Algorithmic Composition of Big Band Music

Michael Tonge

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2007

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:

Algorithmic Composition of Big Band Music

Submitted by: Michael Tonge

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 Batchelor 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:

Abstract

There have been many attempts in the past, of varying success, to try and write programmes that compose music. In this project chaotic systems are used with a novel new harmonisation method to generate rhythm and harmonies and to integrate different passages of the music. Music, which consists of multiple parts, is produced and tested on a sample of people and found to be of respectable compositional quality, though lacking in structure. Therefore song structure is identified as one of the key aspects for future development. From this work it can be concluded that the novel harmonisation procedure used gives rise to some interesting properties that can be exploited within the compositional process.

"Since I have always preferred making plans to executing them, I have gravitated towards situations and systems that, once set into operation, could create music with little or no intervention on my part. That is to say, I tend towards the roles of planner and programmer, and then become an audience to the results"

Brian Eno

Contents

1 Introduction **1**

2 Literature Survey **3**

2.1 Big Bands . . . 3

2.1.1 Big Band Composition . . . 3

2.1.2 Big Band Musical Structures . . . 6

2.2 Compositional Methods . . . 8

2.2.1 Neural Networks . . . 8

2.2.2 Genetic Algorithms . . . 9

2.2.3 Chaotic Systems . . . 10

2.2.4 Others . . . 11

2.3 Evaluation and Method Selection . . . 13

2.4 A Closer Look at Chaotic Systems . . . 14

2.4.1 Conservative Systems . . . 15

2.4.2 Dissipative Systems . . . 15

2.5 A Closer Look at Heuristic Methods . . . 17

2.6 Development Environment . . . 18

2.6.1 CSound . . . 18

2.6.2 The C Programming Language . . . 18

2.6.3 Note Representation . . . 18

2.6.4 Conclusion . . . 19

3 Requirements & Design **20**

3.1 Output Form 20
3.2 Representation of Notes 22
3.3 Representation of Parts 24
3.4 Data Structures Used 25
3.5 Scales and Octaves 27
3.6 Chords 28
3.7 Dynamic Equations Implementation 29
3.8 The Note Creation Process 30
3.8.1 Bass Line 30
3.8.2 Melody 31
3.8.3 Counter Melody 32
3.8.4 Rolling Bass Line 32
3.8.5 Solo Sections 32
3.9 Project Requirements 33

4 Program Development **34**
4.1 Initial Groundwork 34
4.2 Melody 35
4.2.1 The Mapping 35
4.2.2 Sense of Rhythm 36
4.3 Rolling Bass Line 37
4.4 Solo Sections 39
4.5 Linking everything together 40
4.5.1 Chords 40
4.5.2 Harmonisation of Chords 40
4.5.3 Structure 41

5 Evaluation **42**
5.1 Questionnaire Results 42
5.1.1 Questionnaire Conclusions 45
5.1.2 Reflection 57

5.2 Overall Project Evaluation . . . 58
5.2.1 Subject Research . . . 58
5.2.2 Requirements & Design . . . 58
5.2.3 Program Implementation . . . 59

6 Conclusions **60**
6.1 Requirement Review . . . 61
6.1.1 Overall Requirements . . . 61
6.1.2 Melody Requirements . . . 62
6.2 Future Developments . . . 64
6.2.1 Dynamics . . . 64
6.2.2 Solo Part . . . 64
6.2.3 Harmonisation and Chord Selection . . . 64
6.2.4 Song Structure . . . 64
6.2.5 Final Section . . . 65
6.2.6 CSound instruments . . . 65
6.2.7 Links between musical parts . . . 65
6.2.8 Chords to Parts mapping algorithm . . . 66
6.2.9 Introduction Section . . . 66
6.2.10 Chaotic Systems Used . . . 66
6.3 Contribution & Relation to Other Work . . . 67
6.4 Final Conclusion . . . 68

A Code **71**
A.1 File: compose.c . . . 72
A.2 File: AmplitudeValidator.c . . . 111
A.3 File: Instruments.csd . . . 121
A.4 File: Instruments.csd . . . 130

B Questionnaire Results **133**
B.1 Questions . . . 133
B.2 Results . . . 135

List of Figures

5.1 Score part 1 42
5.2 Score part 2 43
5.3 Question 1 Results 45
5.4 Question 2 Results 46
5.5 Question 3 Results 47
5.6 Question 4 Results 48
5.7 Question 5 Results 49
5.8 Question 6 Results 50
5.9 Question 7 Results 51
5.10 Question 8 Results 52
5.11 Question 9 Results 53
5.12 Question 10 Results 54
5.13 Question 11 Results 55

Acknowledgements

Many thanks go to my supervisor Professor John Fitch for his advice and support, and to my girlfriend Charlotte for putting up with all manor of musical attempts being played at random times of the day.

Chapter 1

Introduction

The aim of this project is to construct a program that is capable of the composition of music for a big band in a swing style.

Swing music is an old style of music consisting of multiple sections with soloists. The general structure of a section of swing music can be considered to be of a twelve bar blues style. Therefore this project will make use of the work carried out by Paul Morris in a previous dissertation(Morris, 2005).

The standard form of a simple piece of swing music is as follows:

```
ABCB‘C‘B‘‘C‘‘ .... D
```

This means there are two main parts to the project.

The first is the generation of the twelve bar blues sections, B. This will be based closely on the work done by Paul Morris with a series of modifications designed to allow the choice of chordal progressions and the generation of multiple parts which have a harmonically correct sound when performed together.

The second is the generation of the solo sections, C. This will be based on the mapping of a chaotic system to notes. The chaotic system chosen for this is a standard map system. This was chosen due to its ability to generate values which could be mapped to music that could be considered interesting to a human listener.

The other more minor parts of this dissertation deal with the creation of sections A and D, the introduction and ending. These will be minor parts of the dissertation since they will be heavily based on sections B and C can therefore be added as a future improvement to the project if there is not time to produce them during this dissertation.

During the solo sections this project has a relative degree of freedom. This is because the music is improvised, or made up on the spot. Musicians will often say that their improvisations are based on certain patterns they have developed, rather than being just a set of random notes strung together. This means that improvisation is a very personal

way of playing and no two musicians will play the same. Therefore a major problem within the generation of these sections of music will be attempting to get a machine to model what is in actuality a method of playing which has different rules for every musician. This can also be considered a simplification since, from a musician's perspective, once the rules have been laid out, it is simply a matter of improvising music, no matter what the output produced is.

There are several possible methods for the generation of the music. The simplest and most trivial would be to simply string random notes of random lengths together for each part. This would lead to music being produced which had no structure and where discords were commonplace. This music would almost definitely not have the qualities of a big band playing swing music. Therefore a method that would impose some structure on the music must be selected. The method selected (reasoning is provided in the literature review2) is to use a combination of chaotic systems with heuristic methods. The chaotic systems were selected because they have been known to have the ability to add structure to music. Heuristic methods have been selected for some of the more trivial tasks such as chord selection in order to simplify the task faced by the author. A possible extension to this project would be to investigate the use of chaotic systems for both the generation of parts and the selection of chords.

In the past there have been varying attempts to program computers to compose music - two examples of this are the mathematical statistical approach and one based on chaotic systems. Iannis Xenakis used computers to generate music through a stochastic process (Alpern, 1995). This employed statistical and probabilistic methods to produce music from a list of note densities and probabilistic weights which used a random number generator for specific decisions.

Non-linear dynamic equations that are iterated can be used to generate music (Bidlack, 1992). There are three categories of behaviour of these equations: constant, where all points are the same; oscillatory, where all points stay within a set range and chaotic, where the points are seemingly random. The behaviour of the chaotic systems is most interesting since the data produced bears some resemblance to its past but will never be quite the same again. This method of generation was therefore selected for the generation of the moving parts of the music.

This project shall research the different chaotic methods in order to select the ones most suited to the generation of each part of the big band. The selected methods shall then be explored further and finally shall be used to complete the design and implementation of the final system. The final system design and implementation will be a very iterative process designed to develop the best overall sound. Once the program has been produced it will be tested by asking a number of people to listen to it and comment on the outputted music.

Chapter 2

Literature Survey

This section presents the review of the literature studied prior to, and during, this work.

2.1 Big Bands

This section aims to deal with the questions of what a big band is and what type of music it plays.

2.1.1 Big Band Composition

A big band is composed of seventeen separate parts. These parts are organised into three main sections:

- Rhythm Section
- Saxophone Section
- Brass Section

The brass section can also be considered to be two separate sections: the trumpet section and the trombone section. This distinction is useful because the trombone section can often be found playing in harmony with, or supporting, all other sections of the big band.

Rhythm Section

The rhythm section comprises drums, double bass (or bass guitar), piano and optionally guitar. Although not intended to be heard above the wind instruments, the rhythm section is essential both to the band and to the audience in providing the pulse in the music that is so important for dancers and listeners. The rhythm section is sometimes referred to as

the 'powerhouse' or engine room of the band, as one of its main purposes is to drive the band forward at a steady rate. The rhythm section is sometimes said to provide a large part of the 'swing' in a band. A poorly synchronised rhythm section will not swing and will sound stiff and awkward. This is a common problem with fledgling bands. When playing together properly, the rhythm section achieves what is known in electronics terms as phase-lock, being totally together in tempo and phase. Under these conditions, the rhythm section is said to be 'swinging'.

The drummer is an important member of the rhythm section, who, together with the bass, piano and optional guitar, forms the core of a solid timekeeping unit. The drummer plays fills that accent the horn figures, and provides the basis of the swing feel with a steady broken-triplet figure on the ride cymbal. The drum kit usually comprises bass drum, tom-tom(s), snare drum, a heavy ride cymbal, hi-hat or 'sock' cymbals, crash cymbal(s) and sometimes other specialty cymbals (splash, China boy, pang). The double bass's part should consist of a rolling bass line in a legato style. Bassists normally play a simple four crotchets to a bar part. When combined with the legato playing style, this generates a big band sound.

The guitar is mostly used as a pure rhythm instrument, playing straight time. That is, in a swing tune, the guitarist will often play four beats in every bar. Other music styles (such as ballads or Latin music) may be approached differently but will not be considered in this case, as the algorithm will not be developed to perform non swing types of music. The guitarist sometimes takes solos, but usually less often than the piano.

The role of the pianist in a big band depends on his/her style and the needs of the band. In some groups, such as that of Count Basie, the part played by the piano is minimal, in that the comping only contributes a light specification of the voicing of the chords. More contemporary big bands give a greater role to the piano in accompanying the band, by punctuating various accents; providing responses to other parts; playing countermelodies etc., as well as providing fills in the music. One example of a modern big band pianist is Roland Hanna with the Thad Jones/Mel Lewis Big Band.

Saxaphone Section

The saxophone section represents the 'backbone' of the wind instruments in that it frequently carries the tune or provides backing harmonies underneath a soloist or section solo. Saxes, when playing along with brass in an ensemble, are said to 'soften' the sound of the brass, while giving it support. There are many techniques which can be employed by a saxophone player, such as growling, glissando transitions, multiphonics, overtones or voicing. These techniques are only used occasionally in big band music and are therefore not considered worth modelling within this algorithm. The algorithm should allow easy improvement to include techniques such as these, though.

The lead saxophone plays the highest and most strenuous part. This does not usually include solo work, though, as this is usually given to the second alto or the first tenor

saxophone. The second alto is often an improvisation specialist. He/she will take a large number of the section solos.

The tenor saxophone plays a much greater part in jazz and big band music than any other musical type. It was the pioneering genius of Coleman Hawkins that lifted the tenor saxophone from the traditional role of adding weight to the ensemble, and established it as a highly effective melody instrument in its own right. Many of the greatest jazz musicians from the 1940s onwards have been tenor players.

The second tenor saxophone is a supporting part. It normally plays within its own section to add richer tonal colour and depth to the bass of the parts.

The baritone player usually plays rather simple rhythms in order to maintain the musical pulse of the band. Often, this consists of quarter notes on beats one and three in 4/4 time. The baritone can often be found doubling with the bass trombone, double bass or first alto saxophone to provide additional weight to the bass of the ensemble.

Brass Section

The brass section is a collective term for the trombone and trumpet sections. Quite often these sections play the same phrases and rhythms for a powerful, brassy sound.

The trumpet section usually comprises four (sometimes five) players, each playing a separate part. The section leader is usually the first (or lead) trumpet, who plays the highest and most strenuous part. The lead trumpet is considered the leader of the band when the whole band is playing tutti (in unison, or all the same). This means he/she will be followed in phrasing, articulation, etc., by the rest of the band. The second trumpet player is normally the jazz soloist and so will take a large number of the specialised solos within the music.

The third and fourth trumpets add to the depth of the rest of their section. They are very occasionally given solos.

The trombone section is similar in formation to the trumpet section, except that there are three tenor trombones and one bass trombone. The trombone section provides a deeper sound than that of the trumpets.

The lead tenor trombone is the leader of the section. He/she should be followed by the other section members in phrasing, articulation, etc. He/she plays the highest, most strenuous part in the section and takes some solos.

Either the second or third tenor trombone takes most of the section solos. This is dependant on the composer of the music. The second and third tenor trombones are normally used to add weight to the section sound or to add deeper, richer tonal colours to other band sections.

The Bass Trombone plays the lowest part in the big band and can play in parallel with any section of the band in order to provide a deeper or richer tonal colour.

2.1.2 Big Band Musical Structures

Typical big band Swing music is written in a strophic form, discussed in more detail below, with the same phrases and chord structures repeated several times (often across multiple instruments). Typically, the first chorus of a piece introduces the melody and then each subsequent chorus expands or modifies the original. Most pieces start with some kind of introduction, usually initiated by the rhythm section, with the other sections joining later.

Musical arrangements for big bands often use several common compositional techniques:

- Trumpet parts can be arranged in close harmony (a thickened line) to give a broader impression of the melody.
- Trumpets can play in unison, giving a powerful, penetrating sound that cannot be achieved by a single trumpet. Groups of two or three trumpets are sometimes used in simple harmony.
- The baritone saxophone may play the lead alto part an octave lower to reinforce the melody and provide an effective 'five part' harmony in close harmony saxophone soli.
- The baritone saxophone's part is sometimes written in with the trombones, (especially in bands without a bass trombone), to give extra richness at the bottom of the trombone section.
- On occasions, the baritone sax can combine with the double bass and bass trombone to create very heavy bass lines or riffs.

Strophic Form

Strophic form, or chorus form, is a sectional and/or additive way of structuring a piece of music based on the repetition of one formal section or block. It is the musical analogue of repeated stanzas in poetry or lyrics: where the text repeats the same rhyme scheme from one stanza to the next, the accompanying music is either the same or very similar from one stanza to the next.

Strophic formal music may be considered to have the structures:

```
AAA...
or AA'A''....
or ABA'B'....
```

Each chorus can take the form of a twelve bar blues section.

Twelve Bar Blues

The twelve bar blues is normally based around the chord progression:

```
T T T T
S S T T
D S T T
```

Here, T indicates the tonic, while S is the subdominant and D is the dominant.

These chords provide the basic blues sound and can be used to add a blues sound to any type of music. This is how some modern pop songs gain a slight blues feel without being classed as blues music. Many variations of this basic chord progression are possible, such as the use of seventh chords before a change, or the use of the dominant seventh in the last bar as a turnabout.

Within the blues, the solo part does not have to conform to the standard pattern of having four notes per beat. For example, Bessie Smith's "Empty Bed Blues":

```
D        -     -      -
Woke up this morning with an

G     -   -    -   D - - - D7 - - -
awful aching head

G        -     -      -
Woke up this morning with an

G7    -   -    -   D - - - D7 - - -
awful aching head

A     -        -     A7
My new man had left me

G    -    -     G7 D - - - D - A A7
just a room and an empty bed.
```

2.2 Compositional Methods

Early programs written to compose music were based on random number generators. Probably the most famous of these early compositions was the "Illiac Suite for String Quartet" by Lejaren Hiller and Leonard Isaacson. These early pieces were composed on the principles of the Markov process. The Markov process involves using past events to predict future events. This process was plagued by having either too much, or too little, knowledge of the past. If past knowledge was insufficient then "the result sounded as random as if the notes had simply been selected from an arbitrary set of weighted probabilities" (Smoliar, 1991). If past knowledge was excessive then "what was predicted was nothing more than a replication of the original data from which the probabilities were computed" (Smoliar, 1991).

Many different researchers have attempted to improve this situation with new methods. These encompass ideas such as the use of probabilities; stochastic matrices and heuristics; neural networks and learning algorithms (artificial intelligence); genetic algorithms and chaotic systems. Each of these possible methods will be discussed in more detail below.

2.2.1 Neural Networks

An interesting possibility for the generation of Swing music based on the twelve bar blues is the use of Long Short-Term Memory (LSTM) and Recurrent Neural Networks (RNNs) as proposed by Eck and Schmidhuber (2002). Their proposal was to use the LSTM to add a global structure to the music generated by the previously explored avenue of RNNs. Mozer wrote that, "while the local contours made sense, the pieces were not musically coherent, lacking thematic structure and having minimal phrase structure and rhythmic organisation" (Mozer 1994, cited by Eck and Schmidhuber (2002)). Eck and Schmidhuber used a form of twelve-bar blues as a basis for all of the music generated. The pentatonic scale was used to build the melody and a limit of eight notes per bar was introduced. Training melodies were written by Eck to fit musically with each chord.

Two experiments were carried out. Firstly, the network was presented with the chord structure. This was done to ensure that the LSTM was not using structures in the melody to predict the global chord structures. Once the network could generate one chord cycle, it could then generate any number of cycles. Secondly, both the chords and melody were presented to the network. The network was then given a seed note, or series of seed notes, and allowed to improvise. The network learned the chord structure and the resultant melody was constrained to the chord structures given. The resulting music can be considered to be a reasonable approximation of blues-style music.

The author admits that the experimental setup made the compositional task extremely simple. This can be shown by listening to several outputs from the system, which are very difficult to distinguish from each other.

This method has added an element of structure to the music, but there is still nothing

defining about the music - the notes are constrained to a single octave and no chords are used within the melody.

Another paper proposing the use of hierarchical RNNs for generation of music with structure was produced by Burr and Miyata (1993). This also identified the problems involved in the addition of structure to music, and proposed using layers of interacting RNNs to form a hierarchy that would give the music structure. "A hierarchical recurrent network consists of a collection of sub-networks each of which operates on a different time scale" (Burr and Miyata, 1993). The time scales proposed in this paper are every beat, three beats, twelve beats and twenty-four beats, but only the first two levels were implemented by the author. The use of networks of different granularities could capture local and global structures within the music. These different networks could therefore add an overall structure to the music.

Burr and Miyata (1993) used this method to generate waltzes and demonstrated its ability to organise melodic patterns into a structure. The main problem was that it had a habit of directly including large sections of the melody from the training data. An extension was proposed to add the higher-level sub-networks to the network to try to add greater structure to the music. Ways of eliminating the problem of including segments of training data in the output were also proposed.

Both of these papers were aimed at improving the overall structure of the music, rather than generating the music in the first place. If composition was going to be carried out through neural networks then this problem would need to be addressed. Another problem to address would be selection of chords and the inter-relation between the bass line and melody. However, the main problem is that the quality of the output is totally dependant on the training data.

2.2.2 Genetic Algorithms

Genetic algorithms are a popular approach to music composition. They work by sending small sections of data (genomes), which are then evaluated by a function. In this case, the genomes would be short series of notes. The best genomes are selected by the function and then combined to produce a new genome. The approach taken by Gibson and Byrne (1991) incorporates genetic algorithms with neural networks by using a neural network as the evaluation function. Gibson and Byrne's (1991) method was designed to produce music similar to traditional hymns. The compositional process was simplified by restricting the composition to C major and by producing harmonies that were restricted to only three chords. The composition was broken down into three main parts: rhythm, melody and harmonisation. This method has the interesting feature that two separate neural networks are used to evaluate the melody: one focusing on intervals between pitches, and the other on the overall structure.

Another genetic algorithm for music composition was written by Marques, Oliveira, Vieira and Rosa (2000). This system is different from Gibson and Byrne's (1991) as it simply

uses some basic rules to decide the fitness of the genomes. There are three parts to the evaluation: harmony, which evaluates the intervals between notes played at the same time; tone, which evaluates how suitable a note is for the chosen tone, and scale and melody, which evaluate intervals between consecutive notes. This is a complex system, compared to Gibson and Byrne's (1991) system, among others, since it allows eight octaves of notes, pauses, chords and multiple instruments to make up the music generated. The output of the system is said to sound pleasant to the authors of the work.

Ortega de la Puente, Alfonso and Moreno (2002), however, took a slightly different approach. This was to compose music using grammatical evolution. This differs to normal genetic algorithms in that it translates genotypes grammatically (different name for genomes) into phenotypes in order to evaluate them. The phenotype is then used to define the melody. For this method there are 83 different notes with seven possible lengths. The program was written to consider lengths of notes, pitch and differences between consecutive notes. Ortega de la Puente et al. (2002) intended to improve the program so that it could generate music in the style of well-known composers. This meant it could be adapted to generate music in a big band swing style.

Although all of these methods generate music, they appear to be a step behind neural networks in that they do not consider the introduction of structure to music and are instead concerned with the generation of music which is simply pleasing to the ear. Furthermore, there has been no work on the idea of using fitness functions to consider the structure of large sections of music, rather than the small segments currently examined. This issue could be answered by attempting to produce a swing band composer through the use of genetic algorithms.

2.2.3 Chaotic Systems

Chaotic systems are dynamic equations, produced through an iterated equation. Each iteration is called an orbit. Every output of this equation is returned as the input for the next orbit. Although this is a simple idea, these systems can yield very interesting results when used in the field of algorithmic composition. Bidlack (1992) conducted a range of experiments with these systems. He showed how systems in multiple dimensions can be used to produce values for pitch, amplitude and length. Furthermore, he used figures to show that the regular patterns produced could be incorporated into a bass line, and that a range of more random systems could produce a melody. However, Bidlack (1992) also provides a few warnings about the use of these systems:

1. The start value given to the system is extremely important as some start values can cause the system to tail off to infinity extremely quickly.

2. Precision in the calculation of the values for the different orbits is extremely significant as small rounding errors can cause massive variations in the outputs produced. For example, vastly different results could be found for a system using single precision numbers compared to one using double precision.

3. The usage of the same system on different processors is likely to give very different results.

Bidlack (1992) shows that once these three problems have been solved, chaotic systems can be an extremely promising option for algorithmic composition.

Leach and Fitch (1995) are other users of chaotic systems to generate music. Their paper aimed to use dynamic equations to model and produce a musical interpretation of events in nature.

It has been suggested that these systems may contain an inbuilt structure. This is because "each iterated point is a result of a precise formula which contains no random elements, giving rise to a highly structured orbit which has elements of near repetition within it everywhere" (Leach and Fitch, 1995). This is very interesting since using these systems effectively gives us the inbuilt structure that neural network methods have being searching for.

Although the use of chaotic systems in algorithmic composition is not currently widespread, this may be a sensible area of research: they have strong prospects as far as their use and manipulation to provide structure in music are concerned.

2.2.4 Others

Conversely, Moorer (1972) experimented with heuristics, trying to give the music structure by creating major and minor groups, chosen by using probabilities. For each musical section he either repeated a group or created a new one. Each major group comprised two minor groups. This allowed the composition to have a form (such as ABAC) which could be broken down into minor groups (for instance ABACABBC). Once this overall structure had been chosen, the contents of each minor group were determined by probabilities in order to generate rhythm, chords and melody. Moorer (1972) described the music produced as "strangely alien-sounding" (Moorer, 1972). The reason offered for this was that his method for structuring and generating the music was not the same as the method that humans use. However, this remains unproven, though his method is an interesting idea.

Dubnov, Assayag, Lartillot and Berjerano (2003) conducted some research "to capture some of the regularity apparent in the composition process by using statistical and information theoretic tools to analyse musical pieces" (Dubnov et al., 2003). Their method aims to analyse music provided for the program and then produce stochastic generation rules to re-create the patterns or phrases which have been discovered in the music. They have used two methods: the first is incremental parsing. This method scans the music and builds a dictionary of motifs in the music. It adds every new phrase that differs by a single last character from the longest match that already exists. The second method uses prediction suffix trees. They work in a similar way to incremental parsing, but only add certain motifs to the dictionary. They must appear a significant number of times and must be considered significant to the generation of future music. Experimentation shows that incremental

parsing captures the most musically meaningful motifs but tends to copy the training data. Prediction suffix trees produce music which is significantly different from the training data but they have a tendency to include 'false notes' (Dubnov et al., 2003).

2.3 Evaluation and Method Selection

Most of the methods researched have been struggling with the problem of adding structure to music. However, they have solved the issues of the extreme randomness of the music, and of the repetition of training data. Interestingly, it appears that the papers on genetic algorithms have not yet examined the problem of adding structure to the music.

Neural Networks seem to be a popular, well-used method for algorithmic composition. They have already been considerably successful in generating twelve bar blues (the basis of most big band Swing music) by using LSTM RNNs (Eck and Schmidhuber (2002)). There have been other projects using neural networks, such as BoB by Thom (2000). This is an agent that listens to a musician playing the blues and plays along with him or her in real time.

The method that has been chosen for this project is to use chaotic systems to generate the melody, while stochastic matrices are used to determine the choice of chords. These systems have been chosen because they have been used in the past to implement a twelve bar blues generator. Moreover, it is relatively easy to implement an iterated equation into a dynamical system. Stochastic matrices have been chosen to select chords because they are simple to implement and have provided positive results, such as in Dubnov et al. (2003). In addition, the choice of chaotic systems is a sensible option because of their potential to add structure to the music generated. This structure enables the music to be much more acceptable to the human ear.

The fact that no previous research into the use chaotic systems and stochastic matrices to generate big band Swing music has been found makes this a valid area for new research.

2.4 A Closer Look at Chaotic Systems

Devany (2003) states that dynamic systems have three overall different types of behaviour:

- Fixed Point, where each orbit of the system produces the same output.
- Periodic orbit, where each orbit consists of the same specific set of points.
- Chaotic, where the orbits appear to move aimlessly between an undefined set of points.

As regards the algorithmic composer, the chaotic behaviour will be considered, as this provides a degree of freedom in the outputs generated.

Bidlack (1992) categorises the chaotic behaviour into two further groups - dissipative and conservative:

- In dissipative systems, friction can be considered to occur. The phase space shrinks with respect to time, i.e. the system's outputs converge towards an attractor, which can be a single-point, a periodic, or a chaotic attractor. Examples of dissipative systems are almost any of the earths natural phenomena.
- In conservative systems, energy is conserved. This means that the phase space remains constant. An example of a conservative system is the dynamics of celestial bodies where energy is conserved over time.

Bidlack (1992) notes that a chaotic system can be formulated in one of two ways:

1. Iterated maps in one or two dimensions.
2. Continuous flows in three or more dimensions, which can be represented as differential equations.

Bidlack (1992) made the following observations on the types of chaotic systems: "A dissipative system, once it has settled onto its attractor, maintains a relatively constant behaviour....Conservative sequences, on the other hand, can exhibit much less internal consistency, and are marked by sudden changes in texture and range of values." (Bidlack, 1992). This suggests that a dissipative system may produce a strong bass line, whilst a conservative system is likely to produce a stronger melody, since it will exhibit more changes of texture and value, making the music produced more interesting. Bidlack (1992) also presents his equations and example C code alongside them. This is a sensible starting point for this research.

2.4.1 Conservative Systems

As already stated, conservative systems are those in which energy is conserved. They can be used to produce music that demonstrates sudden changes in pitch, amplitude or any other variable parameter. Some of the possible conservative systems that may be considered for this project are:

1. The standard map. This is a conservatively iterated map, derived from the model of a pendulum with external periodic perturbations. This system is described by the difference equations:

   $I_{n+1} = I_n + K \sin \Theta_n$

   $\Theta_{n+1} = \Theta_n + I_{n+1}$

   where K is the perturbation parameter, while I and Θ are variables taken modulo 2 in the interval from 0 to 2.

   To generate a chaotic standard map, K generally needs to be greater than 1. The value of 1 is only an approximation, though, and the precise behaviour of an orbit depends as much on its starting point as on the value of K.

2. The Hénon-Heiles system. This was originally introduced as a simplified model of the motion of an individual star within a galactic gravitational field. It is a conservative system which can be described by the differential equations of motion:

   $\ddot{x} = -x = 2xy$

   $\ddot{y} = -y = x^2 + y^2$

   where x and y are derivatives with respect to time.

2.4.2 Dissipative Systems

Dissipative systems are those in which friction can be considered to occur. This means that the systems outputs will converge towards an attractor. Once the system is in this attractor's state, the system can be very useful in the generation of a bass line as it will not demonstrate sudden changes.

Some possible dissipative systems that may be considered for use in this project are:

1. The Henon Map. This is an iterated map of the Cartesian plane, expressed as a system of two difference equations:

   $x_{n+1} = y_n + 1 - A{x_n}^2$

   $y_{n+1} = Bx_n$

   where A and B are positive constant parameters and the sequence of (x, y) outputs are points on the plane.

2. The Lorenz system. This is a dissipative system generated from a simplified model of atmospheric turbulence. It can be described by the differential equations:

   $\dot{x} = \sigma(y - x)$

   $\dot{y} = Rx - y - xz$

   $\dot{z} = xy - Bz$

   where σ, R and B are positive constant parameters.

2.5 A Closer Look at Heuristic Methods

Heuristic methods are based on the mathematical areas of statistics and determinism. These are demonstrated by the use of stochastic matrices in such algorithmic composition papers as Moorer's Music and Algorithmic Composition (Moorer, 1972).

Heuristic methods work by determining the choice of the next note based on the previous note. A stochastic matrix is used to define each possible transition from each possible note. Next, a random choice is made from the available options for that note based on the probabilities stored within the matrix. This leads to a highly pre-defined piece of music, as it is entirely dependant on the probabilities initially assigned in the stochastic matrix.

For this project, heuristic methods shall be used simply to determine the choice of chords below the previously generated melodical part. This method of chord selection has been chosen to help control the chords used, while it is also simple to implement.

2.6 Development Environment

The aim of this project is to produce a program that outputs data that could be interpreted as a piece of Swing music as performed by a big band. It would be futile to write code to cause a computer to generate sounds, as this is a topic that has already been researched extensively, therefore any work for this project would simply be replicating previously carried-out research. With this in mind, it is necessary to find a program or a language that can take an input of data and convert it into sound.

2.6.1 CSound

This is a system for producing and manipulating sound and music. It has two languages: the orchestra language, and the score language. The orchestra is used to specify the instruments playing in a piece of music, whilst the score is used to specify when and with what properties (i.e. pitch, amplitude, duration) each instrument will play (Boulanger, 2000). This seems perfect for this system, which could output a CSound score file, which could then be put through the CSound compiler to produce the final output. For the orchestra file, already developed instruments will be used wherever possible, although new instruments could be developed if necessary to provide a complete swing band.

There are resources on the internet for CSound, as well as a CSound textbook on how to use this system. Furthermore, the project supervisor, John Fitch, is an expert on CSound and could possibly help if there were any difficult problems.

This makes CSound the most obvious and simple choice for generating the music for this project.

2.6.2 The C Programming Language

The composer will be programmed in C, since this is a powerful language which the author is already familiar with. This means that there is no need to spend time learning a whole new language. Moreover, C is very widely used and contains all the functionality required for this system.

2.6.3 Note Representation

One issue faced by many of the researchers in the papers read is note representation. A number of the research projects explored used an extremely constrained set of notes. Eck and Schmidhuber (2002) limited themselves to an octave, while Gibson and Byrne (1991) limit themselves to C major. For the production of big band Swing music it is not possible to be limited to a small range of notes or to a single key. A large pitch range (four octaves minimum); a range of note durations; volume changes and key changes are desirable. The CSound score file basically consists of a note list that for each note

includes an instrument number, a start time, a duration and other parameters for that instrument. These parameters often consist of amplitude and pitch among other factors (Boulanger, 2000). CSound can produce a wide range of pitch values, all of which are given as numeric values. Therefore, the notes in the program will also be stored as numbers. CSound allows floating point entries for note start time and duration, whatever length the latter may be.

2.6.4 Conclusion

In conclusion it has been decided to use a chaotic system for the generation of the majority of the music (melody and bass line), with stochastic matrices to determine the choice of chords based on standard chordal progressions. This is due to the very interesting structural properties of chaotic systems, as well as the ease of implementation. The choice of stochastic matrices to determine chordal progressions ensures that the generated music will be appealing to the human ear.

It is intended to use the C language for the composer, and to output the musical data in the form of a CSound score file that can be complied into a playable format.

Chapter 3

Requirements & Design

This section describes what the program needs to do, discusses some of the problems and includes some initial program design. As this is a piece of research, the details of how the program actually works are in the implementation section.

3.1 Output Form

This program should, when started, begin to produce a piece of blues music. This should continue to be produced until it reaches a predetermined piece length. Each time the program is run a different piece of music should be produced.

The starting values given to the equations used to generate the music will have a great impact on the way the system behaves. The system could show cyclic, fixed point or chaotic behaviour, or it could escape off into infinity. Part of the research towards the use of this system would be to determine a way in which the starting conditions may be varied, nevertheless ensuring that the system still behave in the desired way. A decision about how much the starting parameters should be varied would also have to be taken. This is likely to be carried out on a pseudo-random basis.

The fact that the input parameters to the chaotic systems used will be varied means that extremely different outputs may be found between iterations of the program. This could be a good thing in that a different output could be easily generated or it could be bad since a small change could cause the system to fail to produced good music. The main problem with this is that "every piece of sound has its admirers and haters." (Fitch, 2007). The data produced by the equations used is only a very small part of the compositional process because the data must be mapped to notes. If this mapping to notes (covered in the next section) remains constant then the production should retain a similar feel amongst different iterations.

Another big issue with the production of the music is the decision of whether real time production should be attempted. Preliminary testing of CSound for music with a complexity

of ten to twenty instruments playing simultaneously has shown that this is not a suitable idea. Working out of real time also allows each instrument's part to be produced in any order required. For example, a bass line could be produced and then each part could be fitted so that it fits harmonically with the bass line.

3.2 Representation of Notes

The decision about how to store the notes generated is an important one as it will have a knock-on effect throughout the program. There are two main possibilities for the storage of these notes:

- Use an array with space for every single possible time step in the piece.
- Store the notes in a linked list.

At first glance option one appears to be flawed due to the memory requirements. On closer inspection, however, it can be seen that for a piece in which the shortest note is a demisemiquaver, only 32 cells per bar are required. Considering the memory sizes of modern computers, this is not an excessive requirement. The limitation of the piece to a minimum length of demisemiquavers is not a problem since shorter notes are hardly ever seen in any composed big band music. A major advantage of this method is that sections/individual notes are easily located since they are at specific array locations. This possesses a great advantage over a linked list since a linked list does not necessarily have any order. This means that each note would have to be found by linear search. There is also the advantage that the implementation of an array of notes is much simpler and therefore saves on coding time which is an important consideration in an exploratory research project.

Since notes are going to be stored within an array, the storage of chords is more difficult. This is because it requires multiple notes to be stored within one cell of the array. There are two main reasons why this could occur. Firstly, a part could be playing more than one note at once, or more than one part could be playing.

Possible solutions include:

- Having a pointer from each note to another note which would allow a chord to be stored as a set of notes with pointers. This would mean that each cell of the parts array would actually be storing a chord which could consist of an arbitrary number of different notes. This would overcome the first problem.
- Storing a different array for each part. This would mean that it was always clear which part was playing a particular note since all the parts would be kept separately. This would be the simplest solution to the second problem.
- Restricting instruments to only play a single note at one time. This would be a suitable option for all but the piano.

From these possible solutions it was decided that the best possible design would be to use a different array for each part and to restrict the instruments to only playing a single note at a time.

The decision to restrict the performance was taken since it allows for much simpler coding on the majority of the parts. Furthermore, it was decided to simplify the process in this

way since generating a suitable big band piano part would require a whole dissertations work on its own therefore it was decided that other issues such as harmonisation would be more important within this piece of work.

3.3 Representation of Parts

Each part will have several parameters which are specific to itself. These will include range, solo priority and type (which defines whether it is mainly a bass or melody instrument). Storing these parameters makes it much easier to program the music to fit with the multiple parts since tasks such as ensuring the music remains within the playable range of each part are simplified.

The storage of these parts needs to be carried out in a way which ensures that any part can be easily found. The fact that only a maximum of about twenty different parts will be stored makes this process substantially easier. Conducting a linear search through a set of twenty different parts to find the correct one on an infrequent basis does not increase the computational intensity of the program by a significant amount. For this reason, the parts shall be stored in a linked list which can be searched using a linear method when a specific instrument is required.

3.4 Data Structures Used

The following is the data structure used for notes:

```
typedef struct NOTE
{
double start;
double duration;
double pitch;
double *scale;
double amplitude;
} NOTE;
```

The essential data for each note is stored in order to output sufficient information to CSound.

The following is the data structure used for Chords:

```
typedef struct CHORD
{
    int notes[4];
    NOTE *bassNote;
    double octaveBassNote;
    int pitchBassNote;
    int type;
    double probability;
} CHORD;
```

This contains the essential data for each chord. The bass note's information is stored in two separate ways to simplify the code which uses the chord data structures.

The type of the chord is an integer used to simplify the checking of a chord's type. This is because it is necessary to check only one variable rather than the different notes within the chord to ascertain which type of chord it is.

The following is the data structure used for instruments:

```
typedef struct INSTRUMENT
{
NOTE *noteArray[SONG_LENGTH];
char *name;
int instrument;
int type;  //Holds 1 if its mainly a bass instrument or
      //0 if its mainly a melody instrument.
double soloPriority;
```

```
double maxRange;
double minRange;
struct INSTRUMENT *next;
} INSTRUMENT;
```

This contains the essential data for each instrument. The important things to note are:

- The type variable which is used when instruments are selected to play notes from chords.
- The soloPriority double. This should add up to one when the solo priorities of all the instruments playing solos are added together.
- The minimum and maximum range of the parts. This range is designed to limit the instruments' range to an actual playable range in order to ensure that the music produced by this program could actually be performed by a big band in real life.

3.5 Scales and Octaves

In order to generate notes which are part of a scale rather than an arbitrary frequency we must use the CSound method of representation.

The octave number

The offset from the base of the octave where each hundredth is a semitone.

In order to successfully map the notes to scales, it was decided to store the notes within each blues scale in arrays.

```
double cScale[6] = {0, 0.03, 0.05, 0.06, 0.07, 0.10};
double fScale[6] = {0.05, 0.08, 0.10, 0.11, 1, 1.03};
double gScale[6] = {0.07, 0.10, 1, 1.01, 1.02, 1.05};
```

Two further variables were used. These were a pointer to the current scale and a double which stores the current octave being played.

The current octave would be added to the desired note from the current scale to produce an actual note from this scale. This makes it relatively easy to generate the pitch of a note.

A normal blues musician would not restrict himself purely to pentatonic scales. Although the musician's music could be based around these scales, other notes would not be ruled out. This means that this representation is slightly restrictive, but it makes the program more likely to produce music with a blues sound. A possible extension would be to provide the possibility of using notes from outside the pentatonic scales without losing the swing sound of the music.

3.6 Chords

A method for the selection of the chords to be used within the music needs to be defined. This is because when 20 parts are being randomly generated it is not suitable to simply hope that there is some harmony between the notes. Therefore a method based on heuristics will be implemented. This will involve an array which contains the possible chords to be used within the piece and a probability for the selection of the chord. This system will be easily extended by adding new chords. The details of this method have been left vague on purpose in order to allow for innovation during the implementation phases.

3.7 Dynamic Equations Implementation

The equations to be implemented are very simple and just require the arithmetic inputting into the development language. There are two main issues that require consideration within this project:

1. Collecting the data generated.
2. Selecting and supplying the starting values.

It was decided that a global variable for each dimension of the equation's phase space would be used to collect the data. Calling the equation's function would produce one generation of an orbit, after which the global variables would be updated so that the data could be collected by any interested function. This most simple of methods would ensure the quick and easy implementation of the basic parts of the program. This could be improved to make the function return an array of the values from the orbit of the map.

The issue of passing starting values to the function seems trivial, but it is worth pointing out that these values will be passed to the function as parameters. This would therefore allow the starting values for each run of the function to be changed if required. This would also allow different starting values to be used for each instrument during the implementation phase, if necessary.

3.8 The Note Creation Process

This is one of the most complex and difficult parts of this project. To produce data from dynamic equations is simple, but it is the mapping of this data from a numeric domain into notes that determines the sound of the music produced. The method chosen for this will have a huge impact on the sounds generated. An interesting method to consider is to map the data to notes via ranges, so that data within a certain range will be mapped to a certain note. Another possible method to consider is to use the data to determine the offset from the previous note.

3.8.1 Bass Line

The bass line in swing music normally consists of a simple rolling bass line and a set of chords which, whilst showing some musicality, tend to be long chords simply to add support to the background of the piece of music. This set of chords will be dealt with in this section and the rolling bass line will be dealt with in Section 3.8.4.

As mentioned earlier, Bidlack (1992) notes that there are two types of dynamic system: dissipative and conservative. Furthermore he notes that, after a transition phase, a dissipative system will settle toward an attractor and exhibit a fairly constant behavior. This type of behavior is ideal for mapping to a bass line as a bass line should exhibit a relatively constant behavior.

The main tasks of the bass line are thus:

1. To implement a dissipative system in order to produce data to be mapped.
2. To map those values to chords in the correct scale.
3. To ensure that repeating structure of the data fits to the music.

The choice of which dissipative system to use requires a degree of experimentation since each system will behave differently. Good starting points for this are the Henon Map and Lorenz System detailed by Bidlack (1992) as he has also provided C code for their implementation. The mapping of this data into pitch values and chords is also very open for experimentation. As the bass line is quite simple, a straight mapping of ranges of data values to notes is likely to be all that is required.

The task of composing the bass line is complicated by the fact that the data structure needs to be fitted to the bar structure. A simple method to mask this will probably be implemented as this is not a major concern of this project. This method will consist of forcing the notes onto the beat of the bar if it is within $\frac{2}{32}$s of the beat. During the project, though, it is more worthwhile to attempt to find a better solution than to attempt to mask the structure of the chaotic system.

3.8.2 Melody

The melody is the most complex part of the program. It is of main interest to the listener and therefore needs to show the most true big band sound. The complexity given to the melody will depend on the progress made through the harmonisation of the multiple parts which is where the main aims of this project lie. A good blues melody would be expected to show at least some of the following techniques:

- Notes of varying lengths
- Movement over multiple octaves
- Repeated structures
- Various chords

The choice of which points to include from this list will be mainly dependant on time.

Conservative systems are intended to be used for the creation of the melody since, as noted by Bidlack (1992), they show changes in texture and range of values which should hopefully provide an interesting set of data for interpretation into music.

The main tasks and core issues are in the production of the melody are:

1. To implement a conservative system to produce data to be mapped
2. To map from the data to notes in the correct scale
3. To allow repeated note structures, blues style riffs or repeated segments of the piece
4. To allow a variety of note lengths
5. To allow the music to range over multiple octaves
6. To build chords from the data
7. To fit the melody part produced around the bass line
8. To ensure the notes flow from scale to scale within the blues sound
9. To produce a melody with a blues sound
10. To maintain an overall structure in the melody

Experimentation will be done using the Standard Map to determine what sounds good. This will start by implementing a simple individual note selection method before building that up into chords with musical qualities.

3.8.3 Counter Melody

The implementation of the counter melody will be very similar, if not identical, to that of the main melody, though it will have a lower volume. Moreover, the counter melody also does not have to feature in every section of the music and so a pseudo-random number generator will be used to determine whether it appears or not. The percentage chances of a counter melody being present will be subject to research within the project.

3.8.4 Rolling Bass Line

The rolling bass line function shall generate a single part of music which exhibits a change of a maximum of one degree within the blues scale each note. This change shall be randomly generated. Two different methods will be assessed. The first shall be to use a chaotic system to decide whether a movement shall be made and in which direction this movement shall be. The second is to use a pseudo-random number generator seeded on the date and time the program is run. It is my expectation that the second method shall be more suitable since the first method seems to require an unnecessary amount of work.

3.8.5 Solo Sections

The solo section is one of the most interesting aspects of the music. I feel that to implement a good solo algorithm would be a project in its own right. This is due to the extremely large array of techniques which a good modern-day soloist can draw upon. Therefore I have chosen to implement a similar method to that being used for the melody, but with a seeding note to ensure a much more varied output from the Standard Map. This will add far more variation in terms of the musicality of the section.

The solo section could implement many techniques to increase the auditory quality of the music produced, but these are not deemed necessary for this project, which simply aims to compose big band music. For this reason, the ability to output music through CSound directly is just a way of facilitating the implementation process. Therefore these techniques would be added by a human soloist as he or she interpreted round the music within the solo sections.

3.9 Project Requirements

This section contains the overall requirements of the project. This is a research project so the requirements are deliberately vague in order to allow innovation during the development process.

1. Research should be carried out into possible ways of using chaotic systems with herustic methods to carry out compositional procedures.
2. The outputted composition should be a piece suitable for performance by a big band. This means that each part must be constrained to remain within the playable limits of the instruments.
3. The program should output a single piece of music each time it is run.
4. Each piece of music outputted by the program should show differences.
5. The bass parts generated by the program should include a rolling bass line as well as other parts such as drum parts, etc.
6. The music outputted should represent notes of varying lengths.
7. The music outputted should represent notes which span a variety of octaves.
8. The music outputted should contain sections which are of a twelve bar blues style.
9. The music outputted should contain solo sections and an introduction.
10. The music outputted should be of a relatively high complexity and should hold the listener's interest.
11. An overall structure should be visible within the music produced.

Chapter 4

Program Development

4.1 Initial Groundwork

Before beginning the implementation of the main composer the data structures and related methods were developed. This involved a number of minor improvements on the design which were added to simplify other sections of the project. A system based on Morris's (2005) work was implemented to control the movement through scales within the twelve bar blues sections.

The other main item that was implemented at this point was the output to Csound. This involved both researching and finding a number of instruments to play specific parts of the piece and writing a small number of custom instruments where suitable ones were not forthcoming. This mainly involved the development of a range of drum parts and a simple trumpet and trombone parts. These were created since the examples found on the internet were extremely complex and required a large number of varying parameters. As this would have complicated the main program substantially, it was deemed to be unwise. Furthermore, it would have taken the focus away from the main aim of considering the harmonisation of multiple parts within the music.

4.2 Melody

After starting work on the melody section of this program it became apparent that a whole dissertation could be written about this one small area alone. In order to create an authentic blues sound there are many different paths to explore and test. Unfortunately there was not time to explore all of these different aspects within the scope of this one project, therefore a likely avenue for success was picked, based on Morris's (2005) work and this was explored in detail.

4.2.1 The Mapping

The likely path for success was to implement a mapping of the Standard Map in two dimensions. One dimension was used to specify the pitch and the other dimension was used to specify the duration. This was implemented using a variable which holds the current location within the piece and which corresponds to the current location within the array of notes.

Pitch Mapping

The pitch mapping was designed to ensure that multiple octaves of notes could be heard. This meant that the data from the Standard Map was used to specify the offset from the previous note in the music. This was chosen because the melody can show a wide variation in pitches and - since each note in the melody is directly related to the previous note - the melody will hopefully display flow, direction and structure.

The main problem with this was that the melody had a tendency to wander across a very wide range of notes. This would make the music unplayable since it could wander outside the instrument's range. Therefore an upper and lower limit was imposed on the melody. The limits were chosen were to hold it within the range of from middle c to two octaves above. This range was chosen since it corresponds to the ranges of most instruments within a big band. A better system may choose which lines of the music each part will play first and then generate the notes which fit within each specific instrument's range.

Duration Mapping

The duration mapping followed the same method as the pitch mapping, i.e. using ranges. In order to add a sense of rhythm to the music, notes were arranged so that they fell more often on the beats of the bars. This was done by checking if a note was within $\frac{2}{32}$s of a beat and then moving the note onto the beat. This created a better rhythm to the music produced at the expense of limiting the possible outputs slightly. This also generated a better structural link between the melody and bass line since both the bass line and melody were following the same method of forcing notes onto the beats of the bar where possible.

4.2.2 Sense of Rhythm

The sense of rhythm within the melody was added by moving notes within $\frac{2}{32}$s of a beat of a bar onto the beat. This means that the music shows a sense of rhythm which can be compared to that of a human composer. This effect is extremely subtle, but has a strong impact when tested on listeners (formal tests were not carried out but informal tests were). The music was tested on listeners by playing them two samples and asking which sounded better. The samples used were the same piece of music with and without this modification applied. The results of this testing was that approximately 80% of the listeners tested picked the one with the effect applied as the best sounding version.

4.3 Rolling Bass Line

The rolling bass line was first implemented using a simple mapping of a chaotic system (Henon Map) to the notes. This was successful, but was felt to be extremely complex for what should have been a simple operation. This is because there are only three possible choices for each note:

- Move up a note in the blues scale.
- Move down a note in the blues scale.
- Remain on the same note.

These operations can be further simplified by adding a variable concerning the direction of movement:

- continue in the same direction;
- change direction;
- change direction and remain on the same note (this is a simple sub case of point two). This was not implemented in the first attempt but comments from people listening to the music produced showed that it was a necessity to gaining a realistic sounding piece of music.

Once the direction of movement variable was added, it became apparent that implementing the rolling bass line as a simple "if statement" would provide equal, if not better results. This "if statement" employed a random number to determine whether or not a change of direction would occur:

```
x = ((double) (rand() \% 50000))/50000;
if(x < 0.2) {
direction = direction * -1;
}
```

This method was tested and found to provide better results than the chaotic mapping for a less processor-intensive method. Although processor time is not important when using the program for non real time processing, a future improvement may be to use this system to generate music in a real time domain.

The outputted music produced was described by listeners as being extremely good at building up suspense through the movements. Other comments regarding this suspense stated that though the music produced built up the suspense, it did not seem to go anywhere. I believe that this was because the links between different parts of the melody and rolling bass line did not intermingle in a way that generated suitable structure. This was left as a

future development task due to a shortage of available time. I feel that mapping volumes to the music after all the pitches have been generated may go a long way to solving this problem since the program would be able to view all parts of the music and attempt to link them all together through the volume changes. This also fits with the fact that a number of composers believe "that a piece without dynamic changes is as good as no piece at all". This is a quote made by a member of the group which surveyed the final outputs and it really shows how important dynamics are to a good piece of music.

4.4 Solo Sections

The solo section was implemented as a mapping of the Standard Map to notes based on ranges. This was done by basing each note on the previous and using the value from the Standard Map to determine both the offset for the pitch of the next note and the duration of next note. The duration was mapped based on a series of ranges, but the ranges were set to ensure a much more interesting part with a large number of shorter notes (crotchets or less).

Due to time constraints further improvements to the solo sections, such as instrument-specific features for the output (such as vibrato) were not added. The main improvements to the solo sections would be added by a human soloist when the piece was performed. This could include aspects such as volume changes, accenting and even modifying the notes to be performed since most good soloists would use the solo part provided as a guide rather than as the definitive version.

4.5 Linking everything together

This was by far the hardest section of the whole dissertation as a whole since it required both harmonisation of multiple parts and the retention of the musical structure after the music had been harmonised. This meant that it was not possible to simply ensure that every note in the piece was in harmony with every other note since this would make the music relatively bland and boring compared to big band music, which often uses discords to emphasise certain parts of the music. This added additional problems to the program since the matter of when to employ dischords and when to ensure the music is in tune is an extremely difficult problem.

4.5.1 Chords

The chord selection process went through a number of iterations. The first attempt consisted of using a chaotic system to decide the full details of the notes within the chords. This was tested and it was found to be extremely difficult to generate music that showed any form of harmony. This therefore led to the use of the chaotic systems to select the bass note of the chord. Following this, a simple heuristics method was used to select which type of chord should be played at that point. The possible chords were restricted to major or minor triads (with the bottom note repeated an octave up). A limited selection of chords was included to make the programming process easier. The system used for the selection of the chords and their generation, however, was designed to be easily extensible so a future improvement of this software could be to enable the program to generate a greater range of chords and to investigate the probability of each chord occurring within big band music.

4.5.2 Harmonisation of Chords

A number of different methods were trialed to harmonise the chords from different parts. The final method implemented consisted of copying the bass note for the chord (though not the octave it should be played in) and the chord's structure. This meant that the harmonised chord was the same as the bass chord it was harmonised with, just in a different octave. This is not the best solution as it affects the mapping of the chaotic systems to notes. A better solution would be to include the ability to have chords with the first in different positions. This solution was chosen since it simplified the procedure of developing the program and, upon testing, was found to give a strong musical output.

The positioning of discords within the music was almost left to chance since addressing this problem would need a dissertation of its own. The chance of a discord occurring on the first beat of a bar was non-existent since each beat was harmonised. The positioning of discords was left to chance for all notes which were off-beat because it was felt that this gave a relatively high chance of a discord occurring whilst ensuring that the music had a harmonic structure. This had the knock-on effect that the quality of the output decreased, but it was still possible to carry out much more work on the main issues of harmonisation

as a whole.

4.5.3 Structure

A sense of structure was added through the definition of pre-determined solo sections and band sections. The decision to use pre-determined lengths for these sections was made to simplify the programming. Furthermore, the analysis of a number of pieces of big band music showed that set lengths were often used for the repeated sections.

Structure was also added through the choice of chaotic systems for the generation of the musical sections. This is because chaotic systems have already been shown by a number of authors, including Leach and Fitch (1995) and Morris (2005), to have the ability to add structure to music generated from them. One comment when speaking about chaotic systems says that "each iterated point is the result of a precise formula which contains no random elements, the result is a highly structured orbit which has elements of near repetition within it everywhere" (Leach and Fitch, 1995).

The final structural modification was to use a simple initial section to provide an introduction to the music. This follows the common big band style where a drum and bass solo builds up to the main music. This section could be developed to allow each part to join during the introduction so that a more gradual build-up into the music is gained, but this work was felt to not be worthwhile considering the extra time it would have taken.

This pre determined structure was chosen due to its similarities to the real life big band music which the author has played before. These similarities meant that the music should be able to show a realistic big band structure whilst also being easy to program since it was using fixed lengths.

Chapter 5

Evaluation

This section seeks to evaluate the work that has being done through the development stage and to start to draw the work together towards a set of conclusions.

5.1 Questionnaire Results

In order to evaluate the outputs produced by the program it was decided to ask some people to evaluate one of the score outputs. This is shown below:

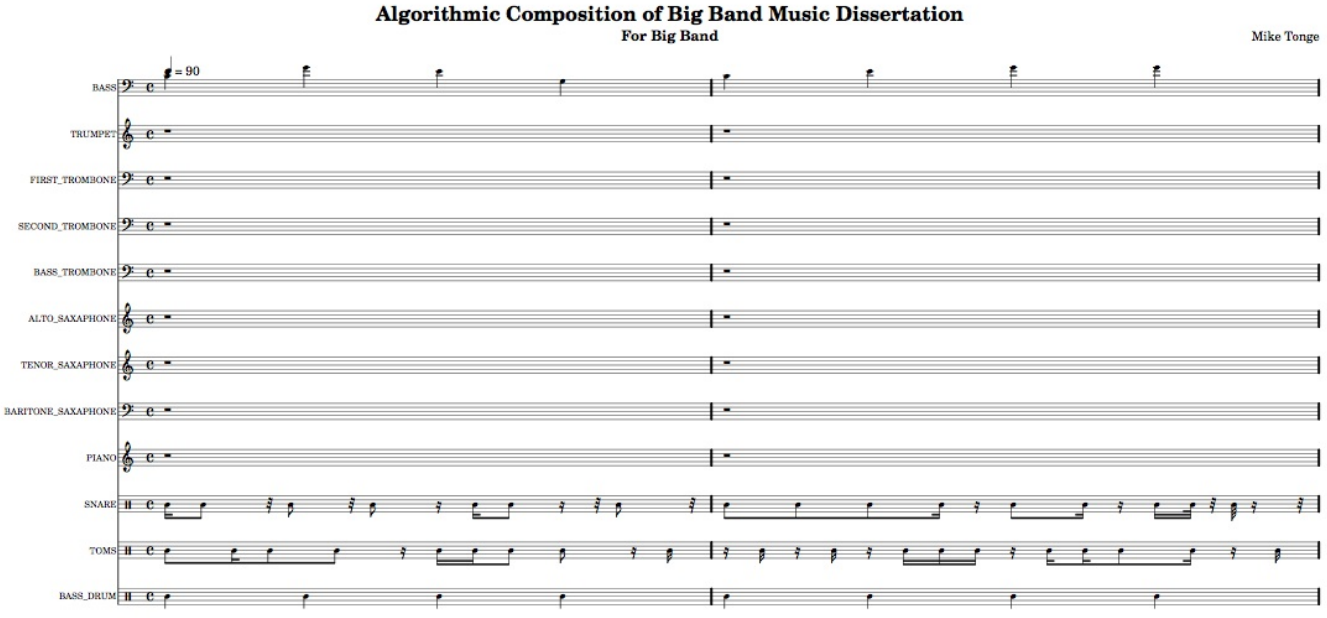

Figure 5.1: Score part 1

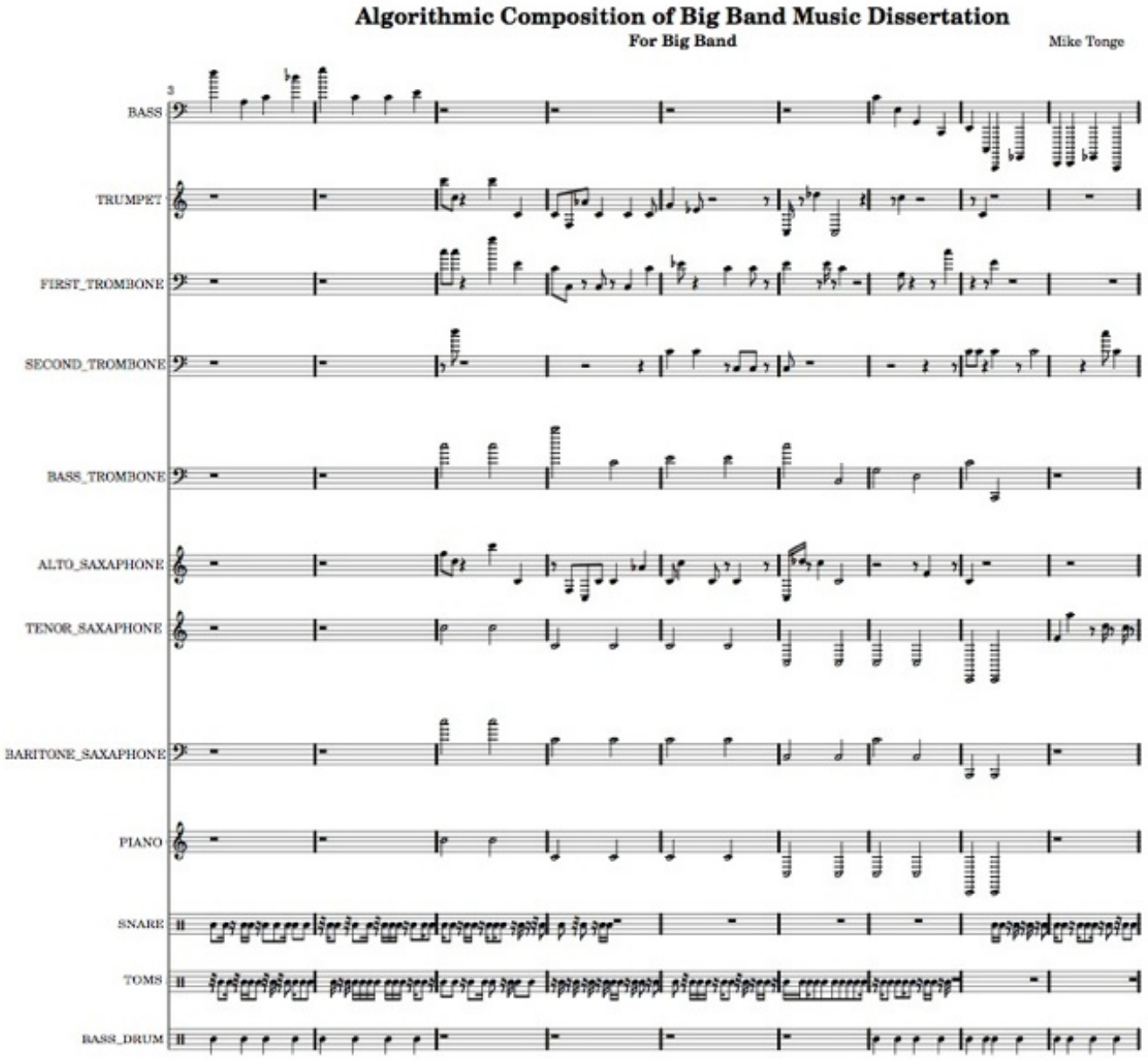

Figure 5.2: Score part 2

The decision to show a score file prior to playing music to the evaluators was taken since all evaluators were members of an orchestra therefore they could be expected to be able to read music just as easily as to listen to the sound. One main benefit of choosing to evaluate the music through the score is that it removes any bias caused by the instrument qualities. This also allowed questions to look in detail at the quality of the instruments used if required.

The people surveyed were then also asked to comment on a recording of a piece of the music since this allowed comments on the quality of the instruments used within the CSound libraries to be acertained.

The decision to ask external individuals to comment on the outputs produced was taken because the author has spent many hours listening to test outputs and so is very accustomed to anything that the system can play. Secondly an unbiased, critical opinion is very beneficial because it provides a fair evaluation of the outputs produced.

Twelve questions were asked and the results were as follows:

5.1.1 Questionnaire Conclusions

1) Please rate your musical knowledge/experience on the scale of:

1. None (1)
2. Beginner (2)
3. Intermediate (3)
4. Advanced (4)
5. Professional (5)

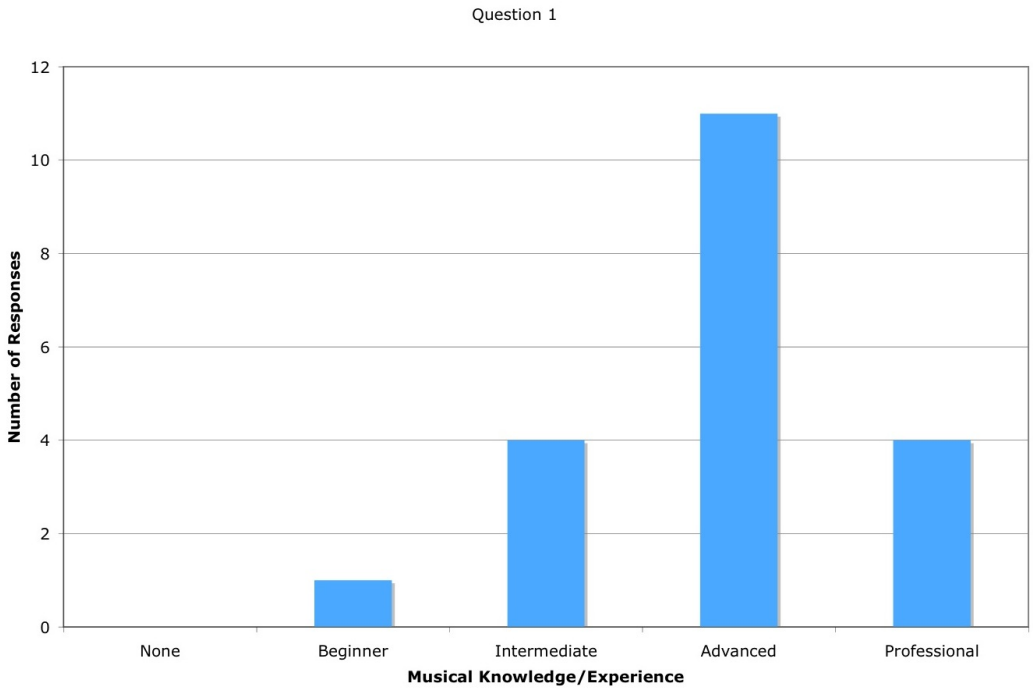

Figure 5.3: Question 1 Results

The people surveyed were mainly within the advanced range of musical experience. I feel that this was suitable for this project since the evaluation needed to centre around harmonisation whose evaluation required musical knowledge.

2) Please rate the structure shown by the music on a scale of 1-10?

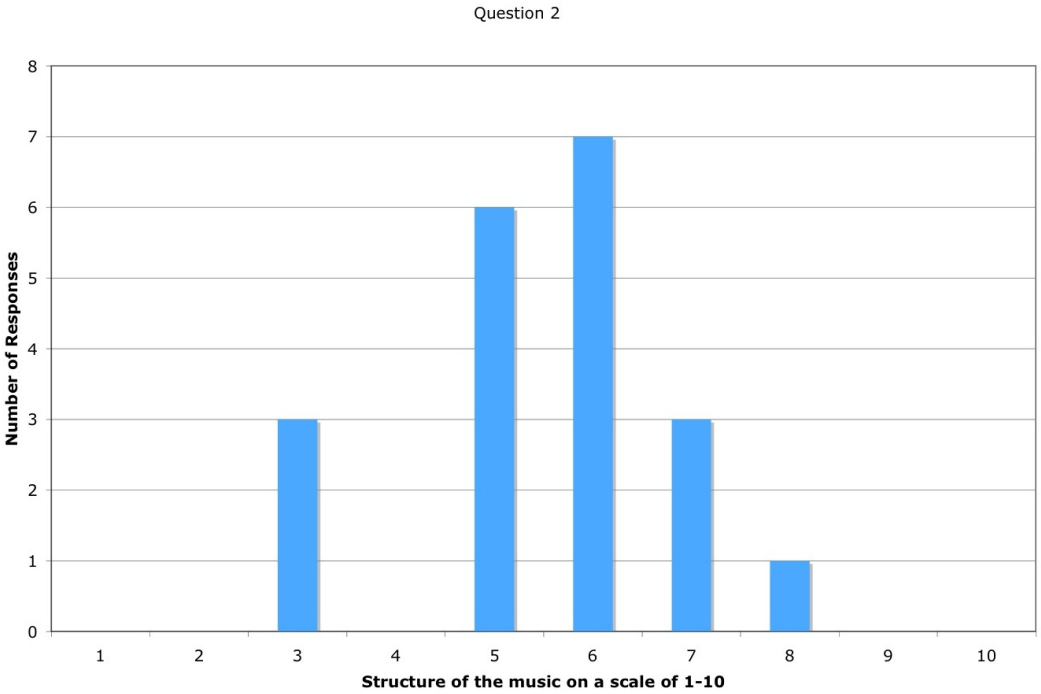

Figure 5.4: Question 2 Results

The structure of the music was rated as approximately a five. This is relatively good since the surveyors are highly qualified musicians. The main comments which were relevant to these scores was that the addition of dynamic changes to the music would enable a much higher score to be received here.

3) Please rate the quality of the musicality of the solo sections on a scale of 1-10?

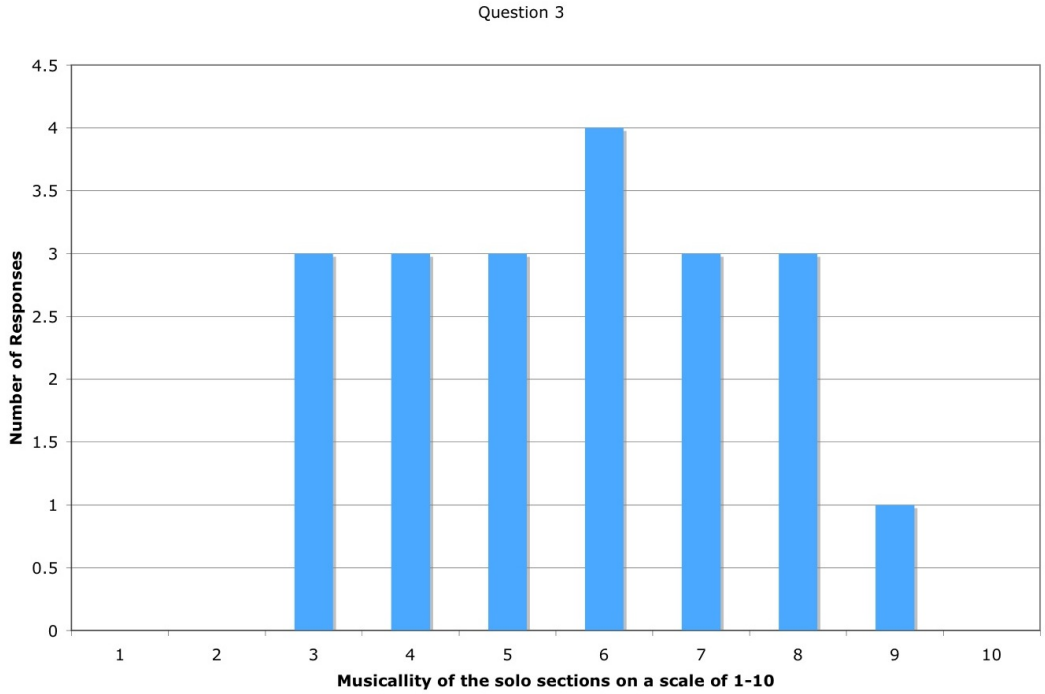

Figure 5.5: Question 3 Results

The musicality of the solo sections received a high average of 5.7. This did have a high standard deviation which represented the fact that people seemed to either love or hate the solo sections produced.

Overall these results were pleasing since the fact that a number of people scored the musicality very highly, including a nine and a few eights shows that some aspects of the production of the solo sections were accurate.

4) Please rate the links between the whole band and the solo sections on a scale of 1-10?

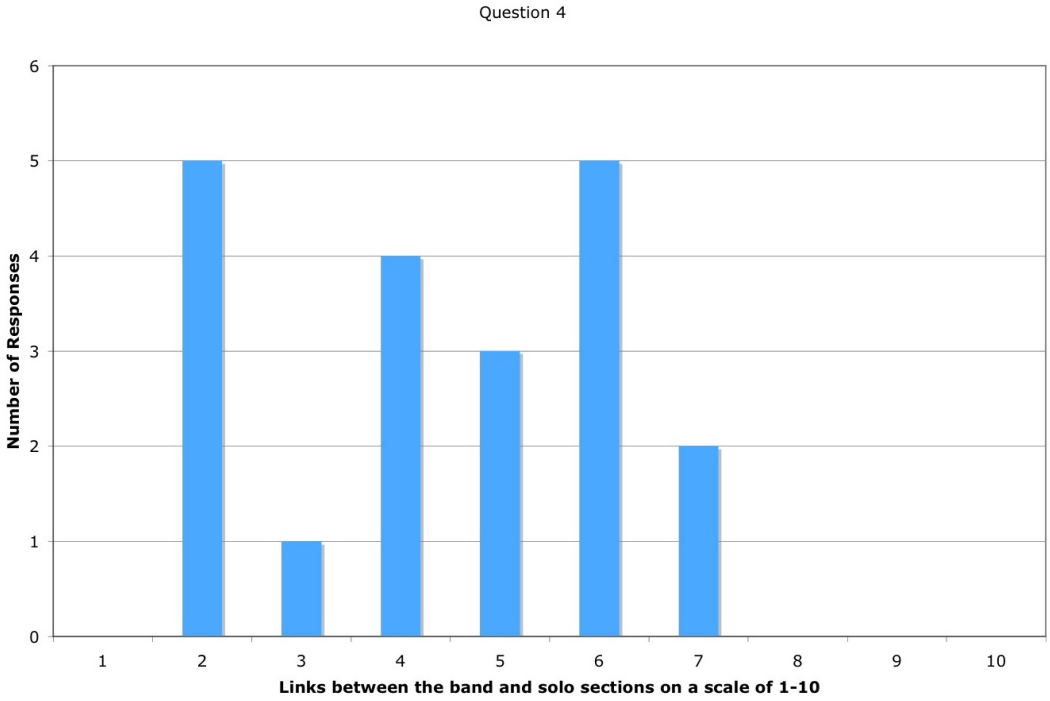

Figure 5.6: Question 4 Results

This scored very badly on the questionnaire since people commented that there wasn't a strong flow between the whole band sections and the solo sections. There were a number of people who rated the program as a two which identifies this as an area where major improvements could be carried out.

5) Please rate the quality of the melody line of the music on a scale of 1-10?

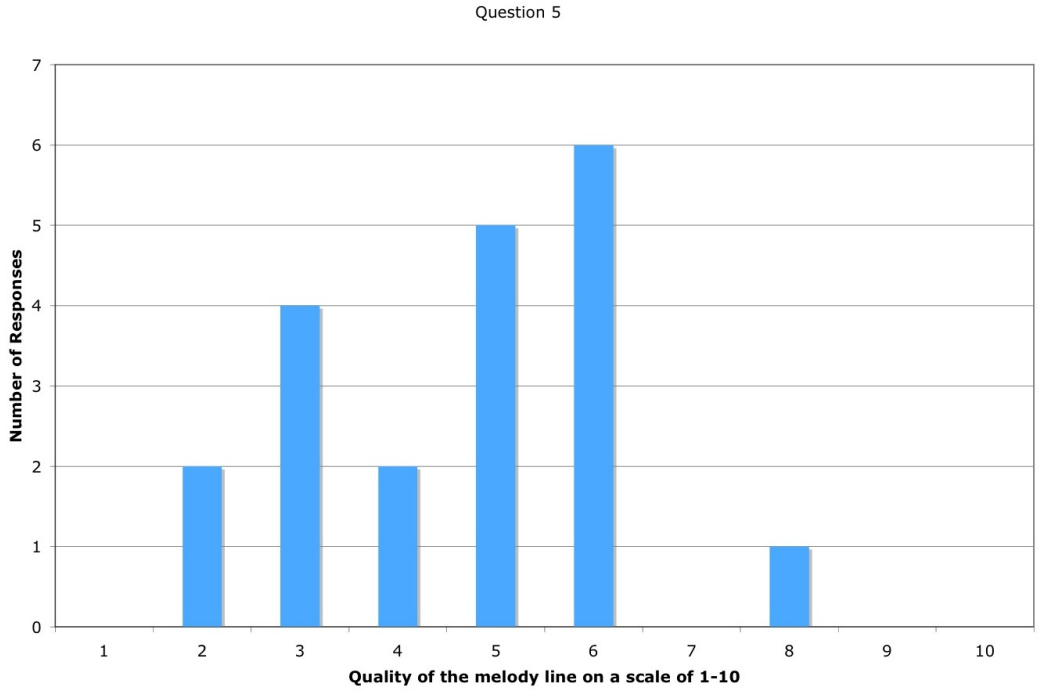

Figure 5.7: Question 5 Results

The melody line of the music had a balanced spread across the range of results though it had a low average. The main comment that people had was that the melody line was too spread throughout the different instruments therefore it was difficult to distinguish it whilst looking at the score. They also identified that this would make it extremely difficult to get a live band to play the part with a correct balance.

6) Please rate the quality of the bass line of the music on a scale of 1-10?

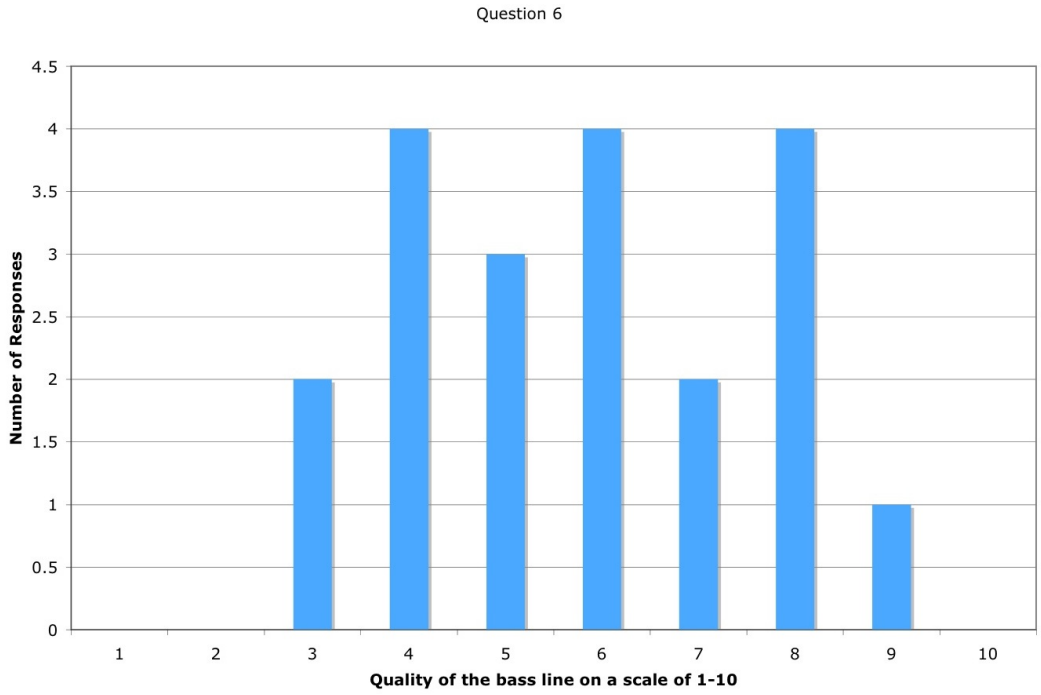

Figure 5.8: Question 6 Results

This received the highest overall average and was the most well received out of all the parts of the music. The evaluators felt that the bass line had a real rolling gait and would make a very strong big band bass line.

7) Please rate how well the bass line interlinks with the melody on a scale of 1-10?

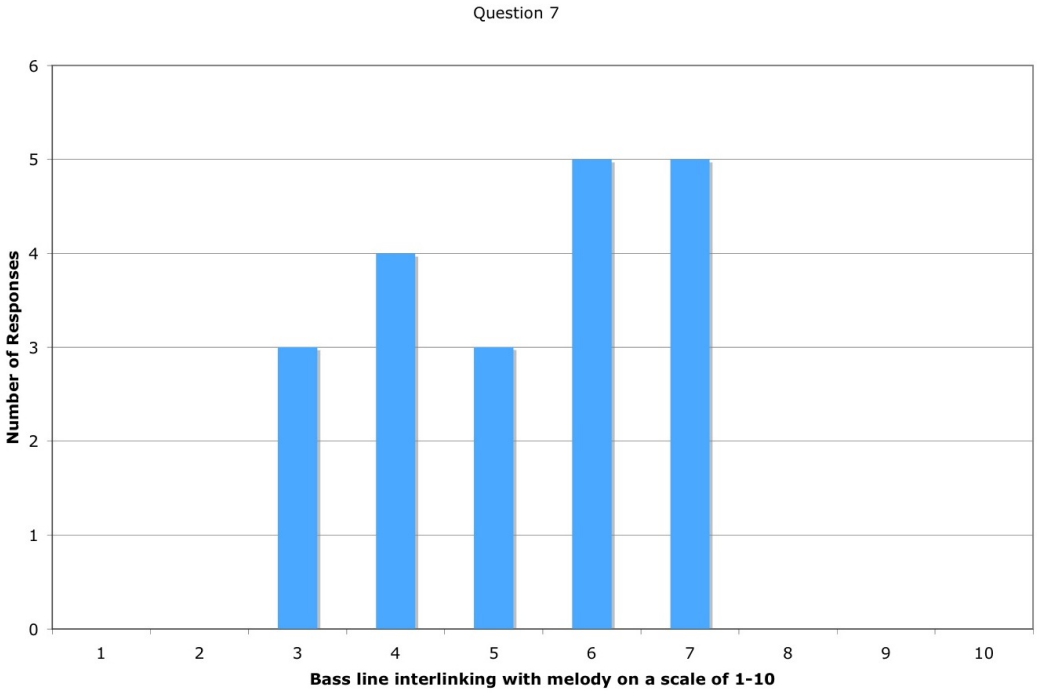

Figure 5.9: Question 7 Results

This question was one where the evaluators seemed to feel there was no right answer. Comments were received which said that though the pitches and tuning were correct the rhythm didn't fit quite aswell as they might have expected. On the other hand comments were received that said that the rhythms fitted really well and a few aspects of the tuning could do with some work. This leads me to believe that this is very much a matter of personal preference so following on from Fitch's comment that "every piece of sound has its admirers and haters"(Fitch, 2007).

8) Please rate on a scale of 1-10 the sense of rhythm shown within the music?

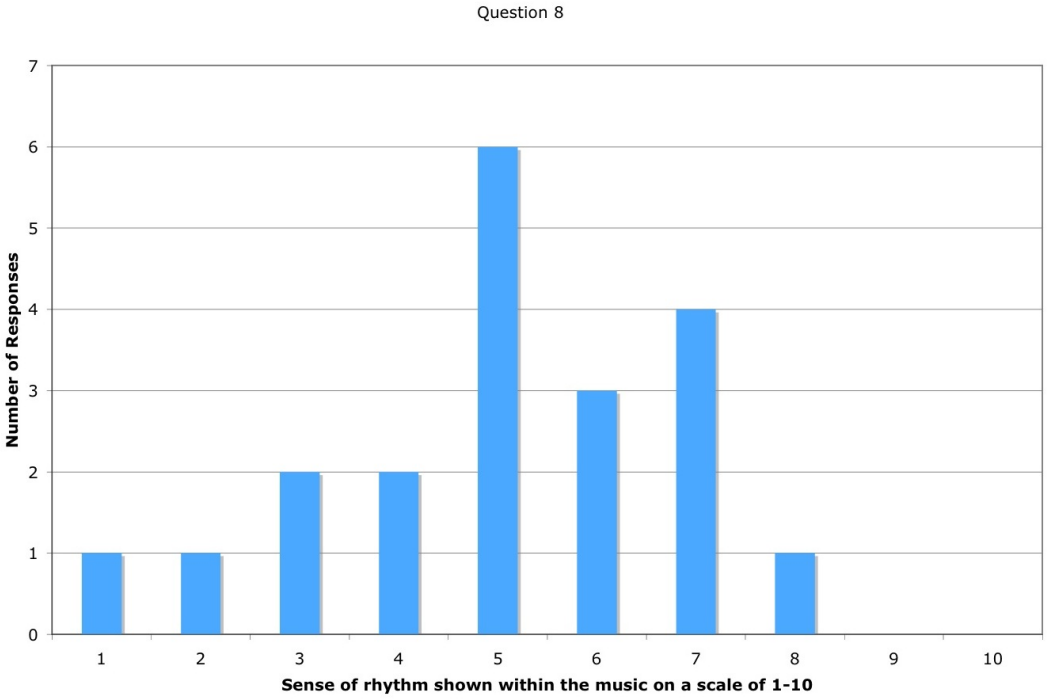

Figure 5.10: Question 8 Results

This again received a very balanced set of results which averaged out to around a five. The wide spread on these results suggests that the rhythm shown within the music, whilst not being liked by all, does generate results which can be liked by the evaluators. The wide range of results here, including a number of very low scores makes a strong match with the results received throughout the questionnaire, especially the ratings received by the melody.

9) Please rate how much you enjoyed listening to the music played on a scale of 1 - 10?

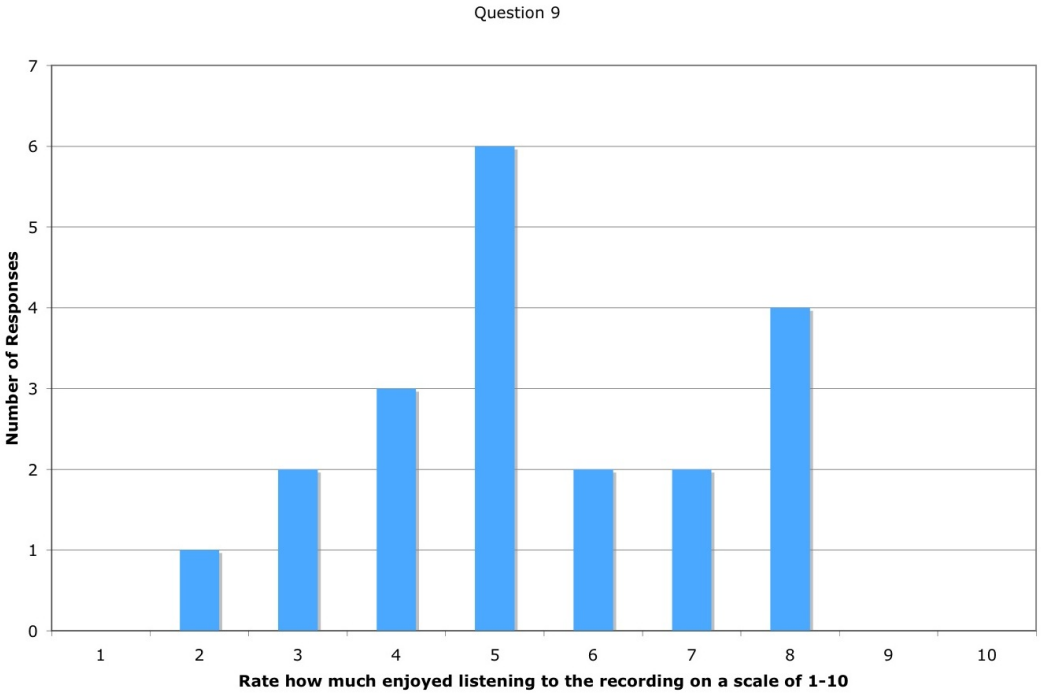

Figure 5.11: Question 9 Results

An average result of approximately five was received which is an extremely strong result on this question, especially when you consider that most listeners believed the music to have being played by a beginner or intermediate band. This result had a very wide range which may have been influenced by the listeners exposure to beginners bands in the past.

10) Please rate the quality of the band you believe would have played the recording of this music?

1. Beginner (1)
2. Intermediate (2)
3. Advanced (3)
4. Professional (4)

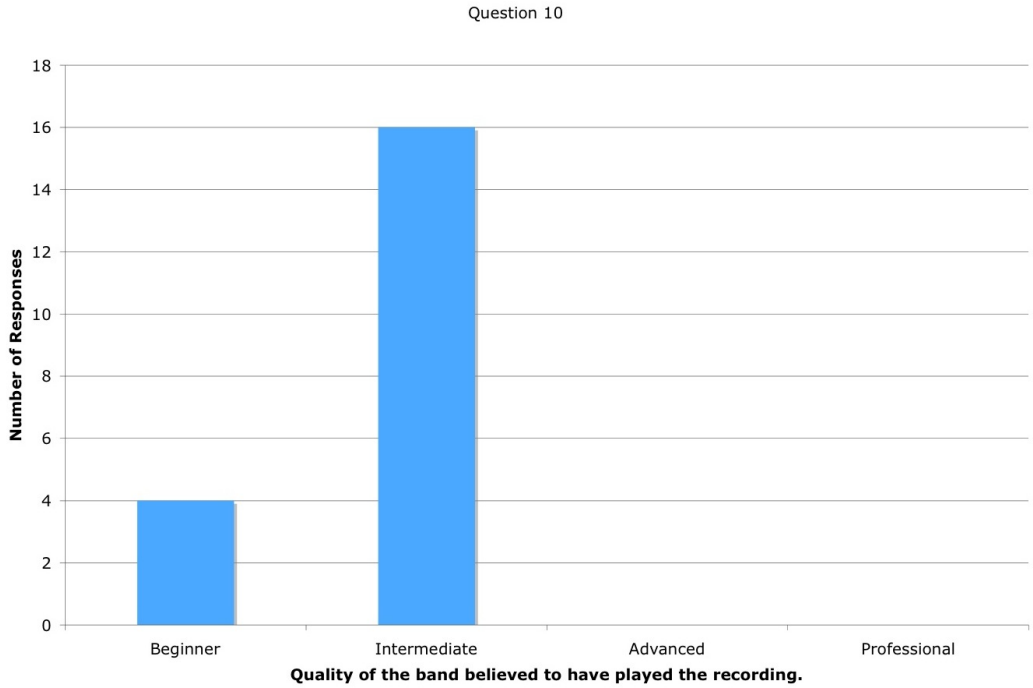

Figure 5.12: Question 10 Results

The answers to this were almost unanimous in that the music had either being performed by a beginner or intermediate standard band. This was as expected since the sound produced by the system sounds extremely computer generated and this was expected to sway the users opinion. This is why the playing of the recording was left until the questions on the score had being asked.

11) Although the sound is obviously computer generated, could you believe if played to you without any prior knowledge, that this was composed by a person, rather than a computer?

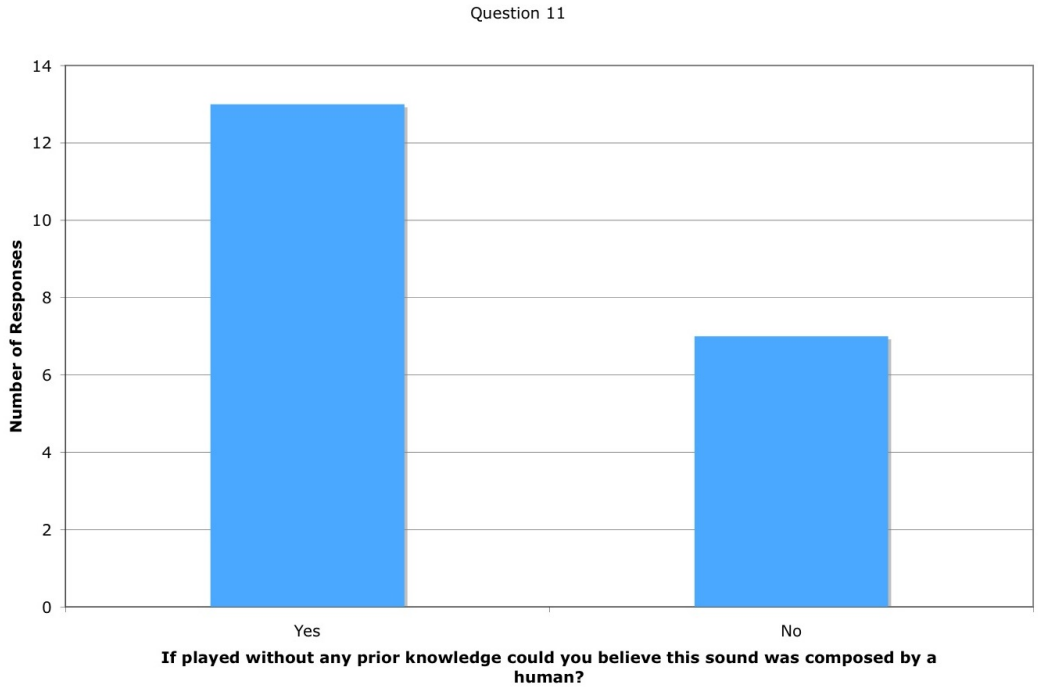

Figure 5.13: Question 11 Results

This question gave a very surprising response as at least half the people surveyed said that they could be fooled into thinking that a human had composed this music. This is much greater amount than expected, but may have been biased by respondents attempting to please the surveyor. If these results are accurate then this proves that we are well on the way to producing a decent automated composer. The comments received at the time suggested that this question should have been split into two separate ones. Firstly to consider the score and secondly to consider the recording generated. The commentees felt this would give a fairer feed back on the outputs of the program.

Unfortunately a couple of comments received during the survey leave me wondering whether the results here were biased by the fact that hardly any of the surveyed people had heard of the field of algorithmic composition and therefore had no knowledge that a computer could be used in this way. This may have influenced peoples views since they would automatically assume that anything had being composed by a human because they didn't realise there was any other way.

12) Any further comments

Numerous additional comments were received. A large number of these centered around the lack of dynamic changes within the music and how this would have affected the answers to other questions.

In response to being played a recording of the score, one individual commented that "The notes were computer generated but also played by a computer; if they were played by a good human musician maybe the music would sound much better". This leads to a future testing plan in which the music should be performed and recorded by a big band. This recording should then be used as the subject of a second survey which should include a much wider range of musical abilities.

5.1.2 Reflection

Overall from this questionnaire we can see that this system produces music that is either quite good or extremely dire, or anywhere within this range. This may be due to the instruments used for the recordings but could also be due to the fact that lines of each part of the music range throughout the available instruments. The feedback received about the bass line has being extremely positive and this could be reused to produce bass lines for future usage. The feedback received about the structure of the music was also extremely positive since it was mainly positive and centred around an already identified issue, the lack of dynamic changes.

5.2 Overall Project Evaluation

The fact that this has been a research project has dictated to a large extent the way in which it has been carried out. There have been three main sections to this project; subject research, requirements and design, and program implementation.

5.2.1 Subject Research

The subject matter for this field is extremely specific. It was found that there is a major lack of basic introductions to this field, so detailed research in books and journals had to be carried out. A number of past dissertations which dealt with sub-fields of algorithmic composition were found to be extremely important initial starting points for the research. The most used resource for this project has to be online databases of journals. These resources provided a good overview of the potential project so that specific mathematical texts could be used for more in-depth research on the chaotic systems and heuristic methods used. The information gained from these sources supplied an extremely strong standing when it came to the requirements and design phase.

Since one of the key aims of this project changed quite dramatically during the implementation process (from producing a program capable of composing big band music in a swing style to producing a system capable of harmonising multiple parts), a number of extra sources were consulted during the project. These gave a brief overview of some of the methods for harmonisation of multiple parts which have already been produced. This overview of the currently available methods exposed the untouched area of research which this dissertation has attempted to fill.

5.2.2 Requirements & Design

The requirements and design stage is an extremely important stage in most software engineering projects. Since this was a research project rather than a software engineering one, this section was relatively brief as opportunities for changes of requirements and major design changes had to feature within the implementation phase. This meant that the requirements produced provided a brief overview of the ranges of possibilities for the implementation. This was an extremely valuable phase since one of the outputs of the phase was a document containing the possible methods for each different phase of the music generation.

The final requirements produced were slightly over-optimistic for the timescale of the project and were therefore reduced slightly during the implementation phase to give a suitable level of complexity and required input time.

5.2.3 Program Implementation

The project was developed in an incremental fashion. This suited the project well since each part of the music could be considered individually. This meant that the schedule for implementation was:

1. Rolling bass line
2. Drum kit - this was implemented in a number of stages since there are multiple drums which could be considered relatively individually.
3. Bass Line
4. Melody
5. Solo Sections
6. Counter Melodies

This worked well since it meant that the issues associated with the section being implemented could be identified easily, as could the problem areas. The development of the bass lines prior to the melody and solo sections was also useful as it could be considered a training process. This is due to the fact that the level of complexity of each section increases as the implementation schedule progresses.

The actual development of the program centered mainly on an experimental method since it allowed the most freedom in terms of identifying solutions and solving the problems that occured. This was done on a trial and error basis rather than by following a specific plan since it allowed the most freedom to experiment with what sounded good. This worked well because a large number of ideas were trialed throughout the project and only the specific ones which worked well were kept in the final implementation.

The code produced by this project has been written to be functional rather than efficient, with the exception of the harmonisation algorithm as the main aim of this project was not software engineering, but rather identifying new techniques and areas for development. This means that the final code is not efficient or anywhere near optimal and therefore it is extremely far from what would be expected from a professional product. However, this has had little noticeable impact in terms of the functional aspects of the final ideas.

The program developed can not be considered to be perfect in any way. This is because it does not implement a number of aspects of normal music which would be expected in most actual performances. These include a large number of issues which have being identified in section 2.1.

The project was successful in meeting one of its main aims which was to develop a simple harmonisation algorithm. This was done through the harmonisation procedure identified in section 4.5.2. This also identified a large number of future issues which means that there is still a lot of work to be done in this field, although this project has provided a number of interesting results for the future.

Chapter 6

Conclusions

The initial aim of this project was to "generate music that could be considered to be in a big band swing style". This aim was changed slightly to be more about the production of a simple harmonisation procedure. Both these aims have been met to a certain degree.

I feel that the production of a simple harmonisation procedure has being fully met since the procedure implemented and tested is simple to implement and provides passable results in terms of the quality of output achieved.

The music produced by this program definitely does not have a professional sound, but it does have a swing feel to it. This swing feel is mainly provided by the rolling bass line. This project never expected to achieve a professional sound and so I am relatively happy with the outputs produced. If performed by a real big band then the music would be playable and would produce an interesting sound, though maybe not a sound which could be considered to be fully reminiscent of a big band.

Considering these points, it is reasonable to say that the overall objectives of this project have been met.

6.1 Requirement Review

In this section the requirements for this project (found in section 3.9) will be reviewed.

6.1.1 Overall Requirements

1. **Research should be carried out into possible ways of using chaotic systems with heuristic methods to carry out compositional procedures.**

   The resulting music was implemented as a mapping of chaotic systems to notes. This involved a small amount of research into the possible ways of using chaotic systems. Heuristic methods were not used to implement the project as a more beneficial alternative (although this was not too dissimilar from these) was found.

2. **The outputted composition should be a piece suitable for performance by a big band. This means that each part must be constrained to remain within the playable limits of the instruments.**

   Measures were taken to implement this solution and the outputted music remained within the playable limits of each instrument, meaning that this requirement has been met fully. I feel that a better solution could have being proposed for this problem, but was not a key aspect of the project.

3. **The program should output a single piece of music each time it is run.**

   This requirement was met in full.

4. **Each piece of music outputted by the program should show differences.**

   Comparing a number of pieces of music which all show different movements within the pitches proves that this requirement has been met. These differences were implemented in such a way as to ensure that they did not affect the structure of the music.

5. **The bass parts generated by the program should include a rolling bass line as-well as other parts such as drum parts, etc.**

   This can be shown through the multiple lines of the music.

6. **The music outputted should represent notes of varying lengths.**

   The music outputted shows differing note lengths depending on which part is being performed. It also shows different note lengths within each specific part to ensure that interest in the music is retained.

7. **The music outputted should represent notes which range over a variety of octaves.**

   This was implemented within the music: firstly, in that different parts can perform in different octaves and, secondly, that parts can range over a number of octaves as long as they remain within their own instrument's playable bounds.

8. **The music outputted should contain sections which are of a twelve bar blues style.**

   The music produced contains twelve bar blues sections which consist of multiple parts playing in harmony. The harmony was generated as defined in section 4.5.2.

9. **The music outputted should contain solo sections and an introduction.**

   The music outputted contains solo sections interspersed within the twelve bar blues sections. It also contains an introduction which consists of a drum part with rolling bass line.

10. **The music outputted should be of a relatively high complexity and should hold the listener's interest.**

    The music outputted is able to show a wide range of different notes and a selection of chords. Despite this, I do not feel this requirement has been fully met since some people described the music as aimless. This could be because of the lack of volume changes within the music.

11. **An overall structure should be visible within the music produced.**

    A pre-defined overall structure can be heard within the music produced since the sections of the music are very strongly defined. I also feel that the use of chaotic systems gives elements of structure to the twelve bar blues sections since each twelve bar blues section shows similarities to the others, but with a number of differences being present each time. This is due to the underlying qualities of chaotic systems which are described in section 2.2.3.

6.1.2 Melody Requirements

The requirements for the production of the melody were met to the standard needed to generate an appropriate style of music for this project. The main requirements and how they were met is detailed below:

1. **To implement a conservative system to produce data to be mapped.**

   The Henon Map was implemented to produce the data to be mapped to notes.

2. **To map from the data to notes in the correct scale.**

   The scales were implemented using the system developed by Morris (2005) within his dissertation project. This ensured that the music generated contained mainly, but not exclusively, notes from the blues scales.

3. **To allow repeated note structures, blues style riffs or repeated segments of the piece.**

   This was not explicitly implemented, but the element of freedom present meant that this could occur. This means that the issue of whether or not repeated note structures,

blues style riffs or repeated segments occur depends on the chaotic systems used to generate the data to be mapped to notes.

4. **To allow a variety of note lengths.**

   A wide range of differing note lengths can be performed by every part within the piece.

5. **To allow the music to range over multiple octaves.**

   It is possible for the melody to range over a range of different octaves.

6. **To build chords from the data.**

   Chords have been implemented in the system by firstly choosing a bass note for the chord first and then selecting a chord structure to be built on top of this. This means the melody consists of a series of chords rather than of a series of notes.

7. **To fit the melody part produced around the bass line.**

   The melody has been fitted around the bass line by two means:

   (a) The melody is constrained in order to be more likely to play a note on the first beat of each bar. This ensures that the music fits together since the bass line is also constrained to the same limits.

   (b) The melody is harmonised to fit with the bass line through the specific harmonisation function.

8. **To ensure that the notes flow from scale to scale within the blues sound.**

   This has been ensured by guaranteeing that the notes converge to a suitable note, one which is found in both scales, at the changeover point.

9. **To produce a melody with a blues sound.**

   Structurally the melody can be considered to have a blues sound because the conditions of its generation ensure that it meets the structural requirements of a piece of blues music. This requirement was checked in more detail by a survey of musicians which was carried out and is detailed in section 5.1.

10. **To maintain an overall structure in the melody.**

    An overall structure is ensured within the melody by mapping all notes from the same chaotic system. This means that the underlying structure of the chaotic system can be seen to come through within the music produced.

6.2 Future Developments

There are a large number of future possibilities. These could include:

6.2.1 Dynamics

The addition of dynamic changes to the music. This could be implemented by mapping a chaotic system over the whole project in a third pass through the music produced. This would give more overall structure to the music and produce music that could be considered to have a greater degree of musicality. In this way, the user assessments of the produced music would be improved.

Another possible method would be to investigate the use of three dimensional chaotic systems and to map one of the dimensions of the music to the dynamics whilst the other two were used for the pitch and length as before.

6.2.2 Solo Part

The development of the solo part to use more techniques (such as accenting) and to provide a greater degree of freedom within the solo parts.

The solo part could also be developed to use a much more random chaotic system since this may provide a good swing solo.

6.2.3 Harmonisation and Chord Selection

The development of the harmonisation aspects of the program to enable it to work for a greater number of chords and to provide different degrees of chords so that more harmonic patterns can be used within the music.

The harmonisation function as a whole could be upgraded to harmonise more notes within the music and to specifically select the points where discords should be found. This may provide a more standard musical sound to the program since most music uses much more harmonic music rather than discords.

6.2.4 Song Structure

The development of a way to automate the song structure selection so that it is generated by a chaotic algorithm. This could be a simple mapping using a chaotic system to determine the lengths of each section, but I would consider this to be excessive. Instead, it may be more suitable to simply use a pseudo-random number generator with specific limits to the output to generate the length of each section.

6.2.5 Final Section

The music in its current state does not have a strong build up to a sensible ending point. This is a significant problem within the program since it means the music seems to finish relatively abruptly. Therefore the development of a function that generates a finishing section for the piece would be an interesting future improvement. This could implement a mapping of a system which converges to a point since the convergence point could be used for the end of the piece. The system chosen would have to be similar to the Henon Map or Standard Map since it would then ensure that a continuous feel could be identified throughout the music.

6.2.6 CSound instruments

Further develop the instruments used within CSound for performance of this music to achieve a more authentic big band sound. This would require extensive mathematical modelling of the required instruments in order to implement them within the mathematical constraints of CSound. There is a wide range of possible starting points for this work, mainly based on the numerous sources of information where people have already generated instruments within most categories of the instrumental families.

Piano

Within the bounds of generating new CSound instruments, consideration must be given to the generation and implementation of a realistic sounding piano part. This is an extremely difficult topic in itself and is a current area of research. This generation of a CSound piano would then need to be followed by the implementation of a generation algorithm for the music. This is because generation of piano parts as there are a great number of techniques which can be incorporated (including pedals, comping and the merging of two notes to give the impression of mediums between notes).

6.2.7 Links between musical parts

Currently the links between each musical part are relatively low due to the random selection of which part should play each note within the melody, bass line and counter melody are very random. There are many different possibilities to improve this section. The main improvements would be to improve the links between the parts by generating them all simultaneously and therefore allowing all of the parts to be adjusted at the same time to fit with each other.

6.2.8 Chords to Parts mapping algorithm

The algorithm used is extremely rudimentary at the moment since it simply selects an instrument that is not playing at the current time, with no concern for which chord pattern line each instrument follows. This could be improved to make instruments stick to a specific line within the chord and to attempt to ensure that parts appear to have a direction and motivation.

Another consideration that could be made in this section is to ensure more section-specific conditions are included. These would include attempting to group sections together when they are playing, and to give the trumpet part, as the band leader, the highest part most of the time.

6.2.9 Introduction Section

The current introduction section consists of a double bass and drum solo. This is a relatively normal introduction section though another method would be to allow each part to join at a randomly selected time to form a build up into the main music. This may provide a better sound to the music since it will add more structure to the music as a whole.

6.2.10 Chaotic Systems Used

Within the current program only two-dimensional chaotic systems have been used. This could be extended to use systems with many dimensions. These extra dimensions could be used for aspects such as the amplitude of the notes or to determine whether a note should be legato, staccato or normal. An analysis of more systems may also reveal ones which are better for big band music since time constraints meant that the selection of which systems to use was a relatively rushed process.

This would make a good future dissertation topic since there is a large amount of room for research and experimentation.

6.3 Contribution & Relation to Other Work

This project showed that methods using chaotic systems could be used to great effect to form complex music. This built on the previous work by Morris (2005) which showed that chaotic systems could be used to generate simple musical structures, such as music consisting of a melody and bass line.

One area that was identified as being of current interest in the literature survey was that of adding structure to music. Eck and Schmidhuber (2002), Burr and Miyata (1993) and Dubnov et al. (2003) have all previously conducted research into the possibilities of adding structure into music through chaotic systems. This previous work was instrumental in the selection of chaotic systems for the main music generation methods. By using chaotic systems in this way, this project has provided some extra weight to the argument that chaotic systems have an inbuilt structure which can be portrayed within music when a suitable mapping is used. This project has also provided weight to the argument that the structure evident within chaotic systems can make interesting and good sounding music.

This project's work produces music which some listeners consider to be interesting, while others find it offensive. This is very different to the work of Eck and Schmidhuber (2002) which produced music which had no major structure, but which was less likely to be considered offensive by listeners. Future work needs to consider the issues of removing these offensive notes within the music, therefore allowing for interesting music that can also be classed as being easy to listen to.

6.4 Final Conclusion

This program has composed music that could be considered to be in the style of a big band playing swing music. The music produced is not rated highly by the listeners who took part in the questionnaire. This seems to be due, in a large part, to the fact that the system does not model additional conditions such as differing volumes.

The project as a whole can be considered a relevant and valuable contribution to the field of algorithmic composition since a majority of the time was spent studying the question of harmonisation of multiple parts within a piece of music and the final program produced implemented a simple to use system for the harmonisation of a variable number of parts.

The other issue that was covered in some detail is the use of chaotic systems to generate music. The findings here matched with the findings of Morris (2005) in that chaotic systems have relevant structural points which can be exploited within the compositional process to produce an underlying structure within the music generated.

These points lead me to believe that the main objectives of this project have been met to a limited degree and a number of new objectives for future work have been identified. Due to this identification of new objectives and the system produced for harmonisation of multi part music I feel that this project has been a success.

Bibliography

Alpern, A. (1995), *Techniques for Algorithmic Composition of Music*, Hampshire College.

Bidlack, R. (1992), *Chaotic systems as simple (but complex) compositional algorithms*, Computer Music Journal, 16(3):33-47.

Boulanger, R. (2000), *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing and Programming*, The MIT Press.

Burr, D. J. and Miyata, Y. (1993), 'Heirarchical recurrent networks for learning musical structure', *Neural Networks for Signal Processing* **Proceedings of the 1993 IEEE-SP Workshop**, 216–225.

Devany, R. L. (2003), *A first course in chaotic dynamical systems*, Addison-Wesley Publishing Company.

Dubnov, S., Assayag, G., Lartillot, O. and Berjerano, G. (2003), *Using machine-learning methods for musical style modelling*, Computer, 36(10):73-80.

Eck, D. and Schmidhuber, J. (2002), 'Finding temporal structure in music: blues improvisation with lstm recurrent networks', *Neural Networks for Signal Processing* **Proceedings of the 12'th IEEE Workshop**, 747–756.

Fitch, J. (2007), *Music and Digital Signal Processing Lectures*, None.

Gibson, P. M. and Byrne, J. A. (1991), 'Neurogen, musical composition using genetic algorithms and cooperating neural networks', *Artificial Neural Networks* **Second International Conference, Volume 1**, 747–756.

Leach, J. and Fitch, J. (1995), *Nature, music and algorithmic composition*, Computer Music Journal, 19(2):22-33.

Marques, M., Oliveira, V., Vieira, S. and Rosa, A. C. (2000), 'Music composition using genetic evolutionary algorithms', *Evolutionary Computation* **Proceedings of the 2000 Congress, Volume 1**, 714–719.

Moorer, J. A. (1972), *Music and algorithmic composition*, Commun. ACM, 15(2):104-113.

Morris, P. (2005), Programming a computer to play the 12-bar blues, Master's thesis, University of Bath.

Ortega de la Puente, A., Alfonso, R. S. and Moreno, M. A. (2002), 'Automated composition of music by means of grammatical evolution.', *APL '02: Proceedings of the 2002 conference on APL* pp. 148–155.

Smoliar, S. W. (1991), *Algorithms for musical composition: a question of granularity*, Computer, 24(7):54-56.

Thom, B. (2000), *Bob: an interactive improvisational music companion*, AGENTS '00 Proceedings of the fourth international conference on Autonomous agents, pages 309-316. ACM Press.

Appendix A

Code

The code is split into four seperate sections:

1. Algorithmic composer. This section presents the code for the algorithmic composer program.
2. Amplitude validator. This section presents the code for the amplitude validator.
3. Instruments. This section presents the big band instruments as implemented in CSound.
4. Instruments. This section presents the function tables for the big band instruments detailed in the previous section.

A.1 File: compose.c

```
/***********************
*                                                *
*              compose.c                 *
*                                                *
***********************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

/*
    1 bar = 32 16th notes
    Each section is a twelve bar blues section.
*/
#define BAR_LENGTH (4 * 2 * 2 * 2)
/*
    12 bars = 384 16th notes
    Each section is a twelve bar blues section.
*/
#define SECTION_LENGTH (BAR_LENGTH * 12)
/*
    4 bars
    Each fill or solo section is a chaotic section consisting of four bars.
    The first fill is an introduction section which consists of a bass solo.
*/
#define FILLSOLO_LENGTH (BAR_LENGTH * 4)
/*
    4 sections.
*/
#define NUMBER_OF_SECTIONS (4)
/*
    The total song length is NUMBER_OF_SECTIONS sections plus NUMBER_OF_SECTIONS fills since one extra fill is used
        for the intro.
*/
#define SONG_LENGTH ((SECTION_LENGTH + FILLSOLO_LENGTH) * NUMBER_OF_SECTIONS)
#define HENON_A 1.4 /*1.076 for a chaotic bass line orbit*/
#define HENON_B 0.3
#define STANDARD_K 2
#define STANDARD_I 2.02
#define STANDARD_T 3.10
```

```
/*Variables to hold and use the note values for the scales*/
double cScale[6] = {0, 0.03, 0.05, 0.06, 0.07, 0.10};
double fScale[6] = {0.05, 0.08, 0.10, 0.11, 1, 1.03};
double gScale[6] = {0.07, 0.10, 1, 1.01, 1.02, 1.05};
double *currentScale;
double octave = 7;

/*Variables for the Henon Map*/
double Henon_x = 0;
double Henon_y = 0;

/*Variables for the Standard Map*/
double S_iota = STANDARD_I;
double S_theta = STANDARD_T;
double S_K = STANDARD_K;

/*The structure for holding the note*/
typedef struct NOTE
{
        double start;
        double duration;
        double pitch;
        double *scale;
        double amplitude;
} NOTE;

/*
 * Variables to hold the allowed chords and their probablities.
 * Types:  1 - Major
 *         2 - Minor
*/
typedef struct CHORD
{
    int notes[4];
    NOTE *bassNote;
    double octaveBassNote;
    int pitchBassNote;
    int type;
    double probability;
} CHORD;

CHORD *bassChordArray[SECTION_LENGTH];
CHORD *melodyChordArray[SECTION_LENGTH];
CHORD *counterChordArray[SECTION_LENGTH];
```

```
CHORD *major7th;
CHORD *minor7th;

/*The structure for holding the part*/
typedef struct INSTRUMENT
{
        NOTE *noteArray[SONG_LENGTH];
        char *name;
        int instrument;
        int type;  //Holds 1 if its mainly a bass instrument or 0 if its mainly a melody instrument.
        int clef;  //Holds 1 if its a bass clef instrument or 0 if its a treble clef instrument.
        double soloPriority;
        double maxRange;
        double minRange;
        struct INSTRUMENT *next;
} INSTRUMENT;

/*Variables to hold and control the parts of the piece*/
int parts;  //Stores the number of parts defined within the piece.
INSTRUMENT *bottomInstrument = NULL;
INSTRUMENT *soloist;
INSTRUMENT *melody;
INSTRUMENT *counterMelody;
/*Specific instruments*/
INSTRUMENT *snare;
INSTRUMENT *toms;
INSTRUMENT *bassDrum;
INSTRUMENT *bass;

/*This is the function to create a new part*/
INSTRUMENT *newInstrument(char *instrumentName, int instrumentNumber, double soloPriority, double maxRange, double
    minRange, int type, int clef)
{
        int i;
        parts++;
        INSTRUMENT *part;
        INSTRUMENT *pointer;
        part = (INSTRUMENT *)(malloc(sizeof(INSTRUMENT)));
        for (i = 0; i < SONG_LENGTH; i++)  {
                part->noteArray[i] = NULL;
        }
        part->next = NULL;
        part->name = (char *) malloc(sizeof(char[256]));
    i = 0;
```

```
    while(i < strlen(instrumentName))  {
        part->name[i] = instrumentName[i];
        i++;
    }
        part->instrument = instrumentNumber;
        part->type = type;
        part->clef = clef;
        part->soloPriority = soloPriority;
        part->maxRange = maxRange;
        part->minRange = minRange;
        if (bottomInstrument == NULL) bottomInstrument = part;
        else  {
                pointer = bottomInstrument;
                while (pointer->next != NULL) pointer = pointer->next;
                pointer->next = part;
        }
        return part;
}

/*This is the function to create a new note*/
NOTE *newNote()
{
        NOTE *note = (NOTE *)(malloc(sizeof(NOTE)));
        note->start = 0;
        note->duration = 0;
        note->pitch = 0;
        note->amplitude = 0;
        note->scale = NULL;
        return note;
}

NOTE *getPreviousNoteOnInstrument(INSTRUMENT *part, int startPosition)
{
    while(startPosition > 0)  {
        if(((int) part->noteArray[startPosition]) != 0)  {
            NOTE *x = part->noteArray[startPosition];
            return x;
        }
        startPosition--;
    }
    return NULL;
}

NOTE *getPreviousNote(int startPosition)
{
```

```
    INSTRUMENT *savedBottomInstrument = bottomInstrument;
    while(startPosition > 0) {
        while(bottomInstrument != NULL) {
            if(((int) bottomInstrument->noteArray[startPosition]) != 0) {
                NOTE *x = bottomInstrument->noteArray[startPosition];
                bottomInstrument = savedBottomInstrument;
                return x;
            }
            bottomInstrument = bottomInstrument->next;
        }
        startPosition--;
        bottomInstrument = savedBottomInstrument;
    }
    return NULL;
}

/*
 * Create a new chord.
*/
CHORD *newChord() {
    CHORD *chord = malloc(sizeof(CHORD));
    chord->bassNote = malloc(sizeof(NOTE));
    chord->type = 0;
    return chord;
}

/*
 * Select a chord to play in a psedo random method and then generate a new chord object of this type and return it
     to the caller.
*/
CHORD *selectChord() {
    double x = (rand() % 10000) / 10000;
    CHORD *selectedChord = newChord();
    if(x < major7th->probability) {
        selectedChord->notes[0] = major7th->notes[0];
        selectedChord->notes[1] = major7th->notes[1];
        selectedChord->notes[2] = major7th->notes[2];
        selectedChord->notes[3] = major7th->notes[3];
        selectedChord->type = major7th->type;
        selectedChord->probability = major7th->probability;
    } else {
        selectedChord->notes[0] = minor7th->notes[0];
        selectedChord->notes[1] = minor7th->notes[1];
        selectedChord->notes[2] = minor7th->notes[2];
        selectedChord->notes[3] = minor7th->notes[3];
```

```
        selectedChord->type = minor7th->type;
        selectedChord->probability = minor7th->probability;
    }
    selectedChord->octaveBassNote = 0.0;
    selectedChord->pitchBassNote = 0.0;
    return selectedChord;
}

CHORD *harmoniseChords(CHORD *bass, CHORD *melody)
{
    double *scale;

    double bassNote = bass->bassNote->pitch;
    double melodyNote = melody->bassNote->pitch;
    double bassNotePitch = bassNote - (int)bassNote;
    double melodyNotePitch = melodyNote - (int)melodyNote;

    if(bassNotePitch != melodyNotePitch) {
        //Check each possible note within the chord for if the melody note fits with a note in the bass chord.
        //First set the scale to be the bass note.

        melody->bassNote->pitch = bass->bassNote->pitch - (int)bass->bassNote->pitch + (int)melody->bassNote->pitch;
    }

    if(bass->type != melody->type) {
        //Correct the melody to match the same chord type.
        melody->type = bass->type;
        melody->notes[0] = bass->notes[0];
        melody->notes[1] = bass->notes[1];
        melody->notes[2] = bass->notes[2];
        melody->notes[3] = bass->notes[3];
    }
    return melody;
}

/*This is the function to print out the piece to Csound format*/
void printSong()
{
        int i;
        NOTE *note;

        INSTRUMENT *savedBottomInstrument = bottomInstrument;

        /*This loop goes through the parts and prints out all the notes*/
        while(bottomInstrument != NULL) {
```

```
printf(";%s\n", bottomInstrument->name);
printf(";instrument start duration amplitude pitch\n");
        for (i = 0;i<SONG_LENGTH;i++) {
    if((i % (SECTION_LENGTH + FILLSOLO_LENGTH)) == 0) {
        printf(";Section\n");
    }
                note = bottomInstrument->noteArray[i];
                if(note != NULL) {
        //Print a different string depending on which type of instrument it is.
                        if(strcmp(bottomInstrument->name, "BASS") == 0) {
                            printf("i%i %1.2f %1.2f %1.2f %1.2f\n",bottomInstrument->instrument,
                                note->start, note->duration, note->pitch, note->amplitude);
                        } else if(strcmp(bottomInstrument->name, "SNARE") == 0) {
            printf("i%i %1.2f %1.2f %1.2f\n", bottomInstrument->instrument, note->start, note->duration,
                note->amplitude);
        } else if(strcmp(bottomInstrument->name, "TOMS") == 0 || strcmp(bottomInstrument->name,
            "FLOOR_TOM") == 0) {
            printf("i%i %1.2f %1.2f %1.2f\n", bottomInstrument->instrument, note->start, note->duration,
                note->amplitude);
        } else if(strcmp(bottomInstrument->name, "BASS_DRUM") == 0) {
            printf("i%i %1.2f %1.2f %1.2f\n",bottomInstrument->instrument, note->start, note->duration,
                note->amplitude);
        } else if(strcmp(bottomInstrument->name, "PIANO") == 0) {
            printf("i18 %1.2f %1.2f %1.2f %1.2f\n", note->start, note->duration, note->pitch,
                note->amplitude);
        } else if(strcmp(bottomInstrument->name, "TRUMPET") == 0) {
            printf("i%i %1.2f %1.2f %1.2f %1.2f\n", bottomInstrument->instrument, note->start,
                note->duration, note->amplitude, note->pitch);
        } else if(strcmp(bottomInstrument->name, "FIRST_TROMBONE") == 0) {
            printf("i%i %1.2f %1.2f %1.2f %1.2f\n", bottomInstrument->instrument, note->start,
                note->duration, note->amplitude, note->pitch);
        }  else if(strcmp(bottomInstrument->name, "SECOND_TROMBONE") == 0) {
            printf("i%i %1.2f %1.2f %1.2f %1.2f\n", bottomInstrument->instrument, note->start,
                note->duration, note->amplitude, note->pitch);
        } else if(strcmp(bottomInstrument->name, "BASS_TROMBONE") == 0) {
            printf("i%i %1.2f %1.2f %1.2f %1.2f\n", bottomInstrument->instrument, note->start,
                note->duration, note->amplitude, note->pitch);
        } else if(strcmp(bottomInstrument->name, "ALTO_SAXAPHONE") == 0) {
            printf("i%i %1.2f %1.2f %1.2f %1.2f 4\n", bottomInstrument->instrument, note->start,
                note->duration, note->amplitude, note->pitch);
        } else if(strcmp(bottomInstrument->name, "TENOR_SAXAPHONE") == 0) {
            printf("i%i %1.2f %1.2f %1.2f %1.2f 4\n", bottomInstrument->instrument, note->start,
                note->duration, note->amplitude, note->pitch);
        } else if(strcmp(bottomInstrument->name, "BARITONE_SAXAPHONE") == 0) {
```

```
                    printf("i%i %1.2f %1.2f %1.2f %1.2f 4\n", bottomInstrument->instrument, note->start,
                        note->duration, note->amplitude, note->pitch);
                } else {
                    printf(";/Error, instrument not printed - %p %p %s.\n", bottomInstrument->name,
                        bottomInstrument, bottomInstrument->name);
                }
                        }
                }
                bottomInstrument = bottomInstrument->next;
        }
        printf("e\n");
        bottomInstrument = savedBottomInstrument;
}

int convertDuration(double duration)
{
    double x = 1/duration;
    x = x * 4;

    int y = (int) x;

    switch(y)  {

    case 3:
    case 5:
        y = 4;
        break;

    case 6:
    case 7:
    case 9:
    case 10:
        y = 8;
        break;

    case 11:
    case 12:
    case 13:
    case 14:
    case 15:
    case 17:
    case 18:
        y = 16;
        break;
    }
```

```
    return y;
}

/*
 * Convert a note into a string for lilypond output.
 */
char *Note_toString(NOTE *n, char *s) {
        char *note = calloc(4,sizeof(char));
        char *octave = calloc(7,sizeof(char));
        if (n != NULL) {

        int intPitch = (int) ((((double) n->pitch) - ((int) n->pitch)) * 100);
        int intOctave = (int) (n->pitch - (((double) intPitch)/100));
        intOctave = intOctave - 6;

                switch (intPitch) {
                        case 0:
                                note = "c";
                                break;
                        case 1:
                                note = "des";
                                break;
                        case 2:
                                note = "d";
                                break;
                        case 3:
                                note = "ees";
                                break;
                        case 4:
                                note = "e";
                                break;
                        case 5:
                                note = "f";
                                break;
                        case 6:
                                note = "ges";
                                break;
                        case 7:
                                note = "g";
                                break;
                        case 8:
                                note = "aes";
                                break;
                        case 9:
```

```
                        note = "a";
                        break;
                case 10:
                        note = "bes";
                        break;
                case 11:
                        note = "b";
                        break;
        }
        switch (intOctave) {
                case -3:
                        octave = ",,,";
                        break;
                case -2:
                        octave = ",,";
                        break;
                case -1:
                        octave = ",";
                        break;
                case 0:
                        octave = "\0";
                        break;
                case 1:
                        octave = "'";
                        break;
                case 2:
                        octave = "''";
                        break;
                case 3:
                        octave = "'''";
                        break;
                case 4:
                        octave = "''''";
                        break;
                case 5:
                        octave = "'''''";
                        break;
        }

        sprintf(s, "%s%s%d", note, octave, convertDuration(n->duration));
} else {
        note = "r";
        octave = "\0";

        sprintf(s, "%s%s32", note, octave);
```

```
        }
        return s;
}

/*
 * Convert the next bar of the piece into a string for lilypond output.
 */
char *Bar_toString(INSTRUMENT *currInt, int songPosition, char *s) {
        int i;
        for (i = songPosition; (i < songPosition + BAR_LENGTH) && (i < SONG_LENGTH); ) {
            char buffer[12];
            if(currInt->noteArray[i] != NULL) {
            strcat(s, Note_toString(currInt->noteArray[i], buffer));
            if(currInt->noteArray[i] != NULL && (i < (songPosition + BAR_LENGTH))) {
                i = i + (currInt->noteArray[i]->duration * 32);
            }
        } else {
            int restDuration = 0;
            while((currInt->noteArray[i + restDuration] == NULL) && (i < songPosition + BAR_LENGTH) && (i <
                SONG_LENGTH)) {
                restDuration++;
            }

            if(restDuration > BAR_LENGTH) {
                restDuration = BAR_LENGTH;
            }

            switch(restDuration) {
            case 32:
                i = i + 32;
                restDuration = 1;
                break;
            case 31:
            case 30:
            case 29:
            case 28:
            case 27:
            case 26:
            case 25:
            case 24:
            case 23:
            case 22:
            case 21:
            case 20:
            case 19:
```

```
            case 18:
            case 17:
            case 16:
                i = i + 16;
                restDuration = 2;
                break;
            case 15:
            case 14:
            case 13:
            case 12:
            case 11:
            case 10:
            case 9:
            case 8:
                i = i + 8;
                restDuration = 4;
                break;
            case 7:
            case 6:
            case 5:
            case 4:
                i = i + 4;
                restDuration = 8;
                break;
            case 3:
            case 2:
                i = i + 2;
                restDuration = 16;
                break;
            case 1:
                i = i + 1;
                restDuration = 32;
                break;
            }

            sprintf(buffer, "r%i", restDuration);
            strcat(s, buffer);
        }
                strcat(s, "␣");

        }
        //Add an end of bar mark
        strcat(s, "␣|␣");
        return s;
}
```

```
/*
 * Print the music out to a lilypond input file.
*/
void printLilyPond(char *composition) {
        if (composition) {
                int i, j, size;
                char outfile[200], *key;
                FILE *f;
                sprintf(outfile,"./%s.ly",composition);
                f = fopen(outfile,"w");
                key = calloc(5,sizeof(char));
        key = "c";

        fprintf(f,"\\version \"2.10.20\"\n");
        fprintf(f,"\\header {\n\ttitle = \"Algorithmic Composition of Big Band Music Dissertation\"\n\tinstrument =
            \"For Big Band\"\n\tcomposer = \"Mike Tonge\"\n}\n");
        fprintf(f,"\\paper{\n\t#(set-paper-size \"a3\" 'landscape)\n\tragged-bottom = ##t\n}\n\\score {\n\t <<\n");
             //<<\n\t\\new StaffGroup
        fprintf(f,"\t\t\\tempo 4=90\n\t\t\\time 4/4\n");

        INSTRUMENT *savedBottomInstrument = bottomInstrument;
        /*This loop goes through the parts and prints out all the notes*/
        while(bottomInstrument != NULL)  {

            if(strcmp(bottomInstrument->name, "SNARE") == 0 || strcmp(bottomInstrument->name, "TOMS") == 0 ||
                strcmp(bottomInstrument->name, "BASS_DRUM") == 0)  {
                fprintf(f,"\t\t\\new DrumStaff \\with {\n\t\t\tfontSize = #-3\n\t\t\t\\override StaffSymbol
                    #'staff-space = #(magstep -3)\n\t\t}\n\t\t{\n\t\t\t\\override DrumStaff.InstrumentName
                    #'space-alist = #'((left-edge extra-space . 2.0))\n\t\t\t\\set DrumStaff.instrumentName =
                    \"%s\"\n\t\t\t\\set DrumStaff.shortInstrumentName = \"%s\"\n", bottomInstrument->name,
                    bottomInstrument->name);
            } else  {
                fprintf(f,"\t\t\\new Staff \\with {\n\t\t\tfontSize = #-3\n\t\t\t\\override StaffSymbol
                    #'staff-space = #(magstep -3)\n\t\t}\n\t\t{\n\t\t\t\\override Staff.InstrumentName #'space-alist
                    = #'((left-edge extra-space . 2.0))\n\t\t\t\\set Staff.instrumentName = \"%s\"\n\t\t\t\\set
                    Staff.shortInstrumentName = \"%s\"\n", bottomInstrument->name, bottomInstrument->name);
            }
            if(strcmp(bottomInstrument->name, "SNARE") == 0 || strcmp(bottomInstrument->name, "TOMS") == 0 ||
                strcmp(bottomInstrument->name, "BASS_DRUM") == 0)  {
                fprintf(f,"\t\t\t\\clef percussion\n\t\t\t\\key %s \\major\n",key);
            } else if(bottomInstrument->clef == 1)  {
                fprintf(f,"\t\t\t\\clef bass\n\t\t\t\\key %s \\major\n",key);
            } else  {
                fprintf(f,"\t\t\t\\clef treble\n\t\t\t\\key %s \\major\n",key);
```

```
            }

            for (i = 0; i <= SONG_LENGTH; ) {
                char *buffer = calloc(1000,sizeof(char));
                fprintf(f,"\t\t\t%s\n", Bar_toString(bottomInstrument, i, buffer));
                i = i + BAR_LENGTH;
                free(buffer);
            }
            bottomInstrument = bottomInstrument->next;

            fprintf(f,"\t\t}\n");
                }
                bottomInstrument = savedBottomInstrument;

                //End the staff group
                fprintf(f,"\t>>\n");  //\t>>\n

        fprintf(f,"\t\\midi {}\n");
                fprintf(f,"\t\\layout { }\n");
                fprintf(f,"}\n");
                fclose(f);
        }
}

/*This function performs a single orbit of the Henon Map*/
void Henon(double A, double B)
{
        /*Taken from Bidlack, R, Chaotic Systems as Simple (but Complex) Compositional Algorithms, Computer Music
            Journal*/
        double new_x, new_y;
        new_x = Henon_y + 1.0 - (A * (Henon_x * Henon_x));
        new_y = B * Henon_x;
        Henon_x = new_x;
        Henon_y = new_y;
}

/*This function performs the orbits of the Henon map a random number of times to move past the transient phase.*/
void HenonXTimes()
{
        int i;
        int x = rand();
        x = x%50000;
        for (i=0; i < x; i++) Henon(HENON_A, HENON_B);
}
```

```
/*This function performs a single orbit of the Standard Map*/
void Standard(double K)
{
        /*Taken from Bidlack, R, Chaotic Systems as Simple (but Complex) Compositional Algorithms, Computer Music
            Journal*/
        double new_iota, new_theta, TWO_PI = 6.283185;
        new_iota = S_iota + (K * sin(S_theta));
        new_theta = S_theta + new_iota;
        S_iota = new_iota;
        S_theta = new_theta;
        while (S_iota <0.0) S_iota = S_iota + TWO_PI;
        while (S_iota >= TWO_PI) S_iota = S_iota - TWO_PI;
        while (S_theta <0.0) S_theta = S_theta + TWO_PI;
        while (S_theta >= TWO_PI) S_theta = S_theta - TWO_PI;
}

/*This is the function that maps the data from the henon map to notes*/
void mapHenon(CHORD *part[], double A, double B, int length)
{
        double min, max; /*These are used to find the maximum and minimum values of the data*/
        double range; /*This holds the range of the values of the Henon Map*/
        int direction = 1; /*This variable holds the direction the bass line should be moving*/
        int barCount = 0; /*This variable is used to hold the position within the current bar*/
        int barProgression = 0; /*This is used to store the progression of the piece through the 12 bar structure*/
        int i; /*Loop control variable*/
        int songPosition = 0;
        int currentPosition = 0;
        int currentNote;
        NOTE *note;
        int m = 1; /*Variable used to change the granularity of the bass line*/
        HenonXTimes();
        range = (1.265227 - -0.754202) / 7;
        /*This is the main loop for processing the pitch values*/
        currentScale = cScale;
        songPosition = 0;
        i = 0;
        octave = 8.0;
        /*Carry on for length of section*/
        while (songPosition < length) {

        /*Code for controlling the scale being played*/
        if (barProgression >= (320*1)) currentScale = cScale;
        else if (barProgression >= (288*1)) currentScale = fScale;
        else if (barProgression >= (256*1)) currentScale = gScale;
        else if (barProgression >= (192*1)) currentScale = cScale;
```

```
else if (barProgression >= (128*1)) currentScale = fScale;
if (barProgression == 384)  {
    barProgression = 0;
}
/*Create new note and set standard values*/
note = newNote();
note->scale = currentScale;
note->amplitude = 6500;

/*This code will bring the song position onto a beat of the bar if it is within 2 32ths of a bar of a beat*/
if (((int)songPosition % 8) >= 6) songPosition = songPosition + (8 -((int)songPosition % 8));
else if ( ( ((int)songPosition % 8) <= 2) && ( ((int)songPosition % 8) >= 1) )songPosition = songPosition -
    (8 - ((int)songPosition % 8));
/*Set the note start*/
note->start = (songPosition/8);

/*This is what maps the pitch values to notes*/
/*Firstly iterate the henon map*/
Henon(A, B);
/*The first two clauses put in the scales root notes if it is the first beat of the bar*/
if ((barCount == 0) && (direction == 1))  {
    note->pitch = octave + currentScale[0];
    currentNote = 0;
} else if ((barCount == 0) && (direction == -1))  {
    note->pitch = octave + currentScale[0] + 1;
    octave--;
    currentNote = 0;
}
/*The rest of the clauses map the pitch value to the correct note*/
else if (Henon_x < (min + range/2))  {
    note->pitch = octave + currentScale[0];
    currentNote = 0;
} else if (Henon_x < (min + range))  {
    note->pitch = octave + currentScale[1];
    currentNote = 1;
} else if (Henon_x < (min + 2*range))  {
    note->pitch = octave + currentScale[2];
    currentNote = 2;
} else if (Henon_x < (min + 4*range))  {
    note->pitch = octave + currentScale[3];
    currentNote = 3;
} else if (Henon_x < (min + 6*range))  {
    note->pitch = octave + currentScale[4];
    currentNote = 4;
} else if (Henon_x < (min + 6*range + range/2))  {
```

```
    note->pitch = octave + currentScale[5];
    currentNote = 5;
} else if (Henon_x < (min + 7*range))  {
    note->pitch = octave + currentScale[0] + 1;
    octave++;
    currentNote = 0;
}

double pitchValue;
double bassValue;

currentPosition = songPosition;
/*Calculate note duration, current position is used to hold the current position so songPosition can be
    updated*/
if (/*S_theta*/ S_iota < (min + 2*range))  {
    note->duration = 0.5;
    songPosition += 4;
    barProgression += 4;
} else if (/*S_theta*/ S_iota < (min + 4.33*range))  {
    note->duration = 1;
    songPosition += 8;
    barProgression += 8;
} else if (/*S_theta*/ S_iota < (min + 6.33*range))  {
    note->duration = 2;
    songPosition += 16;
    barProgression += 16;
} else  {
    note->duration = 3;
    songPosition += 24;
    barProgression += 24;
}

if(currentPosition < length)  {
    part[(int)currentPosition] = selectChord();
    part[(int)currentPosition]->bassNote = note;
    part[(int)currentPosition]->octaveBassNote = octave;
    part[(int)currentPosition]->pitchBassNote = (note->pitch - (int) note->pitch) * 100;
}

//Keep track of where through the array of values you are.
i++;
/*House keeping to keep track of where in the bar the program is*/
barCount++;
if (barCount == 4)  {
    direction = direction * -1;
```

```
            barCount = 0;
        }
    }
}

/*
    This function creates a simple rolling bass line based on a psedo random number.
*/
void rollingBassLine(INSTRUMENT *part, CHORD *bass[], int length, int startPosition)
{
        int direction = 1; /*This variable holds the direction the bass line should be walking*/
        int barCount = 0; /*This variable is used to hold the position within the current bar*/
        int barProgression = 0; /*This is used to store the progression of the piece through the 12 bar structure*/
        int previousNote = 0;
        int i; /*Loop control variable*/
        double x;

        //Define the starting octave to be in the middle of the instruments range.
        octave = (int) ((part->maxRange + part->minRange) / 2);
        if(octave == 0)  {
           octave = 7;
        }

        /*This is the main loop for processing the pitch values*/
        currentScale = cScale;
        for (i=0; i<length; i++)  {
        if((i % 8) == 0)  {
            part->noteArray[(i + startPosition)] = newNote();
            part->noteArray[(i + startPosition)]->start = (double) ((startPosition + i) / 8.0);
            part->noteArray[(i + startPosition)]->duration = 1.1;
            part->noteArray[(i + startPosition)]->scale = currentScale;
            part->noteArray[(i + startPosition)]->amplitude = 6000;
            /*This controls the changes of scale throughout the 12-bar progression*/
            if(barProgression == 128) currentScale = fScale;
            else if (barProgression == 192) currentScale = cScale;
            else if (barProgression == 224) currentScale = gScale;
            else if (barProgression == 288) currentScale = fScale;
            else if (barProgression == 320) currentScale = cScale;
            /*Once the end of 12 bars is reached, reset the counter*/
            if (barProgression == 384) barProgression = 0;
            else barProgression += 8;
            /*This is what maps the pitch values to notes*/
            /*The first two clauses put in the scales root notes if it is the first beat of the bar*/

            //Change direction in a random fashion.
```

```
x = ((double) (rand() % 50000))/50000;
if(x < 0.2) {
    direction = direction * -1;
}
if ((barCount == 0) && (direction == 1))  {
    part->noteArray[(i + startPosition)]->pitch = octave + currentScale[0];
    previousNote = 0;
}
else if ((barCount == 0) && (direction == -1))  {
    part->noteArray[(i + startPosition)]->pitch = octave + currentScale[0] + 1;
    previousNote = 0;
}
/*The rest of the clauses map the pitch value to the correct note*/
else {
    if(direction > 0) {
        if(previousNote == 5)  {
            octave++;
            previousNote = 0;
            part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
        } else {
            previousNote++;
            part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
        }
    } else if(direction < 0) {
        if(previousNote == 0)  {
            octave--;
            previousNote = 5;
            part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
        } else {
            previousNote--;
            part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
        }
    } else {
        printf("Error");
    }
    //Check that the note is within the instruments range.
    if(part->maxRange < part->noteArray[(i + startPosition)]->pitch)  {
        direction = direction * -1;
        if(direction > 0) {
            if(previousNote == 5)  {
                octave++;
                previousNote = 0;
                part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
            } else {
                previousNote++;
```

```
                part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
            }
        } else if(direction < 0) {
            if(previousNote == 0)  {
                octave--;
                previousNote = 5;
                part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
            } else {
                previousNote--;
                part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
            }
        }
    }  else if(part->minRange > part->noteArray[(i + startPosition)]->pitch)  {
        direction = direction * -1;
        if(direction > 0) {
            if(previousNote == 5)  {
                octave++;
                previousNote = 0;
                part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
            } else {
                previousNote++;
                part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
            }
        } else if(direction < 0) {
            if(previousNote == 0)  {
                octave--;
                previousNote = 5;
                part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
            } else {
                previousNote--;
                part->noteArray[(i + startPosition)]->pitch = octave + currentScale[previousNote];
            }
        }
    }
}

//Validate the note against the chord within the bass array.
if(bass != NULL)  {
    if(bass[i] != NULL)  {
        double rollingBassPitch = part->noteArray[(i + startPosition)]->pitch - (int)part->noteArray[(i
            + startPosition)]->pitch;
        double chordBassPitch = bass[i]->bassNote->pitch;
        double chordBassPitch1 = chordBassPitch + currentScale[bass[i]->notes[1]];
        double chordBassPitch2 = chordBassPitch + currentScale[bass[i]->notes[2]];
        double chordBassPitch3 = chordBassPitch + currentScale[bass[i]->notes[3]];
```

```
                    if(rollingBassPitch == chordBassPitch || rollingBassPitch == chordBassPitch1 || rollingBassPitch
                        == chordBassPitch2 || rollingBassPitch == chordBassPitch3)  {
                        //Then the bass line doesn't need modifying.
                    } else  {
                        if(rollingBassPitch < chordBassPitch && rollingBassPitch > chordBassPitch3)  {
                            part->noteArray[(i + startPosition)]->pitch = (int) part->noteArray[(i +
                                startPosition)]->pitch + currentScale[bass[i]->notes[3]];
                        } else if(rollingBassPitch < chordBassPitch3 && rollingBassPitch > chordBassPitch2)  {
                            part->noteArray[(i + startPosition)]->pitch = (int) part->noteArray[(i +
                                startPosition)]->pitch + currentScale[bass[i]->notes[2]];
                        } else if(rollingBassPitch < chordBassPitch2 && rollingBassPitch > chordBassPitch1)  {
                            part->noteArray[(i + startPosition)]->pitch = (int) part->noteArray[(i +
                                startPosition)]->pitch + currentScale[bass[i]->notes[1]];
                        }
                    }
                }
            }

            /*House keeping to keep track of where in the bar the program is*/
            barCount++;
            if (barCount == 16)  {
                barCount = 0;
            }
        }
    }
}

/*This is the function that maps the data from the standard map to notes*/
void mapStandard(CHORD *part[], double K, int length)
{
        NOTE *note; /*Variable to hold a note*/
        double min = 0, /*Minimum value possible for Standard Map*/
        max = 6.283185; /*Maximum value possible for Standard Map*/
        double range; /*Range to be used for duration selection*/
        double prange; /*Range to be used for pitch duration*/
        int currentNote = 0; /*Variable to hold current note*/
        int barCount = 0; /*Variable to hold the position within a bar*/
        double songPosition = 0; /*Variable to hold the position within the song*/
        double currentPosition; /*Second variable to do the above*/
        int barProgression = 0; /*Variable to hold where the song is in the 12-bar structure*/
        int progressionOffset = 0; /*Variable used to hold how many times the song has been through the 12-bars*/
        double original = S_K; /*Variable to hold the systems starting values*/
        /*Calculate the range spaces*/
        range = (max - min) / 7;
```

```
prange = (max - min) / 9;
/*Set the current scale*/
currentScale = cScale;
songPosition = 0;
octave = 8.0;
/*Carry on for length of song*/
while (songPosition < length) {
        /*If 12-bars have been completed amend bar position variables*/
        if (barProgression >= (384*1)) {
                barProgression = 0;
                progressionOffset += 384;
        }
        else {
    barProgression = (songPosition - progressionOffset);
}

        /*This code will add the programmer controlled structure to the music*/
        if (((progressionOffset / 384) == 1) && (barProgression == 256)) {
                S_K = original;
                S_theta = original;
                S_iota = original;
        }
        if (((progressionOffset / 384) == 2) && (barProgression == 0)) {
                S_K = original + 0.001;
                S_theta = original + 0.001;
                S_iota = original + 0.001;
        } else if (((progressionOffset / 384) == 3) && (barProgression == 0)) {
                S_K = original;
                S_theta = original;
                S_iota = original;
        }
        /*Code for controlling the scale being played*/
        if (barProgression >= (320*1)) currentScale = cScale;
        else if (barProgression >= (288*1)) currentScale = fScale;
        else if (barProgression >= (256*1)) currentScale = gScale;
        else if (barProgression >= (192*1)) currentScale = cScale;
        else if (barProgression >= (128*1)) currentScale = fScale;
        /*Get the next orbit, do this more than once to travel through the orbits faster*/
        Standard(K); /*Standard(K); Standard(K); Standard(K);*/
        /*Create new note and set standard values*/
        note = newNote();
        note->amplitude = 7000;
        note->scale = currentScale;
/*This code will bring the song position onto a beat of the bar if it is within 2 32ths of a bar of a beat*/
if (((int)songPosition % 8) >= 6) {
```

```
    songPosition = songPosition + (8 -((int)songPosition % 8));
} else if ( ( ((int)songPosition % 8) <= 2) && ( ((int)songPosition % 8) >= 1) )  {
    songPosition = songPosition - (8 - ((int)songPosition % 8));
}
/* Set the note start */
note->start = (songPosition/8);
/*Calculate the pitch value as an offset from the last*/
if (/*S_iota*/ S_theta < (min + 1*prange))  {
    currentNote += -4;
    if (currentNote < 0)  {
        currentNote = 6 + currentNote;
        if (octave > 6) octave--;
    }
    note->pitch = octave + currentScale[currentNote];
} else if (/*S_iota*/ S_theta < (min + 2*prange))  {
    currentNote += -3;
    if (currentNote < 0)  {
        currentNote = 6 + currentNote;
        if (octave > 6) octave--;
    }
    note->pitch = octave + currentScale[currentNote];
} else if (/*S_iota*/ S_theta < (min + 3*prange))  {
    currentNote += -2;
    if (currentNote < 0)  {
        currentNote = 6 + currentNote;
        if (octave > 6) octave--;
    }
    note->pitch = octave + currentScale[currentNote];
} else if (/*S_iota*/ S_theta < (min + 4*prange))  {
    currentNote += -1;
    if (currentNote < 0)  {
        currentNote = 6 + currentNote;
        if (octave > 6) octave--;
    }
    note->pitch = octave + currentScale[currentNote];
} else if (/*S_iota*/ S_theta < (min + 5*prange))  {
    currentNote += 1;
    if (currentNote > 5)  {
        currentNote = -6 + currentNote;
        if (octave < 8) octave++;
    }
    note->pitch = octave + currentScale[currentNote];
} else if (/*S_iota*/ S_theta < (min + 6*prange))  {
    currentNote = currentNote;
    note->pitch = octave + currentScale[currentNote];
```

```
} else if (/*S_iota*/ S_theta < (min + 7*prange))  {
    currentNote += 2;
    if (currentNote > 5)  {
        currentNote = -6 + currentNote;
        if (octave < 8) octave++;
    }
    note->pitch = octave + currentScale[currentNote];
} else if (/*S_iota*/ S_theta < (min + 8*prange))  {
    currentNote += 3;
    if (currentNote > 5)  {
        currentNote = -6 + currentNote;
        if (octave < 8) octave++;
    }
    note->pitch = octave + currentScale[currentNote];
} else  {
    currentNote += 4;
    if (currentNote > 5)  {
        currentNote = -6 + currentNote;
        if (octave < 8) octave++;
    }
    note->pitch = octave + currentScale[currentNote];
}

/*Make sure the pitch value is legal*/
if (note->pitch < 1)  {
    if (note->pitch <= -0.01)  {
        note->pitch = note->pitch + 0.12;
    } else if (note->pitch >= 0.12)  {
        note->pitch = note->pitch - 0.12;
    }
    note->pitch = note->pitch + octave;
}
/*Calculate note duration, current position is used to hold the current position so songPosition can be
    updated*/
currentPosition = songPosition;
if (/*S_theta*/ S_iota < (min + range))  {
    note->duration = 0.25;
    songPosition += 2;
} else if (/*S_theta*/ S_iota < (min + 3.5*range))  {
    note->duration = 0.5;
    songPosition += 4;
} else if (/*S_theta*/ S_iota < (min + 6.33*range))  {
    note->duration = 1;
    songPosition += 8;
} else if (/*S_theta*/ S_iota < (min + 6.85*range))  {
```

```
            note->duration = 2;
            songPosition += 16;
        } else  {
            note->duration = 3;
            songPosition += 24;
        }

        if(currentPosition < length)  {
            part[(int)currentPosition] = selectChord();
            part[(int)currentPosition]->bassNote = note;
            part[(int)currentPosition]->octaveBassNote = octave;
            part[(int)currentPosition]->pitchBassNote = (note->pitch - (int) note->pitch) * 100;
        }
    }
}

/*This is the function that maps the data from the standard map to notes for a soloist part*/
void mapStandardSolo(INSTRUMENT *part, double K, int length, int startPosition)
{
        NOTE *note; /*Variable to hold a note*/
        double min = 0, /*Minimum value possible for Standard Map*/
        max = 6.283185; /*Maximum value possible for Standard Map*/
        double range; /*Range to be used for duration selection*/
        double prange; /*Range to be used for pitch selection*/
        int currentNote = 0; /*Variable to hold current note*/
        int barCount = 0; /*Variable to hold the position within a bar*/
        double songPosition = 0; /*Variable to hold the position within the song*/
        double currentPosition; /*Second variable to do the above*/
        int barProgression = 0; /*Variable to hold where the song is in the 12-bar structure*/
        int progressionOffset = 0; /*Variable used to hold how many times the song has been through the 12-bars*/
        double original = S_K; /*Variable to hold the systems starting values*/
        /*Calculate the range spaces*/
        range = (max - min) / 7;
        prange = (max - min) / 9;

        /*Set the current scale*/
        currentScale = cScale;
        songPosition = startPosition;

        //Match the starting scale from the previous section.
        NOTE *x = (NOTE *) getPreviousNote(startPosition);
    if(x != NULL)  {
        currentScale = x->scale;
    }
```

```
/*Carry on for length of song*/
while (songPosition < startPosition + length)  {

        /*Get the next orbit, do this more than once to travel through the orbits faster*/
        Standard(K);
        Standard(K);
        Standard(K);
        /*Create new note and set standard values*/
        note = newNote();
        note->amplitude = 8000;
        note->scale = currentScale;
        /*Set the note start*/
        note->start = (songPosition/8);
        /*Calculate the pitch value as an offset from the last*/
        if (/*S_iota*/ S_theta < (min + 1*prange))  {
                currentNote += -4;
                if (currentNote < 0)  {
                        currentNote = 6 + currentNote;
                        if (octave > 6) octave--;
                }
                if(currentNote < part->minRange)  {
        currentNote += 8;
        if (currentNote > 5)  {
            currentNote = -6 + currentNote;
            if (octave < 8) octave++;
        }
                }
                note->pitch = octave + currentScale[currentNote];
        } else if (/*S_iota*/ S_theta < (min + 2*prange))  {
                currentNote += -3;
                if (currentNote < 0)  {
                        currentNote = 6 + currentNote;
                        if (octave > 6) octave--;
                }
                if(currentNote < part->minRange)  {
        currentNote += 6;
        if (currentNote > 5)  {
            currentNote = -6 + currentNote;
            if (octave < 8) octave++;
        }
                }
                note->pitch = octave + currentScale[currentNote];
        } else if (/*S_iota*/ S_theta < (min + 3*prange))  {
    currentNote += -2;
    if (currentNote < 0)  {
```

```
    currentNote = 6 + currentNote;
    if (octave > 6) octave--;
}
if(currentNote < part->minRange)  {
    currentNote += 4;
    if (currentNote > 5)  {
        currentNote = -6 + currentNote;
        if (octave < 8) octave++;
    }
            }
note->pitch = octave + currentScale[currentNote];
    } else if (/*S_iota*/ S_theta < (min + 4*prange))  {
            currentNote += -1;
            if (currentNote < 0)  {
                    currentNote = 6 + currentNote;
                    if (octave > 6) octave--;
            }
            if(currentNote < part->minRange)  {
    currentNote += 2;
    if (currentNote > 5)  {
        currentNote = -6 + currentNote;
        if (octave < 8) octave++;
    }
            }
            note->pitch = octave + currentScale[currentNote];
    } else if (/*S_iota*/ S_theta < (min + 5*prange))  {
            currentNote += 1;
            if (currentNote > 5)  {
                    currentNote = -6 + currentNote;
                    if (octave < 8) octave++;
            }
            if(currentNote > part->maxRange)  {
    currentNote -= 2;
    if (currentNote < 0)  {
        currentNote = 6 + currentNote;
        if (octave > 6) octave--;
    }
            }
            note->pitch = octave + currentScale[currentNote];
    } else if (/*S_iota*/ S_theta < (min + 6*prange))  {
            currentNote = currentNote;
            note->pitch = octave + currentScale[currentNote];
    } else if (/*S_iota*/ S_theta < (min + 7*prange))  {
            currentNote += 2;
            if (currentNote > 5)  {
```

```
                currentNote = -6 + currentNote;
                if (octave < 8) octave++;
        }
        if(currentNote > part->maxRange)  {
currentNote -= 4;
if (currentNote < 0)  {
    currentNote = 6 + currentNote;
    if (octave > 6) octave--;
}
        }
        note->pitch = octave + currentScale[currentNote];
} else if (/*S_iota*/ S_theta < (min + 8*prange))  {
        currentNote += 3;
        if (currentNote > 5)  {
                currentNote = -6 + currentNote;
                if (octave < 8) octave++;
        }
        if(currentNote > part->maxRange)  {
currentNote -= 6;
if (currentNote < 0)  {
    currentNote = 6 + currentNote;
    if (octave > 6) octave--;
}
        }
        note->pitch = octave + currentScale[currentNote];
} else  {
        currentNote += 4;
        if (currentNote > 5)  {
                currentNote = -6 + currentNote;
                if (octave < 8) octave++;
        }
        if(currentNote > part->maxRange)  {
currentNote -= 8;
if (currentNote < 0)  {
    currentNote = 6 + currentNote;
    if (octave > 6) octave--;
}
        }
        note->pitch = octave + currentScale[currentNote];
}

/*Calculate note duration, current position is used to hold the current position so song Position
    can be updated*/
currentPosition = songPosition;
if (/*S_theta*/ S_iota < (min + range))  {
```

```
                        note->duration = 0.28;
                        songPosition += 2;
                } else if (/*S_theta*/ S_iota < (min + 3*range))  {
                        note->duration = 0.28;
                        songPosition += 4;
                } else if (/*S_theta*/ S_iota < (min + 3.5*range))  {
                        note->duration = 0.53;
                        songPosition += 4;
                } else if (/*S_theta*/ S_iota < (min + 6.33*range))  {
                        note->duration = 1.03;
                        songPosition += 8;
                } else if (/*S_theta*/ S_iota < (min + 6.85*range))  {
                        note->duration = 2.03;
                        songPosition += 16;
                }

                /*Code to add the notes into the part array*/
                if (part->noteArray[(int)currentPosition] == NULL) {
            part->noteArray[(int)currentPosition] = note;
        }
        }
}

/*This is the function that maps the data from the standard map to notes for a drum part*/
void mapStandardDrums(INSTRUMENT *part, double K, int length, int startPosition)
{
        NOTE *note; /*Variable to hold a note*/
        double min = 0, /*Minimum value possible for Standard Map*/
        max = 6.283185; /*Maximum value possible for Standard Map*/
        double range; /*Range to be used for duration selection*/
        double songPosition = 0; /*Variable to hold the position within the song*/
        double currentPosition; /*Second variable to do the above*/
        /*Calculate the range spaces*/
        range = (max - min) / 7;

        songPosition = startPosition;

        /*Carry on for length of song*/
        while (songPosition < (startPosition + length))  {
                /*Get the next orbit, do this more than once to travel through the orbits faster*/
                Standard(K);
                Standard(K);
                Standard(K);
                Standard(K);
                /*Create new note and set standard values*/
```

```
                note = newNote();
                note->amplitude = 4000;
                note->scale = currentScale;
                /*Set the note start*/
                note->start = (songPosition/8);
                /*Set the pitch to a standard value since it doesn't matter for the drum kit. */
                note->pitch = 7.0;

                /*Calculate note duration, current position is used to hold the current position so song Position
                    can be updated*/
                currentPosition = songPosition;
                if (/*S_theta*/ S_iota < (min + 0.5*range))  {
                        note->duration = 0.125;
                        songPosition += 1;
                } else if (/*S_theta*/ S_iota < (min + 4*range))  {
                        note->duration = 0.25;
                        songPosition += 2;
                } else if (/*S_theta*/ S_iota < (min + 6.85*range))  {
                        note->duration = 0.5;
                        songPosition += 4;
                }

                /*Code to add the notes into the part array*/
                if (part->noteArray[(int)currentPosition] == NULL) {
            part->noteArray[(int)currentPosition] = note;
        }
        }
}

void createBassline(CHORD *part[], int length)
{
    mapHenon(part, HENON_A, HENON_B, length);
}

void createMelody(CHORD *part[], int length)
{
        mapStandard(part, STANDARD_K, length);
}

void createSolo(int length, int startPosition)
{
        mapStandardSolo(soloist, STANDARD_K, length, startPosition);
}

void createCounterMelody(CHORD *part[], int length)
```

```
{
        mapStandard(part, STANDARD_K, length);
}

void createDrumKit(int length, int startPosition)
{
    mapStandardDrums(snare, STANDARD_I, length, startPosition);
    mapStandardDrums(toms, STANDARD_I, length, startPosition);
    rollingBassLine(bassDrum, NULL, length, startPosition);
}

/*
 * This function creates the CHORD type variables which are used to aid in the selection of chords throughout the
     music.
 */
void createChords()
{
    major7th = newChord();
    major7th->notes[0] = 0;
    major7th->notes[1] = 4;
    major7th->notes[2] = 7;
    major7th->notes[3] = 0;
    major7th->probability = 0.7;
    minor7th = newChord();
    minor7th->notes[0] = 0;
    minor7th->notes[1] = 3;
    minor7th->notes[2] = 7;
    minor7th->notes[3] = 0;
    minor7th->probability = 0.3;
}

INSTRUMENT *selectSoloist()
{
    double i = ((double) (rand() % 10000))/10000;
    double totalChecked = 0;
    INSTRUMENT *savedBottomInstrument = bottomInstrument;
    INSTRUMENT *returnInstrument;
    //Pick the instrument to play the solo depending on the predetermined probabilities.
    while(bottomInstrument != NULL) {
        totalChecked = totalChecked + (double) bottomInstrument->soloPriority;
        if(i < totalChecked) {
            returnInstrument = bottomInstrument;
            bottomInstrument = savedBottomInstrument;
            return returnInstrument;
        }
```

```
                bottomInstrument = bottomInstrument->next;
            }
            returnInstrument = bottomInstrument;
        bottomInstrument = savedBottomInstrument;
        return returnInstrument;
}

INSTRUMENT *selectBassFreeNote(int Position)
{
    INSTRUMENT *savedBottomInstrument = bottomInstrument;
    INSTRUMENT *returnInstrument = NULL;
    //Pick the instrument to play the bass part depending on the solo probabilities.
    while(bottomInstrument != NULL) {
        if(bottomInstrument->type == 1 && bottomInstrument != melody && bottomInstrument != counterMelody) {
            if(bottomInstrument->noteArray[Position] == NULL) {
                NOTE *temp = getPreviousNoteOnInstrument(bottomInstrument, (int)Position);

                if(temp == NULL) {
                    returnInstrument = bottomInstrument;
                } else if(((temp->start + temp->duration) * 8) <= Position) {
                    returnInstrument = bottomInstrument;
                }
            }

            if(returnInstrument != NULL) {
                bottomInstrument = savedBottomInstrument;
                return returnInstrument;
            }
        }
        bottomInstrument = bottomInstrument->next;
    }
    bottomInstrument = savedBottomInstrument;
    return NULL;
}

INSTRUMENT *selectMelodyFreeNote(int Position)
{
    double i = ((double) (rand() % 10000))/10000;
    double totalChecked = 0;
    INSTRUMENT *savedBottomInstrument = bottomInstrument;
    INSTRUMENT *returnInstrument;
    //Pick the instrument to play the melody part depending on the solo probabilities.
    while(bottomInstrument != NULL) {
        totalChecked = totalChecked + (double) bottomInstrument->soloPriority;
        if(i < totalChecked) {
```

```
            if(bottomInstrument != melody || bottomInstrument != counterMelody)  {
                if(bottomInstrument->noteArray[Position] == NULL)  {
                    NOTE *temp = getPreviousNoteOnInstrument(bottomInstrument, (int)Position);
                    if(temp == NULL)  {
                        returnInstrument = bottomInstrument;
                        bottomInstrument = savedBottomInstrument;
                        return returnInstrument;
                    } else if((temp->start + temp->duration) <= (Position/8))  {
                        returnInstrument = bottomInstrument;
                        bottomInstrument = savedBottomInstrument;
                        return returnInstrument;
                    }
                }
            }
        }
        bottomInstrument = bottomInstrument->next;
    }
    bottomInstrument = savedBottomInstrument;
    return NULL;
}

void mapIndividualChordToNote(CHORD *array[], int ChordPositionNumber, double songPosition, int counter, int type)
{
    INSTRUMENT *temp = NULL;

    /* printf("Mapping chord to note\nChord position number %i\nSong position %1.2f\n", ChordPositionNumber,
        songPosition);*/

    if(type == 0)  {
        temp = selectBassFreeNote(songPosition);
        /*printf("Selected bass free note, %p\n", temp);*/
    } else {
        temp = selectMelodyFreeNote(songPosition);
        /*printf("Selected melody free note, %p\n", temp);*/
    }
    if(temp != NULL)  {
        /*printf("Creating new note\n");*/
        temp->noteArray[(int)songPosition] = newNote();
        temp->noteArray[(int)songPosition]->start = songPosition/8;
        temp->noteArray[(int)songPosition]->duration = array[counter]->bassNote->duration;
        currentScale = array[counter]->bassNote->scale;
        int currentNote = array[counter]->pitchBassNote + array[counter]->notes[ChordPositionNumber];
        int octave = array[counter]->octaveBassNote;
        while(currentNote > 12)  {
            currentNote -= 12;
```

```
            octave++;
        }
        temp->noteArray[(int)songPosition]->scale = array[counter]->bassNote->scale;
        temp->noteArray[(int)songPosition]->amplitude = array[counter]->bassNote->amplitude - 1000;
        temp->noteArray[(int)songPosition]->pitch = octave + (currentNote / 100);
    }
}

void mapChordToNote(int sectionNumber, CHORD *chordArray[], int type)
{
    double startPosition = (sectionNumber*(SECTION_LENGTH + FILLSOLO_LENGTH)) + FILLSOLO_LENGTH;
    double songPosition = startPosition;
    int counter = 0;

    //Now map the chords to notes.
    while(songPosition < startPosition + SECTION_LENGTH)  {
        if(chordArray[counter] != NULL)  {
            INSTRUMENT * temp;
            if(type == 0)  {
                temp = selectBassFreeNote(songPosition);
            } else {
                temp = selectMelodyFreeNote(songPosition);
            }

            if(temp != NULL)  {
                temp->noteArray[(int)songPosition] = chordArray[counter]->bassNote;
                temp->noteArray[(int)songPosition]->start = songPosition/8;
            }

            //Map the chord structure to notes.
            mapIndividualChordToNote(chordArray, 1, songPosition, counter, type);
            mapIndividualChordToNote(chordArray, 2, songPosition, counter, type);
            mapIndividualChordToNote(chordArray, 3, songPosition, counter, type);
        }
        counter++;
        songPosition++;
    }
}

// Print out a list of all the chords details.
void printChord(int sectionNumber, CHORD *chord[])
{
    double startPosition = (sectionNumber*(SECTION_LENGTH + FILLSOLO_LENGTH)) + FILLSOLO_LENGTH;
    double songPosition = startPosition;
    int x = 0;
```

```
    //Now map the chords to notes.
    while(songPosition < startPosition + SECTION_LENGTH)  {
        if(chord[x] != NULL)  {

            printf("Chord - Start %1.2f, Duration %1.2f, Pitch %1.2f, Octave %1.2f, Pitch %i \n",
                chord[x]->bassNote->start, chord[x]->bassNote->duration, chord[x]->bassNote->pitch,
                chord[x]->octaveBassNote, chord[x]->pitchBassNote);

        }
        x++;
        songPosition++;
    }

}


void createSection(int sectionNumber)
{
    // Firstly create the set of chords for the bass and melody and counter melody.
    // Then map the chords to actual notes on instruments.

    int count = 0;
    while(count< SECTION_LENGTH)  {
        bassChordArray[count] = NULL;
        count++;
    }
    count = 0;
    while(count< SECTION_LENGTH)  {
        melodyChordArray[count] = NULL;
        count++;
    }
    count = 0;
    while(count< SECTION_LENGTH)  {
        counterChordArray[count] = NULL;
        count++;
    }

    //Select the soloist for the section.
    soloist = selectSoloist();
    melody = selectSoloist();
    counterMelody = selectSoloist();
    while(melody == counterMelody)  {
        counterMelody = selectSoloist();
    }
```

```
if(sectionNumber == 0) {
    /*Iterate the Henon Map to pass the transient phase*/
    HenonXTimes();
    createDrumKit(FILLSOLO_LENGTH, (sectionNumber*(SECTION_LENGTH + FILLSOLO_LENGTH)));
    rollingBassLine(bass, bassChordArray, FILLSOLO_LENGTH, (sectionNumber*(SECTION_LENGTH + FILLSOLO_LENGTH)));
} else {
    createSolo(FILLSOLO_LENGTH, (sectionNumber*(SECTION_LENGTH + FILLSOLO_LENGTH)));
}


/*Iterate the Henon Map to pass the transient phase*/
HenonXTimes();
createDrumKit(SECTION_LENGTH, ((sectionNumber*(SECTION_LENGTH + FILLSOLO_LENGTH)) + FILLSOLO_LENGTH));
createBassline(bassChordArray, SECTION_LENGTH);
/* Generate Melody Chords */
createMelody(melodyChordArray, SECTION_LENGTH);

double i = ((double) (rand() % 10000))/10000;
if(i < 0.5) {
    createCounterMelody(counterChordArray, SECTION_LENGTH);
}

//printChord(sectionNumber, bassChordArray);
//printf("*****************************************\n");
//printChord(sectionNumber, melodyChordArray);
//printf("*****************************************\n");
//printChord(sectionNumber, counterChordArray);
//printf("*****************************************\n");

//Harmonise the chords.
int counter = 0;
while(counter < SECTION_LENGTH)  {
    if(melodyChordArray[counter] != NULL)  {
        //Select the overlapping bass chord.
        int location = counter;
        while(location >= 0 && bassChordArray[location] != NULL && bassChordArray[location]->bassNote != NULL &&
              (bassChordArray[location]->bassNote->start + bassChordArray[location]->bassNote->duration) >=
             melodyChordArray[counter]->bassNote->start)  {
            location--;
        }
        if(bassChordArray[location] != NULL)  {
            melodyChordArray[counter] = harmoniseChords(bassChordArray[location], melodyChordArray[counter]);
        }
    }
```

```
    if(counterChordArray[counter] != NULL)  {
        //Select the overlapping bass chord.
        int locationCounter = counter;
        while(locationCounter >= 0 && bassChordArray[locationCounter] != NULL &&
            bassChordArray[locationCounter]->bassNote != NULL &&
            (bassChordArray[locationCounter]->bassNote->start +
            bassChordArray[locationCounter]->bassNote->duration) >=
            counterChordArray[counter]->bassNote->start)  {
            locationCounter--;
        }
        if(bassChordArray[locationCounter] != NULL)  {
            counterChordArray[counter] = harmoniseChords(bassChordArray[locationCounter],
                counterChordArray[counter]);
        }
    }
    counter++;
}
counter = 0;

//printChord(sectionNumber, bassChordArray);
//printf("*********************************************\n");
//printChord(sectionNumber, melodyChordArray);
//printf("*********************************************\n");
//printChord(sectionNumber, counterChordArray);
//printf("*********************************************\n");

//Create a rolling bass line which fits with the chords already generated.
rollingBassLine(bass, bassChordArray, SECTION_LENGTH, ((sectionNumber*(SECTION_LENGTH + FILLSOLO_LENGTH)) +
    FILLSOLO_LENGTH));

//Map the bass notes
mapChordToNote(sectionNumber, bassChordArray, 0);

//Map the melody notes.
mapChordToNote(sectionNumber, melodyChordArray, 1);

//Now map the counter melody notes.
mapChordToNote(sectionNumber, counterChordArray, 1);


counterMelody = NULL;
count = 0;
while(count< SECTION_LENGTH)  {
    bassChordArray[count] = NULL;
    count++;
```

```
    }
    count = 0;
    while(count< SECTION_LENGTH)  {
        melodyChordArray[count] = NULL;
        count++;
    }
    count = 0;
    while(count< SECTION_LENGTH)  {
        counterChordArray[count] = NULL;
        count++;
    }

}

int main()
{
    int i = 0;
    srand((unsigned int) time(NULL));
    //Create the chords to be used within the piece.
    createChords();
    //instrumentNumber, soloPriority, maxRange, minRange, isBassInstrument
        bass = newInstrument("BASS", 18, 0.0, 10000.0, -10000.0, 0, 1);
        newInstrument("TRUMPET", 11, 0.35, 8.0, 5.06, 0, 0);
    newInstrument("FIRST_TROMBONE", 12, 0.2, 6.08, 4.04, 0, 1);
    newInstrument("SECOND_TROMBONE", 12, 0.02, 6.08, 4.04, 1, 1);
    newInstrument("BASS_TROMBONE", 12, 0.01, 6.07, 4.08, 1, 1);  //Needs changing to use its own instrument due to
        the deeper pitch.
    newInstrument("ALTO_SAXAPHONE", 14, 0.3, 7.08, 5.01, 0, 0);
    newInstrument("TENOR_SAXAPHONE", 14, 0.3, 7.06, 5.10, 1, 0);
    newInstrument("BARITONE_SAXAPHONE", 14, 0.3, 7.06, 5.10, 1, 1);  //Needs changing to use its own instrument due
        to the deeper pitch.
    newInstrument("PIANO", 15, 0.119, 10000.0, -10000.0, 1, 0);
        snare = newInstrument("SNARE", 3, 0, 10000.0, -10000.0, 0, 1);
        toms = newInstrument("TOMS", 1, 0, 10000.0, -10000.0, 0, 1);
        bassDrum = newInstrument("BASS_DRUM", 2, 0, 10000.0, -10000.0, 0, 1);
        //Create each section of the music.
        for(i = 0; i < NUMBER_OF_SECTIONS; i++) {
        createSection(i);
    }
        printSong();

        printLilyPond("Attempt");

        /*
    //Test data to check the maximum and minimum values produced by the henon map.
```

```
    double notes[1000000];
    double min;
    double max;
    int i = 0;

        HenonXTimes();
        for (i=0; i < 1000000; i++) {
                Henon(HENON_A, HENON_B);
                notes[i] = Henon_x;
        }
        min = notes[0];
        max = notes[0];
        for (i=0; i < 1000000; i++) {
                if(notes[i] < min) min = notes[i];
                if(notes[i] > max) max = notes[i];
        }
        printf("Minimum: %lf\n", min);
        printf("Maximum: %lf\n", max);
        */
}
```

A.2 File: AmplitudeValidator.c

```
/***********************************
*                                  *
*          AmplitudeValidator.c    *
*                                  *
***********************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <regex.h>
#define FALSE 0
#define TRUE 1
#define DELIMITERS      " \r\n"
#define MAXIMUM_AMPLITUDE 30000
#define REALMIN 5000
#define REALMAX 30000

FILE *readFile;
FILE *writeFile;

/*The structure for holding the note*/
typedef struct NOTE
{
        char *string;
        double amplitude;
        double startTime;
        double duration;
        struct NOTE *next;
} NOTE;

NOTE *startNote;

/*This is the function to create a new note*/
NOTE *newNote()
{
        NOTE *note = (NOTE *)(malloc(sizeof(NOTE)));
        note->string = malloc(sizeof(char[256]));
        note->startTime = 0;
        note->duration = 0;
        note->amplitude = 0;
        note->next = NULL;
        return note;
```

```
}

/* Read one line from standard input, */
/* copying it to line array (but no more than max chars). */
/* Does not place terminating \n in line array. */
/* Returns line length, or 0 for empty line, or EOF for end-of-file. */

int getline(char line[], int max)
{
int nch = 0;
int c;
max = max - 1;                    /* leave room for '\0' */

while((c = getc(readFile)) != EOF)
        {
        if(c == '\n')
                break;

        if(nch < max)
                {
                line[nch] = c;
                nch = nch + 1;
                }
        }

if(c == EOF && nch == 0)
        return EOF;

line[nch] = '\0';
return nch;
}

/*This is the function to read the notes from the Csound format notes file.*/
void readSong()
{
    char line[256];

    while(getline(line, 256) != EOF)  {
        if(line[0] == ';' || line[0] == 'e')  {
            //Do nothing
        } else  {
            NOTE *new_Note = newNote();
            strcpy(new_Note->string, line);

            //Parse out the start time, duration and amplitude.
```

```
            char* token = strtok(line, DELIMITERS);
            int i = 0;
            while(token != NULL) {
                //Check if the token matches the type for any of the necessary values.
                double x = atof(token);
                if(token[0] == 'i') {
                    //Do nothing
                } else if(i == 1) {
                    new_Note->startTime = atof(token);
                } else if(i == 2) {
                    new_Note->duration = atof(token);
                } else if(x > 100) {
                    new_Note->amplitude = x;
                }
                token = strtok(NULL, DELIMITERS);
                i++;
            }
            new_Note->next = startNote;
            startNote = new_Note;
        }
    }
}

/*
 * Function used by the listsort method.
*/
double cmp(NOTE *a, NOTE *b) {
    return a->startTime - b->startTime;
}

/*
 * This is the actual sort function. Notice that it returns the new
 * head of the list. (It has to, because the head will not
 * generally be the same NOTE after the sort.) So unlike sorting
 * an array, where you can do
 *
 *      sort(myarray);
 *
 * you now have to do
 *
 *      list = listsort(mylist);
 */
NOTE *listsort(NOTE *list) {
    NOTE *p, *q, *e, *tail, *oldhead;
    int insize, nmerges, psize, qsize, i;
```

```
/*
 * Silly special case: if 'list' was passed in as NULL, return
 * NULL immediately.
 */
if (!list) {
        return NULL;
}

insize = 1;

while (1) {
    p = list;
    oldhead = list;                 /* only used for circular linkage */
    list = NULL;
    tail = NULL;

    nmerges = 0;  /* count number of merges we do in this pass */

    while (p) {
        nmerges++;  /* there exists a merge to be done */
        /* step 'insize' places along from p */
        q = p;
        psize = 0;
        for (i = 0; i < insize; i++) {
            psize++;
            q = q->next;
            if (!q) break;
        }

        /* if q hasn't fallen off end, we have two lists to merge */
        qsize = insize;

        /* now we have two lists; merge them */
        while (psize > 0 || (qsize > 0 && q)) {

            /* decide whether next NOTE of merge comes from p or q */
            if (psize == 0) {
                /* p is empty; e must come from q. */
                e = q;
                q = q->next;
                qsize--;
            } else if (qsize == 0 || !q) {
                /* q is empty; e must come from p. */
                e = p;
```

```
                    p = p->next;
                    psize--;
                } else if (cmp(p,q) <= 0) {
                    /* First NOTE of p is lower (or same);
                     * e must come from p. */
                    e = p;
                    p = p->next;
                    psize--;
                } else {
                    /* First NOTE of q is lower; e must come from q. */
                    e = q;
                    q = q->next;
                    qsize--;
                }

                /* add the next NOTE to the merged list */
                if (tail) {
                    tail->next = e;
                } else {
                    list = e;
                }
                tail = e;
            }

            /* now p has stepped 'insize' places along, and q has too */
            p = q;
        }
            tail->next = NULL;

        /* If we have done only one merge, we're finished. */
        if (nmerges <= 1)   /* allow for nmerges==0, the empty list case */
            return list;

        /* Otherwise repeat, merging lists twice the size */
        insize *= 2;
    }
}

/*
    Implement a very simple algorithm to ensure that all combined amplitudes are within the cSound limits.
*/
void simpleValidationMethod()
{
    double maximumAmplitude;
    double minimumAmplitude;
```

```
NOTE *temp = startNote;

//Loop through all the notes
while(temp != NULL) {
    double max = 0.0;
    //At each timestep calculate the maximum possible amplitude being heard.
    NOTE *temp2 = temp;
    while((temp2 != NULL) && (temp2->startTime < (temp->startTime + temp->duration))) {
        max = max + temp2->amplitude;
        temp2 = temp2->next;
    }

    if(max > MAXIMUM_AMPLITUDE) {
        //scale by
        double scaleFactor = MAXIMUM_AMPLITUDE / max;
        NOTE *temp2 = temp;
        while(temp2 != NULL && temp2->startTime < (temp->startTime + temp->duration)) {
            double moddedAmplitude = temp2->amplitude * scaleFactor;
            char *AmplitudeString = malloc(sizeof(char[10]));
            sprintf(AmplitudeString, "%f", moddedAmplitude);

            char * output = malloc(sizeof(char[256]));

            //Tokenise the string and replace the amplitude token
            char* token = strtok(temp2->string, DELIMITERS);
            int i = 0;
            while(token != NULL) {
                //Check if the token matches the type for any of the necessary values.
                double x = atof(token);
                if(token[0] == 'i') {
                    //Do nothing
                } else if(i == 1) {
                    //Do nothing
                } else if(i == 2) {
                    //Do nothing
                } else if(x > 100) {
                    token = AmplitudeString;
                }
                output = strcat(output, token);
                output = strcat(output, " ");

                token = strtok(NULL, DELIMITERS);
                i++;
            }
```

```
                temp2->string = output;
                temp2->amplitude = moddedAmplitude;
                temp2 = temp2->next;
            }
        }
        temp = temp->next;
    }

}

/*
    Implement a complex validation algorithm to ensure that all combined amplitudes are within the cSound limits.
*/
void complexValidationMethod()
{
    double maximumAmplitude;
    double minimumAmplitude;
    int amplitudesArrayLength;

    NOTE *temp = startNote;
    int noOfNotes = 0;
    while (temp != NULL)  {
        noOfNotes++;
        temp = temp->next;
    }

    double amplitudes[noOfNotes];
    temp = startNote;
    int i = 0;
    //Fill the amplitudes array
    while(temp != NULL)  {
        double max = 0.0;
        //At each timestep calculate the maximum possible amplitude being heard.
        NOTE *temp2 = temp;
        while((temp2 != NULL) && (temp2->startTime < (temp->startTime + temp->duration)))  {
            max = max + temp2->amplitude;
            temp2 = temp2->next;
        }
        amplitudes[i] = max;
        i++;
        temp = temp->next;
    }
    amplitudesArrayLength = i;

    i = 0;
```

```
minimumAmplitude = amplitudes[i];
maximumAmplitude = amplitudes[i];
while(i < amplitudesArrayLength)  {
    if(amplitudes[i] < minimumAmplitude)  {
        minimumAmplitude = amplitudes[i];
    }
    if(amplitudes[i] > maximumAmplitude)  {
        maximumAmplitude = amplitudes[i];
    }
    i++;
}

/*i = 0;
while(i < amplitudesArrayLength)  {
    printf("Amplitude: %1.2f\n", amplitudes[i]);
    i++;
}
*/

temp = startNote;
double scaleFactor = (REALMAX / maximumAmplitude);
//Loop through all the notes
while(temp != NULL)  {
    //scale by
    double moddedAmplitude = (temp->amplitude + REALMIN - minimumAmplitude) * scaleFactor;
    char *AmplitudeString = malloc(sizeof(char[10]));
    sprintf(AmplitudeString, "%f", moddedAmplitude);

    char * output = malloc(sizeof(char[256]));

    //Tokenise the string and replace the amplitude token
    char* token = strtok(temp->string, DELIMITERS);
    int i = 0;
    while(token != NULL)  {
        //Check if the token matches the type for any of the necessary values.
        double x = atof(token);
        if(token[0] == 'i')  {
            //Do nothing
        } else if(i == 1)  {
            //Do nothing
        } else if(i == 2)  {
            //Do nothing
        } else if(x > 100)  {
            token = AmplitudeString;
        }
```

```
            output = strcat(output, token);
            output = strcat(output, " ");

            token = strtok(NULL, DELIMITERS);
            i++;
        }

        temp->string = output;
        temp->amplitude = moddedAmplitude;
        temp = temp->next;
    }

}

void validateSong()
{
    complexValidationMethod();
}

/*This is the function to print out the piece to Csound format*/
void printSong()
{
        NOTE *note;
        note = startNote;
        /*This loop goes through the parts and prints out all the now that they have had their amplitude corrected*/
    while (note != NULL)  {
        fprintf(writeFile, "%s\n", note->string);
        note = note->next;
    }
        fprintf(writeFile, "e\n");
}

void printNotes()
{
    NOTE *note;
    NOTE *oldNote;
        note = startNote;
        oldNote = startNote;
    while(note != NULL)  {
        printf("Checking token note = %s\n", note->string);
        printf("Checking token amplitude = %1.2f\n", note->amplitude);
        printf("Checking token startTime = %1.2f\n", note->startTime);
        printf("Checking token duration = %1.2f\n", note->duration);
        note = note->next;
    }
```

```
    startNote = oldNote;
}


/*
 * This program validates the amplitudes of the given input file according to
 * the rules in the validateSong function.
 */
int main(int argc, char *argv[])
{
    srand((unsigned int) time(NULL));

    if(argv[1] == NULL)  {
                printf("Error, please specify a file path to read in as argument one.");
                abort();
        }
    if(argv[2] == NULL)  {
                printf("Error, please specify a file path to write the output to as argument two!");
                abort();
        }

        //Read in the music from a text file.
        readFile = fopen(argv[1], "r");
    readSong();
    fclose(readFile);
    printf("Finished reading\n");

    //Sort the notes into start time order.
    listsort(NULL);
    startNote = listsort(startNote);
    printf("Finished sorting\n");

    //printNotes();

    //Validate the amplitudes
    validateSong();
    printf("Finished validating\n");

    //Write the corrected song out.
    writeFile = fopen(argv[2], "w");
        printSong();
        fclose(writeFile);

        printf("Finished writing\n");
}
```

A.3 File: Instruments.csd

```
sr=44100
kr=4410
ksmps=10
nchnls=2
zakinit   30, 30
gifqcadj  =          6600

;-------------------------------------------------------------------------------
; STANDARD DRUM
;-------------------------------------------------------------------------------
instr 1
        kampenv expseg .0001, .01, p4, .04, .01
        asig rand kampenv
        afilt reson asig, 1000, 100
        aout  balance afilt, asig
        outs aout, aout
endin


;-------------------------------------------------------------------------------
; BASS DRUM
;-------------------------------------------------------------------------------
instr 2
        kfreqenv expseg  50,    .01, 200, .08, 50
        kampenv  expseg  .0001, .01, p4,  .08, .01
        asig     rand kampenv
        afilt    reson asig, kfreqenv, kfreqenv/8
        aout     balance afilt, asig
        outs aout, aout
endin


;-------------------------------------------------------------------------------
; SNARE
;-------------------------------------------------------------------------------
instr 3
        kampenv4  linseg  0, .001, 1,  p3-.021, 1, .02, 0
        kptchenv linseg  100, .01, 300, .2, 200, .01, 200
        asig     pluck   p4, kptchenv, 50, 2, 4, .8, 3
        aout     =       kampenv4*asig
        outs aout, aout
endin
```

```
;-------------------------------------------------------------------------------
; BRASS - TRUMPET
;-------------------------------------------------------------------------------
instr 11
idur    = p3
iamp    = p4
ifenv   = 231
ifdyn   = 231
ifq1    = cpspch(p5)
if1     = 201
ifq2    = cpspch(p5)
if2     = 201
imax    = 5

   aenv   oscili    iamp, 1/idur, ifenv

   adyn   oscili    ifq2*imax, 1/idur, ifdyn
   amod   oscili    adyn, ifq2, if2

   a1     oscili    aenv, ifq1+amod, if1
          outs       a1, a1
endin


;-------------------------------------------------------------------------------
; BRASS - TROMBONE
;-------------------------------------------------------------------------------
instr 12
idur    = p3
iamp    = p4
ifenv   = 231
ifdyn   = 231
ifq1    = cpspch(p5)
if1     = 201
ifq2    = cpspch(p5)
if2     = 201
imax    = 5

   aenv   oscili    iamp, 1/idur, ifenv

   adyn   oscili    ifq2*imax, 1/idur, ifdyn
   amod   oscili    adyn, ifq2, if2
```

```
   a1     oscili    aenv, ifq1+amod, if1
          outs       a1, a1
endin


;-------------------------------------------------------------------------------
; CLARINET
;-------------------------------------------------------------------------------
instr 14
  idur   = p3
  iamp   = p4
  ifenv  = 51                    ; clarinet settings:
  ifdyn  = 52                    ; amp and index envelope see flow chart
  ifq1   = cpspch(p5)*3          ; N1:N2 is 3:2, imax=5
  if1    = 50                     ; duration ca. .5 sec
  ifq2   = cpspch(p5)*2
  if2    = 50
  imax   = p6
  imin   = 2

     aenv  oscili   iamp, 1/idur, ifenv                ; envelope

     adyn  oscili   ifq2*(imax-imin), 1/idur, ifdyn    ; index
     adyn  =        (ifq2*imin)+adyn                   ; add minimum value
     amod  oscili   adyn, ifq2, if2                    ; modulator

     a1    oscili   aenv, ifq1+amod, if1               ; carrier
           outs      a1, a1
endin


;-------------------------------------------------------------------------------
; DOUBLE BASS
;-------------------------------------------------------------------------------
instr   18
;               ihold                              ;turn on note indefinitely
        idur    =       abs(p3)
        ibase   =       cpspch(p4)              ;p4 is keyboard pitch
        iroct   =       octpch(p4)
        irbase  =       octpch(4.09)            ;base of rate scl table
        irrange =       octpch(13.06)-irbase
        iveloc  =       p5                      ;0 <= p5 <= 127
        iop1fn  =       112                     ;param tables for ops
        iop2fn  =       113
        iop3fn  =       114
```

```
        iop4fn  =       115
        iop5fn  =       116
        iop6fn  =       117
        iampfn  =       102                     ;amp/level map function
        ipkamp  =      4000                    ;scale for converter
        irsfn   =       103                     ;rate scaling function
        idevfn  =       108                     ;level/pkdev map func
        irisefn =       104                     ;eg rise rate fn
        idecfn  =       106                     ;eg decay rate fn
        ivsfn   =       110                     ;vel sensitivity fn
        ivelfn  =       109                     ;vel/amp fac map fn
        iveloc  table   iveloc,ivelfn           ;map this note's veloc
        ifeedfn =       111
        ifeed   table   7,ifeedfn               ;0 <= p25 <= 7 (feedbk)
        ifeed   =       ifeed/(2 * 3.14159)     ;dev in radians
        idetfac =       104                     ;max detuning divisor
        imap128 =       127/99                  ;mapping constant 99->127
        irscl   table   (iroct-irbase)/irrange*127,irsfn
        irscl   =       irscl*6
        iop     =       1                       ;start loop with op1
        iopfn   =       iop1fn

loop:
;-------------------------------read operator parameters
        ilvl    table   0,iopfn
        ivel    table   1,iopfn
        iegr1   table   2,iopfn
        iegr2   table   3,iopfn
        iegr3   table   4,iopfn
        iegr4   table   5,iopfn
        iegl1   table   6,iopfn
        iegl2   table   7,iopfn
        iegl3   table   8,iopfn
        iegl4   table   9,iopfn
        iams    table   10,iopfn
        imode   table   11,iopfn
        ifreq   table   12,iopfn
        idet    table   13,iopfn
        irss    table   14,iopfn
;-------------------------------initialize operator
        ihz     =       (imode>0?ifreq:ibase*ifreq)+idet/idetfac
iamp    =       ilvl/99         ;rescale to 0->1
        ivfac   table   ivel,ivsfn

        iegl1   =       iamp*iegl1
```

```
        iegl2   =       iamp*iegl2
        iegl3   =       iamp*iegl3
        iegl4   =       iamp*iegl4

        iegl1   =       iegl1*(1-ivfac)+iegl1*ivfac*iveloc
        iegl2   =       iegl2*(1-ivfac)+iegl2*ivfac*iveloc
        iegl3   =       iegl3*(1-ivfac)+iegl3*ivfac*iveloc
        iegl4   =       iegl4*(1-ivfac)+iegl4*ivfac*iveloc

        irs     =       irscl*irss
        iegr1   =       (iegr1+irs > 99 ? 99 : iegr1+irs)
        iegr2   =       (iegr2+irs > 99 ? 99 : iegr2+irs)
        iegr3   =       (iegr3+irs > 99 ? 99 : iegr3+irs)
        iegr4   =       (iegr4+irs > 99 ? 99 : iegr4+irs)

        irfn    =       (iegl1 > iegl4 ? irisefn : idecfn)
        iegd1   table   iegr1 , irfn                    ; convert rate->dur
        ipct1   table   iegl4 , irfn+1                  ; pct fn is next one
        ipct2   table   iegl1 , irfn+1
        iegd1   =       abs(iegd1*ipct1-iegd1*ipct2)
        iegd1   =       (iegd1 == 0 ? .001 : iegd1)

        irfn    =       (iegl2 > iegl1 ? irisefn : idecfn)
        iegd2   table   iegr2 , irfn
        ipct1   table   iegl1 , irfn+1
        ipct2   table   iegl2 , irfn+1
        iegd2   =       abs(iegd2*ipct1-iegd2*ipct2)
        iegd2   =       (iegd2 == 0 ? .001 : iegd2)

        irfn    =       (iegl3 > iegl2 ? irisefn : idecfn)
        iegd3   table   iegr3 , irfn
        ipct1   table   iegl2 , irfn+1
        ipct2   table   iegl3 , irfn+1
        iegd3   =       abs(iegd3*ipct1-iegd3*ipct2)
        iegd3   =       (iegd3 == 0 ? .001 : iegd3)

        iegd4   table   iegr4 , idecfn
                if      (iegl3 <= iegl4) igoto continue
        ipct1   table   iegl3 , irfn+1
        ipct2   table   iegl4 , irfn+1
        iegd4   =       abs(iegd4*ipct1-iegd4*ipct2)
        iegd4   =       (iegd4 == 0 ? .001 : iegd4)
continue:
                if      (iop > 1) igoto op2
op1:
```

```
␣␣␣␣␣␣␣␣i1egd1␣␣=␣␣␣␣␣␣␣iegd1
␣␣␣␣␣␣␣␣i1egd2␣␣=␣␣␣␣␣␣␣iegd2
␣␣␣␣␣␣␣␣i1egd3␣␣=␣␣␣␣␣␣␣iegd3
␣␣␣␣␣␣␣␣i1egd4␣␣=␣␣␣␣␣␣␣iegd4
␣␣␣␣␣␣␣␣i1egl1␣␣=␣␣␣␣␣␣␣iegl1
␣␣␣␣␣␣␣␣i1egl2␣␣=␣␣␣␣␣␣␣iegl2
␣␣␣␣␣␣␣␣i1egl3␣␣=␣␣␣␣␣␣␣iegl3
␣␣␣␣␣␣␣␣i1egl4␣␣=␣␣␣␣␣␣␣iegl4
␣␣␣␣␣␣␣␣i1ams␣␣␣=␣␣␣␣␣␣␣iams
␣␣␣␣␣␣␣␣i1hz␣␣␣␣=␣␣␣␣␣␣␣ihz
␣␣␣␣␣␣␣␣iop␣␣␣␣␣=␣␣␣␣␣␣␣iop␣+␣1
␣␣␣␣␣␣␣␣iopfn␣␣␣=␣␣␣␣␣␣␣iop2fn
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣igoto␣␣␣loop

op2:␣␣␣␣␣␣␣␣␣␣␣␣if␣␣␣␣␣␣(iop␣>␣2)␣igoto␣op3
␣␣␣␣␣␣␣␣i2egd1␣␣=␣␣␣␣␣␣␣iegd1
␣␣␣␣␣␣␣␣i2egd2␣␣=␣␣␣␣␣␣␣iegd2
␣␣␣␣␣␣␣␣i2egd3␣␣=␣␣␣␣␣␣␣iegd3
␣␣␣␣␣␣␣␣i2egd4␣␣=␣␣␣␣␣␣␣iegd4
␣␣␣␣␣␣␣␣i2egl1␣␣=␣␣␣␣␣␣␣iegl1
␣␣␣␣␣␣␣␣i2egl2␣␣=␣␣␣␣␣␣␣iegl2
␣␣␣␣␣␣␣␣i2egl3␣␣=␣␣␣␣␣␣␣iegl3
␣␣␣␣␣␣␣␣i2egl4␣␣=␣␣␣␣␣␣␣iegl4
␣␣␣␣␣␣␣␣i2ams␣␣␣=␣␣␣␣␣␣␣iams
␣␣␣␣␣␣␣␣i2hz␣␣␣␣=␣␣␣␣␣␣␣ihz
␣␣␣␣␣␣␣␣iop␣␣␣␣␣=␣␣␣␣␣␣␣iop␣+␣1
␣␣␣␣␣␣␣␣iopfn␣␣␣=␣␣␣␣␣␣␣iop3fn
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣igoto␣␣␣loop

op3:␣␣␣␣␣␣␣␣␣␣␣␣if␣␣␣␣␣␣(iop␣>␣3)␣igoto␣op4
␣␣␣␣␣␣␣␣i3egd1␣␣=␣␣␣␣␣␣␣iegd1
␣␣␣␣␣␣␣␣i3egd2␣␣=␣␣␣␣␣␣␣iegd2
␣␣␣␣␣␣␣␣i3egd3␣␣=␣␣␣␣␣␣␣iegd3
␣␣␣␣␣␣␣␣i3egd4␣␣=␣␣␣␣␣␣␣iegd4
␣␣␣␣␣␣␣␣i3egl1␣␣=␣␣␣␣␣␣␣iegl1
␣␣␣␣␣␣␣␣i3egl2␣␣=␣␣␣␣␣␣␣iegl2
␣␣␣␣␣␣␣␣i3egl3␣␣=␣␣␣␣␣␣␣iegl3
␣␣␣␣␣␣␣␣i3egl4␣␣=␣␣␣␣␣␣␣iegl4
␣␣␣␣␣␣␣␣i3ams␣␣␣=␣␣␣␣␣␣␣iams
␣␣␣␣␣␣␣␣i3hz␣␣␣␣=␣␣␣␣␣␣␣ihz
␣␣␣␣␣␣␣␣iop␣␣␣␣␣=␣␣␣␣␣␣␣iop␣+␣1
␣␣␣␣␣␣␣␣iopfn␣␣␣=␣␣␣␣␣␣␣iop4fn
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣igoto␣␣␣loop

op4:␣␣␣␣␣␣␣␣␣␣␣␣if␣␣␣␣␣␣(iop␣>␣4)␣igoto␣op5
```

```
        i4egd1  =        iegd1
        i4egd2  =        iegd2
        i4egd3  =        iegd3
        i4egd4  =        iegd4
        i4egl1  =        iegl1
        i4egl2  =        iegl2
        i4egl3  =        iegl3
        i4egl4  =        iegl4
        i4ams   =        iams
        i4hz    =        ihz
        iop     =        iop + 1
        iopfn   =        iop5fn
                  igoto   loop

op5:            if      (iop > 5) igoto op6
        i5egd1  =        iegd1
        i5egd2  =        iegd2
        i5egd3  =        iegd3
        i5egd4  =        iegd4
        i5egl1  =        iegl1
        i5egl2  =        iegl2
        i5egl3  =        iegl3
        i5egl4  =        iegl4
        i5ams   =        iams
        i5hz    =        ihz
        iop     =        iop + 1
        iopfn   =        iop6fn
                  igoto   loop

op6:
        i6egd1  =        iegd1
        i6egd2  =        iegd2
        i6egd3  =        iegd3
        i6egd4  =        iegd4
        i6egl1  =        iegl1
        i6egl2  =        iegl2
        i6egl3  =        iegl3
        i6egl4  =        iegl4
        i6ams   =        iams
        i6hz    =        ihz
;=============================================================================
                  ;timout  idur,999,final          ;skip during final decay
        k1sus   linseg  i1egl4,i1egd1,i1egl1,i1egd2,i1egl2,i1egd3,i1egl3,1,i1egl3
        k2sus   linseg  i2egl4,i2egd1,i2egl1,i2egd2,i2egl2,i2egd3,i2egl3,1,i2egl3
        k3sus   linseg  i3egl4,i3egd1,i3egl1,i3egd2,i3egl2,i3egd3,i3egl3,1,i3egl3
```

```
        k4sus   linseg  i4egl4 , i4egd1 , i4egl1 , i4egd2 , i4egl2 , i4egd3 , i4egl3 , 1 , i4egl3
        k5sus   linseg  i5egl4 , i5egd1 , i5egl1 , i5egd2 , i5egl2 , i5egd3 , i5egl3 , 1 , i5egl3
        k6sus   linseg  i6egl4 , i6egd1 , i6egl1 , i6egd2 , i6egl2 , i6egd3 , i6egl3 , 1 , i6egl3
        k1phs   =       k1sus
        k2phs   =       k2sus
        k3phs   =       k3sus
        k4phs   =       k4sus
        k5phs   =       k5sus
        k6phs   =       k6sus
                kgoto   output
final:
        k1fin   linseg  1,i1egd4 ,0 ,1 ,0
        k1phs   =       i1egl4+(k1sus-i1egl4)*k1fin
        k2fin   linseg  1,i2egd4 ,0 ,1 ,0
        k2phs   =       i2egl4+(k2sus-i2egl4)*k2fin
        k3fin   linseg  1,i3egd4 ,0 ,1 ,0
        k3phs   =       i3egl4+(k3sus-i3egl4)*k3fin
        k4fin   linseg  1,i4egd4 ,0 ,1 ,0
        k4phs   =       i4egl4+(k4sus-i4egl4)*k4fin
        k5fin   linseg  1,i5egd4 ,0 ,1 ,0
        k5phs   =       i5egl4+(k5sus-i5egl4)*k5fin
        k6fin   linseg  1,i6egd4 ,0 ,1 ,0
        k6phs   =       i6egl4+(k6sus-i6egl4)*k6fin



;----------------------Algorithm 17-------------------------------------;
                if      k1fin > 0 kgoto output
                ; turnoff                ; when carrier oscil(s) done , turn off.
output:
  k1gate  tablei  k1phs , iampfn    ; use ampfn for any carrier
  k2gate  tablei  k2phs , idevfn    ; use devfn for any modulator
        k3gate  tablei  k3phs , idevfn
        k4gate  tablei  k4phs , idevfn
        k5gate  tablei  k5phs , idevfn
        k6gate  tablei  k6phs , idevfn

  a6sig   oscili  k6gate , i6hz , 1

  a5phs   phasor  i5hz
  a5sig   tablei  a5phs+a6sig , 1 , 1 , 0 , 1
  a5sig   =       a5sig*k5gate

  a4sig   oscili  k4gate , i4hz , 1
```

```
  a3phs   phasor  i3hz
  a3sig   tablei  a3phs+a4sig ,1 ,1 ,0 ,1
  a3sig   =       a3sig*k3gate

  a2sig   init    0                ; initialize for feedback
  a2phs   phasor  i2hz
  a2sig   tablei  a2phs+a2sig*(.5*ifeed) ,1 ,1 ,0 ,1
  a2sig   =       a2sig*k2gate

  a1phs   phasor  i1hz
  a1sig   tablei  a1phs+a2sig+a3sig+a5sig ,1 ,1 ,0 ,1
  a1sig   =       a1sig*k1gate
                  outs a1sig*ipkamp , a1sig*ipkamp
endin
```

A.4 File: Instruments.csd

```
f1 0 65536 10 1          ; Sine
f2  0 1024  7 1 1024 1

;-------------------------------------------------------------------------------
; PART 1 DEEP DRONE
;-------------------------------------------------------------------------------
f12 0 1024  7 1 512 .5 512 1                    ; EnvFco
f13 0 1024  7 1  1024 1                         ; EnvRez
f14 0 1024  7 0 114   1 206 .5 640 .5   64 0    ; Amp

f32 0 1024 -7 250 1024 250    ; Fco
f33 0 1024 -7 5   1024 5      ; Rez


;-------------------------------------------------------------------------------
; INSTRUMENT 11 (TRUMPET) FUNCTION TABLES
;-------------------------------------------------------------------------------
f201  0  512  10  1

; envelopes
f231 0 513 7 0 80 1  80 .85  290 .8 63 0       ; amplitude & index envelope



;-------------------------------------------------------------------------------
; INSTRUMENT 14 (CLARINET) FUNCTION TABLES
;-------------------------------------------------------------------------------
f50  0  512  10  1

; envelopes
f51 0 1024  5  .0001 200 1 674 1 150 .0001        ; amplitude envelope
f52 0 1024  5  1 1024 .0001                       ; index envelope


;-------------------------------------------------------------------------------
; INSTRUMENT 18 function tables.
;-------------------------------------------------------------------------------
f101     0       512     10      1
; operator fpoutput level to amp scale function (data from Chowning/Bristow)
f102     0       128     7       0       10      .003    10      .013
        10      .031    10      .079    10      .188    10      .446
        5       .690    5       1.068   5       1.639   5       2.512
```

```
        5       3.894   5       6.029   5       9.263   4       13.119
        29      13.119
; rate scaling function
f103    0       128     7       0       128     1
; eg rate rise function for lvl change between 0 and 99 (data from Opcode)
f104    0       128     -7      38      5       22.8    5       12      5
        7.5     5       4.8     5       2.7     5       1.8     5       1.3
        8       .737    3       .615    3       .505    3       .409    3
        .321    6       .080    6       .055    2       .032    3       .024
        3       .018    3       .014    3       .011    3       .008    3
        .008    3       .007    3       .005    3       .003    32      .003
; eg rate rise percentage function
f105    0       128     -7      .00001  31      .00001  4       .02     5
        .06     10      .14     10      .24     10      .35     10      .50
        10      .70     5       .86     4       1.0     29      1.0
; eg rate decay function for lvl change between 0 and 99
f106    0       128     -7      318     4       181     5       115     5
        63      5       39.7    5       20      5       11.2    5       7
        8       5.66    3       3.98    6       1.99    3       1.34    3
        .99     3       .71     5       .41     3       .15     3       .081
        3       .068    3       .047    3       .037    3       .025    3
        .02     3       .013    3       .008    36      .008
; eg rate decay percentage function
f107    0       128     -7      .00001  10      .25     10      .35     10
        .43     10      .52     10      .59     10      .70     10      .77
        10      .84     10      .92     9       1.0     29      1.0
; eg level to peak deviation mapping function (index in radians = Index / 2PI)
f108    0       128     -7      0       10      .000477 10      .002
        10      .00493  10      .01257  10      .02992  10      .07098
        5       .10981  5       .16997  5       .260855 5       .39979
        5       .61974  5       .95954  5       1.47425 4       2.08795
        29      2.08795
; velocity to amp factor mapping function (rough guess)
f109    0       129     9       .25     1       0
; velocity sensitivity scaling function
f110    0       8       -7      0       8       1
; feedback scaling function
f111    0       8       -7      0       8       7
f112    0   32   -2   99   2   75   37   18   63
        99  70   0   0
        0   0   0.505000   0
        3
f113    0   32   -2   80   2   28   37   42   50
        99  0   0   0
        0   0   0.515000   0
```

```
        1
f114    0     32     -2     68     3    73    25    32    30
        97    78     0      0
        0     0      1.000000     7
        3
f115    0     32     -2     99     2    80    39    28    53
        93    57     0      0
        0     0      0.500000     0
        3
f116    0     32     -2     75     2    99    51    0     0
        99    74     0      0
        0     0      1.010000     0
        4
f117    0     32     -2     87     7    25    50    24    55
        96    97     0      0
        0     0      0.500000     1
        3


; Set the tempo
t 0 80
```

Appendix B

Questionnaire Results

A summary of the questionnaire results are presented here. Actual questionnaires are not included due to data protection issues since the respondant was asked for their contact details to allow followups if necessary.

B.1 Questions

1) Please rate your musical knowledge/experience on the scale of:

1. None (1)
2. Beginner (2)
3. Intermediate (3)
4. Advanced (4)
5. Professional (5)

2) Please rate the structure shown by the music on a scale of 1-10?

3) Please rate the quality of the musicality of the solo sections on a scale of 1-10?

4) Please rate the links between the whole band and the solo sections on a scale of 1-10?

5) Please rate the quality of the melody line of the music on a scale of 1-10?

6) Please rate the quality of the bass line of the music on a scale of 1-10?

7) Please rate how well the bass line interlinks with the melody on a scale of 1-10?

8) Please rate on a scale of 1-10 the sense of rhythm shown within the music?

9) Please rate how much you enjoyed listening to the music played on a scale of 1 - 10?

10) Please rate the quality of the band you believe would have played the recording of this music?

1. Beginner (1)
2. Intermediate (2)
3. Advanced (3)
4. Professional (4)

11) Although the sound is obviously computer generated, could you believe if played to you without any prior knowledge, that this was composed by a person, rather than a computer?

12) Any further comments

B.2 Results

Participant	Question Number										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
A	4	6	4	5	6	8	7	5	4	2	Yes
B	5	7	8	7	6	8	7	6	3	1	No
C	4	6	5	6	5	5	5	8	7	2	Yes
D	3	6	3	4	2	3	6	4	4	2	No
E	5	5	5	6	6	4	5	3	6	1	Yes
F	4	6	7	6	5	7	4	2	6	2	No
G	5	5	6	5	3	6	3	5	8	2	Yes
H	4	3	3	2	3	3	4	1	2	2	Yes
I	3	6	5	4	6	5	6	7	5	2	Yes
J	3	5	6	2	4	7	6	3	5	1	No
K	4	7	7	6	3	4	4	5	7	1	Yes
L	4	6	4	4	6	6	5	6	5	2	No
M	4	3	6	2	3	4	7	5	8	2	Yes
N	4	5	8	3	5	8	4	7	8	2	Yes
O	5	3	9	5	8	9	3	5	5	2	Yes
P	3	7	4	2	2	5	6	4	4	2	No
Q	4	5	3	4	5	4	7	7	3	2	No
R	4	6	6	6	5	8	3	6	5	2	Yes
S	4	5	8	2	4	6	7	5	5	2	Yes
T	2	8	7	7	6	6	6	7	8	2	Yes
Average	**3.9**	**5.5**	**5.7**	**4.4**	**4.65**	**5.8**	**5.25**	**5.05**	**5.4**		