



UNIVERSITY OF  
**BATH**

DEPARTMENT OF COMPUTER SCIENCE

FINAL YEAR PROJECT

# Algorithmic Composition of Jazz

*Author:*

Richard Harris

BSc. Computer Science with Honours

*Supervisor:*

Prof. John Fitch

April 25, 2008

# Algorithmic composition of jazz

Submitted by: Richard Harris

## **COPYRIGHT**

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see <http://www.bath.ac.uk/ordinances/#intelprop>).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

## **Declaration**

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:

## **Abstract**

*JazzGen* is a jazz improvisation system that uses a genetic algorithm to evolve a solo and accompaniment over a user-supplied chord progression. To aid experimentation the system is both modular and heavily configurable via an extensive user interface. This report details the design, development and experimentation of *JazzGen*; various musical rules and constraints are encoded into a fitness function with the aim of producing “good” jazz music. The output is shown to be of generally acceptable quality approaching that of a novice jazz player. In particular, Narmour’s model of melodic expectancy is applied to the generation with pleasing results, and repetition is used to encourage structure. Output is still not close to most human improvisation, but the system shows potential for further development and several extensions are considered.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem description . . . . .	1
1.2	Jazz music . . . . .	2
1.3	Genetic algorithms . . . . .	2
<b>2</b>	<b>Literature Survey</b>	<b>4</b>
2.1	Overview . . . . .	4
2.2	Melody generation . . . . .	4
2.3	Chord progressions . . . . .	11
2.4	The psychology of music . . . . .	12
2.5	The Implication-Realization model . . . . .	14
2.6	Non-pitch aspects of music . . . . .	17
2.7	Conclusion . . . . .	18
<b>3</b>	<b>Requirements and Design</b>	<b>21</b>
3.1	Initial considerations . . . . .	21
3.2	The genetic algorithm . . . . .	25
3.3	User interface . . . . .	29
3.4	Development environment . . . . .	30
3.5	Requirements . . . . .	33
<b>4</b>	<b>Development</b>	<b>37</b>
4.1	Music generation . . . . .	37
4.2	Fitness function . . . . .	41

4.3	Fitness modules . . . . .	42
4.4	User interface . . . . .	45
4.5	Chord progression . . . . .	50
<b>5</b>	<b>Testing and Experimentation</b>	<b>52</b>
5.1	Early fitness function . . . . .	52
5.2	Testing and debugging . . . . .	53
5.3	Genetic algorithm configuration: part one . . . . .	54
5.4	Post-pass . . . . .	59
5.5	Genetic algorithm configuration: part two . . . . .	60
5.6	Mutation . . . . .	61
<b>6</b>	<b>Evaluation</b>	<b>65</b>
6.1	Requirements analysis . . . . .	65
6.2	Future development . . . . .	67
6.3	Conclusion . . . . .	70
<b>A</b>	<b>Sample Lilypond output</b>	<b>73</b>
<b>B</b>	<b>Original fitness function specification</b>	<b>76</b>
<b>C</b>	<b>Code listing</b>	<b>77</b>
C.1	Main classes . . . . .	77
C.2	Genetic algorithm method classes . . . . .	87
C.3	Fitness modules . . . . .	93
C.4	Chord progression . . . . .	95
<b>D</b>	<b>Running the code</b>	<b>101</b>
	<b>References</b>	<b>102</b>

# Acknowledgements

First and foremost, my friends and family. My supervisor, Professor John Fitch, who has provided support and feedback in a most wonderful manner. The holy trinity of energy drinks: Relentless, Rockstar and Jolt Cola, for keeping me alert and creative. Special mention must be made of Dr. Oetker's Ristorante Speciale pizza<sup>1</sup>, not because it was of particular help to this dissertation, but because it is the best pizza ever created.

---

<sup>1</sup>[http://www.oetker.co.uk/wga/oetker\\_uk/html/default/debi-5b2jwr.en.html](http://www.oetker.co.uk/wga/oetker_uk/html/default/debi-5b2jwr.en.html)

# Chapter 1

## Introduction

All musicians are subconsciously mathematicians.

— *Thelonious Monk*

### 1.1 Problem description

Algorithmic composition could be described as “a sequence of rules for combining musical parts into a whole” (Cope, 1993). This definition does not constrain one to the use of a computer to perform the composition, as evidenced by Mozart’s famous Musical Dice Game in 1787; or in 1026, when Guido d’Adrezzo proposed a scheme to assign pitches to vowel sounds in religious texts (Loy, 1989). However, the advent of computers has brought forth much more powerful techniques for music generation: rule and constraint systems, complex mathematical models, statistical databases, and many others.

This project employs a *genetic algorithm* to generate jazz improvisations over a user-supplied chord progression. Such algorithms begin with a *population* of initial subjects (melodic phrases), and slowly evolve them towards more suitable phrases until certain criteria are satisfied. Central to this notion is the *fitness function*, a function that attempts to numerically quantify the quality of each phrase in the population. Conventionally such an assessment is performed by a human, which creates a fitness bottleneck (Biles, 1994) as humans take seconds, if not minutes to evaluate phrases. This project attempts to devise an *objective*, mostly deterministic fitness function that relies on a mix of common-sense constraints, and models such as Narmour’s *Implication-Realization Model* (1990, 1992) for melodic expectancy.

Genetic algorithms suffer from having a large complexity: in addition to the fitness function and its hundreds or thousands of constraints, each stage of evolution (mutation, crossover, selection) can be implemented in a variety of ways, each having a different effect on the final output. This project attempts to solve this dilemma by providing a full user interface, allowing the user to select and experiment with different genetic methods and hear the



associated changes in the musical output. The fitness function will be split up into modules that can be enabled or disabled at will, and the results combined via a weighted average. Changing the weighting of these modules will allow even greater control over the final piece.

Finally, regardless of algorithm, such compositions tend to wander and are usually perceived by the listener as aimless and mechanical. This project examines and implements a variety of techniques to combat this by introducing a higher-level structure, and performing a post-pass on the musical output to remove evidence of its mechanical, computer-generated nature. The overall aim is not necessarily to produce human-quality output, but to explore promising approaches to eventually reaching this goal.

## 1.2 Jazz music

Jazz is a broad genre with divergent musical styles, but most share some central traits: improvisation, swing rhythm<sup>1</sup>, polyrhythms, seventh chords and other complex harmonies that are more dissonant than typical Western music. Jazz is a contradiction: it has a large number of informal rules such as common scales to play over chords, drum and acoustic bass patterns. But these rules are often broken, such as when John Coltrane began playing modal jazz<sup>2</sup>. Jazz provides a basic framework for a composition algorithm, without being so strict as to limit the musical possibility space.

Such algorithms can generate music in two main ways: either generate it iteratively, or generate the entire piece at once. The latter tends to involve more long-term, large-scale planning and constraint satisfaction, but it is not a good fit for jazz music. Improvisation, which is key to jazz, models the former process: based on previous notes, output the next note. Variables such as past experience, mood and ability govern the music produced, and planning ahead is not a major factor.

Finally, jazz has always progressed, from its origins in everything from ragtime to Dixieland, to swing, bebop and free jazz. In the spirit of this progression, this project continues to explore musical techniques, in the hope of progressing the state of algorithmic composition techniques as a whole.

## 1.3 Genetic algorithms

Genetic algorithms (GAs) were originally used with bit strings to find optimal solutions to numerical problems (Holland, 1975). However, they can be used with arbitrary data structures, and in this case melodic phrases are the subjects. Horner et al. (1993) provides

---

<sup>1</sup>A rhythm in which the duration of the initial note in a pair is augmented and that of the second is diminished. Usually 8<sup>th</sup> notes.

<sup>2</sup>Instead of playing over several scales set to a chord progression, modal jazz involves improvising over one musical mode (scale) for the whole song. This allows more freedom of expression, but it is harder for solos to remain interesting without scale changes.

an excellent example of a “simple” GA:

```
Initialize the individuals in the population
While not finished evolving the population
    Assess the fitness of each individual
    Select better individuals to be parents
    Breed new individuals
    Build next generation with new individuals
Loop
```

The initial melodic phrases are generated by some simple procedure (for example, using random numbers). Each iteration of the algorithm selects the “most fit” members of the population, combines and mutates them to build a stronger new population. The nature of the combination and mutation algorithms, as well as the fitness assessment itself, can have profound effects on the output of the algorithm.

Two traits of GAs make them desirable for jazz improvisation: they have a random element in the nature of combination, mutation and selection, which creates unpredictable and varied results; and they rapidly converge to *good* solutions without necessarily finding the *optimum*. While the latter would be an issue for many classes of problem, composition is not as clear-cut and there will be many good solutions, ensuring that each musical output is unique.

## Chapter 2

# Literature Survey

Never play anything the same way twice.

— *Louis Armstrong*

### 2.1 Overview

Algorithmic composition is an active research topic with a large degree of interaction with psychology, engineering and computing. To this end, this report focuses not only on the methodology of music generation (stochastic models, genetic algorithms, etc.) but also on the psychology of music. To quote Papadopoulos and Wiggins (1999), “the big disadvantage of most, if not all, the computational models is that the music they produce is meaningless: the computers do not have feelings, moods or intentions, they do not try to describe something with their music as humans do.”

Another area of background reading focuses on the importance of rhythm and form in composition, which directly contrasts with the importance most current research places on pitch choice. More specifically, Narmour’s *Implication-Realization model* is explored, which posits that only general relations between pitches (small vs. large intervals) are important when determining a song’s emotional affect.

### 2.2 Melody generation

The process of computationally generating music is an open problem with a large range of methods in use; Papadopoulos and Wiggins (1999) outline the six most common composition approaches: mathematical models; knowledge based systems; grammars; evolutionary methods; systems which learn; and hybrid systems.

### 2.2.1 Mathematical models

Mathematical models are commonly used to create non-deterministic compositions from random events; the compiler exerts control over the process by weighting the probabilities of these events, and deciding how to map the model's output to something musically meaningful.

#### Markov chains

Any note in a given piece of music can be predicted from the notes before it; this is the concept of a Markov model. Based on a human-specified or empirically discovered set of probabilities, one or more previous notes (the number corresponding to the *order* of the model) are used as input for the Markov model, and a set of probabilities for the following note are returned. The generator simply selects one based on a weighted random number.

Markov models can be easily trained from a body of music, are simple to implement and efficiently accessed if used with the right data structures. Unfortunately, attempts to use a trained model on new input usually fails because the model is over-specified (Conklin, 2003). One solution is to perform transformations on the training data, such as transposing it into a common key.

Due to their tendency to produce wandering and occasionally repeating sequences, Markov chains are useful for chord progression and rhythm generation, but are no longer often used for solos unless combined with other techniques. For example, the *JiG: Jazz Improvisation Generator* (Grachten, 2001) uses a Markov model with additional constraints on tension and pitch contour over time. Sequences are generated in a base key then transposed to a more appropriate key based on the underlying chord. To further enhance the model, pre-stored “licks”, or short phrases of music, are dynamically included within the output in an attempt to mimic a jazz player's usage of premeditated phrases within an improvisation. This technique is also used in commercial implementations, such as “Band in a Box” (Keller and Morrison, 2007).

One complaint about such models are that they are inherently uncontrollable, that is, it is difficult to map changes in the model to changes in the output. Koenig (1970) developed “tendency masks” to allow the user parametric control, while Jones (1981) dismissed the complaint outright: “This is a needless concern, since stochastic generative schemes may produce results that sound far more ordered than what might be produced by a supposedly deterministic system.” Regardless, such models are unable to explicitly capture higher-level aspects of music as they are based entirely on short-term context.

#### 1/f noise

1/f noise (Bak et al., 1987) or “pink noise” is a ubiquitous frequency distribution found in systems ranging from the flow in the river Nile to the luminosity of the stars. Voss and

Clarke (1978) performed analysis of several works of music, from Bach concertos to the piano rags of Scott Joplin; in all cases, the spectral density below 1 kHz matched the  $1/f$  distribution.

Utilising this property, stochastic music was constructed with the  $1/f$  distribution (rather than a true random distribution, or “white noise”) as the source. Compared with both random and  $1/f^2$  (“Brownian noise”) generation, listeners thought the  $1/f$  music to be far more interesting, and a degree of long term structure was imposed that was lacking from the other distributions. The authors postulate that the  $1/f$  shape could also be used to control other parameters of the sound rather than pitch, with similarly pleasing results.

Rather than being the final output,  $1/f$  noise could be used as an initial seed to a genetic algorithm instead of a uniformly random distribution of pitches. This may result in an improvisation that can reach an aesthetically pleasing state more quickly.

### Chaotic systems

Chaotic systems are constructed by taking an initial value and applying a function to it, then applying the function to its previous output. Repeated iterations, or “orbits”, yields an unpredictable and interesting series of values that can (depending on the initialisation vector and function used) form quasi-repeating, self-similar patterns that mimic human composition. This approach has been used to great effect by Leach and Fitch (1995).

Bidlack (1992) conducted experiments with chaotic systems, mapping their outputs to the pitch, velocity and duration of notes. While the regular patterns of the output lent themselves to bass line generation, Bidlack cautioned against several potential problems:

- Some initial values for a given map may cause it to become *unbounded*; the system will eventually tail off to infinity.
- The level of precision when calculating such functions is extremely important, as small rounding errors can create large variations in the output.
- Due to this, different processors are likely to give different results.

Use of cross-platform arbitrary-precision arithmetic libraries (such as the *GNU Multiple Precision Arithmetic Library*<sup>1</sup>) solves the latter two problems. The first can be solved by experimenting and limiting the range of possible inputs, or fixing it entirely in the case of Morris (2005). Like  $1/f$  noise, the output of a chaotic function alone is not sufficient to pass as a genuine jazz improvisation. It may, however, be useful to have chaotic functions such as the *Standard Map* available as an initial input to a genetic algorithm.

---

<sup>1</sup><http://gmplib.org/>

## Other mathematics

Xenakis (1971) covers many approaches in his book *Formalized Music*, including the use of group theory in the piece *Nomos Alpha*: permutations of the twenty-four isometries of the cube are selected and composited in a “Fibonacci motion”. Ultimately these techniques are mathematically and musically interesting, but of limited use when attempting to model an evolved and complex style of music.

### 2.2.2 Knowledge-based systems

If one can isolate the aesthetic code of a musical genre, this code can be used to create new similar compositions. This is the aim of knowledge-based systems: a series of rules, tests and constraints are defined, and the musical output must conform to these constraints. A core advantage of this approach is that knowledge-based systems are capable of explaining their choice of actions.

Unfortunately, this method requires a tremendous amount of care and attention before yielding satisfying output. Loy (1991) examines an attempt by Schottstaedt (1984) to implement a rule-based expert system for composing counterpoint, and its pitfalls: firstly, several of the rules (derived from Fux, 1725) had ambiguities, or were incomplete. Others needed fine-tuning of weights and constraints until the designer had roughly doubled the number of rules in the system. Even with these amendments, the system was only as good as “a typical freshmen composition student” (according to Loy).

While knowledge-based systems are competent at formalizing some aspects of the compositional process, these aspects are “the most elemental and well-known of Western musical styles”; rule-based systems “do not handle ambiguity gracefully, and are difficult to operate meaningfully in parallel.” Ambiguity is very common in most forms of music, due in part to its inherent parallelism (e.g. between melodic, harmonic, rhythmic and timbral dimensions). Furthermore, expectancy violation is necessary to keep the listener engaged. Meyer (1956) initially developed the theory that music is organised by the interweaving of expectation and surprise – Baroque music, for example, is designed around keeping the listener “off balance, and hence mentally engaged in the unfolding composition.”

### 2.2.3 Grammars

Compositions can also be created by constructing a musical grammar, viewing music itself as a language with a distinctive grammar set. Steedman (1984) produced a generative grammar for chord progressions in jazz twelve-bar blues, while Johnson-Laird (1991) also used grammars for the generation of jazz chord progressions and bass line improvisations.

For melody, grammars tend to be too restrictive and would involve a large number of rules, which is why most research focuses on other methods of melody generation. Structurally, grammars are hierarchical while improvised music is not, and this leads to unclear semantics

and ambiguity (Roads and Wieneke, 1979). “Even the simple phenomenon of a repeated phrase in a piece cannot adequately be expressed using a context-free grammar” (Conklin, 2003).

## 2.2.4 Evolutionary methods

Genetic algorithms (GAs) have proved to be very efficient at dealing with large search spaces, and one such space is that of music composition: pitch, amplitude (loudness), timbre and duration can all be modified within a terrifyingly large range. They are also capable of providing multiple solutions, which is often required in creative domains.

### GenJam

Biles (1994) used such algorithms to model a novice jazz musician learning to improvise; his program, GenJam, outputs solos over a rhythm section accompaniment and reacts to real-time feedback from a human “mentor”.

The generation process concerns strings of “symbols”, representing tuples of pitch, amplitude and duration. Defining measures as individuals, the GenJam algorithm first initialises a population from random notes and combines groups of four into families. The two with the highest fitness are marked as parents, and their offspring replace the weaker two. Children are calculated simply by crossing over random notes (bits) from each parent, and occasionally subjecting a child to mutation (inverting notes, sorting notes in ascending pitch order, etc.) in order to “accelerate learning” and ensure “not just new, but better offspring”.

GenJam’s large search space is reduced by making all notes have a fixed duration, choosing a pre-set scale for each chord in the progression, and dealing with only fourteen pitches at once. Additionally, a human is responsible for choosing the fitness of each measure: while the music is playing, the mentor types ‘g’ or ‘b’ to indicate good or bad music respectively. This feedback is used to determine the best parts of the generation, so they can be propagated to future output.

There are several issues with this approach. Using a human as a fitness function requires him or her to listen to every single program output, and worse still, the mentor could be affected by subjective bias, particularly after hearing a large amount of the output. Mentors can get “used” to the computer-generated, often unnatural sound of a genetic algorithm, and favour aspects of the music that an outsider would hesitate to recommend.

### Other approaches

GAs had previously been used to create thematic bridging between two simple melodies (Horner and Goldberg, 1991) and to generate four part Baroque harmonies from an input

melody (McIntyre, 1994). Rather than relying on a human overseer, these methods utilise an objective fitness function.

Gibson and Byrne (1991)’s method was designed to produce music in the style of traditional hymns; generation was restricted to C major and three-chord harmonies. Marques et al. (2000) developed a more complicated system including eight octaves of notes, pauses, chords and multiple instruments; only simple rules are used in the fitness function:

- **Harmony:** evaluates intervals between notes played simultaneously
- **Tone:** evaluates suitability of note for chosen tone and scale
- **Melody:** evaluates intervals between consecutive notes

According to the authors, the output is of acceptable quality, which is promising given the vastly extended search space and relative simplicity of the fitness evaluation.

### Interaction with neural networks

Unlike neural networks, genetic algorithms do not explicitly deal with the higher-level structure of music. Biles et al. (1996) attempted to integrate an artificial neural network (ANN) as a judge for GenJam’s improvisations, but were unsuccessful:

Upon examining the data, we found numerous situations where two measures were nearly identical in their chromosomes, but had maximally opposite fitnesses.

— *Biles et al., 1996*

Biles recommends the use of “knowledge intensive artificial intelligence techniques” in evaluating fitness, and specifically higher-level form, because “humans listen to music in complex and subtle ways that are not captured well by simple statistical models”. Instead of neural networks, a more complex algorithm could handle higher-level structure, and use these computations to guide the fitness function of note generation such that it follows this structure.

For example, Gibson and Byrne (1991)’s genetic algorithm determined melody quality by looking at both intervals between pitches, and the overall structure. This idea of examining the output from multiple viewpoints is something a genetic algorithm’s fitness function can do very well, and might “more closely simulate human musical thinking” (Papadopoulos and Wiggins, 1999). Other systems that use this approach include (Ebcioglu, 1988; Conklin and Witten, 1995).



## Criticisms

GAs in general still draw criticism as “their operation in no way simulates human behaviour” (Wiggins et al., 1998). Yet Cohen (2002) describes creativity as “not a random walk in a space of interesting possibilities, but ... directed”. Genetic algorithms offer this to a fault: they provide a *search* process that has *direction* via the fitness function. They are known to be extremely effective at dealing with large search spaces (Holland, 1975; Goldberg et al., 1989) and the mutation of GAs removes the limitations of over-specified heuristics or hill-climbing algorithms.

The notion of continual improvement via mutation and crossover is intuitively resonant with the idea of creativity; very rarely does a perfect solution immediately fall into the artist’s lap without further refinement. Jacob (1996) introduces two distinct modes of composition: *inspiration* and *hard work*. The former is more “inspired” but “we do not fully understand it and therefore have a slim chance of reproducing it”. The latter “resembles an iterative algorithm” and yields some similarities to the operation of a genetic algorithm.

### 2.2.5 Systems which learn

Learning systems extract information from musical material supplied by users or the system’s creator, and process this information into a piece of music similar to the examples provided. Such systems do not require any inherent knowledge of the genre of music they are working with. However, such Artificial Neural Networks (ANNs) tend to be used to solve “toy” problems, with simplified domains, compared with knowledge-based approaches (Toiviainen, 2000). ANNs can be useful for analysing music which cannot be expressed symbolically, but for jazz, we lose a great deal by dealing only in aggregated statistics and ignoring the context within which notes are placed.

### 2.2.6 Hybrid methods

Generally, approaches that utilise only one of the above methods produce poor results compared to using a combination of AI techniques. Gutknecht (1992) suggests a “post-modern” attitude of combining AI methods to cover for each others’ weaknesses and capitalise on their strengths. One area of current research is the use of genetic algorithms with ANNs as a cooperating fitness function (Gibson and Byrne, 1991; Spector and Alpern, 1995). These techniques have met with mixed results; Spector warns that “genetic programming will often find and exploit bizarre niches produced by weaknesses in fitness functions”. Papadopoulos and Wiggins (1999) point out that hybrid systems are complicated to implement and time-consuming to verify and validate.

## 2.3 Chord progressions

Integral to most forms of jazz improvisation is a harmonising sequence of chords played simultaneously with the melody. This is known as the chord progression of the song, or in jazz terminology, the “changes”. Tirro (1974) describes the progression as a “chordal framework...derived from a standard repertoire of changes derived from popular songs, blues riffs, and a few jazz originals.” While this interpretation clearly encourages a knowledge-based approach, others such as Steedman (1984) and Johnson-Laird (1991) have used formal grammars for chord progression and bass line generation.

At this point it is worth noting that the chord progression could be generated purely from the melody itself (choosing whatever chords fit the melody best), but it is simpler to generate the harmonic framework first. This allows a consistent, flowing harmony; if the melody was allowed to “run free”, the generated chords might jump about or clash together. Once the progression is set, there are more constraints on the melody, which makes that aspect of generation easier as well.

Steedman’s influential grammar takes the form of recursive “rewrite rules.” A common twelve-bar blues progression is provided as input:

I	I	I	I
IV	IV	I	I
V	V	I	I

A set of six core rules operate on this input to generate any well-formed twelve-bar jazz progression. The most important transformations are converting chords to their dominant or subdominant forms, addition of chromatic passing chords, and alterations such as minor sevenths or altered forms.

Steedman obtained the basis of this grammar from a list of jazz chord progressions by Coker (1964); this set is seen as a wide and representative range of permissive variations of the blues basic form. The process effectively captures the human process of improvisation, by performing substitutions on a simple chord progression until it is harmonically interesting. The grammar is capable of generating all of the sequences in Coker’s study, and can also handle 24-bar form “rhythm changes”, as heard in Gerschwin’s “I’ve Got Rhythm”.

However, the grammar only operates over one twelve-bar block at a time; it is not reactive to user input, and operating over a smaller portion of the progression would harshly limit the range of sequences generated by the grammar. For example, the number of rules to apply could be a user-specified setting (corresponding to how “complex” the progression becomes). If this setting is changed, they will only take effect on the *next* generation of a twelve-bar sequence, constituting a fairly serious feedback delay. In response to this, Chemillier (2004) proposes a real-time adaptation of the grammar, first attempting to precompile all possible chord sequences (which is untenable for sequences of a large size), then limiting this precompilation to a select set of “cadential sequences”; progressions

leading to a final dominant seventh chord are interrupted and extended with more chords ‘on the fly’.

It should be noted that Chemillier’s generation is somewhat simplified. His system ignores diminished seventh chords and the substitution of a rhythmically *unstressed* chord with its subdominant, addressed by rules 2 and 6 respectively. Rule 5 is also ignored, but this is justified because its application could only be found once in the set of sequences taken from Coker.

Steedman himself also addresses weaknesses with the system (1996), in particular its “minute coverage”, the difficulty of writing a parser (although the context-sensitive grammar can be converted to an almost equivalent context-free approach) and the parser’s large search space in certain conditions. It is assumed that these criticisms only relate to the parsing applications of the grammar. Steedman later investigated Longuet-Higgins’ *Theory of Tonal Harmony*, but there do not seem to be any immediate generative properties to derive from this work.

In practice, the author believes the user would be more likely to specify a chord progression than to have one generated randomly; evidence can be seen in virtually every paper on music composition, which generally chooses to use a set chord progression or drop the concept entirely. Given this observation, Steedman’s admission, and the desire to constrain the project’s scope to a reasonable size, the system will only accept a user-specified progression.

## 2.4 The psychology of music

One ever-present field of psychology is the investigation of music: rhythm, pitch, structure, and the various mental effects that they produce. This research can be invaluable to an algorithmic generator when shaping the output to sound genuine and interesting, rather than robotic and meandering.

### 2.4.1 Early studies

The earliest study of emotional reactions to music was conducted by Hevner (1936). Listeners of varying musical backgrounds were asked to check boxes alongside adjectives describing the music they were hearing; these adjectives were grouped as synonyms of eight base descriptors, and the results aggregated. Presenting the listeners with a large number of adjectives, and allowing them to choose as many as desired, reduces the cognitive load of choosing the most fitting word.

The study was one of the first to show that untrained listeners gain as much emotional value from music as trained musicians. With the development of modern genres such as noise, free jazz and atonal music, this seems to be a dubious conclusion at best. One example is the piano works of Art Tatum: they are often dismissed by novice pianists, but more advanced musicians have been intimidated by the technical difficulty of his embellishments

and the progressive nature of his harmonisations.

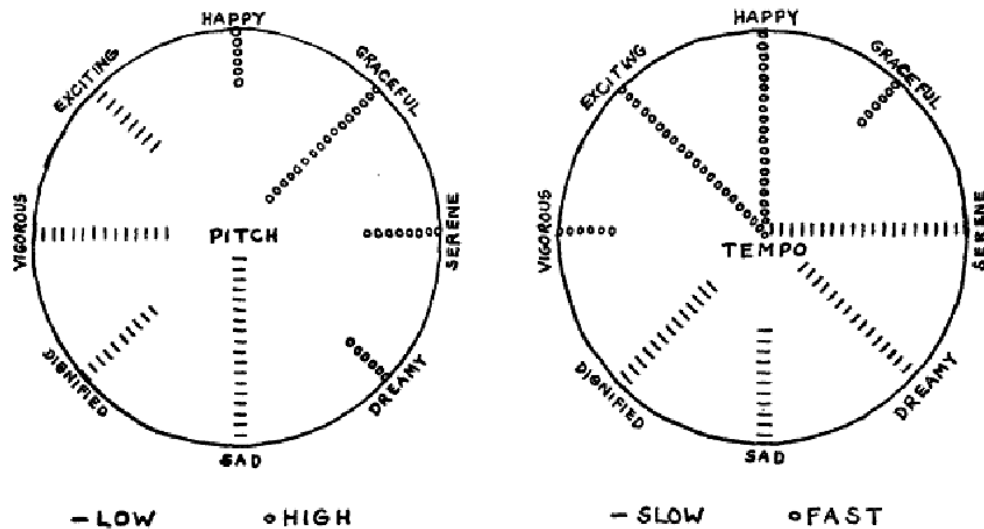


Figure 2.1: Listener response to changes of pitch and tempo (Hevner, 1937)

By design, the study only examines factors like contour, major vs. minor mode, firm vs. flowing rhythm, consonance etc. Each piece played is assigned a score in each of these categories and this weighting is used to determine which adjectives are linked to each musical factor. It is evident, then, that there are factors *not* covered by the study that are also influencing the listeners' adjective choices, making the study unreliable.

Nevertheless, it is a good base for research in the area of “music with intent”. A second study by Hevner (1937) corrects this flaw by having each piece played twice, identical save for one key difference in pitch (transposition) or tempo. Tempo was found to be of critical importance in carrying the expressiveness of music, with fast tempi inciting feelings of grace, vigour, happiness and excitement; while slower tempi produced effects of sadness, serenity and in rare cases, dignity.

Aside from tempo, the studies combined show modality to be the most important aspect of music, with major modes naturally leading to happier (or angrier) emotion. Other aspects have a less concrete effect, particularly with the small sample size of forty musicians that Hevner employed.

#### 2.4.2 Schenkerian analysis

In general, music theory focuses on aspects of pitch rather than rhythm or form. This can be seen in modern atonal theory and, more notably, Schenkerian analysis, a large focus of research into tonal structures. For example, Schenkerian theory regards *every* piece of tonal music to be an embellishment of the I-V chord progression, and that the direction

of a piece can be found by analysing long-range linear pitch motions embedded within. However, pitch is considered by many composers to be a “subordinate aspect of music”:

One can write interesting music in a purely percussive idiom. One cannot write interesting music without rhythm or form. Music without rhythm and form would be ametrical at all levels, with change occurring randomly at random times. It would be pointless, wandering, and ultimately boring.

— *Frederick W. Umminger*

There is an abundance of research that finds fault with Schenkerian theory and other pitch-focused cognitive models. Serialism is based on structures of pitch relation which experiments have repeatedly shown to be “cognitively opaque” (Millar, 1984; Lerdahl and Jackendoff, 1996); they are inconsequential to the music’s cognitive effect. The linear pitch motions of Schenkerian theory have been experimentally shown not to create a sense of implication (Schellenberg, 1997): given a partial linear pitch motion, the listener expects a repetition of the final tone, not a continuation of the motion.

## 2.5 The Implication-Realization model

Eugene Narmour was another opponent of the theory, releasing a criticism of Schenker’s work and introducing his alternative model (1977). Thirteen years later, he published the Implication-Realization model (1990, 1992), based on Meyer (1956)’s work on expectation and Gestalt psychology. Rather than focusing on tonality, this model shows how implications set up expectations for certain realizations to follow.

Implications can be used to develop music with intent that induces feeling. Meyer proposes that emotion is aroused “when an expectation — a tendency to respond . . . is temporarily or permanently blocked.” Expectations are developed “in connection with particular musical styles and of the modes of human perception, cognition, and response.” The outcome of this is that emotion and affect are heightened when a listener’s musical expectations are unfulfilled; this is an important finding, such that “almost all contemporary music theoretic analyses have adopted implicit or explicit ideas of expectation” Schmuckler (1989). Implications can also be used to measure melodic similarity (Grachten and Arcos, 2004; Grachten et al., 2005).

### 2.5.1 The model in brief

Narmour begins with two general hypotheses:

- $A + A \rightarrow A$
- $A + B \rightarrow C$

...where A, B and C are either intervals or pitches. These simple rules indicate that a) hearing two similar items yields an expectation of a repetition of that item, and b) hearing two different items yields an expectation of change. The model then gets more detailed; what follows is a summary of its core tenets.

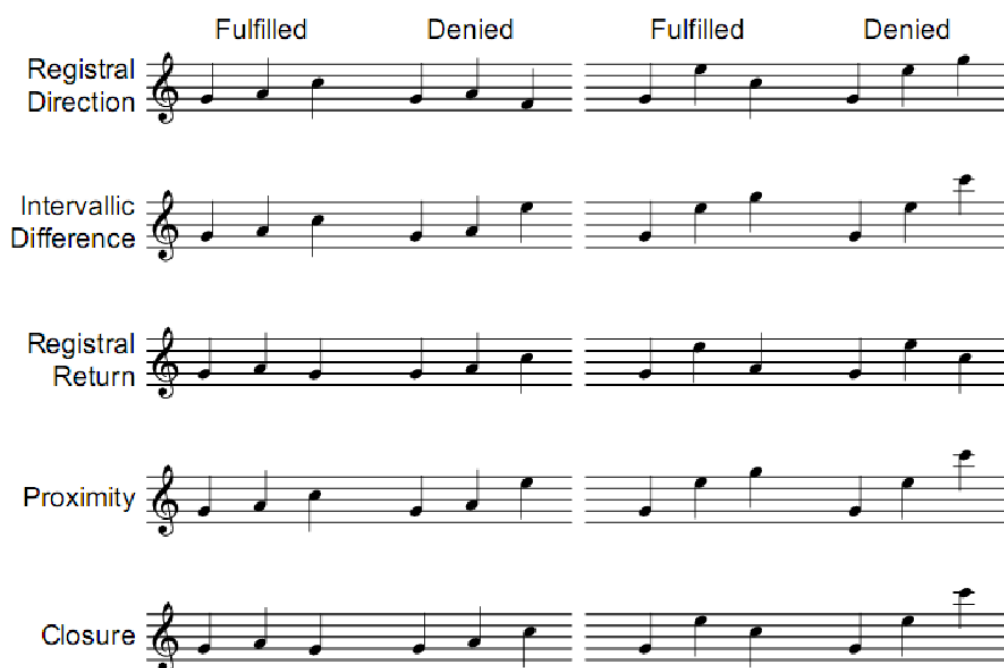


Figure 2.2: Examples of principle fulfilment and denial for small and large intervals (Thompson et al., 1997)

### 1. Registral direction

- Small intervals imply continuation of pitch direction.
- Large intervals imply a change of direction.

In the original model, “large” intervals are deemed to be seven semitones or larger.

### 2. Intervallic difference

- Small intervals imply similar-sized intervals.
  - If the registral direction changes, “small” is defined as the original interval  $\pm 2$  semitones.
  - If there is no change, it is  $\pm 3$  semitones.
- Large implicative intervals imply relatively smaller intervals.

### 3. Registral return

This is the melodic archetype **aba** or **aba'** where the last tone is very similar to the original pitch ( $\pm 2$  semitones), for example A–E–B $\flat$ . Listeners recognise and expect this archetype to occur.

### 4. Proximity

Small intervals have stronger implications and are themselves more implied than larger intervals, consistent with the cross-culture prevalence of small intervals in music (Dowling and Harwood, 1986).

### 5. Closure

Closures define how a listener segments a melody based on pitch direction and interval size. This occurs in two cases:

- Pitch contour direction is reversed.
- A large interval is followed by a smaller interval.

Stronger closures involve both cases occurring simultaneously. In addition, a general sense of closure can be found with any two successive tones where:

- the second tone is longer than the first;
- the second tone occurs on a stronger beat than the first;
- the second tone is more stable in the established key or mode than the first.

## 2.5.2 Justification

As Krumhansl (1995) points out, music-theoretic concepts are seldom subjected to empirical tests; theorists are rarely interested in making their concepts empirically testable. The I-R model, however, clearly states its principles allowing several researchers to perform studies to validate Narmour's theory (Cuddy and Lunney, 1995; Schellenberg, 1996; Thompson et al., 1997). Most support his findings – even across cultural boundaries and classes of musical experience – indicating an innate “genetic code” for musical expectancy:

The principles . . . are presumed to arise from general psychological processes. As a result, they would operate independently of the musical style of a melody and the musical experience of a listener . . . overall, the model was remarkably consistent in its ability to predict response patterns.

— Schellenberg, 1996

Krumhansl (1995) also verified the model for British folk, atonal and Chinese folk music. Conversely, two later cross-cultural studies (Krumhansl et al., 1999, 2000) found that musical cultures shaped the listeners' common principles of expectation in unique ways, so there is no definite consensus.

Schellenberg also shows the model to be over-specified; a revised, simplified version was equally successful at predicting responses among listeners and music styles. In particular, the principle of intervallic difference always implies a small interval, and so has correlation with the principle of proximity. The principle of closure has similar conceptual redundancy, and so the revised model omits two of these principles and modifies the others to reduce explanatory overlap.

I-R is particularly suited for algorithmic composition; its metrics can be easily codified as rules and constraints, and do not require complex analysis of the musical phrase. In his paper, Schellenberg grades each implication type with a score between 1 to 6, with appropriate scaling based on factors such as interval size. These scores could be useful in algorithmically determining the expectancy of a piece; the system could target a specific expectancy score, such as 75%, to ensure that the piece becomes neither too predictable nor too disjoint. It may be useful to implement both the original model and Schellenberg's revised model to see if there are any significant differences in the output.

## 2.6 Non-pitch aspects of music

### 2.6.1 Rhythm

Rhythm and meter are such important aspects of jazz music (swing, syncopation, odd time signatures) that ignoring them is not an option for any serious improvisation system.

Even if no specific rhythm is set, the mind tends to impose patterns on even random series of stimuli (Cohen, 1957) and songs with fixed durations between notes will have a weak rhythm automatically imposed upon them (Cooper and Meyer, 1960). While this is comforting, the problem of rhythm ultimately comes down to choosing which notes are accented and which are not. Cooper claims accented notes must be “similar” to other notes around it in order to create the accent effect, but that there is no correlation between accent and volume or duration. Often accents are played at fixed and stable points in a series of notes; unaccented beats are often slightly displaced when rubato is applied.

Tempo variations can be tricky to implement correctly: Desain and Honing (1993) warn that simply speeding up or slowing down the same sequence of notes will not work. For example, grace notes are more pronounced at slower speeds, while staccato notes would have their duration extended. These changes tend not to be linear but more related to the metrical and rhythmical structure of the piece (Clarke, 1988; Palmer, 1989). Other devices, such as broken chords, may or may not change speed to match the global tempo.

As far as jazz is concerned, swing rhythm is essential for most styles (except for very fast songs or ballads). In general, for each pair of notes the duration of the first is augmented while the second is diminished. The exact ratio of durations is often written as 2:1 or 3:1 but in practice depends on how “hard” the song is swung, and will be a subject of experimentation. Bass players often play “straight” (unswung) eighth notes a little ahead of the beat to keep things “moving forward” (Sabatella, 1995); soloists often play ahead,



or play behind the beat for a more relaxed feel. In general, jazz gives the composer many opportunities to shift notes around the beat for different effects, and the system could account for this.

### 2.6.2 Form

Almost all modern music analyses share one thing in common: they see music as hierarchical. To quote Schenker (1969): “both pitch and rhythm structures are represented in a series of levels, between which relationships of reduction and elaboration operate.” Lerdahl’s (2001) tonal-pitch-space model proposes that tension is a combination of melody dissonance and the position of each musical event in a tree structure; phrases closer to the root of the tree create less tension.

One interesting study by Gotlieb and Konecni (1985) split Bach’s *Goldberg Variations* into pieces and played them in a random order in addition to their original order. Listeners, some of whom were musically trained, did not prefer the original order – a preference for the original order appeared in only 1 of 15 scales. Another study by Cook (1987) examined the effect of a piece beginning and ending on the same key; music students most often preferred the version that ended on a different key, unless the piece was very short. This implies that the effect of *tonal closure* is restricted to fairly short durations, outside of which attempting to end on the same key as you began is a fruitless task.

## 2.7 Conclusion

To progress the field of algorithmic improvisation, factors such as musical tension, intent, expectation and melodic closure should be explicitly referred to and evaluated. Formalizations for these factors are already in place: Lerdahl’s tonal-pitch-space model, psychology research, and Narmour’s I-R model. Additionally, systems which only utilise only one generational approach do not seem to be very effective. If possible, multiple approaches should be combined.

### 2.7.1 Form

Form is an important subject in the field of algorithmic composition, but there is little information on how to generate or enforce a high-level musical structure. From studies such as those by Gotlieb and Konecni (1985) and Cook (1987), naïve attempts at determining high-level form have been unsuccessful, or even counter-productive. Improvisations are generally short and devoid of any particularly strong structure, although musicians will of course think ahead by a few notes or a few bars. Given this situation, and the inability of genetic algorithms to explicitly account for form, the system will place a low priority on giving a high-level form to the output.

### 2.7.2 Choice of algorithm

The most critical decision is the choice of algorithm. In general, chaotic functions, 1/f noise and stochastic methods are too simple for an attempt at generating music that sounds genuinely human. There are simply too many factors going into an improvisation to be sufficiently represented by a random function or a probability matrix. Their support for higher-level form is worse than that of genetic algorithms, and they contain no human knowledge whatsoever. From Biles et al. (1996) it can be seen that two musical objects can share the same statistical data, but have a drastically different effect on the listener, or vice versa; it depends entirely on the metrics chosen. Markov models and other related methods only compound this problem by discarding context (save for one or two previous notes) and averaging statistics across a large number of inputs.

Genetic algorithms (GAs) have been shown to be successful by Horner and Goldberg (1991); Papadopoulos and Wiggins (1998); Marques et al. (2000) and others, even with eight-octave search spaces and simple fitness evaluation. The fitness function itself allows musical phrases to be checked according to arbitrary criteria: rules, constraints, even a human opinion. Unfortunately, efficiency (see Biles, 1994) and subjectivity (Ralley, 1995) make using a subjective fitness function difficult; additionally, any musical composition knowledge would be stored in the user's mind rather than in the system itself, which makes it less useful for future development.

Thankfully objective functions are practical, and from an administrative perspective offer an open-ended goal: the function can be built up with more complexity until there is no time available. The disadvantage is the scope of possibilities when creating such a function, and the experimentation it requires. Whatever the fitness function, the general framework of GAs has been shown to resemble human creativity, or as Goldberg (2002) surmises, "Selection + Mutation = Improvement. Selection + Recombination [Crossover] = Innovation."

Markov chains and other methods are noted for their speed, and their ability to generate real-time output. GAs are rarely real-time, but by deciding not to develop a real-time system, new opportunities are made available: the search space can be made to cover a large range (three to five octaves) rather than decreasing it and possibly compromising output quality; the fitness function itself can be involved and computationally expensive; the genetic algorithm itself can be run for thousands of iterations in the hope of generating a better improvisation. In this light, real-time output does not seem useful enough to justify the concessions that would need to be made.

### 2.7.3 Other methods

GAs require a suitable initial population of candidates. Traditionally, random data is used; the population is shaped and improvised by the algorithm over thousands of iterations. But there is no harm in having the initial population be musically interesting to begin with. A simple example would be using random notes from a particular scale across a particular

subset of piano octaves. More interestingly,  $1/f$  noise or chaotic systems could be used as seeds.

## Chapter 3

# Requirements and Design

When people ask me how is it I was a musician, I facetiously say that I'm a firm believer in reincarnation and in a previous life I was Johann Sebastian Bach's guide dog.

— *George Shearing*

The algorithmic composition system, hereafter titled *JazzGen*, will use genetic algorithms to produce jazz improvisations. But the choice of algorithm is not the only consideration when building such a system.

### 3.1 Initial considerations

There are a number of importance choices to be made before design work can begin.

#### 3.1.1 Output form

Algorithmic composition systems are typically classified into one of three classes:

**Precomputed** The entire musical output is computed at once.

**Real time** The output is computed iteratively at a rate faster or equal to the rate at which the output will be played.

**Linear time** The output is computed iteratively but not necessarily at a rate fast enough to play the output simultaneously.

By splitting the musical output into blocks and running the genetic algorithm one block at a time, real-time output can be achieved (with a small latency). However, such algorithms

typically require a large amount of computation time due to both the complexity of the fitness function and the number of generations required to achieve acceptable output.

*JazzGen* will be a precomputed system: it will perform minor precomputations based on the supplied chord progression, generate the output one block at a time, then convert it to MIDI<sup>1</sup> and play it. It is anticipated that the system could be converted to linear time with the slight complication of developing a suitable buffered MIDI playback library.

Morris (2005) describes two further distinguishing classes of generator:

1. The program produces a different improvisation each time it is executed.
2. The program produces a deterministic piece depending on certain fixed parameters.

Genetic algorithms can trivially be made deterministic via the use of a fixed random seed. The choice comes down to whether the system will be used to compose a piece (Morris, the works of Cope, Xenakis and other algorithm composers) or as a composition *tool* that will continually produce new musical ideas. The inherent unpredictability of GAs (crossover, mutation, and initialisation) lends itself to the latter. Some parameters will be fixed by the user, which serves to constrain the possibility space a little, but ultimately the output produced should be different with each execution.

Additional bass and drum tracks will be composed as an accompaniment to the improvised piano solo. Since *JazzGen* is precomputed, the problem of having to generate several parts simultaneously is absolved, and they will instead be produced sequentially and combined into one MIDI stream. This offers the latter tracks a chance to be reactive to the solo in a more free form way; the bass can ‘cheat’ by playing solo notes in unison, and so on. While obviously not possible in a true improvisation context, anything that makes the output more aesthetically pleasing should be encouraged.

### 3.1.2 Chord progression

Rather than being generated with the solo, the chord progression will be dictated by the user. This allows improvisations over a particular jazz standard or the user’s own choice of progression, assisting in composition or comparison with real improvisations over a similar set of changes<sup>2</sup>. Chord progression generation is a complex subject in itself, and is deemed to be outside the scope of this project.

The system will parse a user-supplied chord progression typed in a human-friendly notation, deduce chord inversions and scales to use, and store this data in a manner that can be easily accessed by the fitness functions. A “repeat” option may be offered, so the user can conveniently input a set of chords and have this set repeated a number of times to form the full progression.

---

<sup>1</sup>Musical Instrument Digital Interface: a protocol for sending musical events to other devices, in this case a software synthesizer.

<sup>2</sup>Jazz term for the chord progression

The initial system should support a reasonable number of chords and scales (exact details are in the requirements): “reasonable” is defined via the author’s musical experience or testing needs. Each chord should map to at least one scale suitable for playing over that chord, so the tonality of the solo can be assessed.

### 3.1.3 Note representation

The MIDI specification defines notes as pitches numbered from 0 to 127, where 0 represents C-1 (an octave below C0) and 127 represents G9. *JazzGen* represents pitches in the exact same way – not just for simplicity, but also to allow flexibility when generating music. While there are only 88 notes on a typical piano, the system may be used for other purposes in the future, where the extreme notes may become useful; other tuning systems, such as microtonal systems, would utilise these pitch numbers or perhaps the values 0–254.

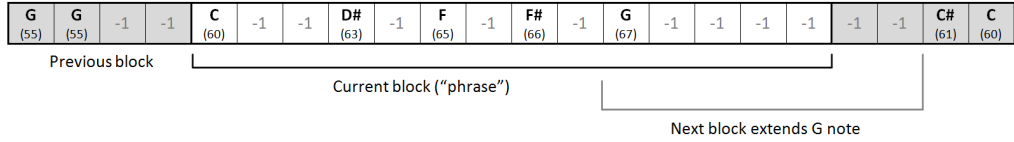


Figure 3.1: Pitch array representation

One pitch value, -1, is reserved to mean “Rest”. A musical phrase is stored as an array of such pitches: either notes from the value of 0-127 or the Rest value (Figure 3.1). This encoding ignores velocity and note duration entirely. The rationale for this is twofold: first, it reduces the search space, and therefore the complexity of the fitness function, mutation and crossover routines. Secondly, it relieves the GA from dealing directly with notes that begin or end in a different block to the one currently being evolved, and the mechanics of splitting them up. Mutations and crossovers will simply shuffle rests around, where they will randomly extend or contract the durations of other notes. Rhythm evaluation will work over multiple concatenated blocks, but the first and last group of rests will be ignored as they are of an indeterminate length.

The note resolution is important: the initial configuration will be that of 32 “notes” to a two-bar phrase, equating to a maximum resolution of 16<sup>th</sup> notes (semiquavers). This is thought to be enough to handle most tempi of music. The genetic algorithm will operate over these two-bar “phrases” individually: each phrase can have an arbitrary sequence of chords in the background (typically two or three). The chords are defined and added separately to the piece, as the solo itself is monophonic; handling multiple notes played at once adds orders of magnitude to the search space, and will not be considered.

Some systems choose to limit their search space by encoding only notes within the current scale. This approach is difficult to manage, especially when multiple chords are present within one GA output. Additionally, it drastically reduces musical possibilities: while the fitness function will encourage notes from the current scale, this vote can be overcome by

high fitness scores in other areas, such as repetition, a smooth melodic contour, or ideal expectancy. There are several reasons why this is important:

- Encoding only notes from the current scale would eliminate unusual and novel musical outputs resulting from dissonance, chromatic passing tones, etc.
- There are often multiple reasonable scales for a given chord, so the choice of which scale to use is ambiguous and should not be heavily enforced.
- Often the previous blocks of output (which the fitness function is able to evaluate) is a better arbiter of which notes to use than an arbitrary scale choice.

Thus, the absolute values 0–127 are used, and all chromatic notes are included.

### Post-pass expression

In this form, the notes would be played staccato at a fixed velocity. To make the piece sound less mechanical, a post-page stage will shape the output in a number of ways:

- Extend notes to cover some of the rest period that follows them.
- Move note start offsets a small, random distance to reduce perception of a “machine pianist”.
- Apply velocity curves around important parts of the phrase.
- Apply *tempo rubato*<sup>3</sup> to these sections to add emotion (Todd, 1989).

The problem is deciding “what is important” when applying these techniques, and this will be addressed during the development stage.

### 3.1.4 Accompaniment

The improvisation alone may be functionally complete, but it will not be attractive to the ears without harmonic and rhythmic context. Thus, a traditional “walking bass line” and percussion should accompany the solo. This backing will be simplistic; developing a realistic and reactive accompaniment is not the primary focus of this project.

The walking bass will echo the root note whenever a chord is played, then proceed to randomly walk up and down the notes of the current chord. This avoids harmonic conflict with the solo and the chord itself. The pattern will be 8<sup>th</sup> notes, played an octave lower than the piano chords.

---

<sup>3</sup>Slightly slowing down or speeding up the solo with respect to the accompaniment. Pianists often employ this technique around important parts of a piece.

The percussion will at first be a simple static drum pattern. If there is time, it will be extended to mimic the rhythm and dynamics of the solo.

## 3.2 The genetic algorithm

The genetic algorithm will be composed of several stages: initialisation, fitness evaluation, selection, crossover and mutation. It will operate over “blocks” or “phrases” of input, roughly two bars long and composed of 32 sixteenth notes in the format outlined on page 23.

### 3.2.1 Initialisation

The algorithm requires an initial population of candidates. These candidates can be completely random, as they will soon be changed by mutation, crossover and breeding (discarding the least fit individuals). However, using candidates that are musically interesting and satisfy *some* of the constraints in the fitness function should lead to faster evolution to an acceptable level of fitness. Several ideas have been considered:

- Using notes from the correct scale and key, instead of purely random notes.
- Limiting notes to a small set of octaves (e.g. the middle four of an 88-key piano).
- Using a more “interesting” source, such as 1/f noise or chaotic functions.

Due to this, the initialisation module should be abstracted out from the algorithm, so it can be replaced with a new one at runtime via the user interface. It is possible that some of the generation methods may need options, such as maximum or minimum octave, input data for the chaotic map, etc. Each module may have a facility to present options to the user.

Grachten (2001) mentions that music with form should revolve around central motifs. One option is to generate a motif and use it as a basis for composition. When improvising, a more natural route is to use previously-played music as the motif. To realise this “history seeding”, some initialisers will be copies of blocks previously generated in the same solo, with more recent blocks preferred so that the repetition is more evident. If necessary, the repeating block may be transposed to a new key to reflect the current chord. The probability of using a previous block as a seed, and the number of previous blocks to consider, should be modifiable by the user.

### 3.2.2 Fitness function

The fitness function is clearly the most important part of the algorithm. It provides direction to the algorithm as it traverses the musical search space. If the mutation, initialisation



or crossover methods are flawed, the search will merely take more iterations before finding a high-fitness individual. If the fitness function is flawed, the definition of “fit” is wrong and the results are meaningless.

Grachten (2001) introduces three major constraints that should be satisfied by a jazz improvisation:

**tonality** the improvisation must be tonal to the key of the music, and thus be predominantly consonant;

**continuity** the melodic contour of the improvisation must be mostly smooth; large intervals are sparingly used and registral direction is not too frequently reversed;

**structure** the improvisation should not be merely be a sequence of non-related notes; in some way, interrelated groups of notes should be identifiable.

In addition, *JazzGen* aims to model expectancy (Narmour, 1990, 1992), which will aid the evaluation of all three tenets. While the first two are simple to qualify, the third is not as clear-cut; *JazzGen* will attempt to measure structure by evaluating rhythmic consistency, note placement, rhythm repetition, and common motifs such as notes travelling up or down the scale. It is also possible that common “licks”, or musical patterns, could be encouraged or placed directly into the improvisation. Finally, there are administrative checks, such as notes falling too far outside the “reasonable” piano range.

Initial development featured a single fitness function with arbitrary, often ill-considered fitness scores. The musical results were surprisingly acceptable, but it quickly became clear that a single function would grow too large and become unwieldy and inconsistent. To combat this, the fitness function will instead be formed from several *fitness modules*, each responsible for a different aspect of evaluation. The overall fitness score  $\bar{x}$  can then be calculated as a *weighted summation*:

$$\bar{x} = \left\lfloor \sum_{i=1}^n w_i x_i + 0.5 \right\rfloor$$

where for each fitness module  $i$  there exists a score  $x_i \in [0, 1]$  and a weighting factor  $w_i \geq 0$ . The user interface should allow the user to adjust fitness module weights or disable the module entirely, which is equivalent to setting  $w_i = 0$ . We then add 0.5 to the score and floor it to yield a rounded integer result suitable for comparison against other scores.

The fitness evaluation will be performed over the current block *plus* the previous  $n$  bars of music, where  $n = 1$  initially, but can be changed by the user. This provides a context to ensure the new block smoothly connects with the old one from both rhythmic and melodic viewpoints. It is possible that  $n$  could be increased to cover the entire output for evaluations of higher-level form, etc. – fitness modules could decide how many blocks to evaluate on an individual basis – but this would be an extension to the project if time allows.

It is important that fitness scores have a linear fitness  $\rightarrow$  value distribution. For example, modules should not be providing high fitness scores to 90% of the population when others are providing those scores to only 10%. Otherwise, modules may have too little or too large an influence on the overall score.

### 3.2.3 Selection

After evaluating the fitness of each candidate, the fittest candidates must be *selected* for breeding (crossover and mutation). The simple approach is to pick the  $X$  fittest individuals of the population, but this can cause premature convergence on poor solutions; traditionally, population diversity is encouraged by occasionally choosing less fit individuals. This also compensates for deficiencies in the fitness function, which is crucial in creative domains.

**Tournament selection** is a popular selection algorithm. Given a tournament size  $k$ , choose  $k$  individuals from the population and select the one with the highest fitness. Repeat until no more selections are needed. If  $k = 1$ , this is equivalent to random selection, but as  $k$  increases, the pressure to choose high-fitness individuals rises. Miller and Goldberg (1996) write that “[tournament selection] is simple to code, easy to implement ... robust in the presence of noise, and has adjustable selection pressure.” Given these qualities, tournament selection will be the only implemented selection method; the tournament size will be a user-configurable option.

### 3.2.4 Reproduction

Reproduction of fit individuals typically involves *crossover* (merging two parents into one child) and *mutation* (modifying the child slightly). The purposes, as outlined by Goldberg (2002), are to “innovate” by combining musical phrases, and to “improve” (at least some of the time) by making minor modifications to the result; these two disparate aims are simultaneously achieved by reproduction. The overall aim is to increase the average fitness of the population with each subsequent generation.

#### Crossover

For simplicity, crossover will occur over byte arrays of notes as defined above: this causes both note duration, as well as position, to change when crossover occurs. Several possible methods are available:

**One-point crossover** Beginning of child is taken from first parent, and the rest from the second (the exact proportions are random).

**Two-point crossover** The beginning and end of the child are taken from the first parent, and the middle from the second.

**Uniform** Notes are randomly copied from the first or second parent, forming an average of  $L/2$  crossover points (where  $L$  = length of child)

**Random mask** Uses a mask to determine whether each byte comes from the first or second parent. Each mask bit has a set probability  $p$  of being a negation of the previous (rather than equal): if  $p = \frac{1}{2}$ , it is equivalent to uniform crossover.  $p > \frac{1}{2}$  creates more crossover points, and  $p < \frac{1}{2}$  creates less.

**Average** Pitches from each parent are merged together; if one parent has a note and the other has a rest, the rest is chosen with probability  $p$ .

Current research indicates uniform crossover is best for large search spaces (Sywerda, 1989; Spears and De Jong, 1991); it creates the most diversification. Unfortunately, it also corrupts perfectly good musical phrases in a much more drastic way than standard one- or two-point crossover, which more closely resembles the human process of combining phrases. This can be partially mitigated with the use of *elitism*, where a set number of fit individuals are copied over to the new generation completely untouched. Nonetheless, it is not possible to settle on just one crossover method; several should be implemented, with an option to select which one is used. This will allow direct comparison of methods and easy examination of their effect on the output.

Note that some methods, such as one-point crossover, create two children. The uniform crossover only creates one. For simplicity, all methods are assumed to produce only one child, and new parents will be continually selected (possibly more than once) until the necessary number of children have been produced.

## Mutation

Mutation is employed for the same reasons that weaker individuals are allowed into the next generation; without introducing noise, the algorithm may quickly converge on a local optimum and slow down or stop evolving entirely. Mutation and low-fitness candidates ensure that there are always more possibilities and directions for the search. Traditionally, mutation is performed by randomly changing bits in the candidate value. Since *JazzGen*'s value is a musical phrase, it makes more sense to randomly adjust pitches in the phrase, and introduce or remove rests. The conventional wisdom is that mutation should be “dumb” and serve only to blindly expand the search space.

Biles (1994) diverged from this viewpoint by introducing “musically meaningful mutation”: mutations existed to reverse and rotate notes, invert pitch values, transpose or sort notes ascending/descending. The aim is to create “not just new, but better” offspring. It is easy to see that sorting notes could give rise to ascending or descending melodic lines that would rarely be generated by random mutation alone. *JazzGen* should expand on this by offering a large range of operations, allowing the user to select which ones should be used; each candidate will be mutated with a random operation from the set of those available. The requirements detail the full list of mutations.

### 3.2.5 Termination

The most common termination conditions are:

- Maximum fitness exceeds minimum bound
- Fixed number of generations reached
- Time limit reached
- Maximum fitness is reaching or has reached a plateau
- Manual inspection
- Combinations of the above

The simplest to implement, and least prone to complication, is a fixed number of generations. *JazzGen* will define a maximum number of generations, which can then be changed according to the user's need for quality vs. the user's patience/available time.

After the algorithm has been finalised, it would be useful to examine fitness scores and see if they plateau regularly (without cycling) or reach a certain minimum value assuredly enough to warrant additional termination conditions. Unfortunately, the fitness value is dependent on the number and the nature of the fitness modules enabled, making this type of analysis difficult.

## 3.3 User interface

Aside from building an algorithmic improvisation system, it is important to be able to experiment with settings and turn on or off various features of the algorithm. This being so, a large amount of effort will be spent constructing a user interface that facilitates quick and easy adjustment of GA settings, input of the chord progression, export to Lilypond, MIDI etc. Ideally these settings will be saved when the application is closed, and restored on start-up. It may also be convenient to allow saving and loading of settings via arbitrary files, enabling the management of multiple generation "profiles".

The chord progression, as mentioned earlier, should be input via a human-friendly format such as *Dm7 / C#7*, and parsed into the appropriate data structure. If there are errors in the progression, such as non-existent chords or note names, they should be detected and displayed to the user. It would be useful to have the option of repeating the chord progression a set number of times to lengthen the improvisation. The tempo and meter of the piece may also be modifiable.

The design of the user interface is not set in stone but it should contain the following main sections, most likely arranged in tabs:

**Fitness functions** A list of available fitness modules will be displayed. The user will be able to enable or disable modules, set their relative importance (weighting), and change module-specific options.

**Genetic algorithm** Contains general options, such as population size, maximum iterations, chance of mutation/crossover, and tournament size. Also determines how many previous blocks to use in fitness evaluation/initialisation.

**Mutation** The user should be able to toggle mutation methods on or off, and possibly modify method-specific options.

**Initialisation** The user should be able to select an initialisation method and modify method-specific options.

**Crossover** The user should be able to select a crossover method, and modify method-specific options.

The top of the interface will contain the inputs for the chord progression, repeat count, tempo and meter. The bottom will contain buttons that trigger various actions: generate the output, play it via a suitable MIDI device, save the piece in SMF format<sup>4</sup>, or convert to Lilypond notation.

A second set of buttons deals with administrative tasks: load or save options to a file (if supported), and a *Close* button to exit the application.

## 3.4 Development environment

To build *JazzGen* in a reasonable time, it is critical that a suitable music generation infrastructure be used to avoid unnecessary work in fields such as sound synthesis, MIDI output or user interface library programming. This work is instead performed by several tools:

### 3.4.1 The Java programming language

Linear time or precomputed systems do not place any strong requirements on generation speed. Nevertheless, if the output takes five minutes to produce, it is going to be very difficult to experiment with GA parameters and compare results. Initial development of *JazzGen* in the Python language proved to be too slow, taking several minutes to execute a few hundred iterations. Despite Java's use of intermediate bytecode, it is a much less dynamic language and many virtual machines employ Just-In-Time compilation to boost the speed further. With the use of primitives and arrays rather than boxed objects or linked lists, *JazzGen* is able to run through thousands of iterations in less than a minute.

---

<sup>4</sup>Standard Midi File, the format used for .mid files.

Java provides automatic memory management and garbage collection, higher-level data structures, and most importantly the Swing user interface library, which the author is familiar with. Despite Java's verbosity, the author was able to achieve rapid iterative development and, in particular, build up a full user interface for the system quite quickly.

Higher-level interpreted languages would allow even faster development, as they forego compilation and add the ability to modify code and fully inspect data structures at run-time. Unfortunately, these comforts result in extremely poor performance for computation-heavy applications.

### 3.4.2 jMusic

Java also offers a very capable music manipulation library known as jMusic. jMusic allows the user to build up *parts* from sequences of notes and *scores* from parts, then convert those scores to SMF (Standard MIDI File) format, play the file via the cross-platform JavaSound library, or even write the score out in CPN (common-practice notation). In particular, it automatically manages the combination of several simultaneous parts (drum, bass, piano) and provides convenience methods and constants for note names, durations and velocities.

### 3.4.3 GNU Lilypond

Unfortunately, jMusic's CPN output is not extensive enough for full scores and chords at this time. Instead, the GNU Lilypond layout engine will be used. It is a fairly simple matter to convert a musical score to Lilypond notation, which can then be converted into a professional-looking PDF (Portable Document Format) file.

### 3.4.4 MIDI

The Musical Instrument Digital Interface, or MIDI, is an industry-standard protocol that expresses music as a series of 'event messages', controlling each note's pitch, tempo, vibrato and other settings. While widespread, MIDI is only capable of defining the placing and characteristics of musical notes; the events must be passed to a synthesizer, sampler or other application to generate the final sound. Writing an application that outputs MIDI is relatively easy, as almost every common programming language has a suitable library. MIDI data can be written to a file, or more commonly sent directly to another device (for example, a synthesizer connected via a MIDI port, or a virtual instrument running on the same computer). This allows the user to select suitable sounds for each musical 'part', but also to change these sounds at any time.

Unfortunately, MIDI has remained relatively unchanged since its introduction in 1983, and the protocol is somewhat limited; the 12-pitch per octave equal temperament tuning system is assumed, and the user is unable to have full control over the timbral qualities of the sound produced. Regardless of this, MIDI seems sufficiently capable of representing the sort of

jazz music a program is likely to generate. MIDI can also be stored to file and played later via any capable device; the jMusic library can output MIDI in both forms.

## 3.5 Requirements

The requirements define and constrain the basic scope of the project, and reiterate the design considerations. Requirement priority is indicated by the use of the words *must*, *should* and *may*.

### 3.5.1 Composition

1. The system must accept a user-supplied chord progression and generate a jazz improvisation that plays over this progression.
2. The improvisation should satisfy the three major constraints according to Grachten (2001): tonality, continuity, structure.
3. A post-pass stage will apply dynamics, note length extension and other humanising factors.
4. The quality of the improvisation should be made as high as reasonably possible in the time available.
5. This improvisation should be accompanied with a walking bass line and/or percussion.
6. The system should support multiple output formats.
  - (a) The system should be able to play the output over a suitable MIDI device in real-time, after composition is complete.
  - (b) The system should be able to save the output to a Standard Midi Format (SMF) file of the user's choosing.
  - (c) The system may be able to export the output to Lilypond score notation format suitable for conversion to PDF.
7. The generation should complete in a reasonable amount of time: fast enough that iterative development and experimentation is possible.

### 3.5.2 Genetic algorithm

8. The genetic algorithm must be initialised with a seed population.
  - (a) The initial method must be “random”: purely random notes with random durations.
  - (b) Previous phrases in the output should also be used as seeds. These phrases may be transposed to the correct key according to the currently playing chord.
  - (c) Other sources of music generation, such as 1/f distributions, normal distributions, or chaotic systems, may be used.



- (d) There may be multiple methods, of which the user can select one at runtime.
  - (e) Methods may be able to present module-specific options to the user.
9. The fitness of an algorithm will be determined via a weighted average of several fitness “modules”, each evaluating a different aspect of the music.
- (a) Each module must have an “enabled” attribute specifying whether it will factor into the fitness evaluation. Disable modules have an effective weight of zero.
  - (b) Each module must have an “importance” value specifying its weight in the average. The default is 100, with reasonable values spread from 0–200.
  - (c) Each module must accept a musical phrase *of arbitrary length* and return a fitness value  $\in [0, 1]$  in floating point format.
    - i. Modules may only evaluate the last  $n$  units of the input phrase when evaluating fitness, as long as this is done consistently over iterations.
    - ii. Modules should aim to have a linear fitness  $\rightarrow$  value distribution.
  - (d) The final fitness value must lie between 0 and  $2^{31} - 1$ .
10. Fitness modules should exist to evaluate:
- (a) Whether notes are in the correct key or scale for the current chord
  - (b) Melodic expectancy
  - (c) Rhythmic consistency (closeness of adjacent durations, notes on certain beat positions)
  - (d) Melodic contour (closeness of adjacent notes)
  - (e) Pitch range of notes (too high, or too low)
  - (f) Note repetition within a phrase

Other modules may be created and evaluated later in the development process.

11. For each chord, there should exist a facility to list one or more reasonable scale(s) to play over the chord, and to specify whether or not a given pitch is in this scale.
- (a) Dominant seventh, major seventh, minor, flat five, augmented, diminished and major chords should be supported, with a variety of common adjustments such as  $\sharp 11$  or  $b9$ .
  - (b) Arbitrary chord extensions, augmented or diminished notes may be supported.
  - (c) For each chord, at least one appropriate scale must be supported. For example,  $C7b9$  can be improvised with the H-W diminished scale.
12. Selection of the population should be performed via tournament selection with a configurable tournament size. Other methods may be implemented.
13. Crossover of parents should occur via single-point crossover. Double-point, uniform or other methods may also be implemented.

14. Mutation of children should occur via the application of a random mutation method.
  - (a) The following methods must be implemented: change of one random note's pitch; transposition of entire phrase by a random amount.
  - (b) The following methods should be implemented: sorting pitches ascending/descending; adding or removing rests from phrase.
  - (c) Other methods may be implemented, such as reversing or rotating pitches/the entire phrase.
15. The genetic algorithm must be executed until a user-specified number of iterations has elapsed. The fittest member of the population must then be added to the output.

### 3.5.3 Post-pass stage

16. The post-pass stage should be applied to the entire output after it has been generated.
17. All notes should be offset by a small, random amount.
18. All notes should be given a normally distributed velocity with a very small standard deviation.
19. Large gaps between notes, if present, should be filled by extending the duration of the earliest note. More complicated rules for extending durations may be implemented.
20. Velocity and tempo (rubato) curves may be applied to specific parts of the output.

### 3.5.4 Accompaniment

21. The walking bass line should be constructed via a random walk algorithm.
  - (a) The bass should echo the root note of a chord whenever one is played.
  - (b) At all other times, the bass should walk over notes in the current chord, only an octave lower, playing 8<sup>th</sup> notes.
22. The percussion should be a static drum loop. It may be reactive to rhythmic or dynamic changes in the solo.
23. Chords should be played with a suitable inversion and octave in order to minimise the distance between the notes of adjacent chords.

### 3.5.5 User interface and interaction

24. The system must be manipulated via a graphical user interface.

25. Input must be provided via the user interface.
  - (a) The chord progression should be input via a human-friendly format and parsed into the appropriate data structure.
  - (b) Parsing errors in the chord progression should be detected and displayed.
  - (c) The interface may provide an option to repeat the given chord progression a specified number of times.
  - (d) The interface may provide a facility for modifying the tempo and meter of the output.
26. The following output commands should be accessible:
  - (a) Generate the improvisation.
  - (b) Play the improvisation over a suitable MIDI device.
  - (c) Save the piece in SMF format.
  - (d) Convert the piece to Lilypond notation.
27. The user will also be able to modify genetic algorithm parameters before generation.

**General options** The general parameters such as population size, maximum iterations, chance of mutation etc. must be editable by the user.

**Fitness function** The fitness function is composed of several *modules*.
  - (a) The user must be able to enable or disable modules.
  - (b) The user should be able to adjust the relative importance of each module in determining overall fitness.
  - (c) The interface should support options specific to each module.

**Mutation** The user should be able to enable or disable mutation methods. The user may be able to modify settings specific to each method.

**Initialisation** The user should be able to select from several initialisation methods, and may be able to set method-specific options.

**Crossover** The user should be able to select from several crossover methods.

**Selection** Tournament selection will be the default selection method, with a modifiable pool size. Other methods may be available as options.
28. All user-specified options should be saved to a file when the program is closed, and automatically restored on startup.
29. There may be a facility to save and load options via arbitrary files.
30. All components should have sensible default values.
31. No interface is required for the accompaniment due to its simplicity and relative unimportance.

## Chapter 4

# Development

I'll play it first and tell you what it is later.

— *Miles Davis*

After designing *JazzGen*, work immediately began on developing the system in Python. While development was swift and enjoyable, the speed of the fitness function was simply too slow – even without a full fitness function, a few hundred iterations took minutes. There are several reasons for this: one is Python's interpreted nature, which prevents the rapid loops present in genetic algorithms from being optimised. The second is Python's *dynamic* nature, which means every single function call and variable is looked up each time it is used – particularly bad for chained lookups such as `gen.chord.bestScale.note[0]`. By switching to Java, and judiciously using bytecode arrays, primitive types and fast array copy routines, the speed was greatly improved.

The final system consists of 50 source files and over 4,000 lines of code, of which around 20% are comments. This chapter will go into some detail about the choices made when developing the core components of the system, and reveal more detailed information about the fitness modules, mutation methods and other areas that were not set in stone at the design stage.

### 4.1 Music generation

This description describes the generation engine behind *JazzGen*, in particular the genetic algorithm.

#### 4.1.1 The `JazzGen` class

The `JazzGen` class is a singleton, invoked whenever music needs to be generated. A `jMusic` score is constructed with the appropriate parts, and the genetic algorithm is invoked to

produce the piano solo. This class also interfaces with jMusic to play MIDIs or write the piece to file, and acts as a bridge between the user interface and the genetic algorithm and jMusic itself.

Due to their simplicity, the bass and drum tracks are also created within this class. The bass track is a simple random walk; it gets the current chord, begins at the root and walks up or down, one note at a time, changing direction around 50% of the time. The walk “bounces” off the start or end of the chord, and is thus limited to only four or five notes. The drum loop, and chords, are almost completely static although the snare drum has a random velocity from 0 to 60, to make it sound slightly more natural.

### 4.1.2 GeneticGenerator

The real work is performed by the **GeneticGenerator** class, which is a singleton; its **run** method accepts an empty jMusic **Phrase** object<sup>1</sup>, which will contain a final sequence of notes when the function terminates. We generate the music two “blocks” at a time, where each block represents two bars and has a length *phraseLen* = 32. This equates to a maximum resolution of 16<sup>th</sup> notes.

The generation is designed to mimic the original algorithm: initialise then evaluate fitness, select, crossover, mutate to form new population, replace old population, repeat. The algorithm terminates after *maxGenerations* iterations, which defaults to 1000. There are a few extra considerations:

- Each round, a number *elitismSel* of the fittest members are copied to the next generation untouched; this prevents mutation and crossover from corroding a particularly good specimen. Their fitness values are cached so they will not be recalculated unnecessarily.
- To do this (and other things), the population must be ranked by fitness. There exists an array of candidates **population** and an array of fitness values **fitness** for these candidates. To produce an array of candidate indices, ordered by fitness:
  1. Produce an array **ordering** = [0, 1, 2, ..., (**fitness.length** - 1)].
  2. Perform a *quick sort* on a copy of **fitness**, duplicating all value swaps on **ordering**.
  3. Thus **ordering**’s indices are now sorted in the same way as the fitness values; the first index corresponds to the lowest score, and so on. Return **ordering**.

The reader is encouraged to send in a better solution if one rapidly comes to mind.

- Each child is mutated with probability *mutationProb* = 0.5.

---

<sup>1</sup>The **Phrase** object is a list of notes, with associated durations and velocities.

- When building up the new population, the old one is untouched; the same parents can be selected multiple times, but are subjected to most likely unique crossover and mutation stages.

Selection is (as mentioned earlier) performed via tournament selection: *tourSize* candidates are picked, and the best is selected; this is repeated  $2 \times \text{populationSize}$  times to build the new population. After all iterations are complete, the most fit candidate is converted into jMusic format and added to the **Phrase** object. The history buffer, used for initialisation and fitness evaluation, is cycled to include the new output.

### 4.1.3 Initialisation

The GA needs an initial population to operate on. If it is not the first block, and given *repeatPhraseProb* probability, a previously-output block is used as an initialisation seed. The block offset to use is defined as

$$\text{offset} = m - \lfloor \text{rand}()^{\text{skew}} m \rfloor - 1$$

where the offset is 0 for most recent, 1 for second most recent etc;  $m$  is the maximum available history (e.g.  $m = 1$  at second block,  $m = 2$  at third block); *rand()* returns a random value between 0.0 and 1.0 and *skew* is a skew factor, generally equal to  $\frac{1}{2}$ . Setting it lower creates more pressure to pick recent bars; setting it closer to 1 tends to a uniformly random distribution.

With probability  $\frac{1}{2}$ , the previously-output phrase is transposed to fit the currently playing chord. For example, a phrase that originally played over C7, moved to D7, would be transposed up two semitones (a transposition of seven semitones or higher is shifted down instead). Where there are multiple chords per block, the transposition shift is calculated for each note individually.

### Initialisation methods

The majority of initial seeds are not reused, but generated anew. This process is encapsulated by the **InitMethod** class, which acts as a function pointer; the user can replace the GA's **InitMethod** with a new one at any time. Each implements a **generate** method which outputs an array of notes. Finally, the **InitMethodFactory** contains the list of initialisation methods, their names, and can create a new **InitMethod** object of the given name on demand. Each of the basic methods are designed to generate notes in the range C3–C6 and choose rests approximately half of the time; completely random pitches and durations would take too long to resolve to something musically useful. The basic methods are:

**InitRandomScale** Pick a random octave and scale note index, and generate notes in the current scale only.

**InitRandomChromatic** Pick *any* note in the central octaves.

**InitOneOverF** Implements the 1/f distribution algorithm by R. F. Voss and featured in (Dodge and Jerse, 1985). Generates notes in the current scale and predominantly in the central octaves.

**InitChaotic** uses the *Chirikov standard map* to generate pitches and durations. This map is defined as:

$$\begin{aligned} p_{n+1} &= p_n + K \sin \theta_n \\ \theta_{n+1} &= \theta_n + p_{n+1} \end{aligned}$$

where  $p_n$  and  $\theta_n$  are taken modulo  $2\pi$ . The constant  $K$  influences the degree of chaos. The range of the first output is between 0 and  $2\pi$ ; this range is mapped to an offset of -3 to 3 (in scale steps) from the previous pitch to deduce the new one, with the first note being the root. The second output has the same range and is mapped to specific durations according to a simple weighted distribution (see `InitChaotic.java`). This continues until the entire phrase has been generated.

The values of  $p$  and  $\theta$  are initialised to 2 (as in Morris, 2005). The value of  $K$  is set to a random value between 0.5 and 2.5, which makes each seed unique.

#### 4.1.4 Mutation

Mutation is performed via the `MutationFunction` class, which accepts a phrase and modifies it in-place. The function consists of several different *mutations*, described below. Each mutation can be enabled or disabled by the user individually; one is selected at random and applied to the child.

Below are the mutation functions implemented, where the note change probability  $p = \frac{1}{2}$ :

**Sort pitches ascending** Sort the *pitches* of the phrase from lowest to highest; leave rests (and rhythm) intact.

**Sort pitches descending** Sort the pitches from highest to lowest.

**Reverse pitches** Reverse the order of the pitches, leaving rhythm intact.

**Reverse phrase** Reverse the entire array, including rests.

**Rotate pitches** Rotate the pitches, leaving rhythm intact. Rotation is between 1 and  $numPitches - 2$  to the right (which also covers all left rotations).

**Rotate phrase** Rotate the entire array between 1 and  $phraseLen - 2$  steps to the right.

**Transpose** Add a random number of semitones, between -3 and 3, to *all* pitches.

**Octave transpose** Transpose the phrase up or down a single octave.

**Restify** Replace each note with a rest with probability  $p$ .

**Derestify** Replace each rest with a note, randomly selected between the lowest and highest note in the current phrase, with probability  $p$ .

**Uniform pitch change** Add between -5 and 5 semitones to each pitch in the phrase with probability  $p$ . Each shift amount has an equal chance of being selected.

**Normally distributed pitch change** Shift each pitch by  $k$  semitones with probability  $p$ ;  $k$  is selected from a discrete normal distribution with mean 0 and standard deviation 3.

General-purpose routines were created to rotate and reverse arrays, as Java seems to be lacking them. Operations over pitches were performed by splitting the phrase into a pitch array and a duration array using the `PhraseInfo` object (described in section 4.2.1, page 42), then re-assembling them. To avoid problems, the sort, rotate and reverse routines check that the phrase contains more than one note, otherwise nothing happens.

#### 4.1.5 Crossover

Crossover methods are implemented as described in section 3.2.4 (page 27). Each method is a subclass of the abstract `CrossoverMethod` class which allows the method to be changed at runtime. The `CrossoverMethodFactory` lists all method classes and can construct them if given a name or index. The default method is set to the uniform crossover.

## 4.2 Fitness function

Evaluation takes place over the entire population with each iteration, and once more after all iterations are complete to select the highest fitness candidate. Of course, any candidates who have been left unchanged since the last iteration keep their existing fitness score.

To begin with, a “fitness subject” is constructed containing zero or more previous blocks (according to the user’s *maxPrevPhrases* setting, the history buffer size, and the current block index) plus a zeroed block at the end. For each member of the population, the candidate is copied to this zeroed area and the entire subject is sent to the fitness function for evaluation. Thus in all blocks except the first, historic context and its compatibility with the current candidate are evaluated.

The fitness function then transforms this subject into a `PhraseInfo` object and calls all of its enabled *fitness modules* with this subject, performing a weighted summation as outlined in section 3.2.2 (page 25); this weighted summation serves as the fitness score for that subject.



### 4.2.1 PhraseInfo

The `PhraseInfo` object is the only argument to the fitness modules, and thus contains all possible information that a fitness module might want to know. This includes note count, phrase count (as there may be multiple blocks present) and the input itself as a byte array.

When the object is constructed, the object is also transformed into two linked arrays of pitches and durations. This format is particularly useful for modules that evaluate melody or rhythm exclusively. The initial rest length is also stored, but is not typically used (as previous blocks may have extended it, making it indeterminate), and nor is the final note duration for similar reasons.

`PhraseInfo` also contains convenience methods such as `getChordAt`, which returns the currently-playing chord for a given index of the subject array. This is necessary because each subject contains multiple blocks, and each block contains multiple chords.

## 4.3 Fitness modules

As with initialisation and crossover methods, fitness modules inherit from a base class (`FitnessModule`) and are managed by a `FitnessModuleFactory`. The base class specifies the module's weight (importance) and whether module is enabled. Its core method, `evaluate`, accepts a `PhraseInfo` subject and returns a floating-point fitness score  $\in [0, 1]$  as described in the requirements. Each module should return a roughly consistent distribution of fitness scores.

A description of each module follows; the `Fm` prefix stands for “fitness module”.

### 4.3.1 FmInfluenceTest

This module is used to track the relative influence of modules in the weighted summation, and to test the genetic algorithm in general. The implementation changes often, but generally encourages every note in the phrase to be C4; the returned fitness is  $\frac{\text{Number of C4 notes}}{\text{phraseLen}}$ .

### 4.3.2 FmKey

Each note in the *most recent* block is checked against the current scale. Fitness is defined as the % of those notes that are in the scale: the fitness score is *not* affected by the number of notes in the phrase. This module is critical in ensuring tonality and is thus given a higher default importance.

### 4.3.3 FmRange

**FmRange** returns a score of 1 if *all* notes are within the user-supplied minimum and maximum octaves (by default, C3–C6 inclusive) – or 0 otherwise. This module prevents melody lines from leaving the reasonable pitch range as they are immediately punished; due to its importance, it is given a higher default weight.

### 4.3.4 FmPitch

Checks each interval (distance between adjacent pitches, in semitones). A user-supplied table maps intervals of 0–12 semitones to scores; intervals above an octave receive a score of zero. By default, scores are based on consonance and proximity (smaller intervals are rated higher). An additional bonus is provided if the phrase begins with the root note. The fitness is represented as the score divided by the maximum possible score for that phrase.

### 4.3.5 FmRhythm

**FmRhythm** carries out a number of rhythmic tests over the subject. It encourages:

**Notes on specific beat positions** e.g. power-of-two subdivisions of the bar

**Adjacent durations of similar length** to encourage consistent rhythm

**Power-of-two note lengths** to discourage overt syncopation

**4<sup>th</sup>, 8<sup>th</sup> and 16<sup>th</sup> notes** to encourage the most common durations

Fitness is defined as the score divided by the maximum possible value. The initial rest and last duration are ignored as blocks either side of the subject may extend them.

### 4.3.6 FmRepetition

This module performs a simple scan of all two- and three-note sequences against all other such sequences. If the pitches of both sequences match, and the sequence does not contain the same adjacent note twice (to discourage repeating the same note for long lengths of time), a bonus is awarded. Fitness is defined as  $\frac{3 \times \text{score}}{\text{maxScore}}$ .

A maximum or high score is difficult to obtain; multiplying the score by three evens out the score distribution and nullifies the effect of *too much* repetition, since we don't really want to encourage that anyway.

### 4.3.7 FmRhythmRep

Performs the exact same routine as **FmRepetition**, only on note *durations* rather than pitches. This experimental module is designed at nurturing structural consistency by looking for duplication of rhythmic patterns within the subject.

### 4.3.8 FmScales

**FmScales** encourages melodic contours that would not naturally evolve otherwise: specifically, chains of ascending or descending notes. Where the note is a single scale step above or below the last, bonus points are awarded – this encourages “scale runs”, as commonly seen in improvisation. The algorithm performs a simple scan of all notes in the phrase, keeping track of sequences where the registral direction (motion of notes up or down) is constant. The final fitness score is equal to the *longest* run, plus bonus points for single scale steps, over the maximum score.

There is an option for this module to only come into effect for approximately  $\frac{1}{8}$  of all blocks; combined with a high weighting, this has the effect of making the improvisation suddenly change into scales at some point in the improvisation, then revert back to regular playing. This has a more pronounced effect on the output than simply encouraging scales at all points in the piece.

### 4.3.9 FmExpectancy

The first implementation of expectancy tries to follow the spirit of Narmour’s original model, with finely-grained bonus scores and score multipliers for specific conditions. For each pair of adjacent intervals  $i, j$  with note durations  $L_i, L_j$ , the following rules hold:

**Registral direction** If  $i$  is large ( $> 6$  semitones) and the registral direction has changed, add 1 to the score. Otherwise, if the direction is the same, add  $6 - i$  to the score – smaller intervals have larger implications.

**Intervallic difference** If  $i$  is small, add  $\frac{2(i+1)}{1+|i-j|}$  to the score – implying similar-sized intervals. Otherwise, if  $i > j$ , add  $i$  to the score – implying smaller intervals.

**Closure** Increase expectancy for these intervals if registral direction is reversed,  $i$  is large and  $j$  is small, or  $L_j > L_i$ .

Additionally, expectancy is increased if **registral return** – the pattern **aba'** where **a'** is  $\pm 2$  semitones of **a** – is found in the phrase.

The final score is then mapped to a value between 0 and 100 via division by an empirically-derived constant, in this case  $3.5 \times \text{phraseCount}$ . The fitness value is defined as

$$1 - \left( \frac{|\text{exp} - \text{targetExp}|}{\text{targetExp}} \right)^{\text{punishFactor}}$$

where *exp* is calculated expectancy, *targetExp* is target expectancy in percent (default: 75), and *punishFactor* determines the score scaling for non-optimal phrases (default:  $\sim 2$ ).

#### 4.3.10 FmExpectancy2

This module implements Schellenberg’s revised model of expectancy (Schellenberg, 1996) which is considerably simpler in nature. There are three tests, all performed over pairs of intervals *i, j*: the *implicative* and *realised* intervals respectively.

**Registral direction** If *i* is large ( $> 6$  semitones) add 2 if the registral direction has changed, 0 otherwise. If *i* is small, add 1.

**Registral return** if *j* is in a different direction to *i*, and  $|i - j| \leq 2$ , add 1.

**Proximity** Add  $12 - j$  to the score if  $j < 12$  – smaller intervals get a higher score.

The module specifies three user-configurable weights, one for each test. By default these weights are 2, 3 and 0.5, so that all tests have the same maximum score of 6. The target expectancy is once again 75% and the final fitness score is calculated as before.

Obviously, only one expectancy model should be enabled at a time. In theory both should result in the same quality of output.

## 4.4 User interface

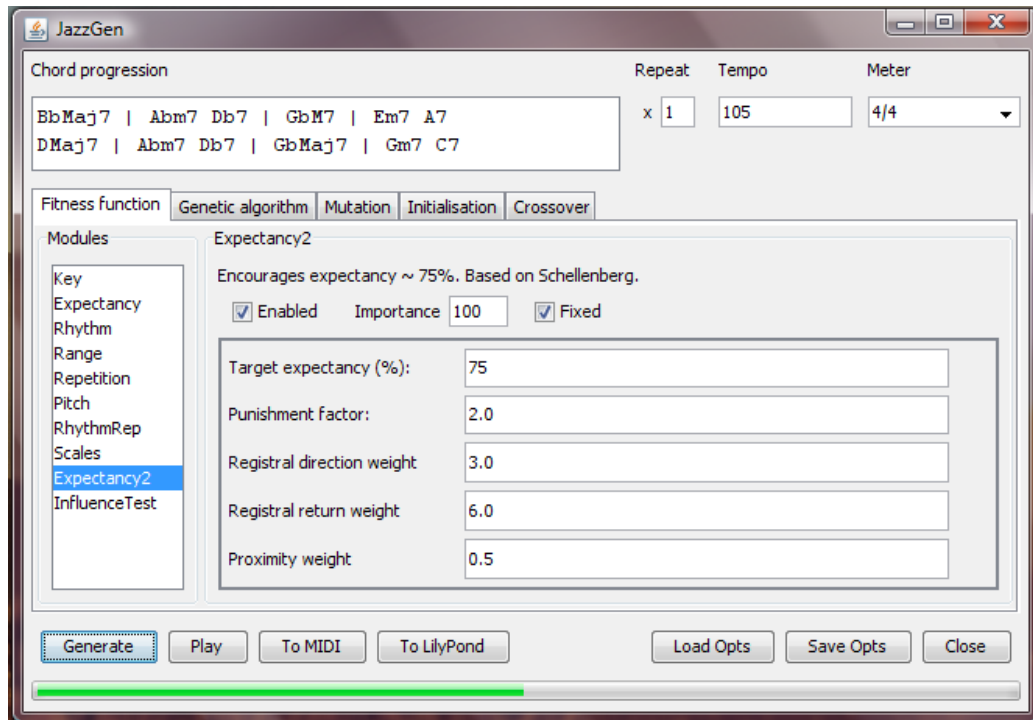
*JazzGen* could certainly be controlled via the command line, but given the large number of choices and options the user can make, this would be unwieldy. Instead, a full user interface allows the user to navigate through genetic algorithm options easily.

The `JazzGenUI` class handles the Java Swing user interface for the system. Figure 4.1 shows the UI with the “Fitness function” tab selected.

The user design has been designed in a simple fashion: the placement of components from top to bottom represents the workflow of generating a composition. First, highly dynamic information such as the chord progression and tempo. Then, the genetic algorithm parameters. Finally, the end-point of the interaction: generate the improvisation, play it or export it. The progress bar at the very bottom of the interface reflects the status of the GA. Each task takes place on a separate thread, so (for example), a new output can be generated while the old one is playing.

To avoid a cluttering of class files, button click and UI component change events are handled by *inner classes* that reside within `JazzGenUI`. The `ActionListener` interface provides a simple way to receive and action UI events.

See Appendix A for sample Lilypond output.

Figure 4.1: *JazzGen* user interface: fitness function tab

#### 4.4.1 Configuration framework

Central to the interface to a set of options: enabled mutations and fitness modules, GA parameters, selected crossover and initialisation methods, etc. These options must be modifiable via the user interface, propagated back to the class concerned, and finally saved to a file on application close for later restoration. To handle all of this, a consistent *configuration framework* is employed across all classes and components.

The `Config` object is responsible for the saving and loading of options. It is a hash table, mapping identifiers (strings in this case) to arbitrary objects or arrays of objects. Using the standard `get` and `put` methods, each class can retrieve and save the options for its own identifier(s).

The identifiers are hierarchical and each module can have several. For example, convenience might dictate storing an array of `doubles` and an array of `ints` separately, instead of boxing all the values and storing them as an array of `Objects`.

#### Saving and loading

The `Config` class uses an `ObjectOutputStream` to serialize itself into a string suitable for saving to disk. It then uses standard Java I/O to store or retrieve itself from an arbitrary

Table 4.1: Typical **Config** class identifiers

Class name	Identifier
CrossoverAverage	<code>c.avg</code>
CrossoverRandomMask	<code>c.rm</code>
FmExpectancy	<code>fm.exp</code>
FmPitch	<code>fm.pitch</code> , <code>fm.pitch.bonus</code>
GenAlgOptionsUI	<code>genopt.int</code> , <code>genopt.double</code>
InitOneOverF	<code>i.1/f</code>
MutationOptionsUI	<code>mutopt</code>

filename. In addition, a default filename `default.opt` is defined: this filename is used to automatically save the configuration when *JazzGen* closes, and restore it when it starts up again.

### Save, load, commit

For every option-bearing object (fitness module, initialisation module, the main interface components) there must exist three standard methods:

Table 4.2: Configuration method list

Method prototype	Description
<code>public void commit()</code>	Parse the user interface and update the method's parameters to reflect the UI values.
<code>public void save(Config c)</code>	Save method's parameters to a <b>Config</b> object. <code>commit</code> should be called first.
<code>public void load(Config c)</code>	Load method's parameters from a <b>Config</b> object.

The methods are defined by the abstract class **GeneticMethod**. In some cases, the objects are constructed with the **Config** object instead of calling `load`.

#### 4.4.2 JazzGen invocation

If *JazzGen* were executed in the same thread as the user interface, the UI would become unresponsive and “freeze” while the output is generated. The algorithm must instead run in a separate thread, but it also needs a way to notify the user interface of its current progress (blocks generated), and to tell the user when it completes.

Two additions to the code base are needed. First, the **JazzGen** object is modified to support an *observer* which can hook-in via the `setObserver` method; *JazzGen* will then notify this object of its progress updates. Then, the observer is created as a subclass of the

**SwingWorker** built-in class: this library automatically handles running code in a separate thread and *safely* dispatching progress updates to the user interface. All that is needed is to implement the `doInBackground` method (which initialises and runs the generator), the `updateProgress` method (which handles progress update messages), and to call the `execute` method of the observer after committing all options.

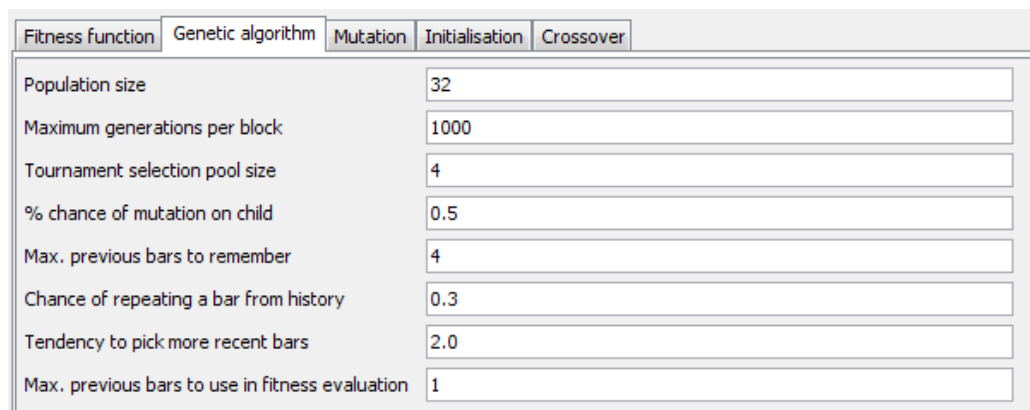
### 4.4.3 Configuration tabs

Each tab controls a different aspect of the genetic algorithm.

#### Fitness function

The fitness function tab is shown in Figure 4.1 (page 46). A list of fitness modules lies to the left; the user may select one by clicking its name. The right side of the tab then updates to show the settings for the selected module. A helpful single-line description of the module is shown, followed by options available to all modules (enabled/disabled, importance). The box underneath these components will contain module-specific settings; each module implements a `getOptionsUI` method which returns a `JPanel` containing this UI.

#### Genetic algorithm



Option	Value
Population size	32
Maximum generations per block	1000
Tournament selection pool size	4
% chance of mutation on child	0.5
Max. previous bars to remember	4
Chance of repeating a bar from history	0.3
Tendency to pick more recent bars	2.0
Max. previous bars to use in fitness evaluation	1

Figure 4.2: *JazzGen* user interface: genetic algorithm options tab

The genetic algorithm tab is displayed in Figure 4.2. This tab contains a simple vertical list of options presented in a consistent format. The most important parameters are listed first, grouped roughly into: general options, selection/mutation, history.

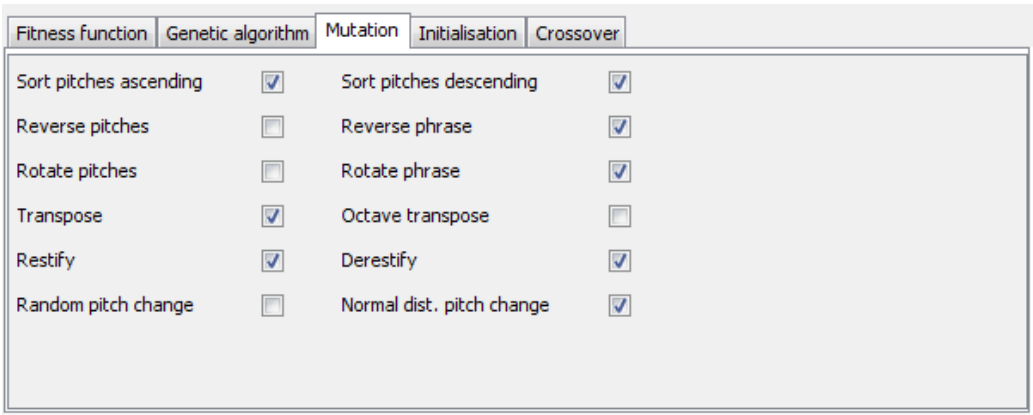


Figure 4.3: *JazzGen* user interface: mutation tab

**Mutation**

The mutation tab is shown in Figure 4.3. Each method is displayed in a simple table; each row contains a pair of related mutations.

**Initialisation**

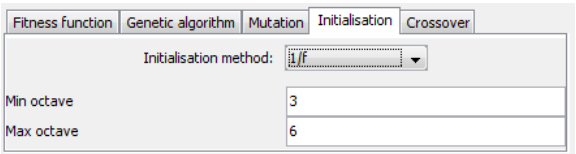


Figure 4.4: *JazzGen* user interface: initialisation tab

The initialisation tab can be seen in Figure 4.4. The choice of method can be selected via the drop-down box. If the method has any options, they are displayed immediately below this box. The most common methods are listed first.

**Crossover**

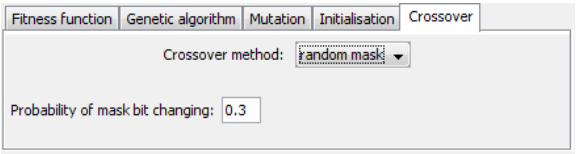


Figure 4.5: *JazzGen* user interface: crossover tab



Figure 4.5 displays the crossover options tab. The layout is identical to that of the initialisation tab.

## 4.5 Chord progression

The chord progression dictates the size of the musical output, and the keys and scales that are used in the improvisation: it is crucial to providing a harmonic context to the improvisation. A system is therefore needed to represent scales, chords and the progression itself. Extensibility is a key concern, as more scales or chords may need to be added at any time.

### 4.5.1 Scales and chords

To implement scales and chords quickly and cleanly, Java `enums` are used. This relatively new feature, introduced in Java 1.5, implements typesafe enums as anonymous subclasses of a base class. The enums can be assigned arbitrary values, or sets of values, via the base class constructor. For example:

```
public enum Scale {  
    Major(new byte[] {0, 2, 4, 7, 9, 11}),  
    Blues(new byte[] {0, 3, 5, 6, 7, 10}),  
    ...  
    public final byte[] offsets;  
  
    Scale(byte[] offsets) { ... }  
    public byte[] notes(byte root) { ... }  
    ...  
}
```

In this case, the value of a scale is an array of its offsets from the root note, in semitones. A utility method is attached to the enum; this returns the offsets for a given root note (by adding the root note to every note in the scale, mod 12, then sorting them in ascending order).

Chord *types* are represented by the `ChordType` class, storing both the offsets of the chord's notes, and the scale best played over the chord type, e.g. `Scale.Locrian`. Finally, the `Chord` class encapsulates the chord type, root note and inversion. Inversions can be positive or negative; this determines how many notes from the right or left of the chord are increased or decreased by an octave respectively. It is possible to, for example, specify a fifth inversion of a dominant seventh chord: the root note is transposed two octaves higher, while the other notes are transposed one octave.

### 4.5.2 Parsing the progression

The user inputs the progression in a simple format, e.g.

```
Em7 F7 BbMaj7 Db7 / GbMaj7 A7 DMaj7  
Dm7 Eb7 AbMaj7 B7 / EMaj7 G7 GMaj7
```

Figure 4.6: Chord progression for *Countdown* by John Coltrane

The / character, | character or newline denotes a new bar. Let  $N$  be the number of chords defined within the bar. If  $N$  is a power of two, the chords are timed to occur evenly across the bar’s duration. If not, round up to the nearest power of two and assume the “missing entries” are repeats of the last chord in the bar. For example, G7 F7 C becomes G7 F7 C C.

### 4.5.3 Voice leading

The `ChordProgression` class encapsulates chord progression information, and performs the parse by accepting a `String` as its constructor. It also maintains a statically-initialised map of `String`  $\rightarrow$  `ChordType` relations; most common chords, from C+ to B#maj7+11, are supported. Sharps are represented by # and flats by b; the input is case-insensitive.

If the root position was used for all chords, the harmony would harshly jump up and down in pitch, as may the solo itself (depending on fitness function implementation). Instead, chords are inverted such that the distance between notes in adjacent chords is minimised.

The inversion is a simple process: the first chord is left at root position. For each subsequent chord, try inversions from -4 to 4 and add the distance between every pair of consecutive notes (with some adjustment). The lowest score inversion is used. However, inversions that bring the chord so low that it would conflict with the bass, or so high that it might conflict with the solo, are given a score penalty. If two inversions have the same score, one is picked at random; this avoids the problem of always picking the higher or lower of the inversions, which would cause the chord progression to consistently move upwards or downwards in pitch over time.

## Chapter 5

# Testing and Experimentation

I have won several prizes as the world's slowest alto player, as well as a special award in 1961 for quietness.

— *Paul Desmond*

With the *JazzGen* system in place, it now remains to tailor the genetic algorithm such that it produces the best output possible in the time available. This process includes testing, finding and fixing bugs in the system, and carrying out tests on elements of the generation. Firstly, however, mention must be made of the original fitness function.

---

*MIDI files, as supplied on the CD in the `midi` directory, are referenced like so: [early1] corresponds to the file `early1.mid`. Except where explicitly stated, all files represent the **first** GA output with the relevant settings, to avoid the hand-picking of superlative outputs that would make the algorithm seem better than it actually is.*

### 5.1 Early fitness function

Originally, the fitness function was a monolithic collection of arbitrarily-defined constraints. Weight values were based on intuition alone. Appendix B (page 76) lists these constraints and their associated rewards and penalties.

[early1,early2] show the output of this fitness function over a simple blues progression, with only basic (normally distributed) pitch mutation, random initialisation and single-point crossover. The output is consistently acceptable, but features too much repetition and not enough variation. At this point the genetic algorithm was rewritten to use a modular fitness function.

## 5.2 Testing and debugging

Before the GA could make any sense at all, several bugs had to be discovered, tracked down and fixed. The test approach consisted of several simple procedures:

- Trialling operations in a separate class to ensure correctness; e.g. checking that the distribution and range of previous history blocks for initial seeding is appropriate.
- Adding `try/catch` blocks around operations that might return array out of bounds or other errors. The `catch` block halts the program and outputs information that explains how the array offset was calculated.
- Debug instrumentation was added around suspicious sections of code.
- Unnecessary features such as mutations, history seeding, and most fitness functions were disabled. Once the basic GA had been seen to work properly, features were enabled on a discretionary basis.

Several sections of code warranted more in-depth testing. With the `MutationFunction` class, a sample musical phrase is run through each of the mutations in turn, with the results displayed so a human can verify their correctness. These test cases are implemented as static `main` methods of the relevant class; running `java MutationFunction` (in this case) will execute the tests for that class.

General testing revealed several *major* flaws in the code base. In some cases these flaws would have severely compromised the quality of the GA if not found, and thus a dedicated test plan turned out to be a prudent idea. A few examples follow:

- Converting a floating point fitness value (between 0 and 1) to an integer *before* multiplying it by the weight. This results in the value being *either* 0 or 1, drastically reducing the direction quality of the GA's random search. With a lower level of a granularity, the GA would take *many* more iterations to reach an acceptable fitness value.
- Forgetting to remove a clause that returned a random fitness value if all weights summed to zero. The weight sum was no longer used but remained initialised at zero, causing random fitness values to be returned for *all* evaluations and effectively reducing the GA to a random number generator.
- Specifying the wrong scale array caused all "is this note in the current scale?" lookups to be performed against the key of C rather than the current key, resulting in the key of C being used at all times.

### 5.3 Genetic algorithm configuration: part one

To begin with, random scale initialisation and single point crossover were used. All mutations were disabled. The most important modules, **FmKey** and **FmRange** were enabled with weights 200 and 250 respectively; the acceptable octave range was set to C2–C7, slightly larger than the initialisation range of C3–C6, to allow for transposition when recycling old output.

The chord progression C7 / F7 / G7 / C7 was used, as the blues scale makes it easier to produce acceptable output given its pentatonic basis. [keyrange1] shows the first output with these settings, and also demonstrates transposed repetition for bars 2 and 4. Two problems were also identified:

- Bars 2 and 4 are exact copies of bar 1 (after transposition), in spite of crossover over thousands of iterations. This was determined to be caused by a) *elitism* preserving exact copies, b) near-identical fitness values for all members, and c) that the members copied via elitism were deterministically picked as the “most fit” due to their position in the population. When the fitness function becomes more complex this should not happen.
- The solo is “cut short” at the end. Rather than try to induce a suitable conclusion in the fitness function, a simple repeat of the last-played chord was added to the end; this wraps up the solo, but is not a perfect solution [keyrange2].
- The solo actually alternates between high and low octaves rapidly, creating an effect of two simultaneous melody lines. This is not a desirable property for fitness evaluation.

Modulo repetition, this output seems to resemble that of the original fitness function. This implies that many of the rules in the original fitness function had a limited or destructive effect on the output, and that any patterns or relationships found in the notes were conjured up by the human mind, as the output is in fact quite close to random noise. It therefore remains to improve this noise into something more meaningful.

Generation via random chromatic notes converged very quickly to a maximum score for FmKey; members with chromatic notes are rapidly eschewed, making the method equivalent to “random scale” at this point in time. 1/f generation [oneoverf] seemed similar to the uniform distribution, but eliminated the low notes that created an impromptu counterpoint, solving the third issue above.

**Chaotic generation** The Standard Map generation was not heavily investigated, but it generated surprisingly advanced output without any evolution [chaotic0] (as shown in Morris, 2005). Evolution introduced rhythmic complexity and repetition, but overall output tended to resemble that of the 1/f distribution [chaotic2000], which is not surprising given that GAs are designed to handle arbitrary initialisation seeds.

To provide more tonality, a **FmChord** module was created to apply a bonus for each note belonging to the current chord. This has a side-effect of encouraging the seventh note on seventh chords played over a major scale [chord]. The module attempts to encourage 50% chord notes per block.

**Elitism** The initial elitism value was set to 4, which means the four highest-fitness members of the population are transmitted to the next generation unaltered. The value of 4 was carefully considered: lower values seemed to lower average fitness values across the population, as the best individuals are lost and cannot propagate; higher values did not affect the mean but lowered their standard deviation, possibly limiting the variety of musical phrases as the “best” were duplicated too often.

### 5.3.1 Repetition

The initial method of encouraging the repetition of two- and three-note sequences yielded poor results. The average module score first hovered between 0.05 and 0.15 and converged quickly, most likely to due the difficulty of evolving repetition through single-point crossover. Uniform crossover proved to be too destructive, so double-point was chosen as a compromise; scores improved to around 0.2, and some structure was apparent [rep1].

To encourage variation the “normally distributed pitch shift” mutation was enabled on children with probability  $\frac{1}{2}$ . The population size was increased to 64, with 4 elitism candidates, and the tournament selection pool was reduced from 4 to 3 to lower the pressure to select high fitness and prevent early convergence. The first output is shown in [rep2]: the repetition is now much more evident.

An alternate module, **FmRepetition2** was created. This module scans the block for repetitions of *arbitrary* length, assigning a score based the average semitone difference. This allows repetitions that are not exact, although they receive a disproportionately lower score. This can be seen in the first and last block of [reptwo1], the first output with the module enabled. Ultimately this module was used over **FmRepetition** as it was more likely to develop structure.

### 5.3.2 Expectancy

One problem with [reptwo1] is the occurrence of low and high notes played together. Although heard as a syncopated rhythm, it is not realistic for a pianist to jump around octaves like this. Expectancy should encourage intervals that we expect to hear in music, and therefore are those commonly played naturally by musicians.

[exp75] shows the first output with **FmExpectancy2** enabled, default weightings, target expectancy of 75%. The melodic contour is now far smoother, with ascending and descending lines that stay close to a local tonal centre. Bar 2 shows a partial repetition of bar 1 (transposed). [exp10] shows an output with target expectancy 10%. Large intervals

are evident, but perhaps due to the repetition module, there is still some structure and predictability in the piece. Conversely, [exp100] has a target expectancy of 100% and seems constricted, not usually willing to travel more than a few notes from the root. Further experimentation confirmed that 75% was an ideal medium.

[exp75] has several issues: firstly, the complete lack of rhythm, which will be addressed later. Secondly, the large number of repeated adjacent pitches, as the proximity rule assigns the highest score to sequences of identical notes. There are several ways to fix this:

1. *Reduce the proximity weighting.* This is undesirable as it leads to unnatural, large-interval phrases.
2. *Reduce the bonus for intervals of size zero.* Such intervals were given a score of 5 instead of 12; the first output can be seen in [expbonus]. It is now devoid of repeated notes, but sometimes outputs have large intervals in them (octave or more) despite the weighting. Despite this, output is pleasing; [expbonus2] shows a nice descending scale line.
3. *Treat repeated notes as extensions to the first.* This is bad for several reasons: it makes true repeated notes impossible, and it allows a melodic fitness module to encourage rhythm. However, it can create an effect of “intentional” pausing [exptie] that *might* be tied to the melody in a meaningful way, although some effort would be needed to make chords and notes coincide rather than staggering after one another.

Method 2 was chosen as it delivers good results in general; method 3 may be useful if rhythmic constraints disappoint. Registral return caused too much three note repetition so its score was reduced to 4.0 (from 6.0); the registral direction effect was perhaps too prominent but scale runs are statistically less likely than other phrases so it is important to encourage them. [expfinal] shows the first output with the finalised weightings.

**FmExpectancy** The original FmExpectancy model did not perform well. It was determined that the revised model (Schellenberg, 1996) is much more explicit in its specification, while Narmour’s is rather vague. Due to this, the author invented several weighting values and score multipliers that may need careful tweaking before good output can be generated with FmExpectancy. As FmExpectancy2 already delivers satisfactory results, the original model will no longer be considered.

### 5.3.3 Fitness analysis

Figures 5.1 and 5.2 show the fitness statistics for [stat1], a two-bar improvisation over D $\sharp$ m7b5. This information was logged in order to see if any part of the current module configuration needed addressing.

Looking at the modules, **FmRepetition2** takes the longest to converge, but all modules reach a fairly static level within 200 iterations. Nonetheless, the average fitness graph

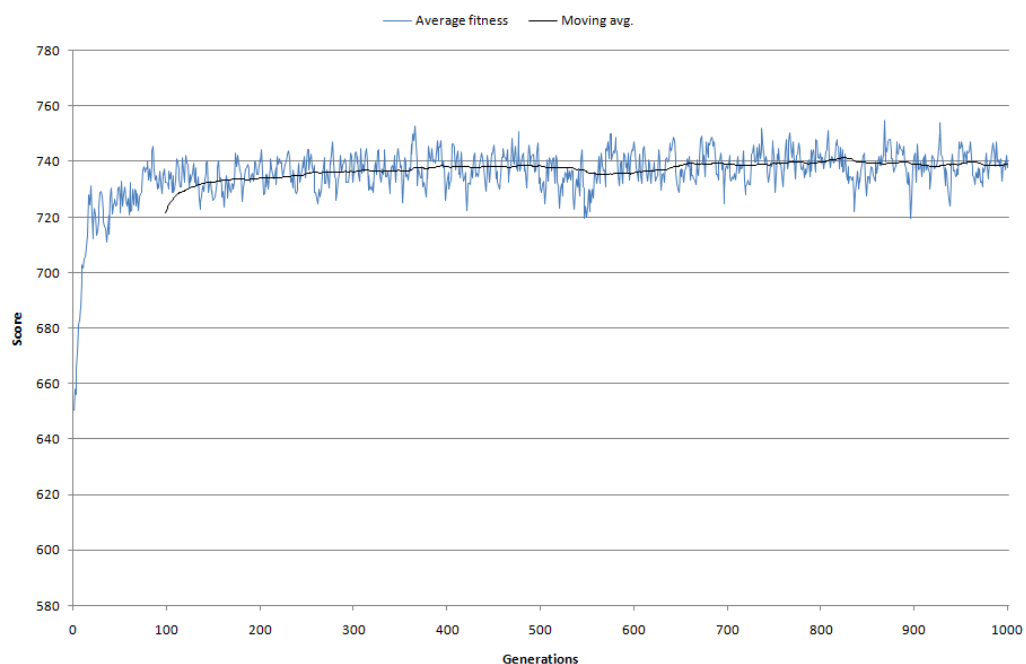


Figure 5.1: Average fitness for population of [stat1]

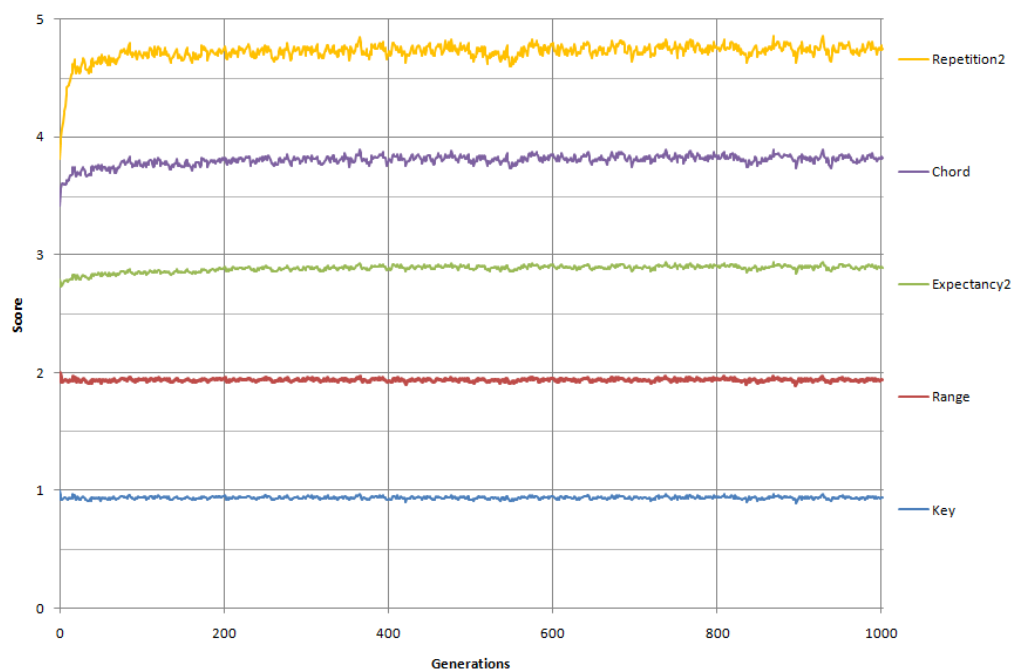


Figure 5.2: Stacked line graph of module scores for [stat1]



clearly shows a steady but slow improvement of fitness (from 720 to 740) over the remaining 800 generations. **FmKey** and **FmChord** converge fast and stay roughly constant. This is a good thing, because 100% on-key notes is not desirable, and also because mutation and crossover are continually introducing off-key notes into the population. **FmRange** is constant at 1.0, which is highly desirable.

Like repetition, **FmExpectancy2** converges to a steady state and does not drop. Although the score then improves extremely slowly, it has already reached a realistic maximum; given that the score is an *average* over the entire population, and that low-fitness specimens are often selected for genetic diversity and to prevent early convergence on local maxima, this is to be expected.

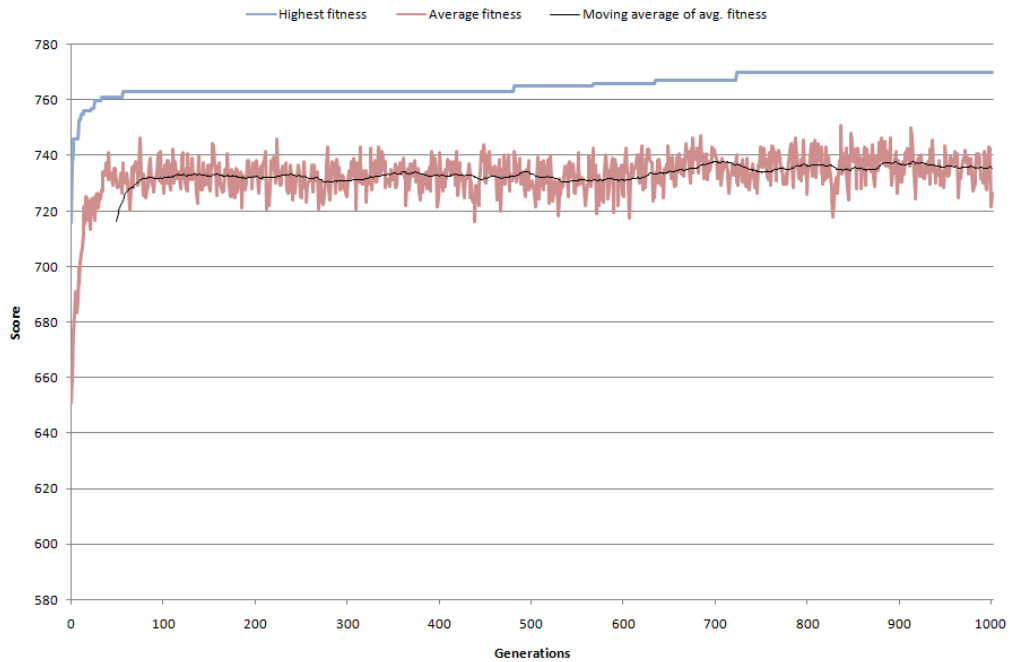


Figure 5.3: Average and best fitness for population of [stat2]

The majority of tested outputs peak at 80%–90% fitness, reaching a plateau after a few hundred generations, and this is echoed in the average fitness graph. This curve can again be explained by taking low-fitness members into account, but the purpose of these members is to increase the variety of the population and ultimately lead to more high-fitness search paths. But does it? Figure 5.3 shows the highest fitness of each generation for [stat2], a solo over C7. Again, the average fitness rises to 740. The *highest* fitness of each generation, due to elitism, jumps up in steps, finally reaching a fitness of 770 at generation 723.

A subsequent test with 10,000 generations yielded a best fitness of 772 after just 2,100 steps (at 1,000 iterations the score was 767). Despite the small improvement, future tests were set to use 2,000 iterations instead of 1,000. However, this plateau is best defeated by

adding more fitness functions, in order to diversify the “fitness landscape”, open up more possibilities and prevent convergence on local maxima (as fewer of them will fit the new criteria).

## 5.4 Post-pass

The initial post-pass lengthens all notes to fill any subsequent rests. This creates an effect that, due to the piano tone, is neither staccato nor legato but rather a mix of the two. The pass also applies a light velocity accent to the first note of each bar and a random, almost imperceptible timing offset is added to every note in an attempt to remove the “mechanical” impression one might get from hearing perfect timing. However, the velocity distribution of the piece was still quite flat, and this made the output less interesting. Rather than add velocity to the fitness function (and vastly increase the search space), it was decided to apply velocity curves during the post-pass stage. A general description of the algorithm follows:

1. When encountering a new “melodic fragment”, reset the velocity to a normally distributed value of mean 90 and standard deviation 20. A melodic fragment begins after a note with particularly long duration (i.e. a rest), or a note that is more than an octave higher or lower than the preceding note.
2. For each note in the fragment, perform a random walk to determine velocity. The shift amount and chance of walk direction changing is tied to the duration of the note; longer durations cause larger shifts and a higher probability of change. If the velocity exits the range 60–120, reflect it back into the range and reverse the walk direction.

This simple algorithm is designed to encourage a base velocity of 90 (*mezzo forte*), with occasionally deviating velocity within a melodic fragment, and possibly drastically different velocities between fragments. It does not try to check for “clever” times to change velocity, but rather aims to sound more “human” than if the velocity were simply constant or completely random. The initial range of velocity 60–120 was far too quiet, and later changed to 80–120: although restrictive, the change in volume is perceptible and aids realism [countdown]. Combined with a complex chord progression, the output is occasionally similar to a human performance.

Jazz players usually “swing” their notes (see Introduction). To achieve this effect, the timings of notes were adjusted while in the 16th note format as described in section 3.1.3 (page 23). For each set of four sixteenth notes, a duration filter is applied, e.g.  $[0.4, 0.3, 0.2, 0.1] \times 4.0 \times [\text{note durations}]$ . The durations must sum to 1, so the total duration of the set is the same. But the first eighth note of the set will have an augmented length, while the latter will be diminished.

Experimentation revealed that aggressive swings sound unnatural as they are applied unilaterally and without consideration of the rhythm already present. Lighter swings, where less emphasis is applied to the first eighth note, made the output sound more authentic with relatively little effort [swing].

## 5.5 Genetic algorithm configuration: part two

### 5.5.1 Rhythm

**FmRhythm** is a module that encourages notes on beat positions, and durations that are of a sensible and consistent length. Enabling the module immediately produced a complex but restrained rhythm that sounded more pleasing than the previous quasi-random rhythmic output. Unfortunately, repeated note sequences – due to either previous block seeding or crossover – generally contain a fitter rhythm than notes that are not copied. This encourages repetition quite strongly, as can be seen in the output of [rhythm1]. Subsequent outputs did not duplicate this, so it could have been a glitch; [rhythm2] shows a combination of consistent rhythm and appropriate pauses between sequences of notes. However, output over complex chord progressions seems to suffer [rhythm3], perhaps because the now harder to fulfill fitness modules are being ignored in favour of satisfying rhythm.

One solution is to employ the **FmNumNotes** module, which encourages a specific number of notes in each phrase. If the target is 12 notes, for example, a range of 8–16 notes will not be heavily punished, but fewer will result in a very low fitness score. This creates more interesting rhythms [rhythm4] but also has its drawbacks. Previously generated blocks tend to satisfy this function fairly heavily, and so end up being reproduced verbatim as those subjected to crossover end up with more or less notes. And for phrases with more chords, the pauses do not work well with the busier sound [rhythm5]: to fix this, the **FmNumNotes** scales its target to the number of chords in the current block, to ensure that chords get at least some melodic activity [rhythm5b]. This also sustains interest in longer solos which have blocks with varying numbers of chords [longsolo].

Regardless, it is important to note that the fitness function *can* be customised to produce output more suited to the desired style and progression. With the addition of rhythm, the GA now outputs pieces with the melodic quality of [countdown], but with a much more regular timing. The emphasis on power-of-two beats and notes on beat boundaries encourages a regular “rhythmic skeleton” around which note durations can shrink or extend to create variation.

**FmRhythmRep** The **FmRhythmRep** module checks for two or three note rhythm repetition, but such rhythm is already generated by **FmRhythm** and history seeding, so no experimentation was performed.

### 5.5.2 Scales

The expectancy module encourages small intervals, but does not heavily specify their direction (Section 4.3.10, page 45). This can prevent the typical runs up and down the scale commonly played by jazz musicians. Enabled all the time, the output seems to favour such runs more often, leading to sustained ascending or descending passages. Due to the relative difficulty of evolving these phrases, other fitness module scores suffer [scales1] and the runs are not particularly apparent in the output. To fix this, the FmScales score was set to alternate between zero and a particularly high score (250) roughly half of the time. One output with these settings is [scales2]: the output starts off routinely, with good repetition. Just when the solo is in danger of repeating itself too often, the FmScales score jumps to 250 in the last bar, and the fitness criteria changes. Previous blocks are no longer good candidates, and a different ending is generated instead.

Many outputs, including [scales2], did not feature scales enough to justify the reduced attention given to the other modules. But changing fitness weights *during* generation is a promising idea, both for introducing variation in the solo, and to stop exact copies of previous blocks from appearing in the output too often. Ultimately FmScales was enabled with a fixed importance of 100, and this leads to some realistic melody lines [scales3].

## 5.6 Mutation

Biles (1994) proposed the use of “musically meaningful mutations” in his seminal GenJam paper. These mutations attempt to speed up evolution by making the output more musically useful: shifting notes forward or backward to cope with timing issues; re-ordering pitches to create ascending or descending lines. To begin with *JazzGen* was configured with the same settings as those used for [scales3]. Mutations were then enabled to see their effect, if any, on the output.

Adding the “restify” and “derestify” mutations compensated for the lack of rhythmic mutation outside of standard crossover [restify]. Rhythms now focused on eighth notes more strongly, there was virtually no syncopation, but the output suffered for it. Looking at the statistics, the non-restify generation has an almost unchanging average FmRhythm score of 0.62, most likely because no meaningful rhythm improvements can result from crossover. The restify generation has a highly-variable distribution with average 0.7, which improves slowly over time. This indicated that the mutations should be enabled, but that the FmRhythm module was incorrectly specified. Surely enough, the module was encouraging quarter notes instead of eighth notes. After some adjustments the FmRhythm score rose to 0.9 and produced much more natural-sounding music [restify2].

Conversely, the “Sort pitches ascending/descending” lead to almost exclusively ascending or descending passages in the output. The fitness functions are designed to encourage such lines when given a base of mostly random pitches; when confronted with pre-sorted melody lines they tend to award a fantastically high score without checking if the melodic contour

is varied enough. To address this, the mutations were simply turned off.

Reversing or rotating pitches or phrases did not seem to have an effect, other than being quite destructive to the average score for FmRepetition2. Similarly, any children given the transposition mutation were discarded almost immediately, because most of their notes were no longer consonant. This was evidenced by the average FmKey and FmChord scores dropping by ~5%, and the discarded children causing the evolution rate to drop. The octave transposition was as damaging as the sort mutations, causing the melody to jump up and down nonsensically.

Currently, only the restify, derestify and pitch change mutations are enabled. Less primitive, “musically meaningful” mutations seem to propagate too far into the population, or have no overall effect after thousands of generations. Therefore *JazzGen* continues to adopt the traditional paradigm of keeping knowledge and search direction in the fitness function, and making the mutations “dumb”.

### 5.6.1 Fitness analysis

One more generation was run with the finalised settings over a standard blues progression: [stat3]

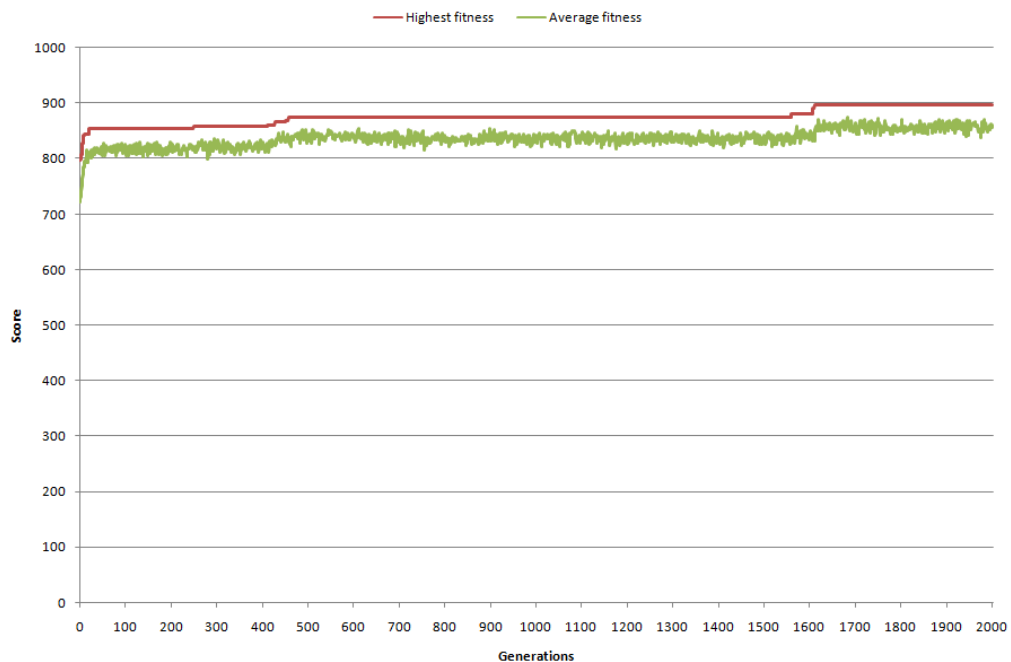


Figure 5.4: Average and best fitness for population of [stat3]

The fitness is similar to the other two samples in that it rises quickly then stabilises, but due to the increased “fitness landscape” there is more opportunity for the score to rise.

Two notable crossovers or mutations bring about sharp rises in the best fitness of the population, and we can also see that the average fitness is highly correlated to the best as elitism propagates the better individuals to others. The rises also indicate that further generations will yield even higher fitness values.



Figure 5.5: Stacked line graph of module scores for [stat3]

The line graph shows that many of the modules, such as FmRepetition2, FmKey and FmChord have roughly static values. This is likely because low-scoring populations can be very quickly mutated to be high-scoring, unlike rhythm or expectancy for example<sup>1</sup>. Major score gains come mainly from sudden jumps in individual module scores, such as FmScales, FmExpectancy2 and especially FmRepetition2, which starts at 0.5 and eventually reaches almost 0.8 after several jumps.

This reveals a flaw in the fitness function itself: the problem of *direction* in the GA's random search. Modules like FmRepetition2 and FmKey, which award high scores for *near*-matches, act as a granular direction for the search: individuals close to a perfect score will continue to evolve – where mutations will improve their score further – while those whose score is decreasing due to bad mutation/crossover will be discarded. The score of all of these modules hovers between 0.9 and 1.0 throughout. In this respect, the GA behaves like a hill-climbing algorithm.

Modules like FmScales and FmRhythm award high scores to phrases that are good, and therefore would be ruined by further mutation/crossover, and low scores to phrases that are

<sup>1</sup>FmRepetition2 awards high scores for phrases a few semitones away from being truly repeating.

bad, and have very little chance of being mutated/evolved into something better. This is due to the nature of what the modules evaluate. The result is that the GA has no method of getting from low fitness to high fitness like it does with the “semitone difference” of FmRepetition2. Instead it waits for an unlikely evolution that produces a very high-scoring candidate, and this change is quickly spread to the rest of the population. The result is a mostly static score graph with occasional jumps.

The conclusion is that fitness modules need to be *directed*, offering a score that is tied to how viable the individual is for future evolution towards a high score, rather than just how good it happens to be *at present*. This is a difficult problem, and something that may be worth investigating in the future.

## Chapter 6

# Evaluation

A real leader faces the music, even if he doesn't like the tune.

— *Anon*

### 6.1 Requirements analysis

The purpose of this section is to review the requirements (page 33) and see if they have been met.

#### 6.1.1 Composition

The system accepts a user-supplied chord progression and generates a jazz improvisation with a walking bass line and percussion accompaniment. The output can be converted to MIDI or Lilypond, or played in real time via a MIDI device.

The genetic algorithm produces music at a reasonable speed: a 64 population, 2,000 iteration, eight-bar solo can be produced in 30 seconds on a medium-spec laptop. This both vindicates the use of Java over Python, and leaves lots of computational room for further extensions to the system.

The output is supposed to satisfy the three constraints of Grachten (2001): tonality, continuity and structure. Tonality is heavily enforced via the FmKey module, yielding music that is 90% scale notes (Figure 5.2, page 58). The FmChord module encourages chord notes which are by definition highly consonant.

Continuity is the domain of FmExpectancy. The proximity check prevents large intervals from frequently occurring, and in general registral direction is not reversed: exactly Grachten's requirements. The melodic contour is further improved by FmScale, which encourages single-semitone scale runs, and the registral return check, which cultivates a tonal centre.



Structure is primarily created via repetition. FmRepetition creates exact or near-exact copies of notes within a block; history seeding reprises old blocks, often in transposed form. The simple restatement, in whole or in part, of a previously played bar instantly creates the effect of “thematic development” (Biles, 1994). FmRhythm also creates a smooth, consistent rhythm; this rhythmic repetition makes notes seem more related due to their position in the rhythm.

### 6.1.2 Genetic algorithm

The GA can be seeded with a selection of generators, including uniform random, 1/f and chaotic. Previous blocks are also used as seeds, with or without transposition to account for key changes. The modules specified have been implemented: FmKey, FmExpectancy2, FmRhythm, FmRepetition2, FmRange. Melodic contour is measured by expectancy. Additionally, FmScales, FmNumNotes, FmChord and FmRhythmRep have been produced and used in experimentation.

A small change has been made: modules can choose to evaluate the current block only, or evaluate the current block and the last  $n$  blocks combined. This reduces the chance of bugs resulting from each module doing block manipulation independently.

The fitness functions utilise a many-to-one chord/scale mapping which supports all listed chords, but does not allow arbitrary extensions; this has so far been sufficient. Only tournament selection was implemented, but it is a flexible and fast selection algorithm and has been satisfactory.

All crossover methods were implemented, but double-point was ultimately used because it is not as destructive as the more exploratory uniform crossover. Similarly, all mutations were coded, but at present only the random pitch shift method is in use.

### 6.1.3 Post-pass

Velocity and tempo “curves” were not applied due to lack of time. However, velocity is changed via a simple random-walk algorithm that has had good results. Notes were given a small, random offset plus a duration multiplier based on swing rhythm, to augment the jazz feel. Finally, note durations were extended to fill any subsequent rests. The sum of these efforts make the output sound more natural.

### 6.1.4 Accompaniment

The chords are played with a simple voice leading algorithm that keeps the “player’s left hand” in roughly the same position. The percussion is a static drum loop, while the bass line plays a random walk over the chord notes, echoing the root note of a chord whenever one is played. These efforts create an enjoyable harmonic and rhythm context without adding complexity to the GA.

### 6.1.5 User interface

The user interface implements virtually all of the requirements, except for the ability to modify meter. Users can modify all generation parameters, save and load options to file, and sensible defaults are provided. Each fitness, crossover and initialisation module has its own UI options which are dynamically displayed on the screen when the module is selected.

## 6.2 Future development

One advantage of an algorithmic composition project is its open-ended scope; from the simple foundation of a genetic algorithm or stochastic system, “layers” of functionality can be added to provide higher-level form and aesthetics, or expand the scope of the algorithm to cover more musical styles or situations. Similarly, *JazzGen* could be expanded in a number of ways if time had allowed.

### 6.2.1 Cultural algorithms

Cultural algorithms (CAs), introduced by Reynolds and Sverdlik (1994) and detailed in (Reynolds, 1994), introduce a second inheritance path via the *belief space*. The belief space contains knowledge that “guides” members of the population towards the optimal solution(s) of the fitness function, by noting the ranges of values, sequences and positions of values, etc. that lead to higher fitness scores. Higher-fitness candidates *influence* the belief space, and in turn, the belief space influences all evolving members of the population.

In *JazzGen*, the addition of a belief space component would allow some of the fitness function modules to be removed, as the belief space codifies their constraints and rules more succinctly. In addition, it should speed up convergence to a high-fitness population as the algorithm remembers what worked well last time, and repeats it. There are, however, several problems: if the belief space is persistent between executions, the high-fitness attributes may be too specific and lead to output that sounds too similar to previous outputs; in any case, the belief space may influence the population too much and lead to less interesting music. Evidently, experimentation would be required to see if CAs would increase the quality of *JazzGen*’s improvisations.

### Subjective fitness function

One advantage to using cultural algorithms is that they provide a mechanism for storing the fitness influence of temporal data, i.e. sequences of notes, rhythms and note ranges at a specific point in the solo. A simple facility could be added to track key presses of ‘g’ or ‘b’ (for “good” or “bad”, as used by Biles, 1994) and increase or decrease the fitness influence score for various metrics based on the currently-playing section of music. The updated belief space would then influence evolving populations of subsequent blocks, to

encourage or discourage elements that the user has commented on. This allows subjective input without the infamous “fitness bottleneck” problem, as the objective fitness functions perform most of the work.

### 6.2.2 Chords and scales

The current chord progression system contains a small number of built-in chords, and a many-to-one map of chords to scales. While this is sufficient for testing, it is not general enough to cover many compositions. A more developed system might support explicit inversions, such as **C7/E**, or explicit scale choices, such as **F7#9(Blues)**. The user could define additional chords as a list of intervals, or simply enter note groupings into the chord progression field, e.g. {**F**, **F#**, **A**, **C**}. Built-in chords could be modified to support an arbitrary combination of augmented, diminished, sharpened, flattened, removed or added notes.

To support these enhancements, the **FmKey** module would be changed to make scale choices that coincide with the last block, or could reasonably be used with the next block. This would mimic a real improviser’s style of playing a single scale or mode over multiple chords, as playing a different scale/mode for each chord is quite difficult in practice. To cope with non-standard chords, “reasonable” scales would be defined as those sharing the most notes with the chord, or having a dissonance distribution that is similar to the scale’s distribution for other chords.

It may also be useful to investigate generating chord progressions from scratch. Tables of common chord choices to succeed other chords are easily acquired and could be used for stochastic generation. A key concept in this generation would be the creation of *tension and release*, where certain chords (or their alterations and extensions) are used to build up dissonance that fades into consonance over time. A chief example of this is the famous ii–V–I progression.

### 6.2.3 Accompaniment

Improving both bass and drums would be a quick way to increase the quality of the output without modifying the main genetic algorithm. The current accompaniment is a simple walking bass line and fixed percussion pattern; in a complete system, these instruments would need to play more intelligent parts to fully simulate a “jam” atmosphere.

The bass line could be generated with a more complex algorithm, such as a Markov chain (McAlpine et al., 1999) or another genetic algorithm. The bass would have additional constraints: its range would be more limited (due to the instrument), it would play slower to mimic the most common playing style, and some kind of harmony with the improvisation and chords would be necessary. The percussion could be generated via a grammar (McCormack, 1996) as it is mostly composed of predefined patterns. The dynamics and rhythm of the improvisation and the placement of chords in the progression could be used

as triggers for more sophisticated patterns, such as fills or syncopation. The overall effect is to augment the structure and rhythm of the solo with percussive harmony, much like a trombone.

It would be useful to explore time offsets in accompaniment; both bassists and drummers often play “ahead of the beat” by an almost imperceptible amount to “push the song forward” (Sabatella, 1995). It is also possible that further instruments could be explored, such as a solo saxophone or full horn section.

#### 6.2.4 Stored “licks”

It would be useful to mix certain “cliché” phrases, or “licks”, into the improvisation; figure 6.1 shows some typical “licks” that would be played over the Dorian scale.



Figure 6.1: Typical “licks” for D Dorian

Typical phrases to play over a D Dorian scale, e.g. Dm7–G7 chord progression.

Licks could be introduced as part of a mutation, to both expand the search space and make dull passages more interesting. The lick would overwrite part of the musical phrase; there may be a simple test that the melodic contour remains smooth, but any harmonic or rhythmic incongruities will be detected by the fitness function. The *JazzGen* block size is currently set to two bars, which is thankfully just enough to cover the most common patterns.

#### 6.2.5 Form and structure

*JazzGen* is current designed for 4–8 bar solos and it shows. The block repetition of *JazzGen* can cause segments of the melody to be restated at several points of the solo. This fools the user into thinking there is a “theme” to the solo, but the reality is much more mundane.

Variation is high and the melody seems to be purposeful, but extend beyond eight bars and the melody will eventually sound meandering as it has no higher-level structure. Similarly, “traits” of the genetic algorithm configuration will begin to become apparent through the regularity of the output, and the sound risks becoming stale.

Motifs, or short themes are often concocted by jazz players and restated several times during a solo. The system could attempt to generate (or find from history) a set of enjoyable themes, then arrange and restate them for maximum impact. Themes could even be mutated or transposed as long as they are continually re-introduced. This would add a sense of structure and purpose to the piece, while also “resetting” the melody line when themes are reprised, to prevent wandering. However, finding high-fitness melodic fragments is difficult and might require a per-note fitness function.

Another variation technique was used with FmScales in section 5.5.2 (page 61). By adjusting the fitness function weights anew for each block, the solo is generated with slightly different parameters throughout. This prevents exact repetition as the exact copies no longer fit the changed fitness landscape, and must mutate to survive. Similarly, the aesthetics of the solo will be different for each pair of bars, and ultimately different for each output, so they will rarely sound stale. Code was added to handle these systems, but the user interface and experimentation will have to be handled later.

Finally, the ending cadence would be replaced with fitness function modules that encourage a real ending to the solo (e.g. fewer notes, return to root note).

### 6.2.6 Emotion

Although the solos of *JazzGen* sometimes appear to have a theme or purpose, no effort was applied to actually giving them one. Research into emotion and music is plentiful and it would be useful to take some of the models for evaluating emotion and place them into the fitness function, for example the early findings of Hevner (1936, 1937) or the work by Kim and André (2004) on affective music.

## 6.3 Conclusion

Expectations for the *JazzGen* project were not high. It has been widely concluded that writing an objective fitness function for music is *hard* (Papadopoulos and Wiggins, 1998; Jin, 2005; Unehara and Onisawa, 2003); the chief aim of the project was not to achieve human-equivalent output but rather explore methods for reaching that eventual goal.

In this, it is felt that *JazzGen* has shown that an objective fitness function can produce results that generate consonant and acceptable “riffs” over chord progressions, combined with a natural-sounding rhythm, repetition and restatement [scales2]. The end result is inconsistent (due to the random nature of the genetic algorithm) but far better than the author’s expectations, and therefore quite promising. After seeing how simple rules like stressing

chord notes, playing notes on beat positions and repeating transposed melodic fragments have aided the realism of the output, the author is convinced that more complicated fitness functions which draw upon higher-level concepts will yield similar improvements; the future development section above covers many of the most important.

At the beginning of the project the genetic algorithm was defined as heavily modular, with a user interface that could support the selection of various methods for mutation, fitness evaluation etc. each with their own set of options. This was primarily to aid my exploration of the system, but is also relevant for extensions: given the small scope of the project, it was decided to design *JazzGen* such that it would be extensible and comprehensive. One can now easily add new fitness modules and functionality.

The genetic algorithm is unpredictable and, early on, exploited problems with the fitness function to produce completely uninteresting output. It took some effort and refinement to get the output to the state where it currently is, and it will take far more effort until the output is of truly acceptable quality. While GAs are resilient to bad methodology to some degree – poor initialisers, crossover or mutation – they are solely guided by the fitness function and it must be precise and correct.

With an objective fitness function the problem is how one defines musical knowledge in this form. The answer was to sidestep the issue and define only the easily-defined aspects of such knowledge, hoping that the genetic algorithm would randomise the rest and that it would sound acceptable. This sort of “guided” or constrained randomness is key to the GA, and many tricks were added to improve the output quality. For example, the velocity curves, swing and FmScales modules created pleasing effects. Complex chord progressions emphasise harmony (which *JazzGen* does very well) and lessen the pressure to create a good melody. Repetition seems to be extremely useful in humanising the piece, especially when used with entire blocks.

Melodic expectancy proved to be an ideal model for improvisation. It is a model which *checks* a phrase, rather than declaring what it should be in advance, which makes it ideal for a fitness function. Changing the target expectancy between 0% and 100% showed drastically different and distinguishable results, and the choice of 75% proved to be the best balance between innovation and conformity. The result is a restrained melodic contour, mostly free of large intervals, that sounds pleasing over virtually any chord progression. However, melodic expectancy is a very general model, and it could be argued that a jazz improviser should seek to implement more domain-specific knowledge. In this *JazzGen* is still very much undeveloped.

Another note is that GAs turned out to be more exact about following the fitness function than the author had realised. For example, the FmKey module awards a score based on the % of notes in the phrase that belong to the correct scale. Rather than allow mutation to create large amounts of dissonance, the score for the FmKey module remained locked at 90–95% consonance in the face of crossover and random pitch changes. In general, despite the large number of enabled fitness modules, they all seemed to have a firm input on the result as the GA mercilessly discards all but the very fittest candidates with each

passing generation. This is but one example of the author's inexperience with GAs; one disadvantage to their use is that they have so many options and choices, that it is easy to produce bad results with misconfiguration.

The author believes that GAs are a good alternative to a traditional rule-based approach, combining elements from stochastic and knowledge-based composition. Modules like *FmScales* and *FmRepetition2* contained checks for musical elements that would be hard to explicitly replicate with a rule-based approach. Even the simple *FmKey* and *FmChord* models encourage a particular quantity of consonant notes, but do so in a way that allows some notes to be dissonant if it fits the purpose of other fitness modules. A simple, non-GA approach would likely roll a dice to determine if each note is consonant or dissonant, at the expense of other concerns. Constraint-based approaches often use a search algorithm similar to a GA to determine answers to complex questions.

Until more effort is spent on the objective fitness function, it is not possible to determine its potential compared to a subjective fitness function. Subjective approaches have their own set of problems: the fitness bottleneck, individual tastes and human error. One use for them may be to manually check inputs to a subjective fitness function and use that to build models that can be used in an objective one. Biles unsuccessfully tried this with a neural network in 1996, but the field is far from closed.

To conclude, the vast majority of requirements were implemented, the output is promising and many routes for future development have been discussed. There is still a long way to go, but at the very least the GA is shown to be a viable method of music composition, and one that can easily improve with more domain-specific knowledge. In particular, the techniques of history seeding and melodic expectancy modelling have produced agreeable results. To this end, the project is considered a success.

# Appendix A

## Sample Lilypond output

```
\version "2.10.0"
\header {
  title = "JazzGen Composition"
  date = "April 25, 2008"
}

\include "english.ly"

PianoSolo = {
  d''16 d''16 c''16 a'16 d''16 c''16 as'16 c''16 d''16 e''16 a'16 c''16
  d''16 g'16 d'16 a16 d'16 a'16 as'16 c''8 d''8 e''16 a'16 g'16 e'16
  d'16 c'16 d'16 e'16 g'8 a'8 g'16 f'16 g''16 d''16 a'16 g'8 f'4
  g'2~g'8. as'8 c''16 d''4 e''4 f''8 f''8 g''16 a''8 a'8 a'8 g'16 g'4..
  a'8 c''8 d''8. a''4 as''4 c''16 d''2 d''2 e'8 g'8 a'8.
}

PianoChords = {
  \clef bass
  <c, e, g, as, >1<c, e, g, as, >1 <f, a, c, ds, >1 <c e, g, as, >1 <g, b,
  d, f, >1 <f, a, c, ds, >1 <c, e, g, as, >1<c, e, g, as, >1
}

Bass = {
  c,4 c,4 e,4 g,4 as,4 as,4 g,4 e,4 f,4 a,4 f,4 f,4 c,4 c4 e,4 g,4 g,4 b,4
  d,4 b,4 f,4 a,4 c,4 ds,4 c,4 c,4 e,4 g,4 e,4 g,4 as,4 g,4
}

DrumsSNBD = \drummode {
  \times 2/3 {bd4 r8} \times 2/3 {r4 sn8}
}

DrumsHH = \drummode {
  hh4 \times 2/3 {hh4 hh8} hh4 hh4
}

piano = {
  <<
  \set PianoStaff.instrumentName = #"Piano "
  \new Staff = "upper" \PianoSolo
  \new Staff = "lower" \PianoChords
}
```



```

>>
}
bass = {
  \set Staff.instrumentName = #"Bass    "
  \clef bass
  <<
    \Bass
  >>
}
hihat = {
  <<
    \set DrumStaff.instrumentName = #"Drums    "
    \new DrumVoice { \DrumsHH \DrumsHH \DrumsHH \DrumsHH \DrumsHH \DrumsHH
      \DrumsHH \DrumsHH  }
  >>
}
snbd = {
  <<
    \new DrumVoice { \DrumsSNBD \DrumsSNBD \DrumsSNBD \DrumsSNBD \DrumsSNBD
      \DrumsSNBD \DrumsSNBD \DrumsSNBD \DrumsSNBD \DrumsSNBD \DrumsSNBD
      \DrumsSNBD \DrumsSNBD \DrumsSNBD \DrumsSNBD \DrumsSNBD }
  >>
}
\score {
  <<
    \override Score.MetronomeMark #'padding = #3
    \tempo 4 = 120

    \new PianoStaff = "piano" \piano
    \new Staff = "bass" \bass
    \new DrumStaff { \hihat }
    \new DrumStaff { \snbd }
  >>
}
\midi { }

```

**JazzGen Composition**

$\text{♩} = 120$

The score is for a jazz composition in common time (C), with a tempo of 120 beats per minute. It features four staves: Piano (grand staff), Bass, and Drums (two staves). The first system contains measures 1 through 4. The Piano part has a complex melodic line in the right hand and sustained chords in the left hand. The Bass part plays a steady eighth-note line. The Drums part features a complex pattern of eighth and sixteenth notes with triplets. The second system contains measures 5 through 8. The Piano part continues with a melodic line and sustained chords. The Bass part continues with a steady eighth-note line. The Drums part continues with a complex pattern of eighth and sixteenth notes with triplets.

Figure A.1: Lilypond output for a sample eight-bar improvisation

## Appendix B

# Original fitness function specification

The following table lists all constraints responsible for generating the earlier MIDI outputs referenced in section 5.1 (page 52). Following each constraint is the associated score modifier: the final fitness value is an integer that may be positive or negative.

Event	Score modifier
Note not in scale	-30
Note is rest <sup>1</sup>	-10
Interval is 12 (octave)	-5
Interval is > 4 semitones	$-\lfloor \frac{interval^2}{16} \rfloor$
Adjacent note lengths are within 2 units of each other	+30
...otherwise	$-(10 + \frac{note\ difference^2}{4})$
Note length is power of two	+15
Note is on 1 <sup>st</sup> or 16 <sup>th</sup> unit	+10
Note is on 1 <sup>st</sup> , 8 <sup>th</sup> , 16 <sup>th</sup> or 24 <sup>th</sup> unit	+5
Note is on 1 <sup>st</sup> , 4 <sup>th</sup> , 8 <sup>th</sup> ... 28 <sup>th</sup> unit	+2
Note is lower than C4	$-\lfloor \frac{C4-pitch}{5} \rfloor$
Note is higher than C7	$-\lfloor \frac{pitch-C7}{3} \rfloor$
Note is on position 30 or 31 <sup>2</sup>	-40
Repetition of any two note pair	+3
Repetition of any three note pair	+10
Bonus per note <sup>3</sup>	+5

<sup>1</sup>Rests are punished as most penalties don't apply to them; otherwise, they would accumulate.

<sup>2</sup>This "discourages syncopation".

<sup>3</sup>This compensates for the average negative value of a note, which would encourage long rests.

# Appendix C

## Code listing

This is a representative sample of code for *JazzGen*. An attempt has been made to focus on interesting, relevant code that serves an important function in the system.

---

*The full code listing can be found in the `src` directory on the CD.*

### C.1 Main classes

#### JazzGen.java

This class generates the chords, bass and percussion, and sets up the `GeneticGenerator` class to generate the main solo.

```
import jm.util.*;
import jm.music.data.*;
import jm.music.tools.*;
import jm.midi.MidiSynth;
import java.util.*;
import java.util.regex.*;
import java.io.*;

/**
 * It's jazz, baby.
 * @author Richard Harris
 */
public final class JazzGen implements JGConstants {
    Score score;
    Part pPiano, pBass, pDrums;
    Phrase phSolo, phBass, phBD, phSD, phHH;
    CPhrase phChords;
    ChordProgression prog;
```

```

Random rand = new Random();
GeneticGenerator gen;
GeneticObserver obs;
double tempo = 105.0;

/**
 * Sets up the JMusic arrangement.
 */
public JazzGen() {
    gen = new GeneticGenerator(this);
}

public void setObserver(GeneticObserver obs) {
    this.obs = obs; this.gen.obs = obs;
}

public void removeObserver() {
    this.obs = null; this.gen.obs = null;
}

/**
 * Generates a new phrase.
 */
public void generate() {
    score = new Score("JazzGen", tempo);
    pPiano = new Part("Piano", 0, 0);
    phSolo = new Phrase(0.0); // first beat
    try {
        gen.run(phSolo);
    }
    catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
    postPass(phSolo);
    pPiano.addPhrase(phSolo);
    phChords = new CPhrase(0.0);
    genChords();
    pPiano.addCPhrase(phChords);
    score.addPart(pPiano);

    pBass = new Part("Bass", ACOUSTIC_BASS, 1);
    phBass = new Phrase(0.0);
    genBass();
    pBass.addPhrase(phBass);
    score.addPart(pBass);

    pDrums = new Part("Drums", 0, 9); // force drum channel (10)
    phBD = new Phrase(0.0);
    phSD = new Phrase(0.0);
    phHH = new Phrase(0.0);
    genDrumLoop();
    score.addPart(pDrums);
}

```

```

/**
 * Play score via MIDI.
 */
public void play() {
    Play.stopMidi();
    Play.midi(score, false);
}

/**
 * Write MIDI to a supplied output stream.
 */
public void writeMidi(OutputStream out) {
    Write.midi(score, out);
}

/**
 * Write LilyPond engraving to the supplied file;
 */
public void writeLilyPond(File out) throws IOException {
    LilyPondOutput.convert(score, prog, out);
}

/**
 * Simple wandering over the chord progression, always playing
 * the chord root note.
 */
public void genBass() {
    // Blithely walk up and down the current chord
    for(int i = 0; i < prog.length(); i++) {
        int[] durs = prog.get Durations(i);
        int j = 0;
        for(Chord c: prog.get(i)) {
            int dur = durs[j]*8 / phraseLen; // number of 8th notes to
            play
            phBass.addNote(new Note(c.rootNote + 36, QN, MF)); // root
            int[] chord = c.notes(3);
            int curPitch = 0;
            int dir = rand.nextBoolean() ? 1 : -1;
            for(int k = 0; k < dur - 1; k++) {
                curPitch += dir;
                if(curPitch >= chord.length) {
                    curPitch = chord.length - 1;
                    dir = -dir;
                }
                else if(curPitch < 0) {
                    curPitch = 0;
                    dir = -dir;
                }
                else if(rand.nextInt(2) == 0) {
                    dir = -dir;
                }
                phBass.addNote(new Note(chord[curPitch], QN, MF));
            }
            j++;
        }
    }
}

```

```

    }
}
// Add unison at end
Chord last = prog.getLast();
phBass.addNote(new Note(last.rootNote + 3*12, QN, F));
}

/**
 * Plays seventh chords.
 */
public void genChords() {
    for(int i = 0; i < prog.length(); i++) {
        int[] durs = prog.get Durations(i);
        int j = 0;
        for(Chord c: prog.get(i)) {
            int[] chord = c.notes(4);
            if(durs[j] == phraseLen) {
                // cover two bars
                phChords.addChord(chord, WN, MF);
                phChords.addChord(chord, WN, MF);
            }
            else {
                phChords.addChord(chord, durs[j]*2*WN/phraseLen, MF);
                // e.g. 16 maps to WN, 8 maps to HN
            }
            j++;
        }
    }
    // Add a unison chord at the end
    Chord last = prog.getLast();
    last.setInversion(0);
    phChords.addChord(last.unison(new int[]{4, 6}), WN, MF);
}

/**
 * Generates a simple 4/4 drum loop.
 */
public void genDrumLoop() {
    for (int i = 0; i < 4; i++) {
        phBD.addNote(new Note(BASS_DRUM_1, 0.666667, FF));
        phBD.addNote(REST, 1.333333);

        phSD.addNote(REST, 1.333333);
        phSD.addNote(new Note(ACOUSTIC_SNARE, 0.666667,
            (int)(rand.nextInt(60))));
    }

    // 'Cooking' ride pattern
    phHH.addNote(51, C);
    phHH.addNote(51, 0.67);
    phHH.addNote(51, 0.33);
    phHH.addNote(51, C);
    phHH.addNote(51, C);

```

```

    Mod.repeat(phBD, prog.length());
    Mod.repeat(phSD, prog.length());
    Mod.repeat(phHH, prog.length() * 2);

    pDrums.addPhrase(phBD);
    pDrums.addPhrase(phSD);
    pDrums.addPhrase(phHH);
}

/**
 * Apply post-pass stage to melody line.
 */
public void postPass(Phrase p) {
    Mod.tieRests(p);
    Mod.fillRests(p);
    //Mod.tiePitches(p);

    // Apply dynamics + small random offset
    randomWalkVel(p);
    Mod.accents(p, 4.0); // 4 crotches per bar
}

// Apply velocity to phrase via random walk
public void randomWalkVel(Phrase p) {
    int vel = -1;
    int lastPitch = -1;
    int dir = 1;
    for(Note n: p.getNoteArray()) {
        if(n.getPitch() != -1) {
            double dur = n.getDuration();
            if(dur >= 2.0 || vel == -1 || lastPitch == -1 ||
               Math.abs(lastPitch - n.getPitch()) > 12) {
                // Reseed velocity (new phrase fragment)
                vel = Math.max(0, Math.min(127,
                    (int)Math.round(rand.nextGaussian()*20.0 + 90.0)));
                dir = rand.nextBoolean() ? 1 : -1;
            }
            else {
                // Shift velocity (random walk) -- long dur -> larger
                // change
                vel += dir *
                    (int)Math.round(Math.abs(rand.nextGaussian()*dur*15.0));
                // If vel reaches edge, "bounce" it; randomly change
                // walk direction
                if(vel > 120) {
                    vel = 120 - (vel - 120);
                    dir = -1;
                }
                else if(vel < 80) {
                    vel = 80 + (80 - vel);
                    dir = 1;
                }
                else if(rand.nextDouble() < dur*0.3) { // longer dur ->
                    // more likely to change

```



```

        dir = -dir;
    }
}
n.setDynamic(vel);
n.setOffset(n.getOffset() - 0.001 + (Math.random()*0.002));
lastPitch = n.getPitch();
    }
}
}
}
}

```

## GeneticGenerator.java

This is the genetic algorithm. Triggered by JazzGen, it interfaces with many other classes to initialise the population, select, mutate, crossover and evaluate the fitness of each generation. Finally, it returns a **Phrase** object containing the melody line.

```

import jm.util.*;
import jm.music.data.*;
import jm.music.tools.*;
import java.util.*;

/**
 * Runs a genetic algorithm to generate a jazz solo over a given chord
 * progression.
 *
 * Coding scheme: 0..127 indicates MIDI notes; -1..-128 is a rest (-1
 * 'canonical')
 * @author Richard Harris
 */
public final class GeneticGenerator implements JGConstants {
    /* Population size (default: same as chromosome length) */
    int populationSize;
    /* Chance of mutation, per child */
    double mutationProb;
    /* Number of 'most fit' parents to transfer to new generation unmodified
    */
    int elitismSel;
    /* Tournament size for tournament selection. */
    int tourSize;
    /* Max. number of generations to run per block. */
    int maxGenerations;
    /* Max. history of successful phrases to remember. */
    int histSize;
    /* Max. previous phrases to include in fitness evaluation ( < histSize)
    */
    int maxPrevPhrases;
    /* % chance of an initialised phrase being one from the history. */
    double repeatPhraseProb;
    /* Tendency to choose more recent phrases when repeating (1=even,
    >1=skew) */
    double repeatPhraseSkew;

```

```

/* Genetic methods */
FitnessFunction fitFunc;
MutationFunction mutFunc;
CrossoverMethod crossoverMethod;
InitMethod initMethod;

ChordProgression prog;
Random rand;
int[] fitness;
GeneticObserver obs;
JazzGen jg;
int progIndex;
int gen; // generation
byte[] history;

public GeneticGenerator(JazzGen jg) {
    this.jg = jg;
    this.rand = jg.rand;

    fitFunc = new FitnessFunction(this);
    mutFunc = new MutationFunction(rand);
}

/**
 * Generates a piano solo for the given chord progression.
 */
public void run(Phrase out) throws Exception {
    prog = jg.prog; // copy over chord progression
    // Generate one block at a time
    history = new byte[phraseLen * histSize];
    fitness = new int[populationSize];
    fitFunc.openLog();
    for(progIndex = 0; progIndex < prog.length(); progIndex++) {
        byte[][] population = generateSeeds(history);
        Arrays.fill(fitness, NoFitness);
        fitFunc.newWeights();
        int[] weights = new int[fitFunc.enabled.length];
        for(int i = 0; i < fitFunc.enabled.length; i++) {
            weights[i] = fitFunc.enabled[i].weight;
        }

        for(gen = 0; gen <= maxGenerations; gen++) {
            calculateFitness(population);
            int[] rankedFitness = sortFitness();
            byte[][] newgen = new byte[populationSize][phraseLen];

            // If elitism is enabled, copy over some candidates
            untouched
            for(int i = 0; i < elitismSel; i++) {
                newgen[i] = population[
                    rankedFitness[rankedFitness.length - 1 - i] ];
                fitness[i] = fitness[ rankedFitness[rankedFitness.length
                    - 1 - i] ];
            }
        }
    }
}

```

```

    }

    // Generate offspring via random parent crossovers + mutate
    for(int i = elitismSel; i < populationSize; i++) {
        byte[] mom = population[select()];
        byte[] pop = population[select()];
        byte[] child = new byte[phraseLen];
        crossoverMethod.crossover(mom, pop, child);
        if(rand.nextDouble() < mutationProb)
            mutFunc.mutate(child);
        newgen[i] = child;
    }

    // Replace old gen with new gen
    population = newgen;
    // Unset fitness for all but elitism (-> popSize - 1)
    Arrays.fill(fitness, elitismSel, populationSize, NoFitness);
}

// Pick the fittest phrase and add it
int bestIdx = getFittest(population, out);
addBlock(out, population[bestIdx]);

// Rotate history
System.arraycopy(history, phraseLen, history, 0, phraseLen *
    (histSize - 1));
System.arraycopy(population[bestIdx], 0, history, phraseLen *
    (histSize - 1), phraseLen);
if(obs != null) obs.updateProgress(progIndex + 1);
}
fitFunc.closeLog();
}

/**
 * Evaluate the fitness of all members, and return the index of the
 * candidate with the highest score.
 */
private int getFittest(byte[][] pop, Phrase out) {
    calculateFitness(pop);
    int bestFit = NoFitness;
    int bestIdx = 0;
    for(int i = 0; i < populationSize; i++) {
        if(fitness[i] > bestFit) {
            bestFit = fitness[i];
            bestIdx = i;
        }
    }
    return bestIdx;
}

/**
 * Return a selection from the fitness table via Tournament Selection. A
 * random

```

```

    * sample of the population is selected, and the highest fitness member
      is
    * returned.
    */
private int select() {
    int maxFitness = NoFitness;
    int maxIndex = 0;
    for(int i = 0; i < tourSize; i++) {
        int idx = rand.nextInt(populationSize);
        if(fitness[idx] > maxFitness) {
            maxFitness = fitness[idx];
            maxIndex = idx;
        }
    }
    return maxIndex;
}

/**
 * Calculate the fitness of the entire population, if it
 * has not already been calculated.
 */
private void calculateFitness(byte[][] population) {
    // e.g. if progIndex = 2, we can only use 2 previous phrases
    int prevPhrases = Math.min(progIndex, Math.min(histSize,
        maxPrevPhrases));
    byte[] fitSubject = new byte[phraseLen * (prevPhrases + 1)];
    try{
        System.arraycopy(history, phraseLen * (histSize - prevPhrases),
            fitSubject, 0, phraseLen * prevPhrases);
    }catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Fitness subject copy error: " +
            Arrays.toString(new int[]{progIndex, maxPrevPhrases,
                prevPhrases, fitSubject.length, histSize,
                phraseLen*(histSize-prevPhrases)}));
        System.exit(1);
    }
    int totalFitness = 0;
    int bestFitness = 0;
    int tfSize = 0;
    // Also store totals of each module score
    float scores[] = new float[fitFunc.enabled.length];

    for(int i = 0; i < population.length; i++) {
        if(fitness[i] == NoFitness) {
            // Fitness is unset, so calculate it
            System.arraycopy(population[i], 0, fitSubject, phraseLen *
                prevPhrases, phraseLen);
            fitness[i] = fitFunc.evaluate(population[i], fitSubject,
                prevPhrases + 1, scores);
            totalFitness += fitness[i];
            tfSize++;
            if(fitness[i] > bestFitness) bestFitness = fitness[i];
        }
    }
    fitFunc.writeLog(gen, scores, totalFitness, bestFitness, tfSize);
}

```

```

}

/**
 * Return an array of population indices, ranked by
 * fitness ascending.
 */
private int[] sortFitness() {
    int[] ordering = new int[fitness.length];
    for(int i = 0; i < ordering.length; i++) {
        ordering[i] = i;
    }
    // Perform quicksort on fitness; their associated indices
    // (population members) are also sorted.
    TwoArrayQS.quicksort(fitness.clone(), ordering, 0, fitness.length -
        1);
    return ordering;
}

/**
 * Generates the initial candidates for the genetic algorithm. These
 * candidates
 * should at least resemble acceptable outputs, and be as varied as
 * possible.
 */
private byte[][] generateSeeds(byte[] history) {
    byte[][] pop = new byte[populationSize][phraseLen];
    initMethod.newPhrase(this, phraseLen);
    for(int i = 0; i < populationSize; i++) {
        if(progIndex > 0 && rand.nextDouble() < repeatPhraseProb) {
            // Copy a phrase from the history
            pop[i] = new byte[phraseLen];
            int maxHist = Math.min(progIndex, histSize); // available
                history
            double x = Math.random();
            x = Math.pow(x, repeatPhraseSkew); // prefer more recent
                melodies
            int offset = maxHist - (int)Math.floor(x*maxHist) - 1; //
                history offset (0 = most recent)
            int histIndex = histSize - offset - 1; // convert so 0 =
                first block, histSize - 1 most recent

            try {
                System.arraycopy(history, phraseLen*histIndex, pop[i], 0,
                    phraseLen);
            } catch (IndexOutOfBoundsException e) {
                System.out.println("Problem using history as seed!");
                System.out.println("maxHist = " + maxHist + "; x = " + x
                    + "; offset = " + offset + "; histIndex = " +
                    histIndex + "; history array size = " +
                    history.length);
                System.exit(1);
            }
        }

        if(rand.nextBoolean() == true) {

```

```

        // Transpose candidate to fit new chord
        for(int j = 0; j < phraseLen; j++) {
            if(pop[i][j] != Rest) {
                byte oldRoot = prog.getChordAt(progIndex -
                    offset - 1, j).rootNote;
                byte newRoot = prog.getChordAt(progIndex,
                    j).rootNote;
                int shift = newRoot - oldRoot;
                if(shift <= -7) shift += 12;
                if(shift >= 7) shift -= 12;
                pop[i][j] = (byte)Math.min(127, Math.max(0,
                    pop[i][j] + shift));
            }
        }
    }
    else {
        // Generate one according to the selected initialisation
        // method
        pop[i] = initMethod.generate();
    }
}
return pop;
}

/**
 * Add the supplied block to the output. Pitch values are converted to
 * jMusic equivalents, and a swing rhythm is applied to note durations.
 */
public void addBlock(Phrase p, byte[] block) {
    // Represents a 16th note
    double twoEightsDur = (EN*16/phraseLen)*4;
    double[] share = new double[] { 0.3, 0.25, 0.25, 0.2 };

    for(int i = 0; i < block.length; i++) {
        int pitch = (block[i] == Rest) ? REST : (int)block[i];

        // Set duration to a share (4/10..1/10) of 4 16ths (2 8ths)
        double duration = twoEightsDur*share[i % 4];

        p.addNote(pitch, duration);
    }
}
}

```

## C.2 Genetic algorithm method classes

### CrossoverDoublePoint.java

This is the basic double-point crossover used for most of the examples in this document.

```

/**
 * The beginning and end of the child are taken from the first parent, and
 * the rest
 * from the second.
 */
public class CrossoverDoublePoint extends CrossoverMethod {
    public void crossover(byte[] mom, byte[] pop, byte[] child) {
        int cut1 = rand.nextInt(child.length);
        int cut2 = rand.nextInt(child.length);
        if(cut1 > cut2) { int t = cut1; cut1 = cut2; cut2 = t; }
        for(int i = 0; i < child.length; i++) {
            child[i] = (i <= cut1) ? mom[i] : (i > cut2 ? mom[i] : pop[i]);
        }
    }
}

```

## InitChaotic.java

The chaotic initialiser, used with the Standard map. See also InitOneOverF.java for the 1/f generation used in most examples.

```

/**
 * Generate one sequence of pitches randomly, using the Chirikov
 * standard map mapped to pitches and durations.
 */
public class InitChaotic extends InitMethod {
    double I, T, K;
    static final double TWO_PI = 2*Math.PI;
    // Durations that can be randomly selected based on chaotic function (1
    // = 16th)
    static byte[] durations = { 1, 2, 3, 4, 6, 8, 16 };
    // Chance, out of 16, that dur will be selected
    static int[] durWeights = { 2, 8, 0, 4, 1, 1, 0 };
    static int weightSum = 16;

    public byte[] generate() {
        byte[] seed = new byte[curLength];
        int pos = 0;
        byte[] scale = getScale(0);
        byte pitch = (byte)(scale[0] + scale.length*5);
        K = 0.5 + (rand.nextDouble() % 2.0);
        I = 2; T = 2;

        while(pos < curLength) {
            standardMap();
            pitch = (byte)Math.max(0, Math.min(255, pitch - 3 +
                (int)Math.floor(T / (TWO_PI/(7))))); // -3..3
            // Convert pitch number to MIDI pitch via scale
            scale = getScale(pos);
            seed[pos++] = (byte)(Math.floor(pitch / scale.length)*12 +
                scale[pitch % scale.length]);
            double Ip = (Math.min(3, Math.max(1.5, I)) - 1.5) / 1.5;

```

```

        byte rhythmSel = (byte)Math.floor(Ip * 15);
        int i = 0;
        // Select duration index i according to weights
        while(rhythmSel >= durWeights[i]) {
            rhythmSel -= durWeights[i];
            i++;
        }
        for(int j = 0; j < Math.min(curLength - pos, durations[i]);
            j++) {
            seed[pos++] = Rest;
        }
    }
    return seed;
}

private void standardMap() {
    I += K*Math.sin(T);
    T += I;

    I = Math.abs(I % TWO_PI);
    T = Math.abs(T % TWO_PI);
}
}

```

## MutationFunction.java

Contains all of the mutation methods, selected at random. User interface is handled by MutationFunctionUI.java.

```

import java.util.*;

/**
 * Randomly mutates a musical phrase.
 */
public class MutationFunction implements JGConstants {
    Random rand;

    // List of available mutations
    static final String[] fnNames = {
        "Sort pitches ascending",
        "Sort pitches descending",
        "Reverse pitches",
        "Reverse phrase",
        "Rotate pitches",
        "Rotate phrase",
        "Transpose",
        "Octave transpose",
        "Restify",
        "Derestify",
        "Random pitch change",
        "Normal dist. pitch change"
    }
}

```



```

};

// Records which mutations have been disabled by the user
boolean[] fnEnabled = new boolean[fnNames.length];

public MutationFunction(Random r) { this.rand = r; }

/**
 * Apply a random mutation to the musical phrase.
 */
public void mutate(byte[] p) {
    int numChoices = 0, fnIndex;
    for(boolean b: fnEnabled) {
        if(b) numChoices++;
    }
    if(numChoices == 0) return; // no mutations enabled

    do {
        fnIndex = rand.nextInt(fnNames.length);
    } while(!fnEnabled[fnIndex]);

    int noteCount;
    byte[] pitches, lengths;
    int irl;

    switch(fnIndex) {
        case 0: // Sort pitches ascending
            sort(p, true);
            break;
        case 1: // Sort pitches descending
            sort(p, false);
            break;
        case 2: // Reverse pitches
            noteCount = getNoteCount(p);
            if(noteCount > 1) {
                pitches = new byte[noteCount];
                lengths = new byte[noteCount];
                irl = PhraseInfo.rhythmTransform(p, pitches, lengths);
                reverse(pitches);
                reassemble(p, pitches, lengths, irl);
            }
            break;
        case 3: // Reverse phrase - including rests
            reverse(p);
            break;
        case 4: // Rotate pitches
            noteCount = getNoteCount(p);
            if(noteCount > 1) {
                pitches = new byte[noteCount];
                lengths = new byte[noteCount];
                irl = PhraseInfo.rhythmTransform(p, pitches, lengths);
                rotate(pitches, noteCount == 2 ? 1 :
                    (rand.nextInt(pitches.length - 2) + 1));
                reassemble(p, pitches, lengths, irl);
            }
    }
}

```

```

    }
    break;
case 5: // Rotate phrase
    rotate(p, rand.nextInt(p.length - 2) + 1);
    break;
case 6: // Transpose
    transpose(p, -3 + rand.nextInt(7));
    break;
case 7: // Octave transpose
    transpose(p, 12 * (rand.nextBoolean() ? -1 : 1));
    break;
case 8: // Restify - randomly replaces notes with rests
    for(int i = 0; i < p.length; i++) {
        if(p[i] != Rest && rand.nextDouble() < 0.2) {
            p[i] = Rest;
        }
    }
    break;
case 9: // Derestify - randomly replace rests with notes
    noteCount = getNoteCount(p);
    byte[] minMax = noteCount > 0 ? getMinMax(p) : new byte[] {
        48, 72 };
    for(int i = 0; i < p.length; i++) {
        if(p[i] == Rest && rand.nextDouble() < 0.2) {
            // Random note is within range of min/max pitch in
            // entire phrase
            p[i] = (byte)(minMax[0] + rand.nextInt((minMax[1] -
                minMax[0]) + 1));
        }
    }
    break;
case 10: // Random pitch change
    for(int i = 0; i < p.length; i++) {
        if(p[i] != Rest && rand.nextDouble() < 0.2) {
            // Uniformly shifts pitch up or down
            p[i] = (byte)Math.min(127, Math.max(0, p[i] +
                (rand.nextInt(11) - 5)));
        }
    }
    break;
case 11: // Normal dist. pitch change
    for(int i = 0; i < p.length; i++) {
        if(p[i] != Rest && rand.nextDouble() < 0.2) {
            // Shifts pitch up or down according to normal
            // distribution
            p[i] = (byte)Math.min(127, Math.max(0, p[i] +
                (int)Math.round(rand.nextGaussian()*2)));
        }
    }
    break;
}
}

/**

```

```

    * Tranpose all pitches the given number of semitones.
    */
public static void transpose(byte[] p, int shift) {
    for(int i = 0; i < p.length; i++) {
        if(p[i] != Rest) p[i] = (byte)Math.min(127, Math.max(0, p[i] +
            shift));
    }
}

/**
 * Reverse an array.
 */
private static void reverse(byte[] p) {
    for(int left = 0, right = p.length - 1; left < right; left++,
        right--) {
        byte t = p[left]; p[left] = p[right]; p[right] = t; // swap
    }
}

/**
 * Sort the phrase's notes, keeping rhythm intact.
 */
private static void sort(byte[] p, boolean ascending) {
    int noteCount = getNoteCount(p);
    if(noteCount > 1) {
        byte[] pitches = new byte[noteCount];
        byte[] lengths = new byte[noteCount];
        int irl = PhraseInfo.rhythmTransform(p, pitches, lengths);
        Arrays.sort(pitches);
        if(!ascending) reverse(pitches);
        reassemble(p, pitches, lengths, irl);
    }
}

/**
 * Rotate an array to the right by a given number of place.
 * rotleft(x, n) == rotright(x, |x| - n)
 * 0 < amt < |p|
 */
private static void rotate(byte[] p, int amt) {
    byte[] chunk = new byte[amt];
    // Copy last amt bytes to chunk, shift p, copy chunk to head
    System.arraycopy(p, p.length - amt, chunk, 0, amt);
    System.arraycopy(p, 0, p, amt, p.length - amt);
    System.arraycopy(chunk, 0, p, 0, amt);
}

/**
 * Reform a phrase from (pitch, length) tuples.
 */
private static void reassemble(byte[] p, byte[] pitches, byte[] lengths,
    int initRestLen) {
    int pos = 0; // position in phrase
    while(initRestLen-- > 0) p[pos++] = Rest;
}

```

```

        for(int i = 0; i < pitches.length; i++) {
            p[pos++] = pitches[i];
            for(int j = 0; j < lengths[i] - 1; j++) p[pos++] = Rest;
        }
    }

    /**
     * Return the number of (non-rest) notes in the phrase.
     */
    private static int getNoteCount(byte[] p) {
        int noteCount = 0;
        for(byte pitch: p) {
            if(pitch != Rest) noteCount++;
        }
        return noteCount;
    }

    /**
     * Returns the minimum and maximum pitch values in the phrase.
     */
    private static byte[] getMinMax(byte[] phrase) {
        byte min = 127;
        byte max = 0;
        for(byte p: phrase) {
            if(p != Rest) {
                if(p > max) max = p;
                if(p < min) min = p;
            }
        }
        return new byte[] { min, max };
    }
}

```

## C.3 Fitness modules

### FmKey

A very simple fitness function (UI and other class methods removed) that simply returns the % of notes that are on-key.

```

public float evaluate(PhraseInfo p) {
    int score = 0;
    // Only consider last phrase
    for(int i = 0; i < phraseLen; i++) {
        if(p.phrase[i] != Rest && p.getChordAt(i).scaleIndex(p.phrase[i]) >=
            0) {
            score++;
        }
    }
    if(p.noteCount == 0) return 0.0f; // no basis for evaluation
    return ((float)score/p.noteCount);
}

```

```
}

```

## FmExpectancy2

Evaluates melodic expectancy according to Schellenberg's simplified model. `regDirCE`, `regRetCE` and `proxCE` are weightings for each of the three metrics involved.

```
public float evaluate(PhraseInfo p) {
    byte[] pitch = p.pitches;
    float score = 0;

    if(pitch.length < 3) return 0.5f; // not enough notes to work with

    // Operate over groups of three items
    for(int i = 2; i < pitch.length; i++) {
        int implInt = pitch[i - 2] - pitch[i - 1]; // implicative interval
        int realInt = pitch[i] - pitch[i - 1]; // realised interval
        int implSgn = (int)Math.signum(implInt);
        int realSgn = (int)Math.signum(realInt);
        implInt = Math.abs(implInt);
        realInt = Math.abs(realInt);

        // Registrational direction
        // If the impl. int is large, assign 1 if different dir, -1 if same dir
        if(implInt > 6) {
            score += regDirCE * ((realSgn != implSgn) ? 2 : 0);
        }
        else score += regDirCE;

        // Registrational return
        // 1. Realised interval must be in different direction
        // 2. Real. interval must be within 2 semitones of implicative interval
        if(realSgn != implSgn && realInt != 0 && implInt != 0) {
            if(Math.abs(implInt - realInt) <= 1) score += regRetCE;
        }

        // Proximity
        if(realInt > 0 && realInt < 12)
            score += proxCE * (12 - realInt);
        else if(realInt == 0) {
            score += proxCE * 5;
        }
    }

    // Max possible score
    float maxScore = (regDirCE * 2 + proxCE * 12 + regRetCE) * (p.noteCount - 2);

    // Normalisation

```

```

    // Return a fitness value based on proximity of exp. to user-defined
    // target value
    float tExp = targetExp / 100.0f;
    return (float)(1.0 - Math.pow(Math.abs((score/maxScore) - tExp)/tExp,
        punishFactor));
}

```

## FmRepetition2

Checks for similar melodic strings within a block, awarding partial scores for near-matches.

```

public float evaluate(PhraseInfo p) {
    if(p.noteCount < 6) return 0.0f; // no basis for repetition
    int maxScore = 0;
    try{

        // Pick two starting points in the phrase
        // Repetition is realistically at least three notes
        for(int a = 0; a < p.noteCount - 6; a++) {
            for(int b = a + 3; b < p.noteCount; b++) {
                // for(scale transpose?) TODO
                int pos = 0;
                int score = 0;
                // Keep increasing pos til we don't see a match
                while(a + pos < p.noteCount && b + pos < p.noteCount
                    && Math.abs(p.pitches[a + pos] - p.pitches[b + pos]) <= 3) {
                    score += 3 - Math.abs(p.pitches[a + pos] - p.pitches[b +
                        pos]);
                    if(p.pitches[a + pos] == p.pitches[b + pos])
                        score += 2;

                    pos++;
                }
                if(score > 25) score = 12; // punish > 5 note repetition
                maxScore = Math.max(maxScore, score);
            }
        }
    } catch(Exception e) { System.out.println(e.getMessage());
        e.printStackTrace(); }
    return (float)maxScore/25.0f;
}

```

## C.4 Chord progression

### ChordProgression.java

This larger class implements voice leading, chord progression parsing, and includes a test method to ensure that both are correct.

```

import jm.util.*;
import jm.music.data.*;
import jm.music.tools.*;
import jm.midi.MidiSynth;
import java.util.*;
import java.util.regex.*;

/**
 * Represents the solo's chord progression, handles user input parsing
 * and encapsulates information about scale and chord note values.
 */
public class ChordProgression implements JGConstants {
    List<List<Chord>> chords;

    /** Map user-input strings to chord types. Many-to-one. */
    static Map<String, ChordType> cmap = new HashMap<String, ChordType>();

    /**
     * Parses a chord progression in the form C7 D7 / Fmaj7 / Ealt ...
     * The chord progression will be repeated the specified number
     * of times.
     */
    public ChordProgression(String progText, int repeat) throws Exception {
        String parseProg = "";
        while(repeat-- > 0) parseProg += progText + "\n";
        parseProgText(parseProg);
        calculateInversions();
    }

    public ChordProgression(String progText) throws Exception {
        this(progText, 0);
    }

    /**
     * Return the exact chord at phrase _phrase_, position _pos_.
     */
    public Chord getChordAt(int phrase, int pos) {
        assert pos < phraseLen;
        assert phrase < chords.size();
        try {
            List<Chord> bar = chords.get(phrase);
            int[] durs = get Durations(phrase);
            int i = 0;
            while(pos >= durs[i]) pos -= durs[i++];
            return bar.get(i);
        }
        catch(Exception e) {
            System.out.println("getChordAt:" + phrase + "/" + pos);
            // trigger error again
            List<Chord> bar = chords.get(phrase);
            System.exit(1);
        }
        return null;
    }
}

```

```

/**
 * Returns a list of all chord lengths (in units) for position pos.
 */
public int[] getDurations(int pos) {
    List<Chord> bar = chords.get(pos);
    int size = bar.size();
    int[] durs = new int[size];
    int pow2 = 1; while(pow2 < size) pow2 *= 2; // Round size up to
        nearest power of two
    int unitSize = phraseLen/pow2;

    // Now assign each chord a size unitSize, assigning the last chord
    any
    // remaining space
    for(int i = 0; i < size - 1; i++) durs[i] = unitSize;
    durs[size - 1] = phraseLen - (size - 1)*unitSize;

    return durs;
}

/* Parse a String chord progression */
private void parseProgText(String progText) throws Exception {
    Pattern chordPat = Pattern.compile("(?i)^([a-g](?:#|b)?)(.*)$");
    chords = new ArrayList<List<Chord>>();
    List<Chord> curList = new ArrayList<Chord>();
    chords.add(curList);

    StringTokenizer toks = new StringTokenizer(progText, " \t\n\r\f/",
        true);
    while(toks.hasMoreTokens()) {
        String tok = toks.nextToken().toLowerCase();
        if(tok.equals("/") || tok.equals("|") || tok.equals("\n")) {
            // Bar delimiter
            if(!curList.isEmpty()) { // possible that someone would
                type /, \n
                curList = new ArrayList<Chord>();
                chords.add(curList);
            }
        }
        else {
            tok = tok.trim();
            if(tok.length() > 0) { // Chord symbol
                Matcher m = chordPat.matcher(tok);
                if(!m.matches()) throw new Exception("Invalid chord: " +
                    tok);

                // Get pitch class
                byte pitch = -1;
                String note = m.group(1);
                for(int i = 0; i < pitchNameFull.length; i++) {
                    if(note.equals(pitchNameFull[i]))
                        pitch = pitchNameIndexes[i];
                }
            }
        }
    }
}

```



```

        if(pitch == -1) throw new Exception("Invalid pitch class
            for chord: " + tok);

        // Get chord type
        ChordType ct = cmap.get(m.group(2));
        if(ct == null) throw new Exception("Chord " + tok + " is
            not in dictionary.");

        curList.add(new Chord(pitch, ct));
    }
}

// Finally, strip out empty chord lists
for(int i = 0; i < chords.size(); i++) {
    if(chords.get(i).isEmpty()) {
        chords.remove(i);
    }
}

/**
 * Sets chord inversions such that voice leading is optimal i.e.
 * notes between chords remain in close proximity.
 */
private void calculateInversions() {
    Chord prev = null;
    // First chord is played in root position
    for(int i = 0; i < chords.size(); i++) {
        List<Chord> bar = chords.get(i);
        for(Chord chord: bar) {
            if(prev != null) {
                int bestInv = getBestInversion(chord, prev);
                chord.setInversion(bestInv);
            }
            prev = chord;
        }
    }
}

/**
 * Return the best inversion to use for a chord, given a previous chord.
 * Takes into account adjacency of notes and octave range.
 */
private int getBestInversion(Chord cur, Chord prev) {
    cur = cur.clone(); // we're modifying this
    int[] prevNotes = prev.notes(3);
    int bestInv = 0; // if all else fails, don't invert
    int bestInvScore = 0;
    for(int inv = -4; inv < 5; inv++) {
        boolean outOfRange = false;
        cur.setInversion(inv);
        int[] curNotes = cur.notes(3);
        int score = 0;

```

```

        // Check each pair of notes
        for(int curNote: curNotes) {
            for(int prevNote: prevNotes) {
                if(curNote <= C2 || curNote >= G3) outOfRange = true;
                int diff = Math.abs(curNote - prevNote);
                score += (30/(diff+1)); // encourage closer notes
            }
        }
        if(outOfRange) score /= 2; // discourage chords muddying solo
        lines
        // If score is equal to current best, randomly decide whether to
        // favour it
        if(score > bestInvScore || (score == bestInvScore &&
            Math.random() <= 0.5)) {
            bestInvScore = score;
            bestInv = inv;
        }
    }
    return bestInv;
}

/* Returns the length of the chord progression, in bars */
public int length() { return chords.size(); }
public List<Chord> get(int i) { return chords.get(i); }

/* Returns the very last chord in the progression. */
public Chord getLast() {
    List<Chord> lastBlock = chords.get(chords.size() - 1);
    return lastBlock.get(lastBlock.size() - 1);
}

public String toString() { /* ... */ }

/* Test */
public static void main(String[] args) {
    try {
        ChordProgression cp = new ChordProgression(
            "C7 Ebmaj7 / C#7 D#7 | E7\nF7 | C C7 / C7 E7aug
            D7dim\nC7/E7+/C7+9 Dmaj7s11\nF C F C G", 1
        );
        System.out.println(cp);
        // Play the progression
        Score s = new Score(130.0); Part p = new Part(0, 0);
        CPhrase pp = new CPhrase(0.0);
        for(int i = 0; i < cp.chords.size(); i++) {
            List<Chord> bar = cp.chords.get(i);
            for(Chord chord: bar) {
                pp.addChord(chord.notes(4), HN, MF);
            }
        }
        p.addCPhrase(pp);
        s.addPart(p);
        //Play.midi(s);
        // Test duration form
    }
}

```

```
        for(int i = 0; i < cp.chords.size(); i++) {
            int[] durs = cp.get Durations(i);
            System.out.println("Bar " + i + ": " + Arrays.toString(durs));
        }
        // Print Lilypond
        System.out.println(LilyPondOutput.toLy(cp));

    } catch(Exception e) { System.out.println(e); }
}

/* Initialise the String -> ChordType map */
static {
    cmap.put("", ChordType.Cmaj);
    cmap.put("maj", ChordType.Cmaj);
    /* ... */
}
}
```

## Appendix D

# Running the code

A precompiled version can be found in the `bin` directory; jMusic must first be installed and added to the Java classpath. *JazzGen* can then be executed with `java JazzGenUI`. Alternately the source can be found in `src`; important files from the `bin` directory may need to be copied over for execution:

**lilypond.template** Required for Lilypond output.

**defaults.opt** Contains sensible options as derived in Chapter 5, Testing and Experimentation.

**fitness.xlsx** Excel 2007 spreadsheet used for stat logging. After generating a phrase `fitness.csv` will be created in the current directory. Copy this data into the first worksheet of `fitness.xlsx` and check the other worksheets for graphs.

## Usage information

After setting the options, click **Generate**. Once the progress bar is full, click **Play** to hear the composition. The user interface has been designed for personal experimentation only and may not be user friendly to others.

# References

- P. Bak, C. Tang, and K. Wiesenfeld. Self-organized criticality: An explanation of the  $1/f$  noise. *Physical Review Letters*, 59(4):381–384, 1987.
- R. Bidlack. Chaotic Systems as Simple (But Complex) Compositional Algorithms. *Computer Music Journal*, 16(3):33–47, 1992.
- J. Biles, P. Anderson, and L. Loggi. *Neural network fitness functions for a musical IGA*. International Computing Sciences Conferences (ICSC), 1996.
- J.A. Biles. GenJam: A genetic algorithm for generating jazz solos. *Proceedings of the 1994 International Computer Music Conference*, pages 131–137, 1994.
- M. Chemillier. Toward a formal study of jazz chord sequences generated by Steedmans grammar. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 8(9):617–622, 2004.
- EF Clarke. Generative Processes in Music. *The Psychology of Performance, Improvisation, and Composition*. Oxford Science Publications, 1988.
- H. Cohen. A Self-Defining Game for One Player: On the Nature of Creativity and the Possibility of Creative Computer Programs. *Leonardo*, 35(1):59–64, 2002.
- J. Cohen. Subjective probability. *Scientific American*, 197(5):128–138, 1957.
- J. Coker. *Improvising Jazz*. Prentice-Hall Englewood Cliffs, NJ, 1964.
- D. Conklin. Music generation from statistical models. *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences, Aberystwyth, Wales*, pages 30–35, 2003.
- D. Conklin and I.H. Witten. Multiple viewpoint systems for music prediction. *Journal of New Music Research*, 24(1):51–73, 1995.
- N. Cook. The perception of large-scale tonal closure. *Music Perception*, 5(2):197–206, 1987.
- G. Cooper and L.B. Meyer. *The Rhythmic Structure of Music*. University Of Chicago Press, 1960.

- David Cope. Panel discussion. In *Proceedings of the International Computer Music Conference*, 1993.
- LL Cuddy and CA Lunney. Expectancies generated by melodic intervals: perceptual judgments of melodic continuity. *Percept Psychophys*, 57(4):451–62, 1995.
- P. Desain and H. Honing. Tempo curves considered harmful. *Contemporary Music Review*, 7(2):123–138, 1993.
- C. Dodge and T.A. Jerse. *Computer Music: Synthesis, Composition, and Performance*. Macmillan Library Reference, 1985.
- W.J. Dowling and D.L. Harwood. *Music cognition*. Academic Press San diegom California, 1986.
- K. Ebcioglu. An Expert System for Harmonizing Four-Part Chorales. *Computer Music Journal*, 12(3):43–51, 1988.
- JJ Fux. *Gradus ad Parnassum*, 1725.
- PM Gibson and JA Byrne. NEUROGEN, musical composition using genetic algorithms and cooperating neural networks. *Artificial Neural Networks, 1991., Second International Conference on*, pages 309–313, 1991.
- D.E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, 2002.
- D.E. Goldberg et al. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Pub. Co Reading, Mass, 1989.
- H. Gotlieb and VJ Konecni. The effects of instrumentation, playing style, and structure in the Goldberg Variation by Johann Sebastian Bach. *Music Perception*, 3(1):87–102, 1985.
- M. Grachten. JIG: Jazz Improvisation Generator. *Proceedings of the MOSART Workshop on Current Research Directions in Computer Music*, pages 1–6, 2001.
- M. Grachten and J.L. Arcos. Using the Implication/Realization Model for Measuring Melodic Similarity. *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI*, 2004.
- M. Grachten, J.L. Arcos, and R.L. de Mantaras. Melody Retrieval using the Implication/Realization Model. *MIREX*, 2005.
- M. Gutknecht. The Postmodern Mind: Hybrid Models of Cognition. *Connection Science*, 4(3):339–364, 1992.
- K. Hevner. Experimental Studies of the Elements of Expression in Music. *The American Journal of Psychology*, 48(2):246–268, 1936.

- K. Hevner. The Affective Value of Pitch and Tempo in Music. *The American Journal of Psychology*, 49(4):621–630, 1937.
- J.H. Holland. *Adaptation in natural and artificial systems*, University of Michigan press. University of Michigan Press, 1975.
- A. Horner and D.E. Goldberg. Genetic algorithms and computer-assisted music composition. Technical report, Technical Report CCSR-91-20, Center for Complex Systems Research, The Beckman Institute, University of Illinois at UrbanaChampaign, 1991, 1991.
- A. Horner, J. Beauchamp, and L. Haken. Machine Tongues XVI: Genetic Algorithms and Their Application to FM Matching Synthesis. *Computer Music Journal*, 17(4):17–29, 1993.
- B.L. Jacob. Algorithmic composition as a model of creativity. *Organised Sound*, 1(03):157–165, 1996.
- Y. Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 9(1):3–12, 2005.
- P.N. Johnson-Laird. Jazz Improvisation: A Theory at the Computational Level. *Representing musical structure, London*, pages 291–325, 1991.
- K. Jones. Compositional Applications of Stochastic Processes. *Computer Music Journal*, 5(2):45–61, 1981.
- R. Keller and D. Morrison. A grammatical approach to automatic improvisation. *Proceedings, Fourth Sound and Music Conference, Lefkada, Greece, July*, 2007.
- S. Kim and E. André. Composing affective music with a generate and sense approach. *Proceedings of Flairs 2004-Special Track on AI and Music*, 2004.
- GM Koenig. Project One. *Electronic Music Report*, pages 32–46, 1970.
- C.L. Krumhansl. Music Psychology and Music Theory: Problems and Prospects. *Music Theory Spectrum*, 17(1):53–80, 1995.
- C.L. Krumhansl, J. Louhivuori, P. Toiviainen, T. Järvinen, and T. Eerola. Melodic expectation in Finnish spiritual folk hymns: Convergence of statistical, behavioral, and computational approaches. *Music Perception*, 17(2):151–95, 1999.
- C.L. Krumhansl, P. Toivanen, T. Eerola, P. Toiviainen, T. Järvinen, and J. Louhivuori. Cross-cultural music cognition: cognitive methodology applied to North Sami yoiks. *Cognition*, 76(1):13–58, 2000.
- J. Leach and J. Fitch. Nature, Music, and Algorithmic Composition. *Computer Music Journal*, 19(2):23–33, 1995.

- F. Lerdahl. *Tonal Pitch Space*. Oxford University Press, 2001.
- F. Lerdahl and R. Jackendoff. *A Generative Theory of Tonal Music*. MIT Press, 1996.
- D.G. Loy. Connectionism and Musiconomy. *Music and Connectionism*, 1991.
- Gareth Loy. Composing with computers: a survey of some compositional formalisms and music programming languages. In *Current directions in computer music research*, pages 291–396. MIT Press, Cambridge, MA, USA, 1989. ISBN 0-262-13241-9.
- M. Marques, V. Oliveira, S. Vieira, and AC Rosa. Music composition using genetic evolutionary algorithms. *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, 1, 2000.
- K. McAlpine, E. Miranda, and S. Hoggar. Making Music with Algorithms: A Case-Study System. *Computer Music Journal*, 23(2):19–30, 1999.
- J. McCormack. Grammar based music composition. *Complex Systems*, 1996.
- R.A. McIntyre. Bach in a box: The evolution of four-part baroque harmony using the genetic algorithm. *IEEE Conference on Evolutionary Computation*, pages 852–857, 1994.
- L. Meyer. *Emotions and meaning in music*. Chicago: The University of Chicago Press, 1956.
- J.K. Millar. *The aural perception of pitch-class set relations: a computer- assisted investigation*. PhD thesis, North Texas State University, 1984.
- B.L. Miller and D.E. Goldberg. Genetic Algorithms, Selection Schemes, and the Varying Effects of Noise. *Evolutionary Computation*, 4(2):113–131, 1996.
- Paul Morris. Programming a computer to play the 12-bar blues. BSc (Hons) dissertation, 2005.
- E. Narmour. *Beyond Schenkerism: The Need for Alternatives in Music Analysis*. University of Chicago Press, 1977.
- E. Narmour. *The Analysis and Cognition of Basic Melodic Structures: The Implication-Realization Model*. University of Chicago Press, 1990.
- E. Narmour. *The Analysis and Cognition of Melodic Complexity: The Implication-Realization Model*. University of Chicago Press, 1992.
- C. Palmer. Mapping musical thought to musical performance. *Journal of experimental psychology. Human perception and performance*, 15(2):331–346, 1989.
- G. Papadopoulos and G. Wiggins. A Genetic Algorithm for the Generation of Jazz Melodies. *SteP*, 98:7–9, 1998.



- G. Papadopoulos and G. Wiggins. AI Methods for algorithmic composition: A Survey, a Critical View and Future Prospects. *AISB Symposium on Musical Creativity*, 1999.
- D. Ralley. Genetic algorithms as a tool for melodic development. *Proceedings of the 1995 International Computer Music Conference*, pages 501–502, 1995.
- R.G. Reynolds. An Introduction to Cultural Algorithms. *Proceedings of the Third Annual Conference on Evolutionary Programming*, pages 131–139, 1994.
- RG Reynolds and W. Sverdlik. Problem solving using cultural algorithms. *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 645–650, 1994.
- C. Roads and P. Wieneke. Grammars as Representations for Music. *Computer Music Journal*, 3(1):48–55, 1979.
- M. Sabatella. A Jazz Improvisation Primer. *ADG Productions, Lawndale, CA*, 1995.
- E.G. Schellenberg. Expectancy in melody: tests of the implication-realization model. *Cognition*, 58(1):75–125, 1996.
- E.G. Schellenberg. Simplifying the implication-realization model of melodic expectancy. *Music Perception*, 14(3):295–318, 1997.
- H. Schenker. *Five Graphic Music Analyses*. Dover Publications, 1969.
- M.A. Schmuckler. Expectation in music: Investigation of melodic and harmonic processes. *Music Perception*, 7(2):109–150, 1989.
- B. Schottstaedt. *Automatic Species Counterpoint*. CCRMA, Dept. of Music, Stanford University, 1984.
- W.M. Spears and K.A. De Jong. On the virtues of parameterized uniform crossover. *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236, 1991.
- L. Spector and A. Alpern. Induction and Recapitulation of Deep Musical Structure. *Working Notes of the IJCAI-95 Workshop on Artificial Intelligence and Music*, pages 41–48, 1995.
- M. Steedman. The Blues and the Abstract Truth: Music and Mental Models. *Mental Models in Cognitive Science: Essays in Honour of Phil Johnson-Laird*, 1996.
- M.J. Steedman. A Generative Grammar for Jazz Chord Sequences. *Music Perception*, 2(1):52–77, 1984.
- G. Sywerda. Uniform crossover in genetic algorithms. *Proceedings of the third international conference on Genetic algorithms table of contents*, pages 2–9, 1989.

- WF Thompson, LL Cuddy, and C. Plaus. Expectancies generated by melodic intervals: evaluation of principles of melodic implication in a melody-completion task. *Percept Psychophys*, 59(7):1069–76, 1997.
- F. Tirro. Constructive Elements in Jazz Improvisation. *Journal of the American Musicological Society*, 27(2):285–305, 1974.
- N. Todd. Towards a cognitive theory of expression: The performance and perception of rubato. *Contemporary Music Review*, 4(1):405–416, 1989.
- P. Toiviainen. Symbolic AI versus connectionism in music research. *Readings in Music and Artificial Intelligence*. Amsterdam: Harwood Academic Publishers, 67, 2000.
- M. Unehara and T. Onisawa. Music composition system based on subjective evaluation. *Systems, Man and Cybernetics, 2003. IEEE International Conference on*, 1, 2003.
- R.F. Voss and J. Clarke. 1/f noise in music: Music from 1/f noise. *The Journal of the Acoustical Society of America*, 63:258, 1978.
- G.A. Wiggins, University of Edinburgh, and Dept. of Artificial Intelligence. *Evolutionary Methods for Musical Composition*. University of Edinburgh, Dept. of Artificial Intelligence, 1998.
- I. Xenakis. *Formalized music*. Indiana University Press Bloomington, 1971.