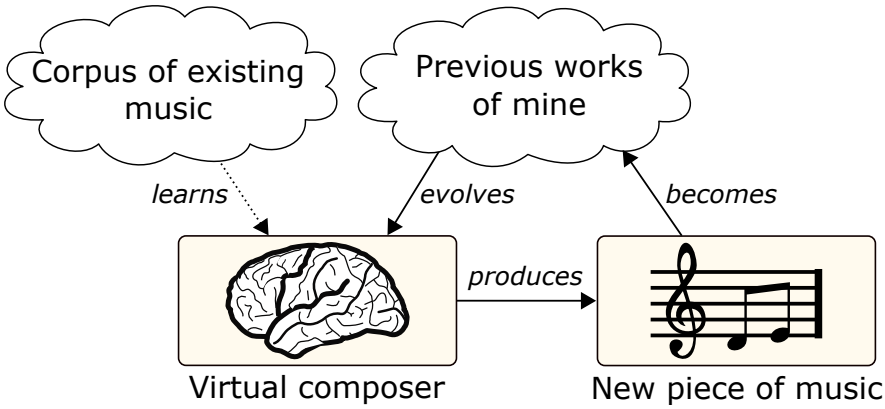


Figure 3: The intuitive way a virtual machine can model the composition process. The composer is modelled by a virtual machine whose thought process is a program. It is influenced by real music (the composition process of which is unknown), and by own previous works (more intimate knowledge).

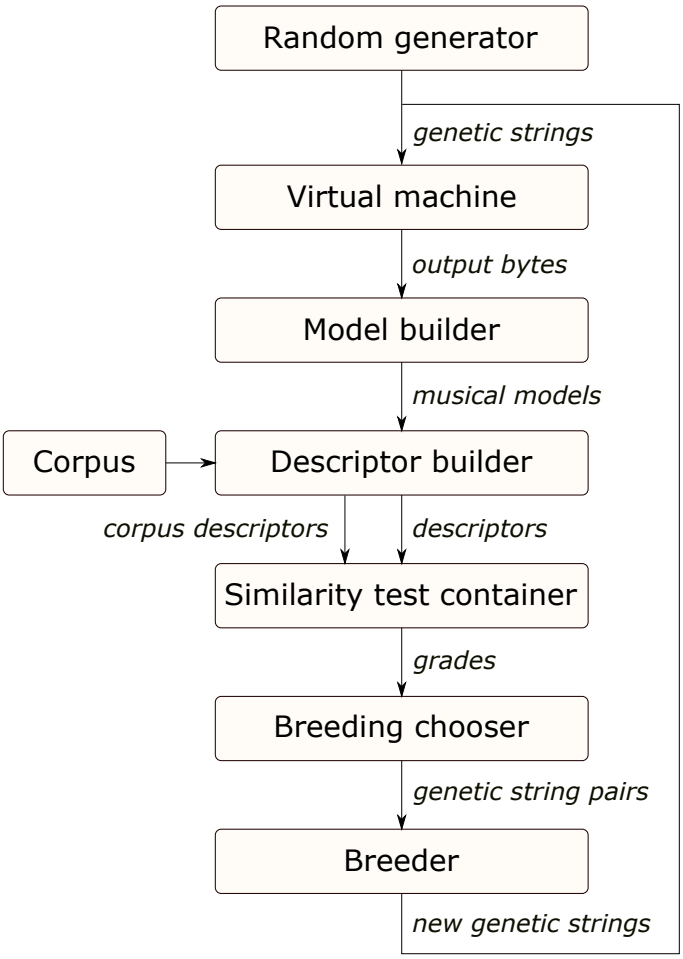


Figure 4: Workflow of the algorithm: A population of genetic strings is interpreted by the virtual machine and the resulting bytes are used to build models; statistical transforms are performed on the models to yield descriptors, which are used in similarity tests to assess fitness of the musical pieces. Based on these grades, genetic strings are bred and mutated to produce the next generation.

properties of the current musical phenotype and those of a chosen corpus of existing music (in this case, a group of Bach keyboard exercises). Our evolutionary algorithm is multi-objective in nature, incorporating tests for a number of different properties (Fonseca and Fleming, 1993).

Our aim here is to establish whether the use of virtual machine and corpus-based similarity tests together will help the system converge towards the properties of the chosen corpus. Although the phenotype in our current system is music, we propose this method as a generic one that can be applied to the evolution of any creative process.

3.1 System overview

Our system follows a conventional genetic programming structure (see Figure 4). An initial population of genotypes (genetic strings) is generated randomly then each individual is run on the virtual machine in turn (see Section 5). The instruction set used by virtual machine indirectly impacts the possible structure of the output music (e.g. branching instructions may dictate repetitions).

The output of the virtual machine is a byte stream that is parsed by a *model builder* to create our phenotype, a musical model (see Section 6). The structure of this model is completely independent of the structure and mechanism of the virtual machine. This two-stage approach to rendering the model deviates from previous musical evolutionary systems in that the genetic string is not directly used to build the phenotype, instead being *interpreted* by the virtual machine.

Each phenotype model is evaluated using a series of tests derived from the corpus. These tests focus on the underlying statistical properties of a model rather than the similarity of the data itself (see Section 7.3). High grades can therefore be achieved not only when a model is identical to a member of the corpus, but also when it is statistically similar to them. To produce these statistics, we subject models to a series of transform methods to produce a *descriptor*. The construct which creates a descriptor from a musical model is a *descriptor builder*. We refer to the complete set of tests as the *similarity test container*.

Based on the assigned grades, the *breeding chooser* chooses pairs of units for crossover and/or survival and the *breeder* splices these pairs to create offspring. The crossover and mutation process operates on the genetic strings producing a new generation of programs for the virtual machine to execute (see Section 5.2).

4 Genetic programming framework

In this section, we present an abstract genetic programming framework capable of modelling all building blocks seen in Section 2.1. The structure of genotypes/phenotypes are left undefined for now, to be concretized in later sections for our specific usage.

The separation of the framework from the musical implementation allows the emergence of a modular design, achieving *separation of concerns (SoC)* (Laplante, 2007). The detailed description of the project structure can be found in Appendix A.

4.1 Population

The exact definition of genotypes/phenotypes are unknown to us at this point; therefore, to retain the object-oriented and typesafe nature of the implementation, empty classes named `Genotype` and `Phenotype` are provided, which can be extended by the user. Section 5.1 and 6 describe how we these entities are made concrete for the composition problem.

Separation of concerns now allows us to construct the `Population` class, which holds the following information:

- *population size* - the number of units in this population (denoted as N);
- *genotypes* - a collection of `Genotype` instances;
- *phenotypes* - a collection of `Phenotype` instances;
- *grades* - the overall fitness test results of each unit, stored as floating-point numbers between 0 and 1 (see Section 4.3);
- *grades per test* - the framework supports a multi-objective context and this is where individual test grades are stored; it is a matrix of size $N \times T$, where T is the number of fitness tests;
- *ages* - each unit is attributed an age so the survival mechanism can discard too old units; for more details, see Section 4.4.

The `Population` class holds this data as separate arrays with colliding indices for code optimization and ease of use.

4.2 Phenotype renderer

We define a *phenotype renderer* as the construct which takes a genotype and constructs the appropriate phenotype from it. Since this process heavily depends on the nature of the genotypes and phenotypes, we can attain another abstraction, simply described as: a phenotype renderer takes genotype(s) and creates phenotype(s).

Therefore the underlying class, `PhenotypeRenderer`, will have an abstract method which takes a `Genotype` argument and returns a `Phenotype`. Since we have defined our `Population`

Code Snippet 1 Running a phenotype renderer on all individuals from a population, without the knowledge of the genotype/phenotype structures. Child classes must implement the `run` method.

```
class PhenotypeRenderer(object):
    '''
    Abstraction of a phenotype renderer, which takes genotype(s)
    and produces the corresponding phenotype(s).
    '''

    def runOnPopulation(self, population):
        '''
        Runs the phenotype renderer to produce phenotypes for each genotype
        in a given population.
        '''
        for phenotypeIndex in range(population.numUnits):
            # create new phenotype (self.run is abstract)
            newPhenotype = self.run(population.genotypes[phenotypeIndex])
            # assign to same index in population
            population.phenotypes[phenotypeIndex] = newPhenotype

        return population
```

as a collection of individuals, the `PhenotypeRenderer` can also take care to create phenotypes from *all* genotypes within a population (see Code Snippet 1).

4.3 Fitness tests

For the evaluation of any phenotype, we use a series of fitness tests. A fitness test (class: `FitnessTest`) is programmatically defined as a method which has a phenotype as input, and a floating-point *grade* as output.

To support multi-objectivity, we define a container (class: `FitnessTestContainer`), which contains a series of fitness tests. We will refer to the overall number of tests as T.

Given a phenotype p_k and test index t, the test $f_t(p_k)$ returns a *grade* $g_{k,t}$, which is a numerical value between 0 and 1. The phenotype's overall grade $\bar{g}_k$ is determined by a linear combination of the individual grades per test. We will refer to the coefficients of this equation as *importances*, since a higher coefficient means the test impacts the overall grade more. Every test f_t is assigned a predefined importance i_t. The importances are scaled to sum to 1 over all tests:

$$\sum_{t=0}^{T-1} i_t = 1 \tag{1}$$

therefore the overall grades will also lie between 0 and 1:

$$g_{k,t} \in [0,1] \quad \Rightarrow \quad \bar{g}_k = \sum_{t=0}^{T-1} i_t g_{k,t} \in [0,1]\,. \tag{2}$$

Tthe `FitnessTestContainer` class allows running all tests on a phenotype or a whole

population, and persists both the individual and overall grades.

4.4 Evolution strategy

After the assessment of a population, a new generation is created based on the grades of the current one. In our case, the size of the population remains the same over time.

4.4.1 Survival of the fittest

When creating generation n using the grades from generation $n-1$, we begin with survivor selection. Each unit's survival is determined by:

1. *their overall grades scaled by a fixed factor* - In our tests, this factor is 15% (details in Section 8.1). For example, a unit with an overall grade of 50% has a 7.5% chance of survival.
2. *their age* - Each unit is attributed an age of 1 upon creation. This value is then increased every times it survives. If it reaches a preset maximum age, it is no longer survived. In our tests, this maximum age is 3.

This probabilistic approach to survival differs from our previous research (Sulyok et al., 2015), where a preset number of highest scoring units were chosen. The impact of this change is detailed in Section 8.5.

4.4.2 Natural selection

The selection of parents for crossover is determined using *complementary phenotype selection* (Dolin et al., 2002). In this process, mothers are chosen based on roulette-wheel selection (Lipowski and Lipowska, 2011) and hypothetical best-case offspring are created by taking the maximum of the two potential parents' grades on each test. Fathers are then chosen through roulette-wheel selection on these hypothetical children rather than the original units.

This method exploits the multi-objective nature of our tests by assigning a high probability to the mating of parents who score high on different tests. The choosing process for survival/crossover is encapsulated in the class `GenotypeChooser` and an example can be seen in Figure 5.

4.4.3 Crossover & mutation

Before phenotype rendering and assessment, the genotypes themselves must be created. The `GenotypeBreeder` class groups together all functionality having to do with creating or altering genotypes. The following methods are declared:

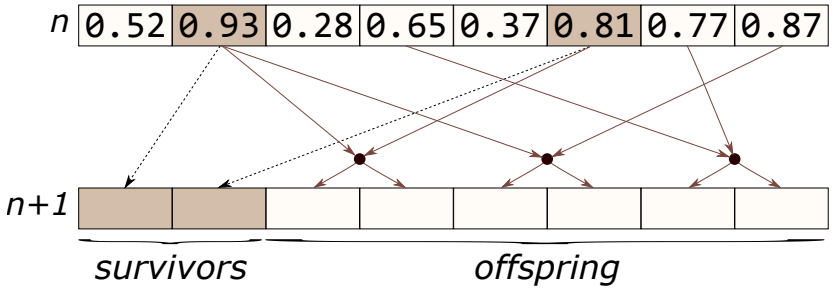


Figure 5: An example of choosing units to build generation $n + 1$ using generation n. The numbers in generation n represent the units' overall grades. The darker units are survived randomly based on their grades and age, while the remainder of the population is filled by choosing pairs of parents for crossover.

- *Creating the first genotypes:* The algorithm's initial population (generation zero) of genotypes is usually created by random means. We associate an abstract method method for this creation: `createGenerationZeroGenotype`; it creates a genotype from no arguments.
- *Crossover:* To create a new generation from an existing one, two genotypes must mate to create a new one. The method `breed` therefore takes 2 genotypes as input, and outputs 2 child genotypes.
- *Mutation:* Any new genotype can go through mutation independent from its parents' genetic code, therefore the method `mutate` will slightly alter a genotype.

Implementation details for our context can be found in Section 5.2.

The `PopulationBreeder` class encapsulates the `GenotypeBreeder` and `GenotypeChooser` instances. It is able to create a generation zero population and to breed a new generation from a previous one. The latter process is performed using the following steps (also seen in Code Snippet 2)

1. We choose the units for survival and crossover (see Figure 5) using the genotype chooser.
2. We survive the chosen units. Upon survival, only the genotypes are copied; they will be rendered and evaluated again, since adaptive phenotype renderers may produce a different phenotype and adaptive fitness tests may give different grades in the newer generation. Survival also increases a unit's age by 1.
3. We use the genotype breeder to perform crossover on the chosen pairs of parents, resulting in two offspring per pair.
4. We mutate the offspring and store them in the new `Population` instance.

Code Snippet 2 Creating a new generation from a previous one: choosing survivors/parents and performing survival, crossover and mutation.

```
class PopulationBreeder(object):
    # ...
    def breedNewGeneration(self, oldPopulation):
        '''
        Breeds a new generation from a previous one. Chooses parents and survivors
        using the genotype chooser, and performs breeding/mutation using the genotype breeder.
        '''
        newPopulation = Population(self.numUnits, self.numTests)

        # choose indices for survivors and parents
        (survivors, survivorCount, parents, parentCount
            ) = self.genotypeChooser.chooseUnits(oldPopulation, self.importances, self.maxAge)
        # survive chosen units to new population
        newPopulation.survive(oldPopulation, survivors)

        for unitIndex in arange(parentCount):
            # select parent genotypes
            mother = oldPopulation.genotypes[parents[0, unitIndex]]
            father = oldPopulation.genotypes[parents[1, unitIndex]]
            # crossover & mutation
            offspring1, offspring2 = self.genotypeBreeder.breed(mother, father)
            self.genotypeBreeder.mutate(offspring1)
            self.genotypeBreeder.mutate(offspring2)
            # assign genotypes to new generation
            newPopulation.genotypes[survivorCount + 2 * unitIndex] = offspring1
            newPopulation.genotypes[survivorCount + 2 * unitIndex + 1] = offspring2

        return newPopulation
```

4.5 Framework example

Before applying the framework to the music composition process, we test it using a simpler example. This allows us to test and measure the performance of the abstractions presented in this section.

The chosen example is finding the maximum value of a 2D function. Although this problem is not typically solved with genetic programming, it is simple enough to test our framework. The following classes are introduced to represent our example:

- Genotype: `TwoDFuncArg` - the arguments of the 2D function: x and y
- Phenotype: `TwoDFuncResult` - the result of applying the function to the genotype: $f(x, y)$
- Phenotype renderer: `SinCosProductPhenoRenderer` - applies the function to create the result from the 2 arguments. In the current tests, we have used

$$f(x, y) = 10 \sin(x) \cos(y) \tag{3}$$

- Fitness test: `TwoDFuncSigmoidMaxFitnessTest` - assigns a grade to a phenotype. We are testing for maximum values and are constrained to give numeric results between 0

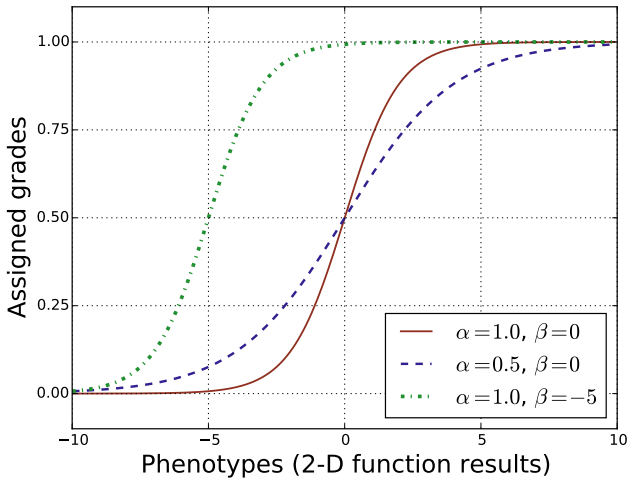


Figure 6: Assessing the output of a function: we scale its value to the 0-1 range using the sigmoid function (see Equation 4).

and 1. To conform to both of these criteria, we use a sigmoid function:

$$s_{\alpha,\beta}(x) = \frac{1}{1 + \mathrm{e}^{-\alpha x+\beta}} \tag{4}$$

where α and β are defined in advance (see Figure 6 for impact of different values).

- Genotype breeder: `TwoDFuncArgBreeder` - contains the edges of our search space ($x_{\min}$, $x_{\max}$, $y_{\min}$ and $y_{\max}$) and provides the evolutionary strategy for arguments:
  - Generation zero arguments are randomly generated by sampling a uniform distribution between $x_{\min} : x_{\max}$ and between $y_{\min} : y_{\max}$.
  - We represent arguments as vectors on a 2D plane where the 2 arguments represent the dimensions. Given that $\vec{p_0}$ and $\vec{p_1}$ are 2 chosen parents, crossover creates 2 offspring using:

$$\vec{o_0} = \vec{p_0} + (\vec{p_1} - \vec{p_0})\alpha \tag{5}$$
$$\vec{o_1} = \vec{p_0} + (\vec{p_1} - \vec{p_0})(1 - \alpha) \tag{6}$$

    where α is a random number between 0 and 1.
  - Mutation generates a random vector $\vec{\beta}$ which is added to an offspring.

These classes are stored in a separate project (see Appendix A) only having knowledge of the framework classes and not the core ones we use for the musical implementation.

Testing with a population size of 20, the example consistently finds the maximum value for the function within the first 50 generations. Most units in the population arrive at the same maximum within the first 100.

5 Virtual composer

We define a virtual machine as a Turing-complete von Neumann register machine which mimics the workings of a processor. It contains the following memory segments (also shown in Figure 7):

- *64KB Random access memory (RAM)* - contains the actual program. Since this is a von Neumann machine, the RAM is also part of the data used by the program, therefore it may overwrite itself during execution.
- *16-bit program counter* - points to the location in the RAM where the next instruction resides; it is capable of addressing any position in the RAM and is increased circularly during execution.
- *256 byte stack* - a segment of memory where the program may push or pop data from the registers or the RAM.
- *8-bit stack pointer* - points to the location in the stack where the program will push to/pop from.
- *8-bit accumulator register* - a register with which arithmetic/logical instructions are performed.
- *16-bit data pointer* - a helper register capable of addressing any position in the RAM.
- *a set of flags* - a status register which contains additional information on the state of the processor; currently only a carry flag is used in arithmetic operations.
- *8 8-bit general purpose registers.*

5.1 The genetic string

Our genotype, the genetic string, is the initial condition of the virtual machine; it encompasses the initial value of the random access memory, the stack and all registers. When a genetic string is fed into the virtual machine, all these segments are set as shown in Figure 7.

Since our virtual machine is a Von-Neumann machine, it can overwrite its own memory while the process executes. To avoid the genetic string being changed in this process, it is

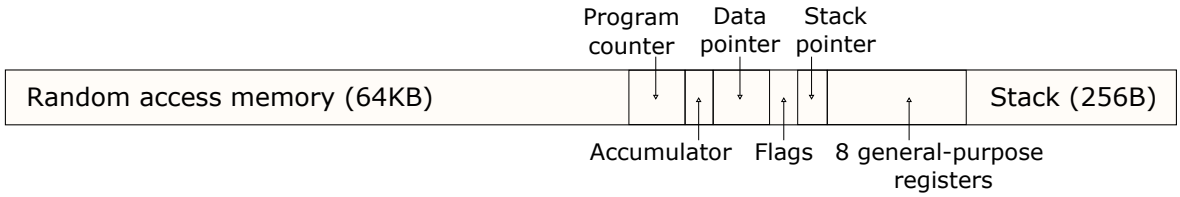


Figure 7: Structure of a virtual machine. It contains a 64KB random access memory (RAM) and a 256B stack, which can be addressed by the 16-bit program counter and the 8-bit stack pointer, respectively. Other registers include 8 general-purpose 8-bit registers, an accumulator, a 16-bit data pointer and a set of flags.

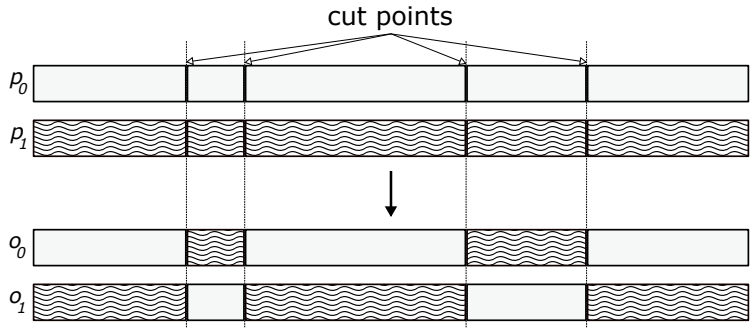


Figure 8: Visualization of the crossover process; random cut points are chosen on the parents and offspring created from their recombination.

always copied rather than moved in the feeding process. This ensures that interpreting the same genetic string multiple times will always result in the same output. Therefore we can safely save, recall or re-evaluate any genotype.

5.2 Evolving genetic strings

As mentioned in Section 1, our subject of evolution is the composer's process rather than the product. Our genetic string represents this process, therefore we are evolving programs running in the same preset virtual machine (Nordin and Banzhaf, 1995).

When creating and breeding genetic strings, we do not concern ourselves with how their underlying data will be used later on, we simply view them as byte arrays. In the first run of the algorithm, the generation zero genetic strings are randomly generated byte arrays of the given size. In all subsequent generations, offspring are spliced versions of their parents. We will refer to the genetic strings of the parents as p_0 and p_1.

With the parents p_0 and p_1 chosen, the *genetic string breeder* builds two new genetic strings o_0 and o_1. It chooses a random number of cut points, separates p_0 and p_1 into chunks and populates the offspring (Figure 8): one chunk is taken from one parent, the next from the other.

Children o_0 and o_1 are then mutated by taking a random number of byte indices, and randomizing those bytes. The ratios of maximum cut points and maximum mutated bytes to the genetic string size are predefined for each run of the experiment.

5.3 Interpreting genetic strings

Running a program on the virtual machine is equivalent to reading 8-bit values from the RAM where the program counter points, and executing the corresponding instructions defined in the instruction set. The program counter is increased before the execution, since a branching instruction may change the program counter's value.

Execution is terminated by reaching either a halt command or a preset maximum number of commands or outputs. The latter conditions were added to avoid infinite loops.

The instruction set contains the following types of instructions:

1. *Data transfer* - copy a segment of the RAM or a register to another segment of the RAM or to a register;
2. *Arithmetic & Logic* - perform simple arithmetic/logical functions on the accumulator register and, optionally, another value from the RAM or a register;
3. *Branching & conditional instructions* - move the program counter to a different location, optionally based on a condition;
4. *Machine control* - internal commands such as halting or pushing/popping using the stack;
5. *Output commands* - outputs a value from the RAM or a register; this is the main customisation we use to make the virtual machine serve our purpose; while all other command types model the composer's thought process, this type represents the actual writing to paper.

For the current test runs, we have defined two different instruction sets, which only differ in their inclusion of immediate addressing. The *immediate* instruction set allows instructions to take the next byte(s) in the RAM as their parameter(s). In these cases, the program counter is increased appropriately with the number of bytes used as parameters. Conversely, the *indirect* instruction set only allows parameters from the registers or a point in the RAM where a register points to. The complete list of instructions and their descriptions can be seen in Appendix C.

Since all possible 8-bit values are mapped to an instruction, any bytearray with the aggregate size of all the segments may be a condition of the virtual machine. Therefore any bytearray with this size may constitute a genetic string. Since a genetic string fully represents a state of the virtual machine, it is ensured that feeding and interpreting the same one multiple times will always give the same results.

The output bytes returned by the interpretation are used to build our musical model, as detailed in Section 6.1.

5.4 Opcode interpretation

We define an *opcode interpreter (OCI)* as the module interpreting the code of a virtual machine. It reads 8-bit values from the RAM, mapping them to our custom instruction set (see full table of used instruction sets in Appendix C). The following scenarios may arise when parsing the bits of an incoming value:

1. We know which instruction it is after n bits where $n < 8$ and the remaining bits get discarded;

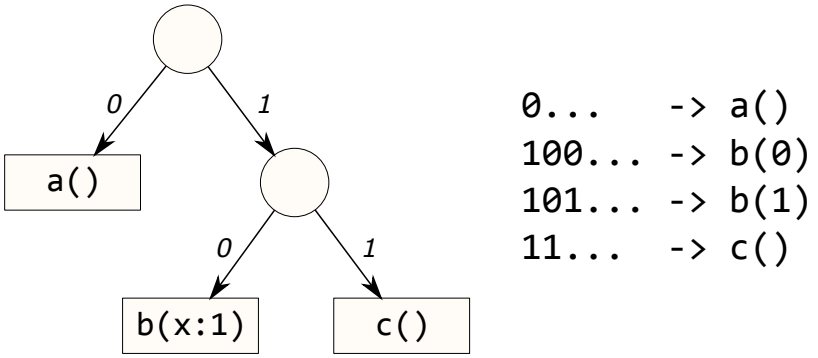


Figure 9: Example of an opcode interpreter's structure. Circles represent pending nodes and rectangles represent assigned nodes with the mapped instruction name. 3 instructions are mapped to the opcode prefixes 0, 10 and 11. `b` is also attributed a parameter of 1 bit. The right side of the figure shows the different cases for incoming opcodes.

2. We know which instruction it is after n bits where $n < 8$ and the remaining bits are used as parameter(s) for the instruction itself;
3. All bits are needed to make an informed decision on which instruction to execute.

5.4.1 Arboreal opcode interpretation

A possible approach to mapping opcodes to an instruction would be to build a full map with all possible opcode values as keys (256 in this case) and instructions as values. Using this approach would result in the same instruction (possibly with different parameters) mapped to multiple opcodes in scenarios 1 and 2. Therefore we use a *binary radix tree* to simplify the representation in these cases. Interpretation is performed using the following steps:

- In the beginning, we are in a *root node*, where we do not yet know the instruction mapped to the opcode.
- We read a bit from the opcode, which takes us down one of two pathways based on the bit. This takes us to the next layer in the tree.
- At any time, we can be in one of two nodes:
  - In a *pending node*, we still are not sure which opcode we have, so we continue reading.
  - In an *assigned node*, an operation has been assigned to the bits read thus far. Therefore we have correctly identified the instruction we must execute, so the rest of the bits can be taken as parameters or discarded. All assigned nodes are leaves in the tree, since no more reading is necessary.

Since we do not know which instruction an opcode is mapped to before reading the first bit, the root node is usually a pending node. Otherwise, the system would only have one possible instruction assigned to the root node.

Figure 9 shows an example of an OCI and its resulting interpretation cases. The OCI contains 3 mapped instructions, called `a`, `b` and `c`. `b` takes a 1-bit parameter called `x` to be read from the subsequent bits after the assigned prefix.

We consider an OCI *complete* if all possible values are mapped to an instruction, i.e. if all pending nodes have 2 children and all leaves of the tree are assigned nodes (the example in Figure 9 is therefore complete).

5.5 Generating the opcode interpreter

Upon exploring implementation options for an OCI, we can differentiate between two possible approaches:

- Representing the previously shown radix tree structure. This would be a more readable/elegant approach, but would reduce performance.
- Using a series of conditions/branches to determine the instruction. This low-level approach would give the best performance, but would be harder to read or modify.

To overcome this problem, we can take the best of both approaches by *generating* the opcode interpreter itself. The input to the generator uses a tree structure, while its output is low-level code filled with conditional statements, which will ultimately be used by the algorithm. The generator uses a *domain-specific language (DSL)* (Fowler, 2008): a specialization of a regular language that restricts its usage to a given set of keywords. DSLs are often used in code generators because they rely on two simple layers: a *model* built up by the keywords (not to be confused with our musical model); and *templates* which represent the structure of output files.

5.5.1 DSL model

The classes used for the binary radix tree modelling of the DSL are as follows:

- `Node` - an abstraction of a node in our tree.
- `PendingNode` - an implementation of `Node` which has 2 children, also of type `Node`. These are the children assigned to bits 0 and 1.
- `AssignedNode` - an implementation of `Node` which holds an actual instruction. It contains the instruction's name, its description and parameters, if any. It has no more child nodes since arriving at an assigned node means we have found the mapped instruction.
- `OpCodeInterpreterModel` - the main model object which contains the root `Node`.

Table 1 shows the global properties of an OCI, Table 2 shows the keywords of the DSL and Code Snippet 3 shows the model for the example OCI as seen in Figure 9.

Property name	Description
`name`	Name of OCI; used in class name patterns and generated documentation.
`counterSize`	Size of program counter in bytes, set by default to 2 (16-bit). The RAM's size is $2^{\text{counterSize}}$ so the program counter can access every byte of it; e.g. an 8-bit program counter will yield a 256-size RAM.
`numRegisters`	Number of general-purpose 8-bit registers, by default 8. They are always stored sequentially in the memory, so 8 8-bit registers can also be used as 4 16-bit registers, 2 32-bit registers etc.
`description`	Text description used in comments and generated documentation.
`outputDir`	Folder where rendered files should be placed.

Table 1: Global properties of the OCI model

Keyword/parameter	Description
`reg`	Adds a new register to virtual machine. Initially only the program counter and the general purpose registers are available.
`-> name`	Register name
`-> size`	Size of register in bytes. By default, 1.
`mem`	Adds an additional memory segment besides the RAM.
`-> name`	Memory segment name
`-> size`	Size of memory in bytes. By default, 256.
`assign`	Assign an instruction to a certain opcode prefix.
`-> opCodePrefix`	Bit string denoting opcode prefix.
`-> instructionName`	Name of instruction: a method stub will be created by this name.
`-> parameters`	Collection of parameter names and the number of bits each parameter uses. By default, unset.
`-> description`	Text description used in comments and generated documentation
`assignBlind`	Assign an instruction to the first available opcode prefix of maximal length.
`-> instructionName`	Name of instruction: a method stub will be created by this name.
`-> parameters`	Collection of parameter names and the number of bits each parameter uses. By default, unset.
`-> description`	Text description used in comments and generated documentation

Table 2: Description of opcode interpreter DSL keywords and their parameters

Code Snippet 3 Configuration steps of the example OCI in Figure 9: setting global properties, adding an extra memory segment, an extra register and assigning opcode prefixes to instructions.

```
oci = OpCodeInterpreterModel(name = 'Example',
                             counterSize = 1, # 8-bit program counter => 256 byte RAM
                             description = 'Example OCI')

oci.mem(name = 'extraMem', size = 256)        # additional memory segment
oci.reg(name = 'extraMemPtr', size = 1)       # pointer which can fully access extraMem

# Assign instructions
oci.assign(opCodePrefix = '0',  instructionName = 'a')
oci.assign(opCodePrefix = '10', instructionName = 'b', parameters = {'x':1})
oci.assign(opCodePrefix = '11', instructionName = 'c')
```

The instruction assignment process will raise an error in the following cases:

- Trying to assign to an opcode prefix already in use. For example, after an assignment to 010, assigning an instruction to 0, 01, 010 or 010... would give an ambiguous interpretation.
- If the model is not complete upon finalizing it (there are uncovered possible opcodes).

5.5.2 DSL templates

Template files represent the structure/formatting of output files, to be populated with variables from the model upon rendering. We define the following templates:

- *The stub of the opcode interpreter* - A header and class file containing properties for the RAM, memory segments and registers, along with the low-level conditional statements which deduce the next instruction by reading and parsing values from the RAM. This file is always overwritten upon regeneration. Code Snippet 4 shows the relevant lines of the header when using the example OCI seen in Figure 9.
- *The child opcode interpreter* - A second source file containing the skeletons for the assigned instructions. Our DSL only does the mapping to the instruction name, the instruction's logic is written afterwards. This file is not automatically overwritten upon regeneration, since the instructions' logic is added by hand.
- *Wrapper* - A Python wrapper for the class seen above (details on tooling can be seen in Appendix B).
- *Hypertext descriptions of the instruction set* - Optional description tables of the instruction set given in HTML or LaTeX format, the latter of which is used in Appendix C.

Using the separation of model and template allows simple alterations. For example, moving the OCI over to another programming language can be achieved by adding new templates, but using the same model.

Code Snippet 4 Relevant lines of output for the C++ header generated from the example OCI. External classes may feed it genetic strings with the size given in the macro in the first line, interpret the program, then recall the output using a getter.

```
#define EXAMPLE_GENETIC_STRING_SIZE 522

/**
 * Class for op-code interpreter Example
 * Description: Example OCI
 */
class ExampleOci
{
public:

  ExampleOci(unsigned int maxCommands, unsigned int maxOutputs, bool haltAllowed);
  ~ExampleOci();

  /**
   * Sets condition of VM using a genetic string (stored in inp).
   */
  void setFromGeneticString(unsigned char* inp, int inp_size);

  /**
   * Interpret RAM commands.
   * Reads bytes from where program counter points, and executes the mapped instructions.
   * Stops if halt flag set or if a maximum number of commands/outputs reached.
   */
  void interpret();

  /**
   * Recall current output.
   */
  void output(unsigned char* outp, int outp_size);

  // ==================================
  // assigned states - main instructions
  // ==================================
  void a(); // 0 -
  void b(unsigned char x); // 10 -
  void c(); // 11 -


private:

  unsigned char _counter; // program counter
  unsigned char* _registers; // general-purpose registers
  // additional registers
  unsigned char _extraMemPtr;

  unsigned char* _ram; // RAM
  // additional memory segments
  unsigned char* _extraMem;

  unsigned char* _output; // output memory block
  unsigned int _outputPtr; // output write pointer
};
```

6 The musical model

The algorithm's phenotype represents a piece of music (*musical model*); it is a set of tracks, each consisting of a set of notes. Each note has the following properties:

1. *Inter-onset interval (IOI)* - The time elapsed between the onset of the previous and current note in a track. For the first note in a track, it is the time between the beginning of the piece and the note onset. The unit of measurement for this property (called a tick) is mapped to sheet music time using the global property *ticksPerQuarterNote* and this is mapped to real-world time using the global property *quarterNotesPerSecond*. In our tests, we use 4 ticks per quarter note (the unit is a 16th note) and 120 BPM.
2. *Onset* - The absolute point in time of the note onset.
3. *Offset* - The absolute point in time of the note offset. IOIs and durations fully represent the timing of a note, onset and offset are included only for auxiliary analysis methods.
4. *Duration* - Time elapsed between the onset and offset of the note.
5. *Pitch* - A numeric value representing the note's pitch between 0 and 127, mapped to the same pitches as defined in the MIDI[1] protocol. The value 69 is associated with the 440Hz concert A, an increase or decrease of one unit representing a pitch rise or fall of one semitone respectively.
6. *Velocity* - The volume (intensity) of the note between 0 and 127. Our current experiments always use velocity 127 because of the chosen corpus (see Section 7.1).

Figure 10 shows an example of a track containing 2 notes; it is represented by an instance of the class `Track` containing a 2-D matrix with 6 rows and n columns where n is the number of notes in a track.

A musical model consists of a set of tracks. It cannot be encoded as a single 3 dimensional matrix since different tracks may have a different note count. Therefore the class `Model` contains a collection of `Tracks`.

[1] Musical Instrument Digital Interface

	Note 0	**Note 1**	
Inter-onset intervals	0	2 (*8th*)	...
Onsets	0	2	...
Offsets	2	4	...
Durations	2 (*8th*)	2 (*8th*)	...
Pitches	50 (*D3*)	52 (*E3*)	...
Velocity	127	127	...

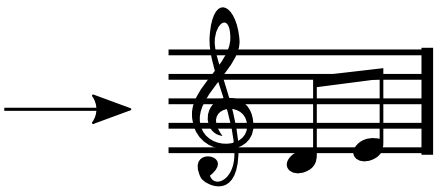


Figure 10: A track from a musical model. Each column represents a note with six properties (rows).

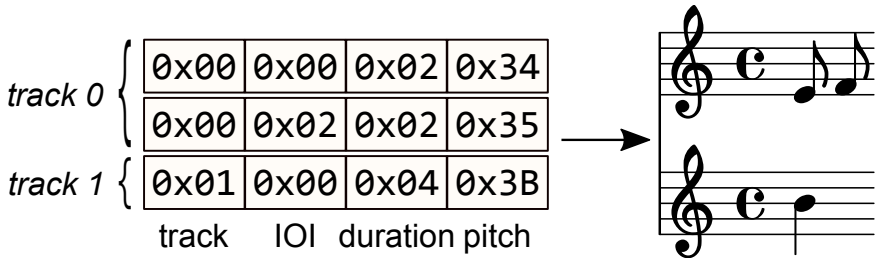


Figure 11: The model builder turns a byte array into a musical model by taking 4-byte chunks and adding a note to the model based on the chunks' values. The first byte denotes the track index and the next 3 are the note properties.

6.1 The model builder

The `ModelBuilder` class transforms any byte array into a musical model. It segments the input into 4 byte chunks, each representing a note. It uses the first byte to determine the track index and the remaining 3 for the note properties (see Figure 11). Velocity can be included optionally by setting the flag `velocityEnabled` (in which case 5 bytes are read per note); we do not use it in our current experiments because of the chosen corpus (see Section 7.1).

The model builder class contains the following properties:

1. `numTracks` - Number of tracks in the output model. The incoming byte representing the track index is masked to constrain this value.
2. `omitZeroDurations, omitZeroPitches, omitZeroVelocities` - Flags which indicate ignoring a note if a relevant property is 0. In our experiments, all of these flags are set. IOI is not included as a flag, since an IOI of 0 represents multiple notes played in unison.
3. `ioiMask, durationMask, pitchMask, velocityMask` - Numbers used to mask incoming bytes when setting note properties; used to constrain their values. In our test cases, we constrain the IOI and duration of each note to a maximum of 16 (the longest possible note is the whole note) and the pitch to 128 (to comply with the MIDI standards).

Our algorithm feeds the output of the virtual machine into the model builder, concluding the two-step process of phenotype rendering: building a musical model from a genetic string.

6.2 MIDI support

Our musical model is largely based on the MIDI format (Stansifer), therefore converting it to a MIDI file becomes a straightforward task. The application supports importing/exporting of any `Model` instance object to/from a MIDI file. We use this functionality in the following cases:

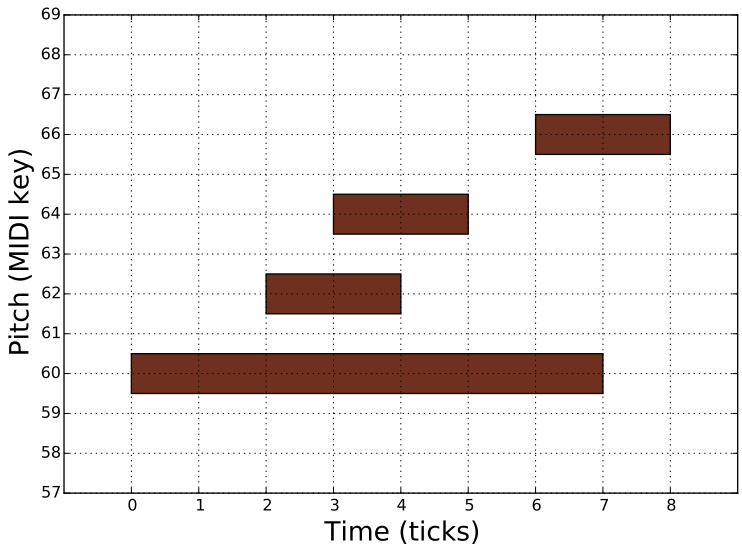


Figure 12: A piano roll example of a model with overlapping notes; this case shows a substantial difference between the MIDI and our model representation, since `NOTE_OFF` events occur between the `NOTE_ON` events in a different order.

1. Our corpus is attached in the MIDI format (see Section 7.1 and Appendix A), therefore we can import it to our own representation for easier comparison (see Section 7.4).
2. The application creates and analyzes the musical models, but does not support playback at this time; therefore exporting to MIDI allows users to listen to the results using an external music player.

We define a *model directory* as a collection of musical models which can be imported from/exported to a set of MIDI files within the same physical directory. Given a path, the `ModelDirectory` class reads all MIDI files and transforms them into our model representation. The set of models may be altered and saved once again as MIDI files. A model directory is used to represent our corpus (details in Section 7.1).

The MIDI protocol is event-based; tracks contain a series of events holding the delta time since the previous event. This is similar to our approach since we use IOIs as the elapsed time since the previous note onset. However, a MIDI note is defined by a separate `NOTE_ON` and `NOTE_OFF` event with the same pitch. Therefore the information about one note is stored in multiple events (possibly more than 2 in the case of overlapping notes). This differs from our approach since we store note durations, making all information regarding a note available in one column of the model matrix. This difference highlights the reason for using a representation different from MIDI for our phenotypes.

Figure 12 shows a model with overlapping notes that emphasizes the difference between the 2 representations. The IOIs of this model using our representation would be $\{0, 2, 1, 3\}$ while the delta times between MIDI events would be $\{0, 2, 1, 1, 1, 1, 1, 1\}$

6.2.1 Importing MIDI files

The `MidiReader` class performs the transformation of MIDI event lists to a musical model using the following steps:

- The global properties for number of tracks and the ticks per quarter note are read in advance.
- The first MIDI track is discarded, since it only contains meta events.
- For each remaining track we:

    1. Initialize a read pointer at the first event of the track.
    2. Find the next `NOTE_ON` event: this will constitute the onset of a new note. The IOI of the note is the aggregate delta times of all events from the read pointer to the event. The read pointer is increased.
    3. Attempt to find a `NOTE_OFF` event with the same pitch. The duration of this note is the aggregate delta times of all events between the `NOTE_ON` and `NOTE_OFF` events. If no event is found with the same pitch, we assume the note to last until the end of the track, i.e. its duration is the aggregate delta times of all events after the `NOTE_ON`.
    4. Repeat steps 2-3 until no more `NOTE_ON` events are found.

6.2.2 Exporting MIDI files

The `MidiWriter` class performs the transformation of a musical model to a set of MIDI events. This transformation must retain the correct order of `NOTE_ON` and `NOTE_OFF` events. For example, Figure 12 shows a model with overlapping notes where the order of offsets is not identical to that of the onsets (the first note ends after the second and third notes).

To ensure the correct order of events and retain linear parsing of the model, we introduce a *note queue* object (class: `NoteQueue`). This queue holds the notes whose `NOTE_ON` events have been added to the MIDI output, but not yet their `NOTE_OFF` events. Along with each note, it stores the onset and offset time relative to the time pointer. The following operations are defined:

- *dequeuing a note* - return the note with the smallest relative offset time (not to be confused with dequeuing in a FIFO queue);
- *delaying the queue* - reduce the relative onset/offset of each note in the queue by a fixed number; this process represents the increase of the time pointer.

The transformation steps are as follows:

- The first MIDI track is added containing a single meta event: a tempo setting of 120BPM (we currently do not support different tempos).

- A MIDI track is added for each model track. The events are built using the following steps:

    1. A track name meta event is added.
    2. An empty note queue is initialized.
    3. We enqueue the next note, setting its relative onset to its IOI and its relative offset to the sum of its IOI and duration.
    4. We dequeue all the notes whose relative offset is smaller than the relative onset of the current note (its IOI). For each read note, we add a `NOTE_OFF` event using the relative offset as the delta time, and delay the note queue with the same value.
    5. A `NOTE_ON` event is added for the current note and the queue is delayed with its relative onset.
    6. We repeat steps 3-5 for each note in the model track.
    7. The note queue is flushed, i.e. step 4 is performed for all remaining notes.

7 Corpus-based Similarity Tests

The fitness of any rendered phenotype is determined by a series of similarity tests. As mentioned in Section 3.1, all tests aim to evaluate how statistically similar a model is to the models in the corpus. This allows a large space of phenotypes to achieve high scores on the tests, not just the ones identical to a member of the corpus.

Here we take a holistic approach to building these tests, i.e. we test fundamental properties of our models instead of properties specific to music theory. We wish to find out whether looking only at the fundamental distributions' similarities allows the system to converge towards musical traits without the need to specifically test for them. This approach allows us to keep the generic aspect of this algorithm, so non-musical models may also benefit from it.

We define two types of tests: those revolving around a single property, such as total duration or number of notes per track, and those revolving around statistical property distributions.

7.1 Chosen corpus

We have chosen Bach's *Inventions and Sinfonias*, comprising 30 keyboard exercises, as our corpus. This catalogue of musical pieces was chosen for the following reasons:

- *Relatively short pieces of equal length* - For this research, we wish to test if, given a corpus of short pieces, our system will converge to create pieces of similar length. Using this constrained length can help demonstrate that our algorithm finds the appropriate solution subspace where pieces of these lengths live.
- *Constant tempo* - We have not included tempo as a property which changes within the models (it is preset to 120 BPM) and we do not test for it.
- *Similar style* - Stylistically, all pieces of the corpus are similar in phrasing and property distribution. By using them, we once again limit our solution subspace and test whether or not the algorithm can find it.

The corpus has been extracted from a set of MIDI files[2] and are included as supporting material for the current work (see Appendix A). The pieces have been generated from a score rather then recorded by a human player. As a result they contain no *musical expression* such as changes in dynamics and tempo (e.g. all notes have velocity 127). For this reason, we do not configure velocities of notes when using the model builder.

The corpus pieces all contain a single track of polyphonic music. We have performed the following transforms to the pieces before running our tests:

1. *Adapting the ticks per quarter note* - As mentioned in Section 6.1, we use the 16th note as our unit for a tick. The MIDI files used a different value for this property, therefore

[2]Downloaded from `www.midiworld.com/bach.htm`; last download date: February 2015

the property has been changed and all note onset/offsets have been adapted to retain the tempo.

2. *Separating the corpus into 2 tracks* - All pieces have been separated into 2 tracks based on their pitch (see Section 7.6). This allows a comparison of the algorithm's results when using a single and dual track corpus that does not differ in musical content, but allows evaluation of statistics across multiple tracks (see Section 8.2).

The different versions of the corpus are stored in different directories (see Appendix A) and the algorithm allows the configuration at startup of which corpus to use (see Section 6.2 and Appendix B).

7.2 Normal distribution tests

In our first corpus-based tests, we evaluate two single-value properties of a model: the *total duration* and *number of notes per track*.

To test similarity, we assume a normal distribution for both properties. The `NormalFitnessTest` class extends `FitnessTest` and, given a mean μ and deviation σ, evaluates incoming values using the following equation:

$$\mathcal{N}_{\mu,\sigma}(x) = \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{7}$$

It omits the scaling by $\frac{1}{\sigma\sqrt{2\pi}}$ used in the PDF of a normal distribution so an output of 1 (perfect grade) is possible. This results in the area under the curve not summing to 1, but this is not problematic since we are using this normal as a reference, not as a distribution we sample.

$$\int_{-\infty}^{\infty} \mathcal{N}_{\mu,\sigma}(x)\mathrm{d}x \neq 1 \tag{8}$$

$$\mathcal{N}_{\mu,\sigma}(\mu) = 1 \tag{9}$$

The `NormalFitnessTest` class can also evaluate multiple normals at once. Given a vector of means $\boldsymbol{\mu}$, deviations $\boldsymbol{\sigma}$ and an input vector $\mathbf{x}$, all of size D, the class outputs a single numeric value: the average grades from each normal.

$$\mathcal{N}_{\boldsymbol{\mu},\boldsymbol{\sigma}}(\mathbf{x}) = \frac{1}{D}\sum_{i=0}^{D-1} \mathrm{e}^{-\frac{(x_i-\mu_i)^2}{2\sigma_i^2}} \tag{10}$$

When evaluating a multitrack model's length, only the global length is involved ($D = 1$); we do not examine the length of each track individually. The number of notes is assessed on each track, i.e. $D = n_t$ where n_t is the number of tracks. Therefore, analyzing the corpus on the algorithm's startup is equivalent to measuring the mean and deviation of the overall lengths and the individual number of notes. Both tests will return only one numeric grade, therefore the number of normal tests does not depend on the number of tracks in the input.

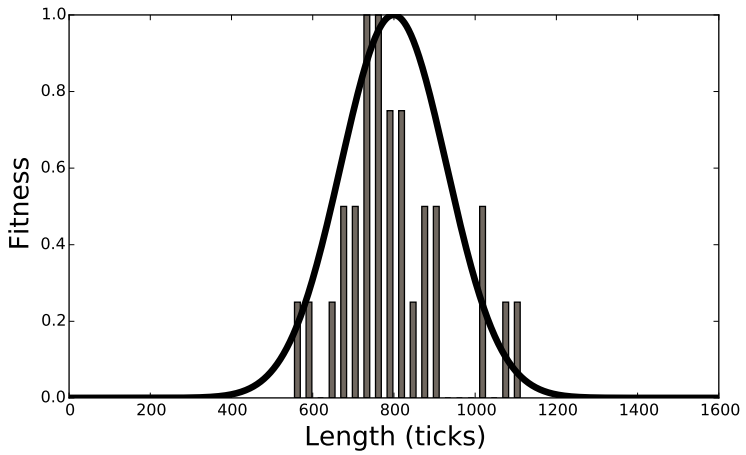


Figure 13: Deduced normal distribution test for musical model length. The length distribution for the pieces in the corpus (histogram bars) determines the grading schema we use for input models. The thick overlaid line shows the deduced normal curve.

This allows us to use the same importances for tests regardless of the number of tracks (more details in Section 7.6 and 8.1).

The presented 2 normal tests constitute the first two tests in our similarity test container. Figure 13 shows the deduced normal of the length test and the histogram of the lengths of the single track corpus, to show the correlation.

Since both the lengths and note counts are integers, the `NormalFitnessTest` class precalculates all values of the normal from 0 to 2μ and stores them in a list. Incoming values are evaluated simply by looking up the appropriate value in the list, returning 0 if the input is below 0 or above 2μ. This results in a significant performance improvement.

7.3 Model descriptors

Instead of comparing the musical pieces directly, we measure the correlation of their statistical properties. These properties are obtained using a series of transforms which yield the *descriptor* of a piece. The remainder of our tests look at similarity between the descriptors of the input and that of the corpus.

The transforms are applied to the IOI, duration and pitch of an input model. Onset/offset have been omitted because IOI and duration fully represent the time axis of a model; velocity has been omitted since it is always constant. Each of the 4 transforms gives a size 128 result; this results in a descriptor being a matrix with 128 columns and $4n_tn_p$ rows, where n_t is the number of tracks and n_p is the number of tested properties (in this case, 3).

The transforms presented in the following subsections were chosen based on the musical qualities they may intuitively represent.

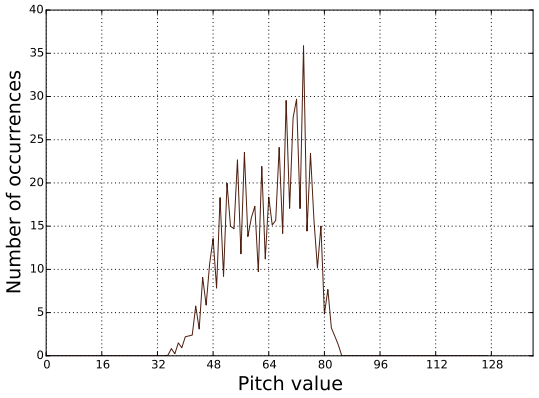


Figure 14: The mean pitch histogram of all members of the single track corpus. The histogram shows no notes outside a certain mid-range.

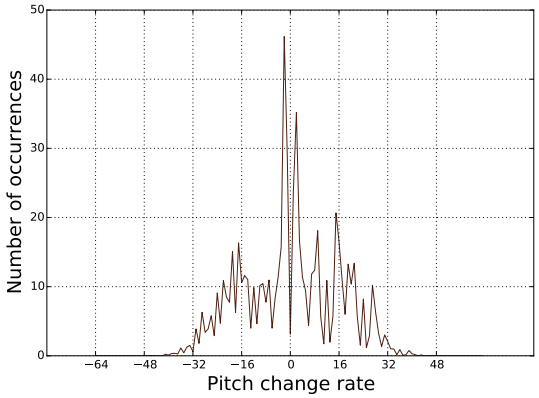


Figure 15: The mean pitch differential histogram of the single track corpus. Measures the distribution of the rate of pitch change between consecutive notes.

7.3.1 Histogram

The first descriptor row is populated with the input data's *histogram*: the distribution of different values. Given an input $x[n]$ of size N, the histogram is given by

$$h_x[v] = \sum_{n=0}^{N-1} \begin{cases} 1 & \text{if } x[n] = v \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

We measure the discrete histogram values from 0 to 127, since every property in our model is a positive integer no larger than 127. The test measuring the correlation of the histogram row rewards high similarity of distribution. To visualize the intuition, Figure 14 shows the mean pitch histogram of all members of the single track corpus. It shows no energy outside of a mid-range interval, which shows that real-world pieces are constrained to a certain pitch range. Therefore the test will penalize digression from this rule.

To widen the search space, the histogram shown in Figure 14 is not the only one we measure similarity to; instead we use clustering to find K representative histograms of the corpus (see Section 7.5).

In a multitrack case, the descriptor would contain separate rows for each track and therefore would reward per-track similarity (see Section 7.6). This mimics the real-world multi-instrumental pieces, where separate instruments' notes often lie in different pitch ranges or the two hands of a pianist cover different areas of the keyboard.

The intuition can be extended to the other properties as well. For example, testing similarity in IOI and duration distributions could highlight the tendency of bass line sparsity: the track assigned to the bass line plays fewer and longer notes.

7.3.2 Histogram of first differential

Histograms focus only on property distribution and therefore ignore the notes' temporal locations. The histogram of the first differential accounts for the immediate rate of change in time. In real music, consecutive notes in a track rarely have large jumps between extrema in any property. The similarity test of this transform therefore discourages large changes of a property from one note to the next.

$$h_{\Delta x}[v] = \sum_{n=1}^{N-1} \begin{cases} 1 & \text{if } x[n] - x[n-1] = v \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

We measure the signed rate of change, therefore we sample $h_{\Delta x}$ between -64 and 63, resulting once again in 128 values. Figure 15 shows the histogram of the differential of the pitches in the single track corpus. Similarly to the pitch histogram, it shows the rate of change constrained between a fixed range, therefore the associated test will discourage quick jumps from very high to very low notes and vice versa. The histogram shows a very small value for $h_{\Delta x}[0]$; we can deduce that keeping the pitch constant throughout consecutive notes occurs rarely in the corpus and is discouraged in our models.

The keyboard exercises in our corpus are played with two hands, but our analyzed data is a merged list of notes. This results in many consecutive notes from different hands, which intuitively show a large pitch jump. This results in the energy in the Figure located around ± 16. The symmetry also arises from the hand changes, since every left-to-right transition is met with an opposite right-to-left one (more details in Section 7.6).

7.3.3 Spectrum

First differential histograms only account for immediate temporal changes and thus ignore an important musical quality: repetition. Real music is often characterized by the reoccurrence of small motifs or larger segments. The Fourier transform of the properties highlights these repetitions and its similarity test penalizes the models with no segment reoccurrence. The discrete Fourier transform is defined as:

$$X_x[k] = \sum_{n=0}^{N-1} x[n] e^{-ink\frac{2\pi}{N}} \tag{13}$$

The number of notes N is not known in advance; it depends on the number of output commands occurring in the genetic string interpretation. Therefore we input a zeropadded version of the note properties using $N = 2048$; any input above this size is truncated. Since the normal test related to the number of notes (see Section 7.2) gives 0 for anything above 2μ, we have chosen N as the first power of 2 above the cutoff value.

$$N = 2^{\lceil \log_2(2\mu) \rceil} \tag{14}$$

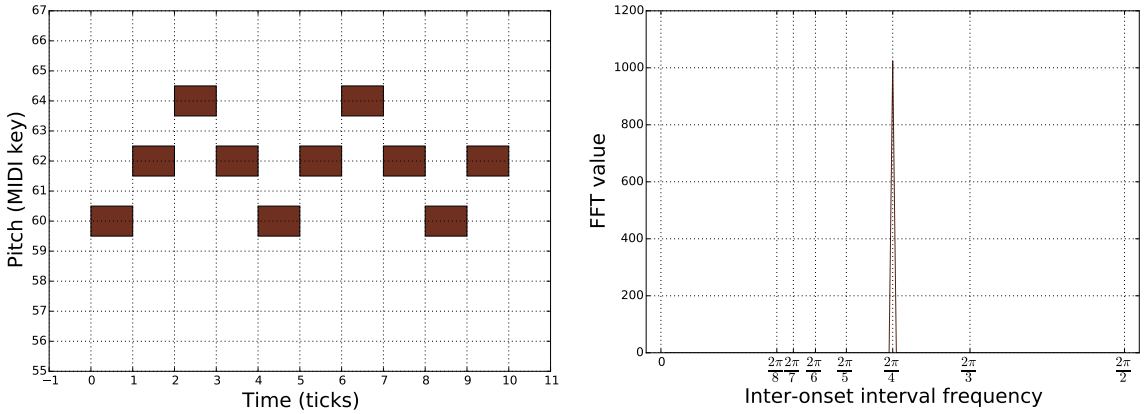


Figure 16: Spectrum analysis example. The left image shows the piano roll representation of input notes, while the right shows the result. We omit $X_x[0]$ and measure only the repetitions with frequencies in $(0, \pi]$. In this case, 4 notes are repeated in a sinusoidal pattern, resulting in spectral energy at $\frac{2\pi}{4}$. The input has been tiled to avoid side lobes of $X_x[0]$ caused by zero-padding.

The mean number of notes in the corpus is $\mu = 633.5$, resulting in $N = 2048$. The maximum number of notes is also influenced by the VM's `maxOutputs` property, which is also set not to exceed $4N$ (since 4 bytes are used per note; see Section 5.3 and Section 8.1).

Our input to the Fourier transform is real, resulting in a Hermitian symmetric output. Therefore the spectrum is fully represented by the first $\frac{N}{2} + 1$ values. We subsample the spectrum to a size of 256; this results in all relevant information being stored in 129 values. This is 1 more than the size of a row in our descriptor.

The spectrum in $k = 0$ represents the sum of all values:

$$X_x[0] = \sum_{n=0}^{N-1} x[n] e^{-in0\frac{2\pi}{N}} = \sum_{n=0}^{N-1} x[n] \tag{15}$$

Our input contains exclusively positive values, therefore $X_x[0]$ will always be disproportionately larger than all other values in the spectrum. It also contains no information about repetitions; only the constant needed to retain energy. Since we only use the spectrum for repetitivity analysis (no synthesis), we can safely discard $X_x[0]$, resulting in 128 values we store in the descriptor row.

Figure 16 shows a Fourier transform example using a series of notes with a sinusoidal pattern. In this case, 4 notes are repeated in this pattern, which results in spectral energy at a period of $T = 4$, i.e. at a digital frequency of $\omega = \frac{\pi}{2}$. The transform always relies only on one property (in this case, pitch); therefore using different IOIs/durations would still give the same result.

The sinusoid's amplitude is 2 and it is centred around the pitch value 62. Therefore, if we had not omitted $X_x[0]$, it would have energy 31 times larger than the peak of the repetition, distorting the information we are really looking for in the spectrum. On the other hand, this

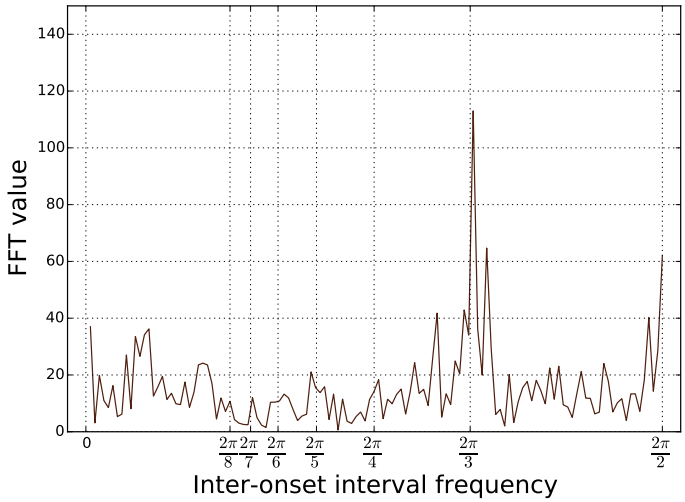


Figure 17: The inter-onset interval spectrum of the first member of the single track corpus. The diagram shows peaks at periods of 2 and 3, favouring patterns of length 3. Models showing a similar repetitivity score high on the similarity test associated with this property.

omission discards the centre pitch; notes with the same sinusoidal pattern around any other pitch value would give the same results.

Figure 17 shows the IOI spectrum as measured for the first member of the single track corpus. It shows 2 distinct peaks at $\omega = \frac{2\pi}{3}$ and $\omega = \frac{2\pi}{2}$, which represent periods of $T = 3$ and $T = 2$, respectively. This analysis shows a tendency of repeating short IOI patterns. The peak at $T = 3$ is larger, showing there are more repeating patterns formed of 3 notes than there are formed of only 2.

7.3.4 Spectrum of first differential

The last property we include in the descriptor rows measures the repetitivity of the rate of change within a property. We measure this property identically to the simple spectrum.

$$X_{\Delta x}[k] = \sum_{n=1}^{N-1} (x[n] - x[n-1]) e^{-ink\frac{2\pi}{N}} \tag{16}$$

The 4 presented transforms are measured for IOI, duration and pitch, resulting in $12n_t$ descriptor rows and therefore 12 grades.

7.4 Descriptor correlation tests

Comparing 2 descriptors is performed using the *Pearson correlation coefficient* (r) (Rummel, 1976). Correlation is measured row-by-row, so different importances can be assigned to the correlation of different properties. The results are averaged across tracks, resulting in $4n_p$ (in this case, 12) grades.

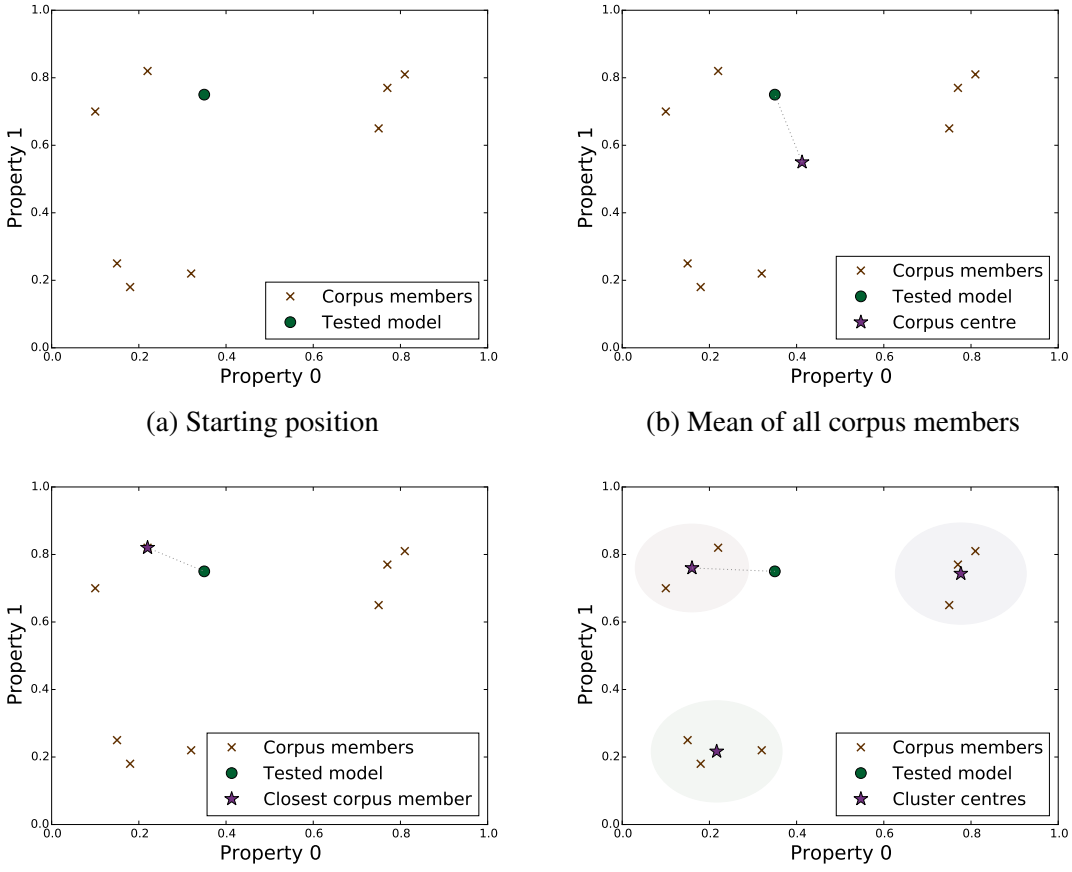
Given the ith lines from descriptors D and E, denoted by D_i and E_i, and their mean values denoted by $\bar{D}_i$ and $\bar{E}_i$, their correlation coefficient is given by the following equation:

$$r_{D_i,E_i} = \frac{\sum_{i=0}^{127}(D_i - \bar{D}_i)(E_i - \bar{E}_i)}{\sqrt{\sum_{i=0}^{127}(D_i - \bar{D}_i)^2}\sqrt{\sum_{i=0}^{127}(E_i - \bar{E}_i)^2}} \tag{17}$$

The resulting coefficient always lies between -1 and 1. Positive values imply positive correlation, negative values imply negative correlation and a value of 0 implies no correlation. In our tests, we return 0 for negative values because we do not wish to obtain negative fitness scores. This approach allows us to return grades between the values of 0 and 1 without having to normalize the descriptors themselves.

7.5 Corpus clustering

We have defined correlation between two descriptors but our corpus consists of 30 different musical models, all of which have a different descriptor. Therefore a method must be used to incorporate the data from all 30 corpus members into the test. To verify how well this incorporation works, we evaluate the members of the corpus themselves using our tests.



(a) Starting position

(b) Mean of all corpus members

(c) Closest corpus member

(d) Clustered

Figure 18: Testing an incoming model against the corpus

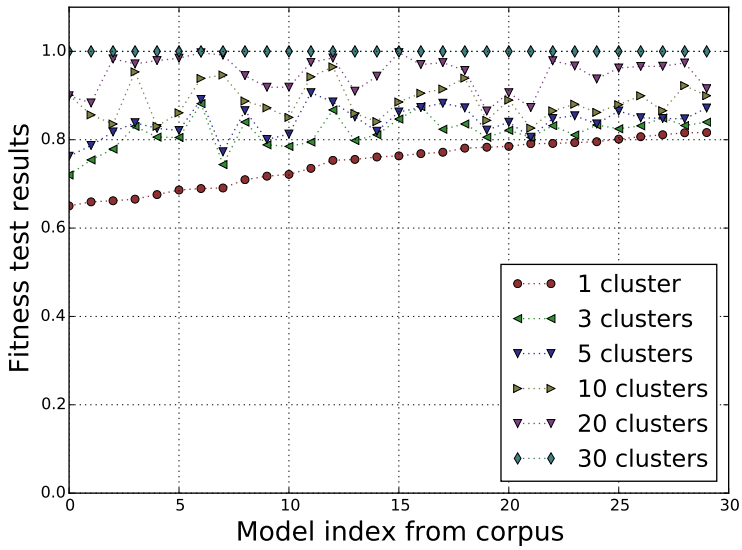


Figure 19: Descriptor similarity test results of the single track corpus for different numbers of clusters; the models have been sorted by correlation when using $K = 1$.

A possible reference descriptor could be the centre of all descriptors from the corpus (Figure 18b). This would be a *single niche* (Mahfoud, 1995) within our solution space. Experimenting with this approach reveals that testing the corpus itself gives relatively small grades (between 65 and 82%). This suggests that taking just one niche narrows down our solution space so much that even the corpus cannot achieve a high enough grade.

Another possible approach is to use all corpus members as niches, i.e. measure an input model's correlation to each of the corpus members and take the maximum value (Figure 18c). In this case, all corpus members score 100%. However, this can become computationally expensive, and might broaden the solution space too much.

To obtain the best of both approaches, we *cluster* our corpus, partitioning the descriptors into K subsets (Figure 18d). Afterwards we can use the centres of these clusters as reference descriptors, and test for the maximal correlation with a centre.

The clustering happens offline, before the evolutionary algorithm starts. It only needs to be done once therefore we do not have to worry about the computational expense. We use *K-means clustering* (Jain and Dubes, 1988) substituting the typically used geometrical distancing with correlation.

We denote the input as x with size N to be clustered into K subsets ($K \leq N$). Using the correlation r as defined in Equation 17, the steps of the clustering algorithm are as follows:

1. Define the cluster centres as the first K descriptors:

$$\mu_k = x_k \quad \text{for each} \quad k \in \{0, \ldots, K-1\} \tag{18}$$

2. *E-step:* Classify each descriptor into one of the K partitions by taking the maximal

correlation (instead of the minimum distance):

$$c_n = \underset{k}{\operatorname{argmax}}\, r_{x_n,\mu_k} \quad \text{for each} \quad n \in \{0,\ldots,N-1\} \tag{19}$$

3. *M-step:* Find the new centres of gravity by taking the mean descriptor of each partition. Assign these as the new cluster centres:

$$\mu_k = \underset{c_n=k}{\operatorname{mean}}\, x_n \quad \text{for each} \quad k \in \{0,\ldots,K-1\} \tag{20}$$

4. Repeat steps 2-3 until no change occurs between iterations, or the number of steps reaches a maximum value (to avoid infinite oscillation).
5. Return cluster centres μ_k and which cluster each input is classified to (c_n).

This approach allows the flexibility of changing the value of K between runs. The two methods proposed initially can also be achieved by setting $K = 1$ or $K = 30$, respectively. Figure 19 shows the achieved grades of the corpus using different values for K.

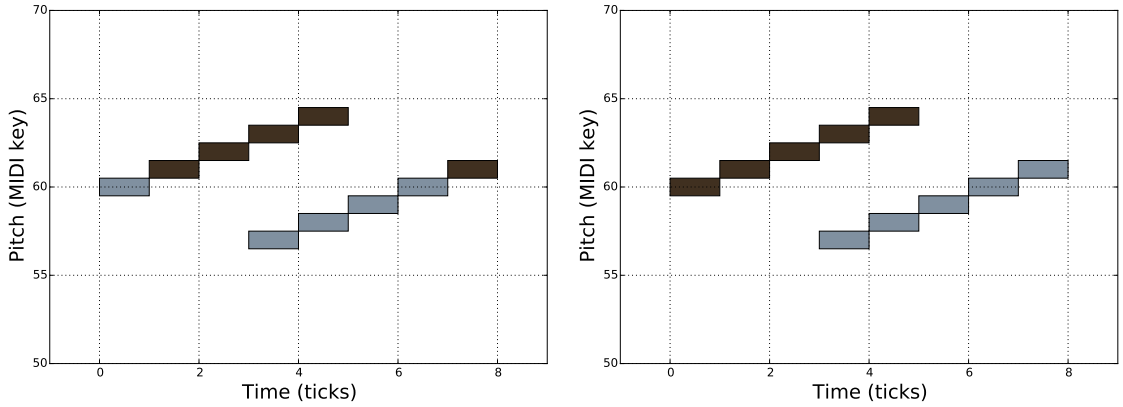
7.6 Using a multivoice corpus

We explore the differences between using single and multitrack corpora. The Bach keyboard exercises (also used in Sulyok et al. (2015)) all comprise single track pieces. Including a second corpus for multitrack experimentation may dilute the results because of the different distributions/lengths/meters. The ideal way to omit result discrepancies due to these differences would be to include the same corpus in both single and multitrack versions.

Our corpus only uses one instrument but the keyboard is played using two hands. The notes played by the left and right hand could be placed into different MIDI tracks to simulate 2 different instruments. Therefore we attempt to *separate* our MIDI files into 2 tracks to create a second version of the same corpus.

Previous works on MIDI separation were based on separating pieces into *contigs* which are regrouped based on pitch proximity (Chew and Wu, 2004); using a learned decision tree (Kirlin and Utgoff, 2005); real-time separation based on pitch proximity (Madsen and Widmer, 2006); or Kalman filtering (Hadjakos and Lefebvre-Albaret, 2009).

We however opt to once again use a clustering mechanism to reuse many steps seen in Section 7.5. The number of clusters is defined as $K = 2$ and the properties of interest are pitch and onset time. Since we need a cluster separation point at every point in time, we do not view the space of interest as a 2-dimensional space with 1 dimension for pitch and 1 for time. Instead, we perform clustering only on pitches, taking onset time as a weighting factor when calculating distances from a centre. Therefore there will always be a separation point in time above which notes are assigned to 1 cluster and below which they are assigned to the other.



(a) First iteration: first and last notes assigned incorrectly

(b) Second iteration: assignments correct due to onset difference weighting

Figure 20: Example of the 2 iterations of the pitch clustering algorithm using a simple example. The plots show a piano-roll style representation.

Given x is the note pitches and t is the note onsets, the adapted algorithm steps are:

1. Start out with a separation point in between the maximum and minimum pitch in the input:

$$c_n = \begin{cases} 0 & \text{if } x_n \leq \frac{1}{2}\left(\max_m x_m + \min_m x_m\right) \\ 1 & \text{otherwise} \end{cases} \tag{21}$$

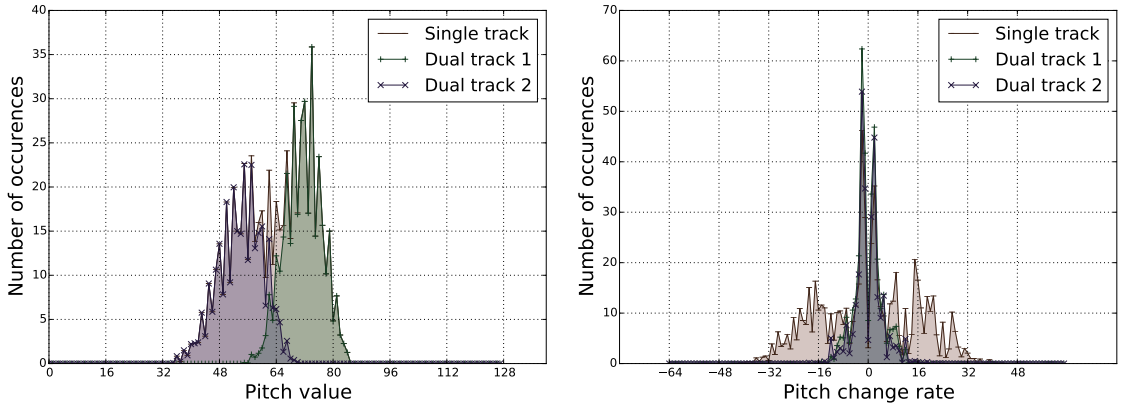
2. Build the *log-normal* shaped weights used for onset differences: $w(\Delta t)$. The mean of the log-normal gives the time difference under which we assume notes played in unison (which should be separated) and the variance allows control of the weighting: how much onset difference still impacts the classification of a current note.
3. Combine the *E-step* and *M-step* since now there are no cluster centres; distancing is based on the onset differences:

$$c_n = \underset{k}{\operatorname{argmin}} \frac{\sum_{c_m=k} \left(|x_n - x_m| \times w(|t_n - t_m|)\right)}{\sum_{c_m=k} w(|t_n - t_m|)} \tag{22}$$

4. Repeat step 3 until there is no change or maximum number of iterations is reached.

Figure 20 shows an example of the separation process for a small example. In the first iteration, the mid point between the minimum and maximum pitch is used to separate the notes. This gives unintuitive results for the first and last notes. In the second iteration, the onset weighting allows the closer notes to rectify this.

Figure 21 shows the resulting pitch histograms of the original and the resulting dual track corpus. Two disjoint pitch ranges have emerged for the cluster members with a small overlap in the middle. The right subfigure shows the histogram of the first differential of pitch. The



(a) Pitch histogram

(b) Histogram of first differential of pitch

Figure 21: The mean histograms of pitch and pitch differential of all members of the single and dual track corpora. The 2 tracks of the dual track corpus show a different pitch range, but they add up to the single track one, since they comprise the same notes. Much of the energy in the differential is lost after separating the corpus, since hand changes are no longer included.

two overlaid plots of the dual track corpus cover a much smaller range than the single track one. Much of the energy at around ± 16 is lost due to not including the jumps from left hand to right hand and vice versa.

8 Experiments & results

Our experiments explore the space of possible parameters to measure the impact of changes. The questions this exploration aims to answer is: “Which set of parameters gives the highest grades consistently?”, “Which set of parameters gives a good tradeoff between performance and high grades?” and “Which parameters help the grades accurately represent musical quality?”.

8.1 Tested configurations

For each set of parameters, we have run 20 different tests, each time allowing the algorithm to reach a target of 20000 generations. Common parameters include a survival probability of 15%, maximum age of 3, maximum cut point ratio of 0.1%, maximum mutation ratio of 2%, number of corpus clusters of 5 and the VM halting conditions of 60000 commands or 2600 output bytes. The importances of the 14 similarity tests are also common (see Figure 22).

The following parameters differ in each test run:

1. *the population size* takes the values from $\{16, 32, 64, \ldots, 1024\}$;
2. *the number of tracks in the corpus* is either 1 or 2 using the different versions of the same corpus as described in Section 7.1;
3. *the VM instruction set* is either immediate or indirect as described in Section 5.3.

This results in a total of 28 different configurations. Each run exports its state to disk every 1000 generations, and the highest scoring models are also exported as MIDI and at-

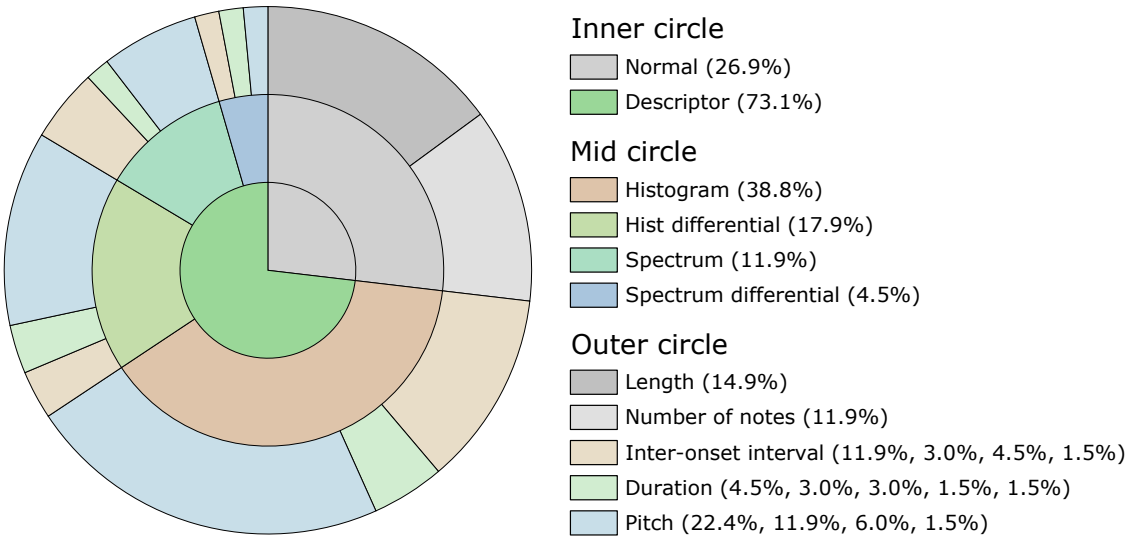


Figure 22: The importances used in our tests: the inner circle separates the normal and descriptor-based tests and the mid/outer circles categorize the descriptor-based tests by the used statistical transform and the input model property, respectively.

	Immediate OCI				Indirect OCI			
Population	**Single track**		**Dual track**		**Single track**		**Dual track**	
size	Mean	Max	Mean	Max	Mean	Max	Mean	Max
16	0.385	0.489	0.352	0.455	0.379	0.483	0.360	0.479
32	0.435	0.580	0.411	0.550	0.424	0.532	0.415	0.598
64	0.446	0.602	0.429	0.616	0.476	0.608	0.431	0.636
128	0.482	0.648	0.429	0.654	0.474	0.583	0.432	0.609
256	0.483	0.662	0.450	0.689	0.476	0.628	0.459	0.640
512	0.492	0.679	0.471	0.706	0.502	0.626	0.476	0.673
1024	0.504	0.667	0.490	0.757	0.513	0.641	0.514	0.732

Table 3: Mean and maximum grades of the populi in their last generation for each different configuration, averaged over the 20 runs.

tached as supporting material (see Appendix A). This allows further analysis and listening to the results.

One of the configurations is identical to our previous research (Sulyok et al., 2015) with only the survival mechanism altered (see Section 4.4). The comparison of the results is discussed in Section 8.5.

8.2 Overall results

Table 3 shows the mean and maximum grades of the last generations in each test run. Analyzing the results, we can draw the following conclusions:

- Larger population sizes result in both better mean and maximum grades (further discussed in Section 8.3).
- Using the dual track corpus results in smaller mean values for the populi, but larger maxima (further discussed in Section 8.4).
- The inclusion of immediate addressing to the VM instruction set seems to make no real difference in the results. The mean and maximum values show an average difference of 0.5% and 2%, respectively.

8.3 Using different population sizes

The experiments show better grades emerging from larger population sizes. Even the 2 largest population sizes (512 and 1024) show a significant difference: a 2.8% and 2% rise in maximum and mean grades, respectively. However, this exponential population size increase also increases runtime exponentially, since all computationally expensive methods are performed on each individual in a population.

Therefore we attempt to estimate if increasing the population size further would continue to give significantly better results. Figure 23 shows the results in Table 3 averaged over all

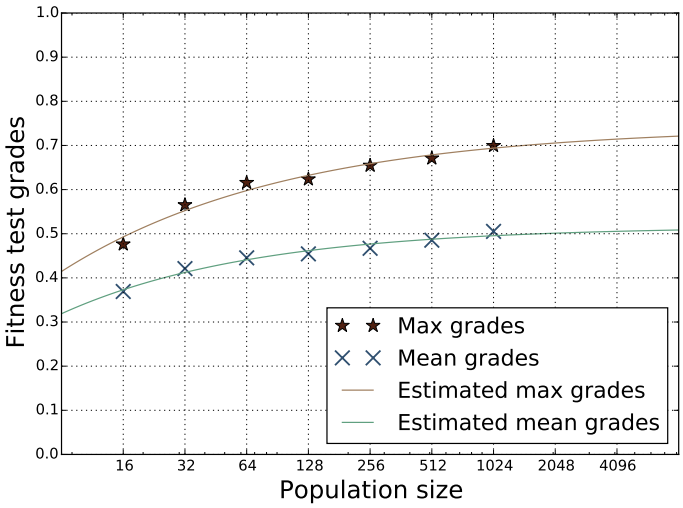


Figure 23: Maximum and mean grades at the last generation, averaged over all test runs. The lines show an estimated curve fit over these samples, which deduce that further increasing the population size would not give significantly better results.

configuration dimensions except the population size. Plotted over them is an estimated curve fit to these target points using linear regression.

Intuitively assessing Figure 23, the curve seems to move in an inverse exponential manner of shape

$$f_\alpha(x) = 1 - \alpha^{-\log_2 x} \tag{23}$$

where x is population size. We use its logarithm since our figure shows the X-axis on a logarithmic scale. α determines the steepness of the curve. f_α tends towards 1 for any value of α strictly larger than 1, which further enhances its fitting, since our fitness tests are constrained between 0 and 1.

$$\lim_{x\to\infty} f_\alpha(x) = 1, \quad \alpha > 1 \tag{24}$$

Attempting to fit f_α (finding the best value for α) to our data results in large errors. This might signal that the algorithm, in its current form, is unable to reach 100% fitness with any population. Therefore we attempt to scale the function:

$$g_{\alpha,m}(x) = m \times (1 - \alpha^{-\log_2 x}) \tag{25}$$

$g_{\alpha,m}$ therefore tends towards m, the hypothetical highest grade we can achieve with the algorithm.

$$\lim_{x\to\infty} g_{\alpha,m}(x) = m, \quad \alpha > 1, \quad 0 < m \leq 1 \tag{26}$$

Fitting $g_{\alpha,m}$ separately for the mean and maximum gives adequately small errors: an

x	1024	2048	4096	8192
$g_{\max}(x)$	0.694	0.706	0.714	0.721
$g_{\text{mean}}(x)$	0.495	0.501	0.505	0.508

Table 4: Estimated maximum and mean grades for higher population sizes.

average of 0.01 and 0.006 respectively. The estimated functions are:

$$g_{\max}(x) = 0.743 \times (1 - 1.313^{-\log_2 x}) \tag{27}$$

$$g_{\text{mean}}(x) = 0.516 \times (1 - 1.377^{-\log_2 x}) \tag{28}$$

This shows the algorithm as capable of reaching a maximum grade of 0.743 and a mean of 0.516. Table 4 shows estimated values for $g_{\max}$ and g_{mean} for higher population sizes. The values suggest further exponential increase of the population size would not result in significantly better results.

8.4 Single vs. dual track corpus

As mentioned in Section 7.6, the corpus members have been clustered into 2 tracks and separate experiments have been performed using the single and dual track versions. Figure 24 shows the resulting grade progressions averaged over all dimensions except the number of tracks. The dual track corpus shows a slightly larger maximum but lower mean. Their progressions through the generations are almost identical, signalling that the number of tracks does not impact the speed at which pieces are able to evolve.

The musical models in a population have the same number of tracks as the corpus used

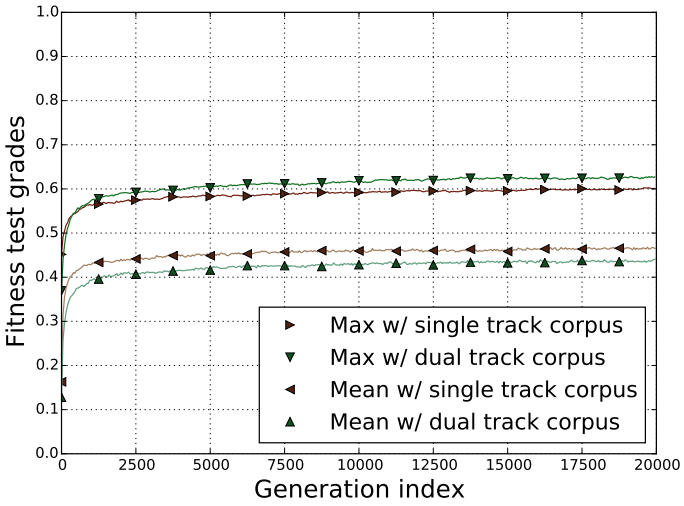


Figure 24: Mean and maximum grade progression over the generations when using single and dual track corpus. The dual track corpus shows a slightly higher maximum, but lower mean.

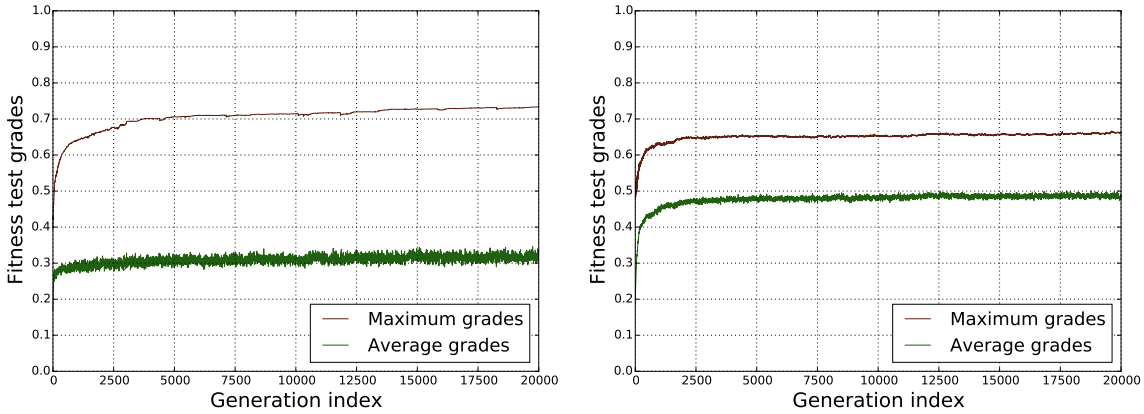


Figure 25: Comparison of mean/maximum grade progression using guaranteed and probabilistic survival. The left plot shows a previous iteration, while the right uses one of our current runs with the same configuration. The results show a slightly smaller maximum but a much larger mean, signalling that probabilistic survival allows a more diverse population to emerge, with the cost of not allowing high-scoring but fragile units to survive.

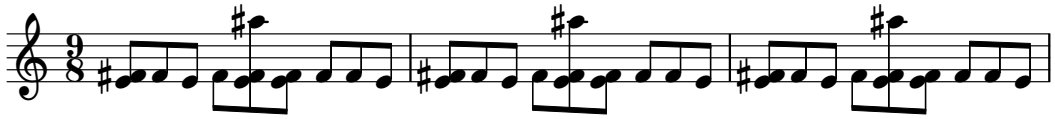
in their associated run (see Section 6.1); their descriptor's size is also proportional to it (see Section 7.3). Therefore, when comparing 2 descriptors in the dual track case, we are assessing correlation of data twice as large, intuitively making correlation more difficult to emerge. The slightly lower means in the dual track case seem to confirm this intuition, but the maximum grades are surprisingly higher.

8.5 Impact of survival mechanism

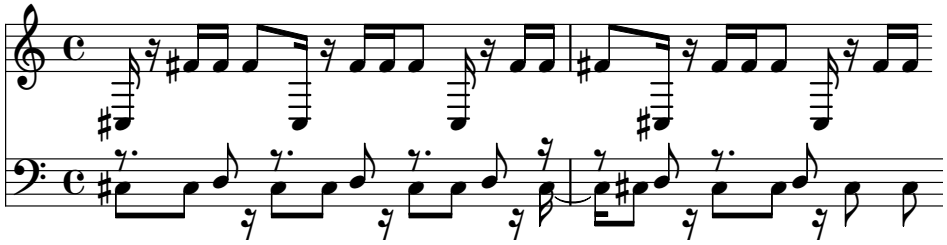
As mentioned in Section 4.4, the survival mechanism has been changed since our previous research (Sulyok et al., 2015); unit survival is now based on chance rather than guaranteed for the highest scoring units. We compare the results from our previous research to the current one to measure the effects of this change. The configuration in Sulyok et al. (2015) is identical to one of the current ones: using a single track corpus, a population size of 256 and allowing immediate addressing. The only difference is that 40 test runs were performed in the previous iteration.

Figure 25 shows the side-by-side comparison of the previous results and the current ones using the same configuration. Both plots show the progression of the maximum and mean grades over the generations, averaged over all runs with the aforementioned parameters. The maximum values seem to suffer a loss (the last generation's values are 0.733 for the old and 0.662 for the new) but the mean grades show a significant improvement (0.324 for the old and 0.482 for the new).

This indicates that guaranteed survival tends to single out one or more marginally favourable units, who are always propagated through the generations. Since genetic strings are fragile when faced with crossover/mutation (even a small change may drastically alter the resulting musical model), guaranteed survival does not allow as much population diversity.



(a) A single track result. It shows a small repeating pattern over the entire duration of the piece. The time signature has been added manually to visualize the pattern.



(b) A dual track result, where 2 different repeating patterns emerge on the 2 tracks. The motifs have a different length (6 and 5 ticks), resulting in an interesting rhythmic pattern when overlaid.

Figure 26: Example results

Probabilistic survival will eventually block the survival of a unit whose genetic string is strong on its own, but is too fragile when facing change.

8.6 Listening to the output

Examining the resulting MIDI files (see Appendix A) allows us to evaluate, albeit subjectively, the musicality of the results. Their durations are almost universally within the boundaries dictated by the corpus statistics, although many achieve this with a smaller than desired number of notes (i.e. the notes are on average longer or there are rests).

Some musical traits emerge in the results, especially related to repetition. Almost all MIDIs contain motifs of a few notes repeated multiple times, and some show variation on the repeated theme. The algorithm successfully finds "shortcuts": small motifs consisting of a few notes which, when repeated many times, approach the statistics of the corpus. However, these results are not particularly musical (see example in Figure 26a), suggesting the need for more restrictive tests.

Although repetition and variation emerge, other musical properties such as *harmony*, *melody* or *entropy* are somewhat lacking. This suggests the need for further fitness tests inspired by music-theory (see Section 9.1).

The multitrack results seem more musical than the single track ones. They also exhibit small repeating patterns but these may easily have different lengths on the 2 tracks, resulting in interesting rhythms when overlaid (see Figure 26b).

A collection of randomly chosen segments from our results can be found under Appendix D.

9 Conclusion

The space of possible music is vast, even when analysing only a few properties and omitting synthesis and performance elements. Even then, there is no universal location of a "subspace of good music"; it is a subjective evaluation. But the results in the field show promise; to date, many musical pieces have been composed *assisted* by evolutionary algorithms. Waschka (2007) has always viewed these algorithms as auxiliary tools for composer inspiration instead of standalone virtual composers.

In this paper we have demonstrated a novel approach to evolutionary music composition: evolving the composition process rather than the product. We have deviated from many of the previous research in the area by incorporating elements of linear genetic programming and by a novel approach to using corpora.

By using a Turing-complete virtual machine, we have successfully modelled a composer; programs running on the VM represent the genotypes and the resulting musical pieces represent the phenotypes. This separation allowed us to model the vast space of possible music indirectly using the instructions of the VM.

Unlike many of the previous approaches, we do not attempt to narrow this space by constraining parameters (e.g. using only pitches within a diatonic scale) or by setting favourable initial conditions (e.g. using the corpus as the initial population). Instead, we give the virtual composer freedom and attempt to guide his search. We have completed a first step in this guidance by providing a set of niches: a corpus of real music. The algorithm is told that musical pieces that are statistically similar to these pieces may be deemed "good".

By exploring the space of possible parameters, we can conclude that 1024 is an optimal population size, immediate addressing in the VM does not influence the results and that using a multivoice context gives a more detailed description of the corpus and therefore helps more interesting pieces to emerge.

Our results are promising, demonstrating that while this approach succeeds at converging towards the properties of the pieces in the corpus, is still only produces partially musical results. Some traits such as repetition and variation do arise, but there is still much room for improvement. In its current form, our algorithm may still be suitable for providing inspiration to real composers.

9.1 Future work

The algorithm achieves high scores with very small motifs repeated many times, which dictates the need for more restrictive tests, especially revolving around entropy. Examining the similarity of entropy to the corpus could potentially force the evolution strategy to discard these simplistic results and search in other directions.

The small recurrent motifs also suggest that only a small portion of the RAM is used in a loop when rendering phenotypes. We propose the exploration of the number of bytes touched

during interpretation; this may dictate the sufficiency of a smaller RAM size. A larger RAM using only a small number of bytes may behave unpredictably when facing crossover: it may have its active code either scattered throughout the genetic string, leading to substantial changes on crossover; or clumped together in one segment, possibly leading to no change on crossover. Besides changing the RAM size, the entropy tests may discourage the branching instructions, increasing the number of used memory, diminishing the need for the change.

The results are also highly disharmonic and lacking a set metrical structure. We suggest the future inclusion of similarity tests more inspired by music theory, such as rewarding the emergence of a diatonic scale.

Even though we model the composer's process as a linearly executed program running on a processor, experimenting with classic tree-like genetic programming or with different types of interpreters (e.g. based on functional programming) may reveal improvements.

Our current representation of the musical model is in function of note index, not time. Transforming a model to a function of time would allow the inclusion of new tests, e.g. measuring the similarity of number of notes played in unison at any time.

In a multivoice context, our descriptor correlation tests only measure the similarities per track. Tests which measure overall statistics similarity might combine the strengths of the single- and multitrack approaches.

Improvements could also be made by a different choice of corpus. Since the current one had no musical expression, it would be interesting to explore what musical pieces played by human musicians could teach our system to do. Incorporation of velocity into the model properties would then allow us to test for accentuation and expressive tempo changes.

A Supporting material

The public website associated with this research project can be found under: `http://csabasulyok.bitbucket.org/emc`. The site contains a hall of fame: cropped MIDI results which exhibit interesting musical traits. It also hosts the results from our previous research (Sulyok et al., 2015).

The current project's source code is hosted online on *BitBucket* at the following URL: `https://bitbucket.org/csabasulyok/emc`. Attached to this document is a snapshot of the Git repository along with the current experiments' highest scoring MIDI files (see folder structure in Table 5) and a compiled distributable of the application (for Windows 7 x64).

Folder name	Description
`dist`	Compiled distributable of the application.
`doc`	LaTeX source code and figures for relevant documents
`-> emc-common-img`	Common images used throughout the different documents
`-> emc-proposal`	Project proposal
`-> emc-specification`	Project specification
`-> emc-report`	The current report
`-> emc-pres`	The associated presentation
`midi`	MIDI files of corpus members
`-> bach_original`	The originally downloaded versions of the Bach corpus (see Section 7.1)
`-> bach_1track`	Single track version of the corpus
`-> bach_2track`	Dual track version of the corpus (processed as seen in Section 7.6)
`results`	MIDI results of current experiments
`-> random`	Random MIDI files sampled from highest scoring models included in Appendix D
`src`	Python/C++ source folder
`-> emc-framework`	Abstract genetic programming framework (see Section 4)
`-> emc-example`	Example application of framework (see Section 4.5)
`-> emc-vm`	Opcode interpreter generator with models and templates (see Section 5.5).
`-> emc-cpp`	C++ sources for computationally expensive steps of the algorithm (e.g. opcode interpreter, model builder, descriptor).
`-> emc-core`	Main Python project

Table 5: Folder structure of supporting material

The MIDI result files hold configuration information in their name. For example, the file

`emc_ociImmediate_multiTrack_pop0128_time20150708-203840_gen19999_score0.684`

is the highest scoring unit from a run using the immediate VM instruction set, the dual track corpus and a population size of 128. Only files from the last generation have been included.

B Tools and build management

The following tools have been used in development:

- Code implemented using *WinPython 2.7.9* and *Visual C++ 9.0*;
- IDE: *Eclipse Kepler CDT* with *PyDev 3.5* and *TeXlipse 1.5*;
- Version control: *Git 1.9.4*

The project uses the following external libraries:

- *SWIG/distutils* - used to compile our C++ code for Python compliance (Sauvage, 2003);
- *Django* - web framework used for template rendering in the OCI generator (see Section 5.5);
- *midiparse* - a Python library for MIDI import/export [3];
- *Numpy.i* - a SWIG library for NumPy support [4]
- *FFTW 3* - C++ FFT library used in descriptor building (see Section 7.3).

The main entry point of the algorithm is `src/emc-core/src/emc/demo/algrun.py`. This file contains the default configuration of a run. Each configuration can be overwritten by a command-line argument in the shape `key=value` (see possible arguments in Table 6). The algorithm can also be started from the Eclipse launcher *emc-algrun*.

A distributable of the algorithm can be built using the Eclipse launcher *emc-dist* which uses *py2exe*. A compiled distributable for Windows 7 x64 is included in the supporting material and can be launched using `emc.exe`. Help about command-line arguments may be queried using `emc.exe -h`.

[3] Downloaded from https://github.com/blob8108/kurt/blob/master/examples/midiparse.py; last download date: February 2015

[4] Downloaded from https://github.com/barbagroup/pygbe/blob/master/util/numpy.i; last download date: December 2014

Name	Default value	Description
`corpusPath`	`'corpus/bach_2track'`	Folder containing corpus MIDI files.
`numClusters`	5	K - number of corpus clusters (see Section 7.5)
`ociClass`	*immediate*	Python class name of OCI interpreter. The generated Python wrapper for the C++ class must be used (see Section 5.5).
`maxCommands`	60000	Maximum number of VM instructions to be executed before halting
`maxOutputs`	2600	Maximum number of VM bytes to be output before halting
`numUnits`	256	Population size
`maxAge`	3	Maximum times any unit may be survived (see Section 4.4)
`survivalProb`	0.15	Weight of survival probability applied to overall grades (see Section 4.4).
`targetGeneration`	20000	The algorithm will run until it reaches this target generation.
`numberOfRuns`	1	How many times the algorithm shall be run. Used to link multiple runs together sequentially.
`outputDir`	`'./output'`	Folder to store output models/MIDI files.
`loggingCycle`	50	The algorithm will print status information to the console every `loggingCycle` generations.
`cycleSize`	250	The algorithm will save the status of the algorithm together with the best MIDI file every `cycleSize` generations.

Table 6: Possible command-line arguments of application

C Opcode interpreter instruction sets

C.1 The immediate opcode interpreter

Opcode prefix	Instruction name	Params	Description
00000	movR_DH	r:3	Copies high byte of dataPtr into given register.
00001	movDH_R	r:3	Copies value of given register into high byte of dataPtr.
00010	movR_DL	r:3	Copies low byte of dataPtr into given register.
00011	movDL_R	r:3	Copies value of given register into low byte of dataPtr.
00100	movR_DPtr	r:3	Copies 1 byte from memory where dataPtr points, into given register.
00101	moviR_DPtr	r:3	Copies 1 byte from memory where dataPtr points, into given register, and increases dataPtr.
00110	movDPtr_R	r:3	Copies value of given register to the memory where dataPtr points.
00111	moviDPtr_R	r:3	Copies value of given register to the memory where dataPtr points, and increases dataPtr.
01000	movR_Next	r:3	Copies next byte after call into given register.
01001	movR_A	r:3	Copies acc into given register.
01010	movA_R	r:3	Copies given register into acc.
01011	xchA_R	r:3	Exchanges values of acc and given register.
01100000	movA_DH		Copies high byte of dataPtr into acc.
01100001	movA_DL		Copies value of acc into high byte of dataPtr.
01100010	movDH_A		Copies low byte of dataPtr into acc.
01100011	movDL_A		Copies value of acc into low byte of dataPtr.
01100100	movA_DPtr		Copies 1 byte from memory where dataPtr points, into acc.
01100101	moviA_DPtr		Copies 1 byte from memory where dataPtr points, into acc, and increases dataPtr.
01100110	movDPtr_A		Copies value of acc to the memory where dataPtr points.
01100111	moviDPtr_A		Copies value of acc to the memory where dataPtr points, and increases dataPtr.
01101000	xchA_DPtr		Exchanges values of acc and the memory where dataPtr points.

Opcode prefix	Instruction name	Params	Description
01101001	xchiA_DPtr		Exchanges values of acc and the memory where dataPtr points, and increases dataPtr.
01101010	movDPtr_Next		Copies next byte after call into memory where dataPtr points.
01101011	movDH_Next		Copies next byte after call into high byte of dataPtr.
01101100	movDL_Next		Copies next byte after call into low byte of dataPtr.
01101101	movA_Next		Copies next byte after call into acc.
01101110	cplC		Inverts value of carry flag.
01101111	inc_A		Increments value of acc.
01110	inc_R	r:3	Increments value of a given register.
01111	dec_R	r:3	Decrements value of a given register.
10000000	dec_A		Decrements value of acc.
10000001	inc_D		Increments value of dataPtr.
10000010	dec_D		Decrements value of dataPtr.
10000011	cplA		Inverts value of acc bitwise.
10000100	anlA_Next		Performs bitwise and between acc and next byte after call.
10000101	orlA_Next		Performs bitwise or between acc and next byte after call.
10000110	anlA_DPtr		Performs bitwise and between acc and memory where dataPtr points.
10000111	orlA_DPtr		Performs bitwise or between acc and memory where dataPtr points.
10001	anlA_R	r:3	Performs bitwise and between acc and given register.
10010	orlA_R	r:3	Performs bitwise or between acc and given register.
10011	addA_R	r:3	Adds value of given register to acc.
10100	addcA_R	r:3	Adds value of given register to acc, with carry.
10101	subA_R	r:3	Subtracts value of given register from acc.
10110	subbA_R	r:3	Subtracts value of given register from acc, with borrow.
10111000	addA_Next		Adds value of next byte after call to acc.
10111001	addcA_Next		Adds value of next byte after call to acc, with carry.
10111010	subA_Next		Subtracts value of next byte after call from acc.

Opcode prefix	Instruction name	Params	Description
10111011	subbA_Next		Subtracts value of next byte after call from acc, with borrow.
10111100	addA_DPtr		Adds value of memory where dataPtr points to acc.
10111101	addcA_DPtr		Adds value of memory where dataPtr points to acc, with carry.
10111110	subA_DPtr		Subtracts value of memory where dataPtr points from acc.
10111111	subbA_DPtr		Subtracts value of memory where dataPtr points from acc, with borrow.
11000000	rlA		Rotates acc left with 1 bit.
11000001	rrA		Rotates acc right with 1 bit.
11000010	rlcA		Rotates acc left with 1 bit, using carry flag.
11000011	rrcA		Rotates acc right with 1 bit, using carry flag.
11000100	pushDH		Pushes high byte of dataPtr to stack.
11000101	popDH		Pops high byte of dataPtr from stack.
11000110	pushDL		Pushes low byte of dataPtr to stack.
11000111	popDL		Pops low byte of dataPtr from stack.
11001000	pushA		Pushes value of acc to stack.
11001001	popA		Pops value of acc from stack.
11001010	sjmpNext		Short jumps program counter with value of next byte.
11001011	jmpNext		Absolute jumps to memory address given by next 2 bytes.
11001100	jmpD		Absolute jumps to value of dataPtr.
11001101	callD		Pushes program counter, and then jmpD.
11001110	callNext		Pushes program counter, and then jmpNext.
11001111	ret		Pops program counter from stack.
11010000	jnc_D		jmpD if carry bit is not set.
11010001	jc_D		jmpD if carry bit is set.
11010010	sjnc_Next		sjmpNext if carry bit is not set.
11010011	sjc_Next		sjmpNext if carry bit is set.
11010100	jnc_Next		jmpNext if carry bit is not set.
11010101	jc_Next		jmpNext if carry bit is set.
11010110	jnzA_D		jmpD if acc is not zero.
11010111	jzA_D		jmpD if acc is zero.
11011000	sjnzA_Next		sjmpNext if acc is not zero.

Opcode prefix	Instruction name	Params	Description
11011001	`sjzA_Next`		`sjmpNext` if `acc` is zero.
11011010	`jnzA_Next`		`jmpNext` if `acc` is not zero.
11011011	`jzA_Next`		`jmpNext` if `acc` is zero.
11011100	`csjneA_Next_Next`		`sjmpNext` if `acc` and next byte are not equal.
11011101	`cjneA_Next_Next`		`jmpNext` if `acc` and next byte are not equal.
11011110	`cjneA_Next_D`		`jmpD` if `acc` and next byte are not equal.
11011111	`halt`		Signals the VM to halt.
11100	`outR`	`r:3`	Outputs the value of a given register.
11101	`outDPtr`	`n:3`	Outputs next n bytes from where `dataPtr` points.
11110	`outiDPtr`	`n:3`	Outputs next n bytes from where `dataPtr` points, and increases `dataPtr` with n.
11111	`outNext`	`n:3`	Outputs next n bytes from memory.

Table 7: The instruction set description of the immediate OCI. The parameter mapping represents name and bit count.

C.2 The indirect opcode interpreter

Opcode prefix	Instruction name	Params	Description
00000	movR_DH	r:3	Copies high byte of dataPtr into given register.
00001	movDH_R	r:3	Copies value of given register into high byte of dataPtr.
00010	movR_DL	r:3	Copies low byte of dataPtr into given register.
00011	movDL_R	r:3	Copies value of given register into low byte of dataPtr.
00100	movR_DPtr	r:3	Copies 1 byte from memory where dataPtr points, into given register.
00101	movDPtr_R	r:3	Copies value of given register to the memory where dataPtr points.
00110	movR_A	r:3	Copies acc into given register.
00111	movA_R	r:3	Copies given register into acc.
01000	xchA_R	r:3	Exchanges values of acc and given register.
01001000	movA_DH		Copies high byte of dataPtr into acc.
01001001	movA_DL		Copies value of acc into high byte of dataPtr.
01001010	movDH_A		Copies low byte of dataPtr into acc.
01001011	movDL_A		Copies value of acc into low byte of dataPtr.
01001100	movA_DPtr		Copies 1 byte from memory where dataPtr points, into acc.
01001101	movDPtr_A		Copies value of acc to the memory where dataPtr points.
01001110	cplC		Inverts value of carry flag.
01001111	inc_A		Increments value of acc.
01010000	dec_A		Decrements value of acc.
01010001	inc_D		Increments value of dataPtr.
01010010	dec_D		Decrements value of dataPtr.
01010011	cplA		Inverts value of acc bitwise.
010101	anlA_RPtr	rp:2	Performs bitwise and between acc and memory memory where given register points.
01011	anlA_R	r:3	Performs bitwise and between acc and given register.
01100	orlA_R	r:3	Performs bitwise or between acc and given register.
01101	addA_R	r:3	Adds value of given register to acc.
01110	addcA_R	r:3	Adds value of given register to acc, with carry.
01111	subA_R	r:3	Subtracts value of given register from acc.

Opcode prefix	Instruction name	Params	Description
10000	subbA_R	r:3	Subtracts value of given register from acc, with borrow.
100010	orlA_RPtr	rp:2	Performs bitwise or between acc and memory memory where given register points.
10001100	anlA_DPtr		Performs bitwise and between acc and memory where dataPtr points.
10001101	orlA_DPtr		Performs bitwise or between acc and memory where dataPtr points.
10001110	addA_DPtr		Adds value of memory where dataPtr points to acc.
10001111	addcA_DPtr		Adds value of memory where dataPtr points to acc, with carry.
100100	addA_RPtr	rp:2	Adds value of memory where given register points to acc.
100101	addcA_RPtr	rp:2	Adds value of memory where given register points to acc, with carry.
100110	subA_RPtr	rp:2	Subtracts value of memory where given register points from acc.
100111	subbA_RPtr	rp:2	Subtracts value of memory where given register points from acc, with borrow.
10100000	subA_DPtr		Subtracts value of memory where dataPtr points from acc.
10100001	subbA_DPtr		Subtracts value of memory where dataPtr points from acc, with borrow.
10100010	rlA		Rotates acc left with 1 bit.
10100011	rrA		Rotates acc right with 1 bit.
10100100	rlcA		Rotates acc left with 1 bit, using carry flag.
10100101	rrcA		Rotates acc right with 1 bit, using carry flag.
10100110	pushDH		Pushes high byte of dataPtr to stack.
10100111	popDH		Pops high byte of dataPtr from stack.
10101000	pushDL		Pushes low byte of dataPtr to stack.
10101001	popDL		Pops low byte of dataPtr from stack.
10101010	pushA		Pushes value of acc to stack.
10101011	popA		Pops value of acc from stack.
101011	pushR	r:2	Pushes value of a register to stack.
101100	popR	r:2	Pops value of a register from stack.
101101	sjmpR	r:2	Short jumps program counter with value of register (R0-R3).

Opcode prefix	Instruction name	Params	Description
101110	jmpR	rp:2	Absolute jumps to memory address given by pair of registers.
10111100	jmpD		Absolute jumps to value of dataPtr.
10111101	callD		Pushes program counter, and then jmpD.
10111110	ret		Pops program counter from stack.
10111111	jnc_D		jmpD if carry bit is not set.
110000	scallR	r:2	Pushes program counter, and then sjmpR.
110001	callR	rp:2	Pushes program counter, and then jmpR.
11001000	jc_D		jmpD if carry bit is set.
11001001	jnzA_D		jmpD if acc is not zero.
11001010	jzA_D		jmpD if acc is zero.
11001011	halt		Signals the VM to halt.
110011	sjnc_R	r:2	sjmpR if carry bit is not set.
110100	sjc_R	r:2	sjmpR if carry bit is set.
110101	jnc_R	rp:2	jmpR if carry bit is not set.
110110	jc_R	rp:2	jmpR if carry bit is set.
110111	sjnzA_R	r:2	sjmpR if acc is not zero.
111000	sjzA_R	r:2	sjmpR if acc is zero.
111001	jnzA_R	rp:2	jmpR if acc is not zero.
111010	jzA_R	rp:2	jmpR if acc is zero.
111011	outRPtr	rp:2	Outputs byte from where a pair of registers is pointing.
11110	outR	r:3	Outputs the value of a given register.
11111	outDPtr	n:3	Outputs next n bytes from where dataPtr points.

Table 8: The instruction set description of the indirect OCI. The parameter mapping represents name and bit count.

D Resulting MIDI files

In this section we present some of the models chosen randomly from our results presented in Section 8. The script performing the selection is included under the name `randresults.py`. It uses the following steps:

1. collects the highest scoring models from all test runs (a total of 560 models);
2. only uses the model with an overall grade above or equal to 70% (resulting in 122 models out of 560);
3. chooses 10 models randomly;
4. crops the selected models to a length of 96 ticks (6 bars) with a randomly selected start tick;
5. outputs the resulting cropped files (attached to current document; see Appendix A).

Figures 27 through 36 show the resulting models together with information about their associated test run.

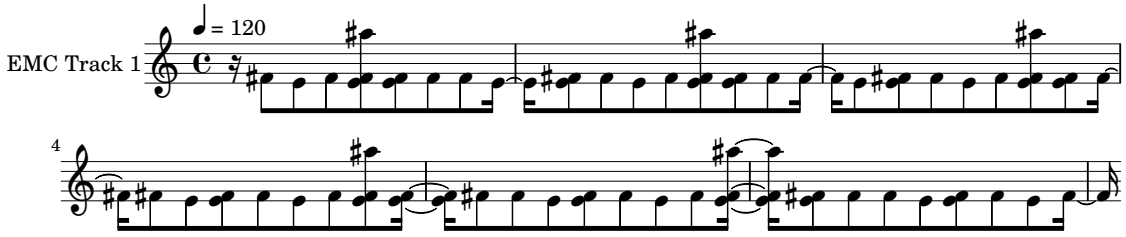


Figure 27: Resulting model from a test run using the indirect OCI, single track corpus, pop size of 256, cropped from 697 to 793 ticks. This model achieved an overall grade of 0.747.

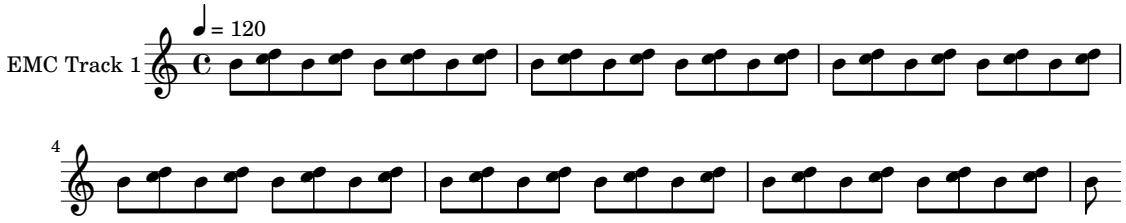


Figure 28: Resulting model from a test run using the immediate OCI, single track corpus, pop size of 512, cropped from 622 to 718 ticks. This model achieved an overall grade of 0.794.

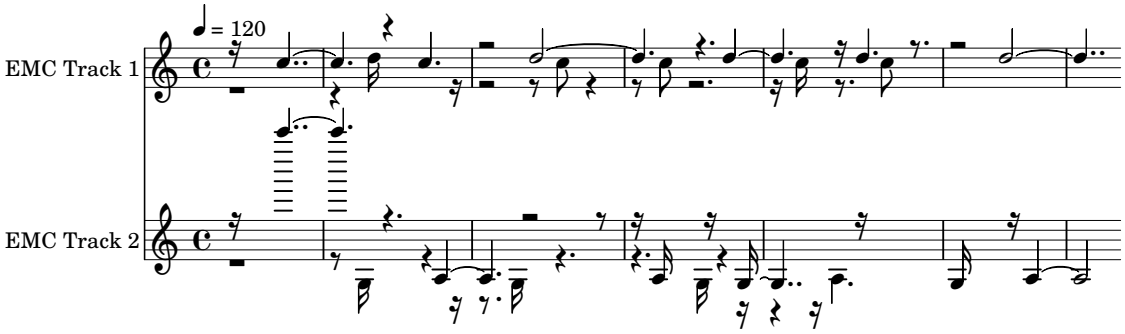


Figure 29: Resulting model from a test run using the indirect OCI, dual track corpus, pop size of 1024, cropped from 4 to 100 ticks. This model achieved an overall grade of 0.801.

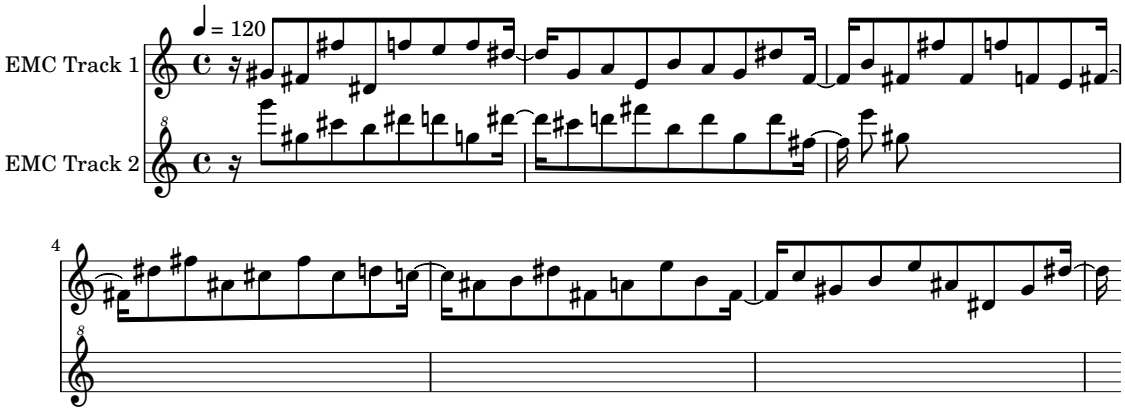


Figure 30: Resulting model from a test run using the indirect OCI, dual track corpus, pop size of 1024, cropped from 543 to 639 ticks. This model achieved an overall grade of 0.874.

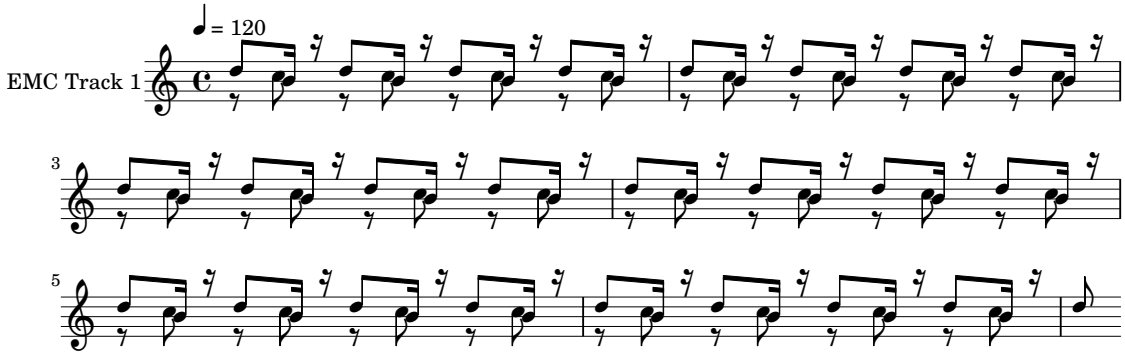


Figure 31: Resulting model from a test run using the immediate OCI, single track corpus, pop size of 512, cropped from 12 to 108 ticks. This model achieved an overall grade of 0.780.

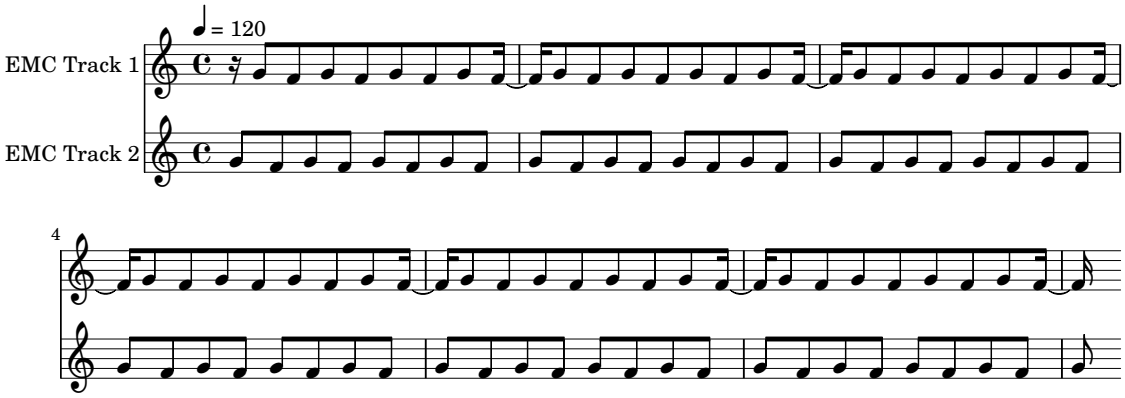


Figure 32: Resulting model from a test run using the indirect OCI, dual track corpus, pop size of 1024, cropped from 121 to 217 ticks. This model achieved an overall grade of 0.771.

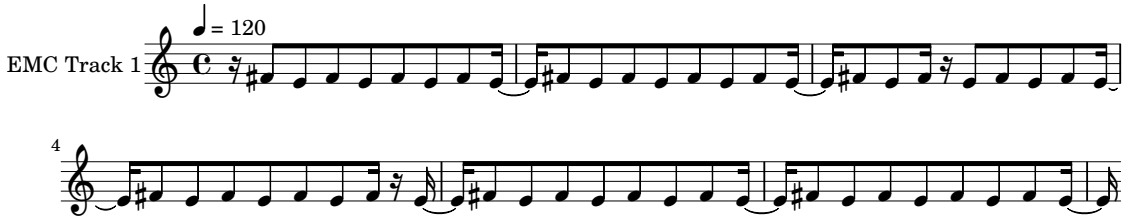


Figure 33: Resulting model from a test run using the immediate OCI, single track corpus, pop size of 256, cropped from 625 to 721 ticks. This model achieved an overall grade of 0.711.

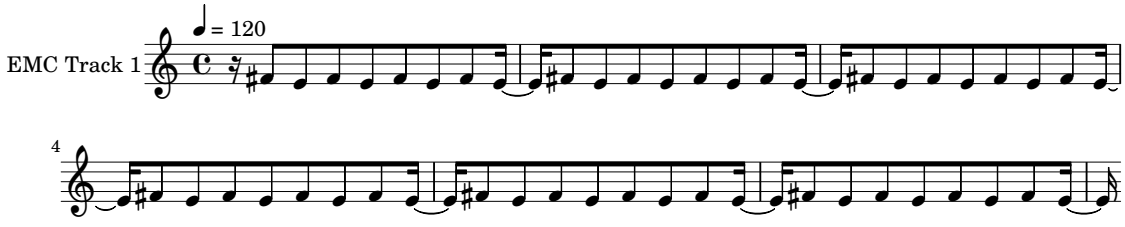


Figure 34: Resulting model from a test run using the indirect OCI, single track corpus, pop size of 64, cropped from 731 to 827 ticks. This model achieved an overall grade of 0.724.

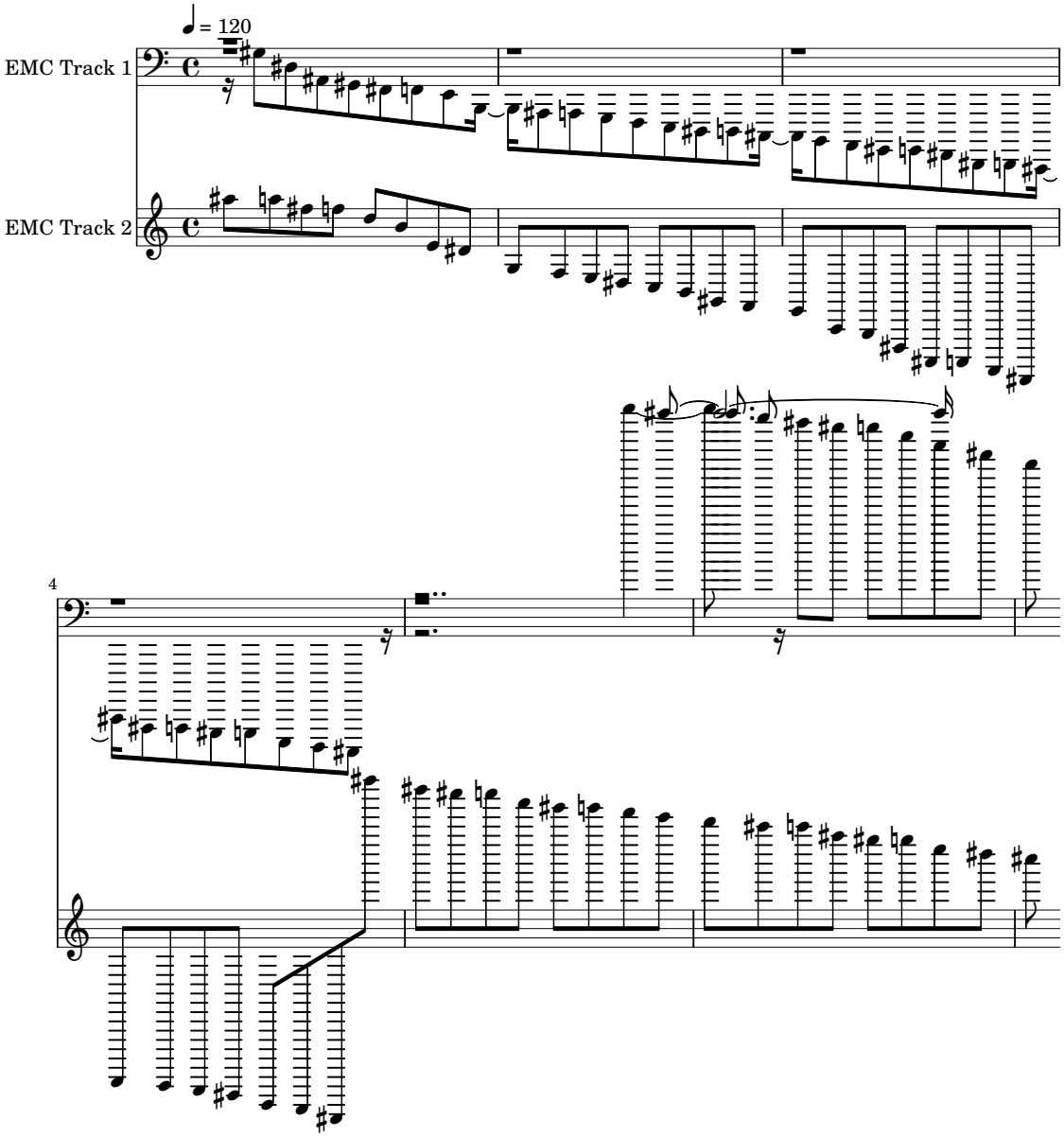


Figure 35: Resulting model from a test run using the indirect OCI, dual track corpus, pop size of 256, cropped from 388 to 484 ticks. This model achieved an overall grade of 0.720.

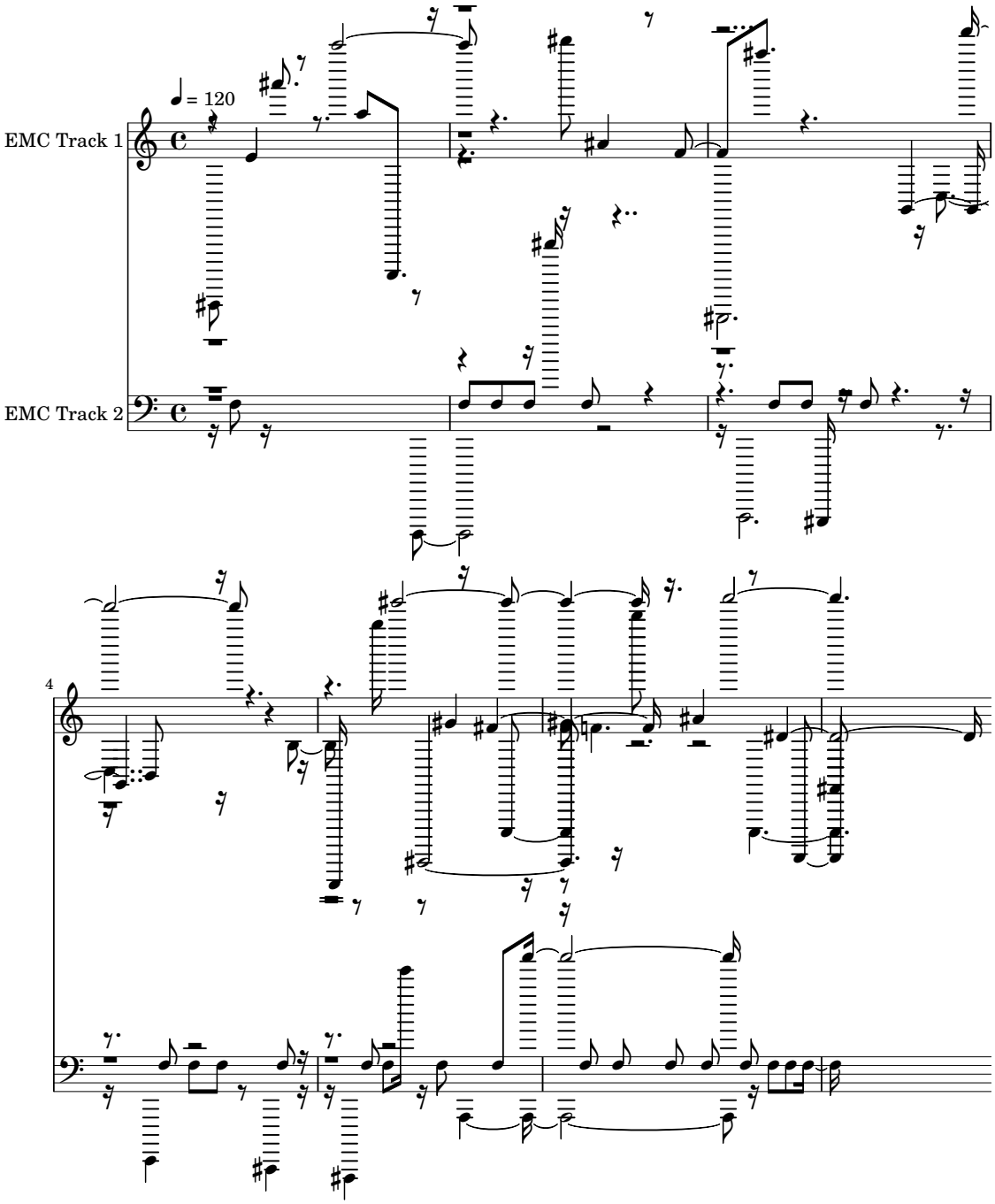


Figure 36: Resulting model from a test run using the immediate OCI, dual track corpus, pop size of 1024, cropped from 137 to 233 ticks. This model achieved an overall grade of 0.743.

References

M. Alfonseca, M. Cebrian, and A. Ortega. A simple genetic algorithm for music generation by means of algorithmic information theory. In *IEEE Congress on Evolutionary Computation*, pages 3035–3042. IEEE, 2007. URL http://dblp.uni-trier.de/db/conf/cec/cec2007.html#AlfonsecaCO07.

J. Biles. GenJam: A genetic algorithm for generating jazz solos. In *Proceedings of the 1994 International Computer Music Conference, ICMA*, pages 131–137, 1994.

M. F. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 1441940480, 9781441940483.

C.-C. J. Chen and R. Miikkulainen. Creating melodies with evolving recurrent neural networks. In *Proceedings of the INNS-IEEE International Joint Conference on Neural Networks*, pages 2241–2246, Piscataway, NJ, 2001. IEEE. URL http://nn.cs.utexas.edu/?chen:ijcnn01.

E. Chew and X. Wu. Separating voices in polyphonic music: A contig mapping approach. In U. K. Wiil, editor, *CMMR*, volume 3310 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004. ISBN 3-540-24458-1. URL http://dblp.uni-trier.de/db/conf/cmmr/cmmr2004.html#ChewW04.

N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc. ISBN 0-8058-0426-9. URL http://dl.acm.org/citation.cfm?id=645511.657085.

P. Dahlstedt. Autonomous Evolution of Complete Piano Pieces and Performances. In *9th European Conference on Artificial Life*, 2007.

R. De Prisco, G. Zaccagnino, and R. Zaccagnino. A multi-objective differential evolution algorithm for 4-voice compositions. In *Differential Evolution (SDE), 2011 IEEE Symposium on*, pages 1–8, April 2011. doi: 10.1109/SDE.2011.5952053.

B. Dolin, M. G. Arenas, and J. J. M. Guervós. Opposites attract: Complementary phenotype selection for crossover in genetic programming. In *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, PPSN VII, pages 142–152, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44139-5. URL http://dl.acm.org/citation.cfm?id=645826.669282.

P. Donnelly and J. Sheppard. Evolving four-part harmony using genetic algorithms. In *Applications of Evolutionary Computation*, volume 6625, pages 273–282. Springer Berlin Heidelberg, 2011.

M. Dostál. Musically meaningful fitness and mutation for autonomous evolution of rhythm accompaniment. *Soft Computing*, 16(12):2009–2026, 2012. ISSN 1432-7643. doi: 10.1007/s00500-012-0875-8. URL http://dx.doi.org/10.1007/s00500-012-0875-8.

A. Eigenfeldt. The evolution of evolutionary software: Intelligent rhythm generation in kinetic engine. In *EvoWorkshops*, volume 5484, pages 498–507. Springer, 2009. ISBN 978-3-642-01128-3. URL http://dblp.uni-trier.de/db/conf/evoW/evoW2009.html#Eigenfeldt09.

A. Eigenfeldt. Corpus-based recombinant composition using a genetic algorithm. In *Soft Computing*, volume 16, pages 2049–2056. Springer, 2012.

C. M. Fonseca and P. J. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 416–423, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. ISBN 1-55860-299-2. URL http://dl.acm.org/citation.cfm?id=645513.657757.

M. Fowler. Domain specific language. "http://www.martinfowler.com/bliki/DomainSpecificLanguage.html", 2008.

P. Gibson and J. Byrne. Neurogen, musical composition using genetic algorithms and cooperating neural networks. In *Artificial Neural Networks, 1991., Second International Conference on*, pages 309–313, Nov 1991.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN 0201157675.

A. Hadjakos and F. Lefebvre-Albaret. Three methods for pianist hand assignment. In *Proceedings of the 6th Sound and Music Computing Conference*, pages 321–326, 2009.

P. Hartmann. Natural selection of musical identities. In *International Computer Music Conference*, 1990.

A. Horner and D. E. Goldberg. Genetic algorithms and computer-assisted music composition. In R. K. Belew and L. B. Booker, editors, *ICGA*, pages 437–441. Morgan Kaufmann, 1991. ISBN 1-55860-208-9. URL http://dblp.uni-trier.de/db/conf/icga/icga1991.html#HornerG91.

D. Horowitz. Generating rhythms with genetic algorithms. In *AAAI*, volume 94, page 1459, 1994.

B. Jacob. Composing with genetic algorithms. In *International Computer Music Association*, pages 452–455, 1995.

A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-022278-X.

P. B. Kirlin and P. E. Utgoff. Voise: Learning to segregate voices in explicit and implicit polyphony. In *ISMIR*, pages 552–557, 2005. URL http://dblp.uni-trier.de/db/conf/ismir/ismir2005.html#KirlinU05.

J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

P. A. Laplante. *What Every Engineer Should Know About Software Engineering (What Every Engineer Should Know)*. CRC Press, Inc., Boca Raton, FL, USA, 2007. ISBN 0849372283.

A. Lipowski and D. Lipowska. Roulette-wheel selection via stochastic acceptance. *CoRR*, abs/1109.3627, 2011. URL http://arxiv.org/abs/1109.3627.

R. Loughran, J. McDermott, and M. O'Neill. Grammatical evolution with Zipf's law based fitness for melodic composition. In J. Timoney, editor, *Proceedings of The 12th Sound and Music Computing Conference*, Maynooth, Ireland, 26 July - 1 Aug. 2015. URL http://ncra.ucd.ie/papers/smc2015_loughran.pdf.

R. M. MacCallum, M. Mauch, A. Burt, and A. M. Leroi. Evolution of music by public choice. *Proceedings of the National Academy of Sciences*, 109(30):12081–12086, 2012.

S. T. Madsen and G. Widmer. Separating voices in midi. In *ISMIR*, pages 57–60, 2006.

S. W. Mahfoud. Niching methods for genetic algorithms. Technical report, 1995.

J. M. Martins and E. R. Miranda. Emergent rhythmic phrases in an A-Life environment. In *Proceedings of ECAL 2007 Workshop on Music and Artificial Life (MusicAL 2007)*, pages 11–14, 2007.

R. McIntyre. Bach in a box: the evolution of four part baroque harmony using the genetic algorithm. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 852–857 vol.2, Jun 1994. doi: 10.1109/ICEC.1994.349943.

E. R. Miranda and J. A. Biles. *Evolutionary Computer Music*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 1846285992.

T. Murata and H. Ishibuchi. MOGA: multi-objective genetic algorithms. In *Evolutionary Computation, 1995., IEEE International Conference on*, volume 1, pages 289–, Nov 1995. doi: 10.1109/ICEC.1995.489161.

P. Nordin and W. Banzhaf. Evolving Turing-complete programs for a register machine with self-modifying code. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 318–327, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-370-0. URL http://dl.acm.org/citation.cfm?id=645514.657920.

C. Ó. Nuanáin, P. Herrera, and S. Jorda. Target-based rhythmic pattern generation and variation with genetic algorithms. In *Proceedings of The 12th Sound and Music Computing Conference*, Maynooth, Ireland, 30/07/2015 2015.

M. O'Neill and C. Ryan. Grammatical evolution. *Evolutionary Computation, IEEE Transactions on*, 5(4): 349–358, Aug 2001. ISSN 1089-778X.

S. Phon-Amnuaisuk, A. Tuson, and G. Wiggins. Evolving musical harmonisation. In *In ICANNCA*, 1999.

J. Reddin, J. McDermott, and M. O'Neill. Elevated pitch: Automated grammatical evolution of short compositions. In *Applications of Evolutionary Computing*, volume 5484 of *Lecture Notes in Computer Science*, pages 579–584. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-01128-3.

J. D. F. Rodriguez and F. J. Vico. AI methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research*, pages 513–582, 2013.

R. J. Rummel. Understanding correlation. *Honolulu: Department of Political Science, University of Hawaii*, 1976.

S. Sauvage. Python + Windows + distutils + SWIG + gcc MinGW. "http://sebsauvage.net/python/mingw.html", 2003.

R. Stansifer. The MIDI File Format. "http://cs.fit.edu/~ryan/cse4051/projects/midi/midi.html".

C. Sulyok, A. McPherson, and C. Harte. Corpus-taught evolutionary music composition. In *Proceedings of the 13th European Conference on Artificial Life*, pages 587–594, York, UK, 2015.

K. Thywissen. Genotator: An environment for exploring the application of evolutionary techniques in computer-assisted composition. *Org. Sound*, 4(2):127–133, June 1999. ISSN 1355-7718. doi: 10.1017/S1355771899002095. URL http://dx.doi.org/10.1017/S1355771899002095.

N. Tokui and H. Iba. Music composition with interactive evolutionary computation. In *Proceedings of the 3rd international conference on generative art*, volume 17, pages 215–226, 2000.

M. Towsey, A. Brown, S. Wright, and J. Diederich. Towards melodic extension using genetic algorithms. *Educational Technology & Society*, 4(2), 2001. URL http://dblp.uni-trier.de/db/journals/ets/ets4.html#TowseyBWD01.

A. M. Turing. Computing machinery and intelligence. In *Mind*, volume 59, pages 433–460, 1950. URL http://cogprints.org/499/.

T. Unemi. SBEAT3: A tool for multi-part music composition by simulated breeding, 2002.

R. I. Waschka. Composing with genetic algorithms: GenDash. In *Evolutionary Computer Music*. Springer London, 2007.

R. P. Whorley, C. Rhodes, G. Wiggins, and M. T. Pearce. Harmonising melodies: Why do we add the bass line first? In *International Conference on Computational Creativity*, pages 79–86, Sydney, 2013. URL http://research.gold.ac.uk/9795/.