The Development and Implementation of Algorithms for Composition from Chaotic Numerical Systems

Nicholas Charlton

Bachelor of Science in Mathematics and Computer Science with Honours

University of Bath

May 2005

The Development and Implementation of Algorithms for Composition from Chaotic Numerical Systems

Submitted by Nicholas Charlton

Copyright

Declaration

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

Signed.....

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

Signed.....

Abstract

Modern algorithmic composition utilises the processing power and speed of computers to produce musical pieces, given any level of initial information defined by the user. This project focuses on one specific concept in this field, that of encapsulating the behaviour of chaotic numerical systems musically, through strict algorithmic manipulation. For consistency, the pieces produced are written for solo piano. This dissertation documents the development of these algorithms and their implementation in a succinct, portable and user-friendly system, and concludes with a discussion of the musical possibilities posed by such a concept.

Acknowledgements

I would like to thank my supervisor, Professor John ffitch, for introducing me to the field of algorithmic composition and the relevance of chaos in music and art. I would also like to thank Vicky for her proof reading and support throughout.

Contents

List of Figures

Contents of the CD

- Dissertation.pdf – *This document in PDF format*
- Cygwin.exe – *Setup file for UNIX emulator*
- Cygwin – *Directory containing setup files for LilyPond etc.*
- Manual.pdf – *User manual*
- ChaoticComposer – *Directory containing program*
    - compose.exe – *Main Program*
    - Interface – *Directory*
        - ChaoticComposer.exe – *User interface program*
    - Compositions – *Directory for storage of compositions*
        - LogEx – *Directory with example from Logistic map*
        - HenEx – *Directory with example from Hénon map*
        - StaEx – *Directory with example from Standard map*
        - LorEx – *Directory with example from Lorenz map*
        - RosEx – *Directory with example from Rossler map*
- Source Code – *Directory containing all appropriate code*
    - Interface – *Directory*

1 Introduction

Although music, like most concepts, has clear roots in basic mathematics, a deeper link between the two areas has generated interest in both communities since early investigations by the Ancient Greeks. This project will aim to investigate the possibilities and the scope of automated composition given minimal user input by means of the development of generalised algorithms. These algorithms will, in particular, take a minimal number of parameters defining a chaotic system and map an implicitly selected subset of the system's output to a finite piece of music.

This particular branch of algorithmic composition has often been explored in the recent past, but has generally incorporated a high level of objectivity throughout the composition process. The composer would, perhaps, examine the behaviour of one or two specific chaotic orbits and compose music based on, or inspired by these orbits, and would, in general, manipulate figures and behavioural shapes to generate the best results. The paradigm of completely automated composition, given an arbitrary chaotic orbit has, however, seldom been investigated. It is precisely this that shall form the main aim of this project.

The algorithms themselves will aim to preserve the dynamic behaviour of the chaotic systems while maintaining a reasonable level of structure and musicality in the composition. It is the geometric beauty of some of these chaotic orbits that inspires this idea of automated composition, and that beauty should be retained as well as possible in the music generated.

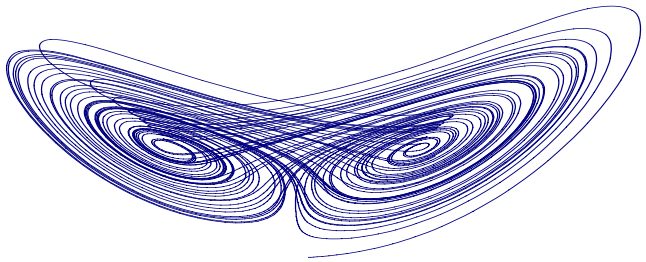

Fig. 1.1 Lorenz's "Butterfly" Map

The second half of this project shall focus on the physical implementation of these algorithms, in order to create concrete musical scores from user-specified chaotic systems.

2 Literature Review

2.1 Nonlinear Dynamics and Chaos

Chaos is the study of certain iterated nonlinear systems under particular initial conditions, which display in some sense seemingly random, but bounded, behaviour. The theory of chaos has generated a lot of interest over the last century from those attempting to model the complicated behaviour of certain natural phenomena (Lorenz, Hénon et al.), attracting most attention for the eventually spectacular variations in a system's output generated by very slight changes in its initial conditions (known as sensitive dependence). The iterations of a map are a means of showing progression in time, the value of each representing the state of the system after so many time increments.

2.1.1 Definitions

In order to understand and classify chaos, it has been rigorously defined in clear mathematical terms, and derived from universally recognised concepts in linear and nonlinear algebra. Although the idea of chaos and definitions involved can be applied with little additional work to the complex domain and in some respect generalised to apply to all spaces, science is most concerned with its applications in the n-dimensional real domain. The following definitions (Alligood et al. 1996) concern a continuously differentiable map $f : \mathbb{R}^n \to \mathbb{R}^n$ for some $n \in \mathbb{N}$.

Let $x \in \mathbb{R}^n$. Then the **orbit of** x **under** f is the set of iterations of applications of f to the previous element, that is

$$\{ f^k(x) : k \in \mathbb{N} \cup \{0\} \} = \{ x, f(x), f^2(x), \ldots \}$$

where f^k represents f being applied k times.

A point $p \in \mathbb{R}^n$ is called a **periodic point** of f if $\exists k \in \mathbb{N}$ such that $f^k(p) = p$. The orbit of p under f is called a **periodic orbit**.

It is necessary to quantify the difference in orbits of two initial values of x very close together, in order to see the effect of variations in initial conditions. The **Lyapunov number** $L(x)$ represents the average difference in the distance between orbits after each iteration. For example, a Lyapunov number of 2 for an orbit of $x_1 \in \mathbb{R}^n$ under f would suggest that for a initial value of x close to x_1, each iteration of f would double the distance between the corresponding elements in the two orbits. The **Lyapunov exponent** $h(x_1)$ is defined to be

$$h(x_1) := \ln[L(x_1)]$$

An orbit $\{x_1, x_2, x_3, ...\}$ is called **asymptotically periodic** if it converges to a periodic orbit, i.e. there exists some periodic orbit $\{y_1, y_2, ..., y_k, y_1, y_2, ...\}$ such that

$$\lim_{n\to\infty} |x_n - y_n| = 0$$

Although the definitions vary slightly for generalised n-dimensional maps, the ideas behind a chaotic system are consistent throughout domains, and can be simplified most in one-dimension.

A bounded orbit of some initial value $x_1 \in \mathbb{R}$ under a map $f : \mathbb{R} \to \mathbb{R}$ is said to be **chaotic** if

1. it is not asymptotically periodic,
2. the Lyapunov exponent $h(x_1) > 0$.

The second condition implies that for an orbit to be considered chaotic, no other initial value should give an orbit which at any point moves closer to the first.

2.1.2 Chaos in Multiple Dimensions

As this project is not so concerned with the classification and existence of chaos in certain systems, but more so with its applications, it will not be necessary to go into detail about its rigorous definition in the n-dimensional real domain. Instead it will suffice to carry through the ideas from one dimension where some basis for investigation has been established.

2.1.3 Differential Equations

For an initial value problem with an autonomous differential equation $\dot{x} = \frac{dx}{dt} = f(x)$ where $f : \mathbb{R}^n \to \mathbb{R}^n$, and initial value $x_0 \in \mathbb{R}^n$, the **flow** is defined to be a function $F : \mathbb{R}_+ \times \mathbb{R}^n \to \mathbb{R}^n$ of time and the initial condition which represents the set of solutions to the differential equation. $F(t, x_0)$ represents the value of the unique solution, given initial value x_0, at time $t > 0$ (Alligood et al. 1996).

The flow of a differential equation is a simple representation of the development of the system over time, given certain initial conditions, and as with the iterated systems, can display chaotic behaviour. The main difference between the two types of system is that while iterated maps give the state of the system at discrete time intervals, flows are mapped totally from continuous time values. For computational purposes, however, these flows can be sampled to give a countable set of values, taken at discrete time intervals:

$$F(t, x_0) \to \{F(0, x_0), F(\Delta t, x_0), F(2\Delta t, x_0), ...\} \text{ for some time interval } \Delta t > 0 .$$

In this form the same criteria can be applied, in some sense, as with iterated maps for the classification of the system as chaotic. This is, however, in no way a reliable and rigorous approach for classification. It is only really useful when the presence of chaos is known and a comparison with iterated maps is desired. In an attempt to generalise chaotic systems for computational purposes, this technique will serve as a useful tool.

The most famous example of a system of differential equations with chaotic flows is the Lorenz map, developed from a simplified model of atmospheric turbulence and described by three differential equations

$$\begin{aligned} \dot{x} &= -\sigma(x - y) \\ \dot{y} &= rx - y - xz \\ \dot{z} &= xy - bz \end{aligned}$$

where $\sigma, r, b > 0$ constant. When the flow of this system is plotted for certain initial conditions and constants, taking (x, y, z) to be a vector in $\mathbb{R}^3$, it becomes apparent that it is chaotic, and produces the well-known butterfly shaped plot. It has become somewhat common practice in numerical experimentation with this model to fix $\sigma = 10$, $b = 8/3$ and to vary r, the 'Raleigh number', in order to produce the most interesting results (Frøyland 1992).

2.1.4 Computation

For computation and formality purposes, iterated nonlinear systems can be represented as systems of difference equations. The orbit $\{x, f(x), f^2(x), ...\}$ can be represented by the difference equation

$$x_{n+1} = f(x_n)$$
$$x_0 = x$$

where the increment on n represents progression by a fixed time interval. This equation fits nicely into an iterative loop, which simplifies computation and representation of data (Bidlack 1992).

Differential equations may be manipulated to mimic difference equations, using numerical solutions, such as *Euler's Forward Method:*

$$x_{n+1} = x_n + \Delta t \cdot f(x_n)$$
$$x_0 = x(0)$$

for sufficiently small $\Delta t > 0$. This method is utilised by Bidlack (1992) in calculating a discrete approximation to the Lorenz map, taking $\Delta t = 0.01$.

2.2 Algorithmic Composition

2.2.1 History

In some sense, it could be argued that all conventional composition follows algorithms, as it is generally intended for the pleasure of the human ear, which to some extent likes to hear predictable sound progressions and structures. Even in the development of some of the most unconventional pieces, some sort of algorithm is followed, albeit an unconventional one. When this is taken into consideration, it seems a perfectly natural desire for humans to pursue higher levels of algorithmic involvement, implementing more formal rules and processes, which minimise the creative input required from the composer.

The concept of composing music using formalisms dates back as far as the Ancient Greeks, with Pythagoras, Plato and Ptolemy all expressing and exploring an interest in the relation between music and mathematics (Maurer 1999). To refer to this exploration as "algorithmic composition", however, would require a great stretch of the definition, as the composer would rarely do more than take inspiration from numerical phenomena in the natural world (Maurer 1999).

It was during the Middle Ages that pure formalisms were first recognised and documented as genuine techniques for composition, with examples of automated mappings to melody, rhythm and form from such sources as religious texts and the roll of dice (Mozart's "Dice Music") (Maurer 1999, Järveläinen 2000).

Although algorithmic composition was ever present in the music community, particularly during the early 20th Century when composers were exploring many unorthodox paradigms (Arnold Schönberg, Anton Webern, Iannis Xenakis, John Cage et al.), it was the advent of the modern computer that generated the most interest in the subject. The composer had the freedom to experiment with techniques and approaches while the computer dealt with the data processing. This made it far easier and quicker to implement an idea, and greatly expanded the scope of what was possible in the field.

2.2.2 Techniques

A number of techniques have been developed by composers, in an attempt to either imitate the human composition process or to bring new and alternative ideas to the field. There is a great deal of overlap between a number of these approaches, each utilising similar fundamental ideas in such areas as balancing "creativity" and formality.

Randomness

Perhaps the simplest approach to algorithmic composition, the idea of generating notes based on random events alone is a very idealistic one and is in practice, unsurprisingly, rarely successful. Taken to a higher level, however, where melodic fragments are pieced together with the help of randomly generated numbers or events, the results can be considerably more acceptable (assuming the fragments are well written). This method has been explored by numerous composers, most notably Mozart (*Musikalisches Wurfelspiel* or "Dice Music") and more recently John Cage (*Reunion*), making use of the roll of dice and a game of chess respectively (Alpern 1995).

Stochastic

In many ways the next natural step up from a purely random approach, stochastic composition combines statistics and random number generation to produce pieces in a certain vein, depending on composer-defined probabilities. Iannis Xenakis pioneered this method, using a computer to generate sets of probabilistic data and then make decisions for composition using a random number generator (Xenakis, 1963 cited by Alpern 1995).

Fractals and Chaos

A concept reminiscent of the Ancient Greek interest in nature's relevance in music, fractals and chaos are most commonly used to model natural world phenomena and so have been of interest to composers since their discovery. Although fractals are of some interest, they lack the variety and subtlety which comes with composing with chaos, which has the potential to produce some very interesting results (Leach and Fitch 1995). This field will be investigated in more depth in the next subsection.

Cellular Automata

An automaton will generate an output state based on its input state. This output will then be fed back into the automaton and the process repeated. The use of such machines in composition leads to pieces in which the nature of a phrase is governed by that of the preceding phrase (possibly along with other previous phrases or elements). Multiple automata can run concurrently to govern the behaviour of various aspects of the music, perhaps until some final state is achieved (Järveläinen 2000).

Rule-based and Grammars

Although in practice, most compositional methods incorporate a set of rules, predominantly rule-based systems have generally been used to generate pieces in some specific style. A list of rules and restrictions will be defined for the algorithm to follow, possibly allowing a small degree of random input for creativity, in order to produce a number of similar pieces. These rules can range from trivial standards for western tonal music to complex structural guidance, and depending on the requirements of the composer, can constitute lists ranging in length from just a few simple guidelines to hundreds, or even thousands, of specific instructions. These lists will generally be implemented along with some other technique with outside influence to maintain a reasonable degree of variety (Burns 1996, Järveläinen 2000).

Artificial Intelligence

The idea of using artificial intelligence (AI) to compose involves the implementation of a rule-based algorithm, from which the program will "learn" new rules as it progresses according to its success with trial data. The result will be a larger set of rules from which the system will be able compose successfully with far greater ease (Burns 1996, Maurer 1999).

2.2.3 Chaos

The relatively new mathematical field of chaos has provoked a lot of interest in the musical community over the past 20 years, and has been explored as a tool for algorithmic composition in a number of different ways (Pressing 1988, DiScipio 1990, Gogins 1991, all cited by Alpern 1995; Bidlack 1992, Leach and Fitch 1995). As Leach and Fitch (1995) point out, the natural behaviour of chaotic systems relates closely in many ways to the compositional idiom of a human.

Further investigation into these systems reveals perhaps the most interesting aspect for composition – near repetition. The average composer will, consciously or not, repeat themes throughout his music, without necessarily repeating phrases exactly, adding a sense of identity and direction. The near repetition of chaotic systems serves as an ideal tool for mimicking this process, while their never landing twice on the same point provides variety and progression.

In his 1992 paper for the *Computer Music Journal*, Bidlack described perhaps the most trivial application of chaotic systems to music, with each element of an orbit being mapped directly to a single note. In one dimension the value of the element would simply represent the pitch, while an extension to two dimensions would provide an additional domain for mappings to dynamic levels. He suggested that these values could be mapped to any parameter of a note ("such as pitch, dynamic level, rhythm, and instrumentation" (Bidlack 1992)), but in his experiments restricted hypothetical results to pitch and dynamics. It was, however, proposed that in general, this simplistic approach is not sufficient as a reasonable compositional algorithm and these systems should be used to generate, arrange and manipulate higher level events, such as note sequences.

This idea was put into practice by Leach and Fitch (1995) when they analysed a finite orbit of a chaotic system in one dimension for local and global maxima. This analysis was applied to a generalised idea of "major events" in composition using event trees. Local maxima in the orbit would be considered smaller, localised major events, while global maxima would represent the central major events. This concept encapsulated the key goal in automated composition – a strict universal algorithm that took into consideration true creative compositional ideas without loss of computational generality. In this particular case, the essence of the chaotic orbit, and hence that of the natural phenomenon it represents, is maintained in the composition without sacrificing musical quality.

It is at this point that the overlap in techniques for algorithmic composition becomes important. The work done by Leach and Fitch has supplied a pure algorithm for mapping chaos to musical structure, which provides the foundations of the composition, but more is needed. At some stage, outside influence is required to build on what has so far been generated, a user-defined sequence of notes, for example, which can be manipulated by the event tree. Alternatively, a set of rules could be followed for the generation of a sequence of notes, which are

otherwise random. The point is that at some stage in its development, a good algorithm will lose its purity and in some sense its generality, and hence for an algorithm to produce the best results, it must combine the key aspects of a number of compositional techniques.

It is common for composers to concentrate on one or several particular examples when investigating such a subject as chaos, and as a result many pieces are generated through algorithms biased towards these examples (Little 1999). Although the output of these algorithms when applied to the corresponding systems (in this case) is generally of relatively high quality, there is a lack of universality. Application of the algorithms to different systems will generally not give desirable results, a clear flaw in their classification as "compositional algorithms".

2.2.4 Scope and Success

As mentioned in the previous subsection, the success of composing from chaos with pure, generalised algorithms has been somewhat limited by restrictive nature of the criteria. Leach and Fitch (1995) have shown that a good basis for composition can be generated by pure algorithms, but without the inclusion of a rule-set or grammar, a random element, or some other separate technique, quality of output is in the hands of the behaviour of the chaotic system. Purists may argue that results after outside influence are not a true representation of the chaotic systems that inspired them, but it could also be said that any level of manipulation of data in an algorithm will destroy its purity.

When it comes down to it, it should always be the composer and the composer alone who decides on the level and the nature of manipulation of data, as it is their own work that is to be produced. This adds weight to the argument that music produced in this way should be considered a composition to be attributed to the composer of the algorithm.

Bidlack (1992) has shown that it is possible to write algorithms using only linear mappings to manipulate the data from any chaotic orbit, and that these will produce a different composition for each different chaotic system. The success in terms of quality of this level of generality and simplicity, however, is questionable, and further investigation is required into the potential of this level of

algorithm. It is very easy in development of this kind to lose sight of generality to give “nicer” results, but it must be maintained for optimisation.

2.3 Implementation

2.3.1 Previous Developments

Given the interest that chaos has generated in the computer music community, it is unsurprising that bespoke software has been developed for the processing and manipulation of chaotic systems in view of composition. Two particular examples encountered during the research process for this project are *Chaosmuse* (Alpern 1995) and *XComposer* (Leach and Fitch 1995), both of which involve some level of user-interaction in addition to defining nonlinear systems for manipulation.

Chaosmuse is written in pure ANSI C and takes equation systems and minimal musical parameters from the user via a text-based, interactive interface. The program manipulates the systems to produce musical pieces in audio form, or other formats for the user’s further development. Although this software has a basis for composition with chaos, Alpern (1995) notes that pieces produced lack structure and direction, “In it’s current state, the music it produces tends to wander aimlessly, with key shifts happening only by accident, if at all” (Alpern 1995). He suggests the inclusion a larger scale decision process to deal with issues of structure and form, citing in particular a rule-based algorithm of some kind.

XComposer takes a slightly different approach, and implements the ideas raised by Leach and Fitch (1995) about chaotic systems as tools for event tree generation (as discussed previously [2.3]). The software allows the user to define a number of systems of chaotic formulae, how they influence one another (modelling interference between chaotic systems in the real world), a rhythm list, and how they are utilised to generate a melody. This system is somewhat more developed than *Chaosmuse* (as it was in 1995), as it already involves some level of rule-based influence, defined by both the user and the program, “The present version uses a default scale tree based on the major scale, and is calibrated to use movement through chords to provide a sense of direction and resolution” (Leach and Fitch 1995).

2.3.2 Algorithms

As this project will be heavily rooted in theory, the implementation of the algorithms will take a backseat through the main development, before taking precedence when it comes to the representation of the data. In view of this, the language used for development should as clear and flexible as possible, and should allow for a rigorous implementation of algorithms. It will do little more than process mathematical systems and convert the output to a suitable form for passing on for representation. The main additional requirement will be the capacity to create a user interface of some description. Portability of the system produced is also a key issue, and one that the chosen language should address, preferably through platform transparency.

In light of these requirements, it seems natural to choose to implement the algorithms in C. This would allow for potential development on most platforms and would produce a portable executable file.

2.3.3 Representation

The representational side of this project cannot be simplified like the algorithmic side, as it will require complicated conversions from an effectively numerical representation of music to formats recognised as standardised representations, such as audio output and sheet notation. Ideally this process should be incorporated into the coding for the algorithmic side, in order to contain the entire program neatly and consistently in a simplistic and compact form. This, however, would involve investigation and learning far beyond the bounds of this project, and would take over completely from algorithm development as the central agenda.

Therefore, outside help is required. The final program should incorporate some external software to process the output from the algorithms and return an appropriate musical representation of the data. It should have the ability to produce sheet music, as well as some audio representation, depending on the requirements of the user. It should employ a logical input structure, so that the data can be converted for processing as easily as possible, and should produce output which is as portable as possible.

Investigation into the field has revealed a number of possible candidates, most of which will produce output in either audio format or sheet notation, but few which combine both capabilities. *LilyPond* is open source software developed very recently as a platform for the interpretation and notational engraving of coded musical scores. It takes a very logical system of code as input, which includes syntax for just about any conventional musical object the user could desire. Output comes in the form of sheet music in PDF, PS and DVI formats, and a MIDI file will be written at the user's request, in which the piece is performed at a user-specified tempo.

The clear drawback of using such a system, however, is the loss of portability. Although *LilyPond* has been developed to run on most common platforms, a slightly different version must be acquired for each. For the confines of this project, however, the complications this issue brings will be overlooked, given their inferiority against the advantages of using *LilyPond*.

2.4 Summary

In order for this project to achieve its goal of producing generalised algorithms for composition from values in chaotic orbits with minimal human interference, it must take into consideration all the previous work in the subject. It is apparent that an algorithm consisting solely of mapping individual values explicitly to notes will not suffice if any reasonable level of quality is desired in the output, and that a deeper level of manipulation, taking influence from internal and external sources, is necessary. Meanwhile, it is important to keep in mind the historic reason for investigation of this kind, the pursuit of representing natural phenomena in musical form, in trying to balance algorithmic purity with quality of output as music. It would be very easy to allow external rules and restrictions for the manipulation of data to overpower the "creative" input of the chaotic systems in the pursuit of music of the highest possible quality, and undoubtedly a significant conflict will arise in monitoring the balance between these.

An important question to consider will be that of the level of human interaction. How much or how little control should the user have over the algorithms? An answer will come with experimentation into the qualitative potential of algorithms with little or no user-defined parameters.

The ultimate goal of this project is to find an optimal balance between all of these conflicting ideas, while above all, trying to produce good quality, representative music.

3 Requirements Specification

3.1 Introduction

3.1.1 Overview

Although there are two distinct streams of development involved in this project – algorithm development and algorithm implementation – it is very important that they are developed concurrently to optimise engineering efficiency. This section will aim first to lay out high level user requirements in both fields individually before combining them in more specific, in depth system requirements. These will outline the integration of the musical development and the system for its implementation, and specify the expectations and constraints of the system as a whole.

3.1.2 Terminology and Definitions

System – There may be some ambiguity throughout the document with this term, with it being used to describe at times the numerical equations which are central to the project, but more generally the piece of software being developed concurrently with the algorithms. The aim will be for the context to extinguish this ambiguity.

Shall/should – In the context of requirements, the term *shall* describes a compulsory requirement, while *should* refers to an optional, though preferable property, for example

The system should…

or

The system shall…

3.2 Musical Requirements

3.2.1 Performance Constraints

As previously specified, the music produced shall be for performance by solo pianist, and as a consequence, clear performance constraints are already in place, based on the design of a piano and the physical limitations of the pianist. Although over the years, experimentation in performance has led to a great variety and freedom of style and ornamentation, far beyond the expectations of the original creators of the instrument, the time and scope constraints of this project shall limit the variety of the output to a fairly standard set of rules.

The first constraints to be taken into account are those of the piano itself. The standard full size piano consists of 88 keys each tuned a semitone apart in the standard Western tonal scale.

A level of objectivity is introduced when considering the constraints of the performer, for which there are no strict universal rules in place. The particular abilities and attributes of the composer or the performer govern the boundaries of the performance, which may be decided by such seemingly trivial matters as the size of their hands or their choice of fingers for playing notes in a sequence or chord. With this in mind, the rules governing the performance constraints of the compositions of this system shall aim to be as generalised and realistic as possible without severe loss of creativity:

- The piece shall be split into two parts, one for each hand, as is the convention in piano music;
- Each hand shall not be required to span more than fifteen semitones to play two or more notes simultaneously;
- If three notes are to be played simultaneously by one hand, the second note shall be no more than ten semitones from the first and the third note shall be no more than seven semitones from the second;
- Each hand shall not be required to play more than five notes simultaneously.

Further performance constraints which are more difficult to quantify, such as intervals (both in pitch and in time) between successive notes and pace of movement, should also be taken into consideration when developing the compositional algorithms.

3.2.2 Quality Requirements

Previous work in this field has shown that a direct mapping from the numerical output of chaotic systems to musical elements, in particular individual notes, produces pieces with little to no discernable structure, and which are often unpalatable as music. In order to produce pieces of a reasonable quality, a set of basic structural requirements shall be adhered to.

The first restriction shall be on the key used in the composition. Given the introductory level of this project, the output shall be confined to the major and minor keys of Western tonal music. Occasional ventures outside the given key often produce more interesting results, and so this should be taken into account in the development of the algorithms.

The algorithms should include some notion of musical harmony and/or dissonance, as it is can be very important in creating a piece which will be of interest to listeners.

Potentially more important than the tonal structure of the piece is the dynamic structure. In order to retain some kind of discernable shape, the piece shall be subject to a standard time signature, as would traditionally be found in piano music.

The overall structure of a piece, along with its phrases and note sequences, gives it an identity and a sense of objective and direction, which is key if it is to spark any interest at all. The listener’s perception of an unstructured piece, however interesting the individual phrases or note sequences, is analogous, to some degree, to the point made at the beginning of this subsection. The algorithms shall construct, at each level of the piece, a discernable structure, although this should not be to such an extent that the creativity is hampered. More specifically, the use of repetition or near repetition, a key characteristic of the numerical systems under investigation, at any level immediately induces structure.

The length of the final piece shall be between two and eight minutes. Ideally, this should not cut down or prolong compositions, but shall be in place as a guideline when approaching the algorithms.

3.3 User Requirements

With the automated production of music a prime objective of this project, user interaction shall be minimal.

The system shall contain a basic user interface for the definition of numerical systems through its equations and parameters. A window based interface would be preferable, though a text input alternative would be acceptable.

The system shall be encapsulated in one program, so that a single click or command shall produce the desired results with no further user interaction.

The music produced should reflect as well as possible the chaotic systems on which it is based, as outlined in chapter 2. The interpretation and measurement of the success of this aspect need not be strict and well-defined, but instead should reflect the creativity involved in the development.

The music produced shall be available in any format for which LilyPond has capabilities (MIDI, PDF, PS, DVI and TEX), which include portable audio and notated representations of the music. The decision on which representation(s), if any, should be utilised shall be that of the user, and all formats produced shall have their filenames defined by the user-entered title of the piece.

3.4 System Architecture

The system shall consist of three main processes:

1. A graphical user interface for the declaration of chaotic systems and output parameters (written in Visual Basic);

2. An implementation of the compositional algorithms combined with a conversion to LilyPond source format (written in C);
3. LilyPond processing of the composition (if required).

Ideally, the system should be contained in a single entity written using a combination of C and C++, consisting of a number of different objects or processes, but limited programming knowledge and experience, and the use of an external program such as LilyPond make this impossible within the time constraints. Instead, the final system as a whole should be taken as a prototype, with the first and third processes as temporary solutions to the problem of user-interaction.

3.5 System Requirements

The user interface shall allow the selection of a template for a predefined dynamical system, prompting for parameters and initial data, and should incorporate a means for the user to define custom systems. Continuous systems shall be rewritten as discrete systems, using Euler's method with user-defined time step.

The numerical data generated by the dynamical system shall be processed in a clear and consistent way by a number of manipulative algorithms. These algorithms should organise the data so that it may be converted into a musical score object. The algorithms involved shall attempt to preserve the shape and identity of the original data while manipulating it in a useful way.

As previously mentioned, the concurrent development of the compositional algorithms and the system for their implementation is very important in ensuring efficient results both in the development process and in the final program. With this in mind, the primary objective shall be the implementation of a method for generation of LilyPond source code. The system shall take as input numerical data from the compositional algorithms and shall interpret it as a musical score. Each note, bar, and the score itself shall be stored as a compound data type with all necessary information, and the score shall be in a format which can easily be printed to a LilyPond source file.

Further appropriate manipulation should be applied to the data once in the form of a score, in order to improve the fluidity of the music and give the score a more discernable structure.

Once composed, the score, bar and note objects shall be interpreted as music in conjunction with the established LilyPond format and written to an external text file with extension *ly*.

The LilyPond program shall be called, if either sheet music or audio output is required, on the LilyPond source file generated. All output from a single composition (source, sheet music, audio and log files) shall be collected into a directory with the user-defined name of the composition.

4 Algorithm Development

4.1 Preliminaries

Before the algorithmic development, some computational formalities should be addressed. The numerical systems with which this project will deal produce theoretically infinite orbits, i.e. those that continue forever. Computationally this is clearly impractical and requires some finite interpretation, which to begin with, shall take the form of the first sufficiently many iterations. A balance must be found in this number, between processing time and freedom for manipulation and sampling, and shall be subject to experimentation in the implementation side of the project.

4.2 Simple Numerical Manipulations

4.2.1 Direct Mapping

The obvious place to begin the development of the algorithms for this project would seem to be a direct mapping from the numerical values of a one-dimensional iterated chaotic system to the keys of a piano, progressing in equal time intervals, as with the dynamical system. For generalised computation, this first requires the orbit to be scaled down to the interval [0,1]. This function should then be multiplied by 87 and rounded to the nearest integer. This will give a new orbit of the same cardinality but now with integer elements in the range [0,87] (corresponding to the 88 keys of a piano). This orbit can be interpreted as a sequence of notes, each of the same duration, for example a quaver.

The music produced by this mapping has one immediate flaw, ignoring for now the issue of quality - any pair of consecutive notes could be up to 87 semitones apart, making the piece potentially unplayable. To combat this temporarily, the orbit shall be restricted to a smaller set of notes in the middle of the piano, perhaps two octaves below and two above middle C.

4.2.2 Orbit Division

The sub-behaviour of a number of chaotic orbits analysed seems to be, to some extent, polarised (see fig. 4.1, 4.2). In an attempt to reflect this in the music, the "top half" of the orbit shall be mapped to right hand, and the "bottom half" mapped to the left hand. In order to achieve this, the median value of the orbit shall be found, and two new orbits of half the size of the original shall be created. All values greater than the median shall be added to one orbit (with original order preserved), and all those lesser added to the second. These two new sets should be normalised and scaled up in the same way as before, and will produce two sequences of notes which can be played concurrently, one by each hand.

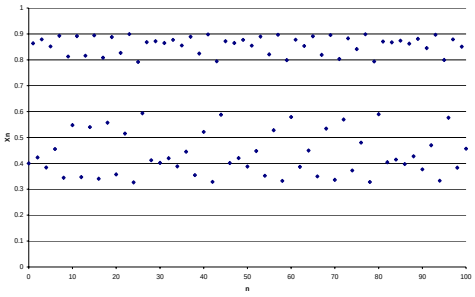

Fig. 4.1 First 100 iterations of the Logistic map with parameter $a = 3.6$ and initial data $x_0 = 0.4$.

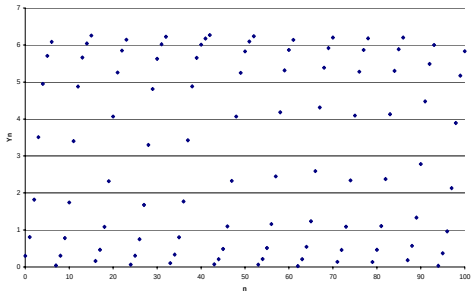

Fig. 4.2 First 100 iterations (y_n) of the Standard map with parameter $k = 0.7$ and initial data $(x_0, y_0) = (0.3, 0.3)$.

4.2.3 Note Duration

With structure still a distant goal, the piece at this stage would benefit from the injection of some level of dynamic variety. As many popular chaotic systems are in at least two dimensions, this would seem an opportune moment to introduce a second dimension. Much in the same way as with pitch, the second dimension of the numerical system (if available) shall be normalised independently of the first dimension and scaled up to a musical time interval. In order to retain some level of structure (although potentially still unrecognisable to the human ear), the time intervals shall be the fixed single standard musical durations *semibreve*, *minim*, *crotchet*, *quaver* and *semiquaver*.

4.2.4 Key Signature

The idea of tonal structure shall be addressed for the first time in the form of the key signature. Although this may be subject to review at a later point, the choice of key signature shall, at this early stage, be an effectively arbitrary process. It shall simply be a case of mapping the normalised initial data (in the first dimension) to a note in the chromatic scale. For simplicity at this stage, only major scales shall be considered. The orbit that would previously have been mapped to the chromatic scale shall now be mapped to the appropriate major scale in the same way.

4.3 Event Trees

4.3.1 Background

Although up to this point, the behaviour of the chaotic systems has been preserved as purely as possible, the musicality aspect has been neglected. The concept of *event trees* and their generation from numerical orbits as described by Leach (1995) shall form the basis of work which shall attempt to form musical elements based on the shape and form of chaotic orbits. Leach proposed that an event tree be built recursively on subsets consisting of local maxima. From the original orbit, all local maxima (i.e. elements greater than both their predecessor and successor) should be collected into an orbit which should then be subject to the same dissection. This should be repeated until an orbit containing only one element remains, and then each element of the original orbit given a value as an event corresponding to "how far" it progressed (*0* for original orbit, *1* for second orbit, etc.). This would form an event tree corresponding to the numerical orbit, which would demonstrate its behaviour on some level in a succinct manner. Moved on a stage, this process can be repeated for local minima to generate a second event tree representing another aspect of the system's behaviour.

4.3.2 Interpretation

The issue of how to interpret these trees musically is a very subjective one, and shall be investigated as the necessity for structure increases. One idea proposed is that the more important an event (the greater its value), the longer its

corresponding note should be. On a purely experimental level (as is consistent throughout this development), the second dimension shall, for now, be discarded in order to allow the first dimension to dominate. The original orbit shall be mapped to both the treble and the bass clef separately, with the maxima and minima event trees dictating the note durations respectively.

Although, on first listen, it would appear that some of the most interesting music so far is produced by this method, there is actually very little variety, particularly in terms of the sequences of notes used. An approach which may add some tonal structure and fluidity to the composition could be to completely ignore, for the moment, the actual values of the original orbit. Instead, the event tree in the first dimension could be used to generate pitches, while that in the second dimension (if applicable) could deal with note durations. To begin with, the importance values of each event shall be mapped independently to pitches and durations, and although this will not produce music of any interest, it will give some idea as to how useful event trees may be as the development progresses.

4.4 Harmony

Despite the presence of polyphony in the work produced so far, harmony has been neglected. The importance values from the event tree in the first dimension shall be used to rectify this, dictating the number of notes which should comprise each chord. The pitch of the notes shall be chosen by the numerical value as before, and non-zero event importance values shall group together sufficiently many subsequent notes to fill a chord.

This approach does not solve the problem of harmony, but demonstrates that the structure of the orbit should not be disregarded so recklessly. Instead, each element of the orbit shall be mapped to a single note as before, and those with non-zero event importance values shall be extended to chords. Unfortunately, this causes similar problems to the previous idea, in that rather than chords and harmony resulting from multiple simultaneous notes, the lack of interaction between distinct notes produces disjointed, erratic note progression.

4.5 Selective Performance

In an attempt to find a balance between monophonic fluidity and harmonic depth, a completely new approach shall be formulated. One unit of duration (for example of quaver) shall be allocated to each event (in the first dimension), and all events with zero importance values shall be interpreted as rests, unless otherwise specified. When a non-zero event is encountered, a number of notes corresponding to its importance shall be performed, starting with the numerical value from that event, followed by the values from sufficiently many subsequent events. If amongst these subsequent events, one has non-zero importance, an extra note shall be added one fifth higher and a second, harmonic count started from the value of the event's importance. For example:

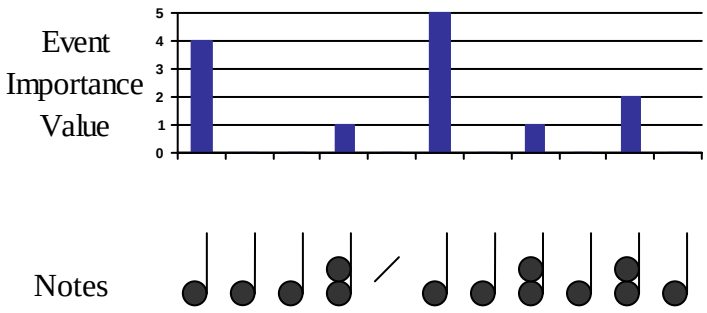

Fig. 4.3 A segment of an event tree and the dynamic and harmonic shape of the corresponding note sequence.

An obvious extension of this idea would be the introduction of dynamic variety. The event, or indeed the numerical value, in the second dimension may be used to determine the length of each note, although zero-importance events should remain short to reflect their nature. The allocation of one unit of duration to each event shall remain in place to preserve the dynamic reflection of the system, which means that some notes may overlap. In this situation, non-zero events shall take priority, and any zero events in their path will be engulfed and disregarded. Any non-zero events which would otherwise be engulfed shall add to the polyphony of the note.

4.6 Bar-by-Bar Composition

4.6.1 Significance of Events

The quest to preserve the dynamic representation of the chaotic systems has so far stood in the way of the development of dynamic structure. Notes are simply played successively until the end of the orbit is reached, and the bars are effectively transparent. In a first experimental step towards rectifying this, a non-zero event shall signal the start of a new bar, and a single note in the bass clef shall represent this event while the successive notes corresponding to zero events are preformed in the treble clef. Note durations shall, again, be taken from the values or importance values of events in the second dimension, while pitches shall be derived from values in the first dimension.

A small comment on the quality on the music produced using this method would be the nature of the beginning of many of the bars. As the prompt to begin a new bar is taken from a non-zero event in the first dimension (representing a locally maximal value) while the pitches of the notes are taken from the values themselves, the bars in all pieces begin high and then fall until the next new bar. Where available, non-zero events from the second dimension shall take over this duty, leading to a greater tonal variety. This does, of course, mean that each bar is likely to begin with its longest note, but this has a much more natural feel than before.

4.6.2 Structured Bass

The additional structure introduced by this approach is a distinct improvement, and does not seem to interfere with the element of near repetition, but is of little interest in terms of note sequences. The lack of movement in the bass makes the piece very stale and unpalatable after a few seconds. The question of balance between preserving the behaviour of the chaotic system and generating palatable music arises here: Could the introduction of arpeggio or a walking bass line, built around the note generated by the non-zero event, be considered excessive external manipulation? On one hand it would reduce the variety and creativity between distinct chaotic systems, but it would also add a sense of direction and chord progression.

The extension of a single note for each bar in the bass line to a simple arpeggio (e.g. tonic, dominant, next tonic, dominant) does add a much needed sense of direction and, most importantly, musicality by maintaining the pace of the piece and outlining naturally occurring phrases. It does, however, give the piece a very cold, artificial feel, and extinguish almost all discernable variety between compositions. Thus it is clear what is required – the left hand should be more structured than the right, perhaps to the extent where no dynamic freedom is allowed and only pitches may vary, but should at the same time take a reasonable degree of influence from the chaotic orbit. A very simple, but potentially effective solution may be to fix pitch intervals and allow values to determine the direction of progression. If the next value is within ±0.1 of the current value, then the note shall be extended. If it is above this interval, the pitch shall be raised by seven semitones, while if it is below, the pitch shall be lowered by five semitones. Of course these numbers are purely experimental, and simply reflect the fact that six of every seven pitch shifts should stay within the major scale.

4.7 Time Signature

Now that the score has a proper bar-by-bar structure, it would seem an opportune moment to develop time signatures. As with key signature, the ideal scenario would be for the time signature to reflect the development of the dynamical system over time by changing as the orbit diverges significantly. To begin with, however, the number of iterations between the first two non-zero events shall be taken as the number of crotchets per bar. This will reflect the shape of the beginning of the orbit, approximately preserving the time between important events (~one crotchet per iteration).

4.8 Final Algorithm

Although the development of the algorithms is far from what could be considered complete, the set which, at its current level, has produced the most preferable results, is as follows:

1. The user-defined map is iterated to produce an orbit.
2. The first 100 iterations are discarded.
3. Each dimension of the orbit is independently normalised.
4. The key signature is chosen by multiplying the first element of the first dimension of the normalised orbit by eleven and rounding to the nearest integer.
5. A ‘maxima’ event tree is generated from each dimension of the normalised orbit.
6. The time signature is chosen by inspecting the number of iterations between the first two non-zero events in the first (or second if available) dimension, and scaling down if necessary.
7. Points of near repetition in the orbit are found.
8. Each element of the orbit before the first point of near repetition is converted to a note in the treble clef:
  8.1. The value in the first dimension is scaled up to the appropriate piano key via a standard transformation (see section 5.5.2).
  8.2. The value in the second dimension is scaled up in a similar way, but to three octaves lower. If it is not available, the note is obtained by taking 36 from the first note (8.1).
  8.3. The duration of the note is chosen by taking the value in the second dimension away from four and raising it as a power of two. If no second dimension is available, the duration is set to four (a crotchet).
  8.4. If the event in the second (or first, if one-dimensional) dimension has non-zero importance:
    8.4.1. Any ‘open’ bar is completed with rests and a new bar is started.
    8.4.2. The first pitch (8.1) and the duration (8.3) are used to generate the first note in the treble clef of new bar (the duration is scaled down if it would otherwise not fit).
    8.4.3. The second pitch (8.2) is used to generate the first note of the bass clef with duration four (a crotchet).
    8.4.4. The bass clef is then filled, note by note, by comparing the value in the second dimension of the subsequent element with that of the current element (see section 4.6.2). The bar is filled by a single note, or set of tied notes, if the orbit is one-dimensional.
  8.5. If the event has zero importance:
    8.5.1. The note generated in 8.4.2 is added to the ‘open’ bar (after any necessary scaling down to fit).

8.5.2. If it is the first note of a new bar, the bass clef of the previous bar is repeated.

8.6. If the bar is full in the treble clef, the pair of bars is stored.

9. If less than 128 bars have been stored, step 8 is repeated for elements between the next pair of near repetition points.

5 Implementation

5.1 Overview

With the design of the user interface heavily dependent on the layout of the main compositional system, and LilyPond a pre-written external program, the first objective shall be the design of a system for the concurrent implementation of the compositional algorithms. Once this is established, constraints on prioritising shall be loosened, and there will be freedom to develop the user interface simultaneously. Due to limited programming experience and time constraints, the platform on which the system will be implemented shall be of less importance than perhaps would be preferable. Instead, the system shall be developed using languages and tools in my repertoire, with its physical implementation to be decided accordingly.

5.2 Representation of Score

5.2.1 Introduction

In order to simplify the conversion of a score generated by the compositional algorithms to LilyPond source code, its abstract representation shall be as clear and as well defined as possible. Each note shall be stored with information on pitch, duration, and any other necessary attributes that may become apparent during the development, all in LilyPond format. Likewise, each bar shall hold information on the notes it contains, as well as anything else concerned with the nature of the bar. The score itself will require the greatest quantity of information, with details of key and time signatures and detailed structure, on top of any other attributes that may arise.

5.2.2 Note

In LilyPond, a solitary note is represented by its ‘letter’:

- c

- `d`
- ...

followed by accidentals:

- sharp = `is`
- flat = `es`

followed by the octave:

- 4 octaves below middle C = `,,,`
- 3 octaves below middle C = `,,`
- 2 octaves below middle C = `,`
- octave below middle C =
- octave above middle C = `'`
- 2 octaves above middle C = `''`
- ...

followed by the duration:

- semibreve = `1`
- minim = `2`
- crotchet = `4`
- quaver = `8`
- …
- dotted semibreve = `1.`
- double-dotted semibreve = `1..`
- …

So, for example, a dotted crotchet of pitch A flat above middle C would be written

```
aes'4.
```

These attributes all need representing in the structure of the data type *Note*, and for simplicity of conversion shall all be of the form necessary for LilyPond. The note itself (letter with accidentals) shall be stored in a *String* as above, as shall the octave, while to begin with the duration shall be represented as an *int*. Although this is not ideal for conversion, it will simplify any necessary manipulation, and shall be reviewed as the development progresses. Any additional requirements on the data type shall be incorporated appropriately as they arise.

```
struct Note {
     char *note
     char *octave
     int duration
}
```

5.2.3 Bar

The next logical step up from the *Note* type, the *Bar* object shall be built around the set of notes it contains. In LilyPond, notes are written one after another and separated by white space, meaning that little extra work must be done by the *Bar* itself. It shall, to begin with, consist of simply an ordered array of *Notes*, but is likely to be extended later in the development to include extra attributes associated with an individual bar. To begin with, the main purpose of this object is organisation of notes.

```
struct Bar {
     Note **notes
}
```

5.2.4 Score

A structured score in LilyPond is a far more complicated entity than the note, and requires far more interaction and decision. The restriction of this project to piano music helps to cut down the quantity of information which must be stored in the data type *Score*, eliminating parameters such as instrumentation, timbre and stave layout. The LilyPond syntax for a generic piano score is not important at this stage, and will be incorporated later when creating the LilyPond source code, however any details specific to a particular score, such as time signature, key signature and the set of notes (or bars) which comprise the score shall be held by the *Score* object. As a piano score, two ordered arrays of bars shall be stored (one for each staff).

```
struct Score {
     Bar **right
     Bar **left
```

```
    int *time
    char *key
}
```

5.3 Numerical Manipulation

A number of auxiliary tools are employed in the process of composition, all of which manipulate numerical data independently of the algorithms and the Score structure used.

5.3.1 Generating and Sampling Orbits

There is a clear, well defined method for generating chaotic orbits already in place for discrete systems, which simply involves iterating the given map on the given initial data, recording the output at each stage until an orbit of sufficient size has been generated. As previously described, continuous systems can be solved and approximated as discrete systems using Euler's method, and so shall be manipulated in this way given a user-defined value for time increment Δt .

For the moment, the program will only be concerned with a set of predefined dynamical systems, all incorporating parameters to be defined by the user. Each map shall be implemented as a function, which will be iterated to generate an orbit.

In the case of continuous systems, some sampling may be beneficial in order to maintain a level of consistency between the orbits. A discrete approximation will typically move and vary slowly, while a discrete system will usually "jump about" (see fig. 5.1, 5.2).

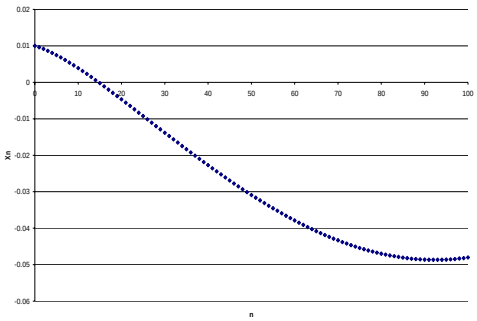

Fig. 5.1 First 100 iterations (x_n) of the Rossler Map with parameters $a = 0.2$, $b = 0.2$, $c = 5.7$, time step $\Delta t = 0.02$ and initial data $(x_0, y_0, z_0) = (0.01, 0.01, 0.01)$.

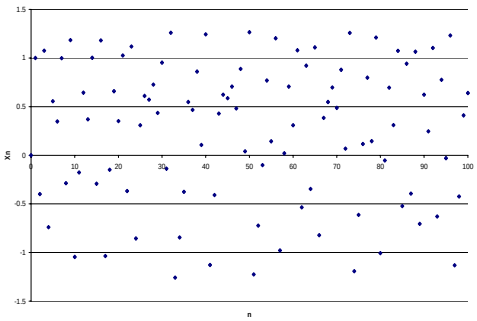

Fig. 5.2 First 100 iterations (x_n) of the Hénon map with parameters $a = 1.4$, $b = 0.3$ and initial data $(x_0, y_0) = (0, 0)$.

A possibility would be to take every k^{th} element from the orbit, and form a new orbit which to some extent retains the shape of the original but which moves around in a more interesting way.

Another small addition which may help to eliminate any later problems would be to discard the first few iterations, say 100. Many systems do not immediately settle into their near repetitive patterns of behaviour, such as the *Lorenz* map in the third dimension with initial data $z_0 = 0$ (Fig. 5.3). This never returns to a point near its initial point and so would never display near repetition of this point.

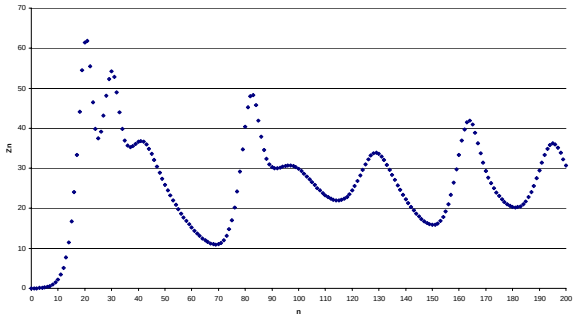

Fig. 5.3 First 200 iterations (z_n) of the Lorenz Map with parameters $\sigma = 10$, $b = 8/3$, $r = 28$, time step $\Delta t = 0.02$ and initial data $(x_0, y_0, z_0) = (1, 1, 0)$.

5.3.2 Normalising

A function which will scale an arbitrary finite orbit down to the interval [0,1] is required. The supremum of the orbit should be mapped to 1, while the infimum should be mapped to 0, and each value in between mapped proportionally. For a one-dimensional orbit A, the injective map

$$x_{\max} := \sup A$$
$$x_{\min} := \inf A$$
$$f : \mathbb{R} \to [0,1]$$
$$f(x) = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

will produce the desired result. This map then needs to be iterated for each dimension of the given orbit. The only missing component is finding the minimal and maximal elements of the orbit, which is a simple task involving checking each element progressively against the greatest (or least) previous element.

5.3.3 Dividing

Another simple function which takes an orbit and divides it in half, this employs a simple sorting algorithm to the first dimension (specifically, insertion sort), finds the median element of the sorted orbit and splits it in half. Each half is stored in a new array of half the length of the original orbit.

5.3.4 Finding Repetition

The ambiguous nature of this task means that some level of external influence is required. Since, by definition, chaotic orbits never repeat, a decision on what can be considered ‘near repetition’ is important. Given a finite orbit, the user (or process) must decide how many repetitions are required, upon which an algorithm will be repeated until sufficiently many are found.

The algorithm will consider the first point of the orbit, x_0, and will search progressively until another point, say x_k, is found in the neighbourhood

$(x_0 - \varepsilon, x_0 + \varepsilon)$ for some small $\varepsilon > 0$ (or the appropriate multidimensional extension). The search will then be continued for a point in the neighbourhood $(x_k - \varepsilon, x_k + \varepsilon)$. This is repeated until either sufficiently many 'repetitions' have been found, or the end of the orbit is reached. If the latter is the case, the algorithm is repeated, slowly increasing ε, until enough points are found.

5.4 Event Trees

5.4.1 Overview

As described in the corresponding section of chapter 4, event trees characterise one-dimensional orbits by representing local and global maxima (or minima). For the purposes of this project, the trees will be built in a slightly simpler way to that proposed by Leach (1995). The events which comprise these trees will not be connected and will stand independent of one another.

5.4.2 Event

An *Event* is a very simple object, which requires only the value it represents and its importance as part of a tree.

```
struct Event {
     double value
     int importance
}
```

5.4.3 Event Tree

An *Event Tree* is effectively just an ordered array of *Events*, the size of which corresponds to that of the orbit from which it is created. When a tree is initiated with an orbit of size n, n *Events* are created with values taken from the orbit and importance values set to zero. The tree is then grown (up or down, corresponding to maxima or minima), with direction specified in initiation.

```
EventTree {
     Event **tree
     int size
     int direction
}
```

5.4.4 Growth

The recursive algorithm described in chapter 4 has a very simple implementation. The value of each event is checked against that of its predecessor and successor, and if it is greater (or lesser) than both the importance value is incremented and the event is added to a new array. When the end of the tree is reached, the algorithm is repeated for the new array until the size of the array is reduced to 1.

5.4.5 Multidimensional Trees

There is no definition for multidimensional growth of event trees, and so there shall be a separate tree for each dimension, each of which shall be grown independently. This also requires the additional field of dimension (an integer) to the definition of an *Event Tree*.

5.5 Composition

5.5.1 Overview

Over the process of this project, the ideas involved in the algorithm development have advanced and undergone a number of dramatic changes. To reflect this development, this section will document the implementation at each stage and will demonstrate the significant modifications made in the system.

5.5.2 Development

The first venture in the composition development is a direct mapping from the normalised orbit to a sequence of notes. The orbit is scaled up to the interval [0,87] and these values are then passed through the function `convert(double **input, Bar **output)`. This function takes an orbit of arbitrary size and

dimension and converts each element to a note. It inspects the value in the first dimension (between 0 and 87) and decides which note and in which octave this value corresponds to on a piano. It uses a *switch* statement on the value under integer division by 12 and modulo 12, to select the octave and the note respectively.

An issue arises here in setting the name of the note for accidentals. Depending on the key signature, an F sharp may be a G flat, for example, and this distinction is important when printing the note as LilyPond source for notated output. Though for now, with no key signature in place, this is unimportant, it will need to be dealt with in the near future.

Assuming that the *note* and *octave* have been set in the *Note* object, the *duration* is set to 8 (representing a quaver) and the *Note* is added to an array. The process is repeated for each element of the orbit until 8 notes have been created (meaning that a *Bar* can be filled, assuming the time signature is set to a default $\frac{4}{4}$). At this point a *Bar* object is initiated using a copy of the array of notes generated so far and added to an array. The *Note* array is then emptied and the process begins again, and is repeated until the end of the orbit is reached. The array of *Bars* is copied to *right* component of the *Score*.

The implementation of the restriction to the middle of the piano is very simple. The orbit is scaled again from the interval [0,87] to the interval [15,63] (or similar) and the above algorithm implemented.

Splitting the orbit in half makes use of the `divide(double **input, double **top, double **bottom)` function, which implements the algorithm for division described in section 5.2.3, splitting the orbit according to values in the first dimension. These are normalised and scaled as before and each passed to the `convert` function. The array of *Bars* resulting from the top half is set as the *right* component of the *Score*, and that from the bottom half as the *left* component.

The use of pointers and dynamic memory, although extremely useful and efficient, has caused a number of problems up to this point. Each function, to minimise the number of parameters being passed around, has calculated the size

and dimension of the orbit (and related arrays) using *while* loops on the elements of the arrays. If *NULL* (or zero) is returned when the elements are called, it is assumed that the end of the array has been reached and so the size has been calculated. It has become evident that this method is unstable and that it would be far simpler to pass and store details such *size* and *dim* throughout the program. As such, a *size* element shall be added to the *Score* and *Bar* objects to keep track of the number of *Bars* and *Notes* respectively. The *size* and *dim* of an orbit will also be passed to any function which will manipulate it.

The mapping of the second dimension (if available) to the duration of the note shall be included in the conversion algorithm. The normalised values (in the interval [0,1]) will be scaled up to the interval [0,4] and rounded to the nearest integer. This value will then be taken away from 4 and then raised as a power of 2, to give a *duration* value from the set $\{1,2,4,8,16\}$. This will mean that the greater the original value, the longer the corresponding note.

In restricting the output to a specific major scale, a number of numerical manipulations are necessary. The key signature of the score is chosen in the same way as a note in the conversion algorithm, but the mapping from the normalised orbit to a specific subset of keys is a more complicated procedure. The orbit is first scaled up to the interval [0,28] and rounded, now representing four octaves of an arbitrary major scale. These values are further scaled up, depending on which note of the scale the represent (tonic, super-tonic, etc.) to the interval [0,48], filling the corresponding values of the chromatic scale. An integer is then added in the interval [0,11] depending on the key (0 for C, 1 for C sharp, etc.), and finally the digits are interpreted as notes and octaves.

It is apparent that this attempt to manipulate and interpret values simultaneously and at such an early stage presents problems. A simpler approach, and one which shall be employed for the remainder of this project, shall be to store the notes as integers representing the actual keys on a piano, and to interpret them as late as possible. This also means that a *flat* component needs to be stored in the *Note* object, to tell the interpreting function whether to use sharps or flats.

```
struct Note {
     int note
     int duration
```

```
    int flat
}
```

Using the implementation of event trees as described in section 5.4, it is a fairly simple process to map a tree representing the maxima of the first dimension to the treble clef and one for the minima to the bass clef. The numerical values to be mapped to pitch (as before, except scaled to stay within two octaves above or below middle C respectively) are stored in the *Event* objects, with the importance values which are to be mapped to duration in the same exponential way as before (truncating those greater than 4).

The next step of replacing the numerical values in the first dimension with the importance values for pitch, and introducing the importance values of the second dimension for duration is again a simple one. The set of importance values in the first dimension shall be scaled up to the major scale of the score, meaning that zero events are mapped to the tonic and non-zero events are mapped in single steps upwards (super-tonic for one, mediant for two, etc.).

The introduction of chords poses a problem in the representation of the score. There is no mechanism in place to tell the printing algorithm (see section 5.6) to group notes together as chords. The LilyPond representation (see fig. 5.4) encourages the presence of polyphony to be stored as an element of the *Note* objects it involves. A simple solution would be to include an extra integer in the definition of the *Note* object, *chord*, which will be set by default to zero, but incremented to dictate the number of successive notes with which it will share a chord.

```
<< c'4 e'4 g'4 >>
```

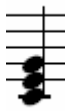

Fig. 5.4 The LilyPond syntax for a chord and the resulting graphical interpretation.

The *chord* values of the *Notes* generated from the non-zero events shall be set to one lower than the corresponding importance values and everything else applied as before. The issue of how to interpret this chord value for LilyPond source code shall be dealt with later (see section 5.4).

The first mention of rests appears here, and it is clear that a method of storing them as part of the *Bars* they inhabit is required. The obvious solution is to consider them as notes (as LilyPond does), and so store them as *Note* objects, with a zero note value. This means that all other notes now need to be incremented to the interval [1,88].

The algorithm which chooses notes for performance according to preceding event values requires simply the addition of a number of variables to remember how many notes should be played, etc., and a *switch* (or nested *if-else*) statement to decipher these variables on each iteration of the conversion loop.

Focus is, for the first time, placed on the individual bars which comprise the score in the algorithm which maps each non-zero event to a new bar. Until now, the set of bars for the right hand and that for the left hand have been completely independent and it has been left down to luck for them to relate to each other in any way. With bar-by-bar structure in mind, it seems an important step to bring the *Bar* arrays together and to fill each pair of bars simultaneously. This requires the grouping of the two *Bar* arrays into one two-dimensional array, which will also ensure that both arrays are of equal length.

The proposed algorithm itself calls for a simple analysis of each event and a series of *if* statements to determine the resulting behaviour. Additional information, such as note pitch and duration, can be derived using previously developed algorithms and then inserted where necessary. When a bar is almost full and the note generated from the next event has a duration greater than there is space for, it would be preferable for the note to be split and ties across the bar line. This, however, would require a great deal more programming, and so, for now, the note shall be scaled down to fit.

The issue of rests in notated form is, although a minor problem, something which could benefit from some extra attention. Until now, bars which have required padding with rests have simply had sufficiently many quaver-rests inserted. This may have the required audible effect, but results in untidy sheet music. Instead, the algorithm shall create an array of rests, starting with the longest and gradually scaling them down to fit. This array will then be reversed and appended to the set of notes in place for that bar.

The algorithm for the development of structured dynamic progression in the bass clef requires that notes be extended to include an extra crotchet at a time. Although this is very simple to begin with (simply a case of converting a crotchet to a minim), it quickly becomes complicated, with notes of strange lengths and varies time signatures. Instead, the notes shall simply be tied together to represent notes of a greater length, requiring the addition of a *tie* component in the *Note* object. This will simply be true or false (1 or 0 in *C*) depending on whether that particular note shall be tied to the following note. Although, as previously with the rests, this will result in a slightly untidy score, it will greatly reduce the development time from what precious little time is available.

5.6 LilyPond Source Generation

With the structure of the score in fairly good condition after composition, relatively little needs to be done to convert the data into LilyPond source code. There are three basic functions which must be defined for this purpose, printing *Note*, *Bar* and *Score* objects to a file. The `printLilyPond(char *composition, Score *s, int sheet, int tempo)` function shall perform the formalities of creating a *FILE* pointer and opening a file given a user-defined filename, with extension '*.ly*'.

The key signature of the composition, stored as an integer as part of the *Score* object, is passed through a simple *switch* statement to convert it to the appropriate *String*. The following is then printed to the file:

```
\version "2.4.3"
\score {
     \new PianoStaff <<
          \new Staff {
               \key [key signature] \major
               \time [time[0]]/[time[1]]
```

where *time[0]* and *time[1]* refer to the first and second integers stored as the time signature in the *Score*.

The next step is to print the bars for the treble clef. This requires the function `Bar_toString(Bar *b, char *s)`, which is called *n* times for a *Score* with size parameter *n*. This function checks whether or not the notes in each bar are played simultaneously (the *chord* element of the note is non-zero), and then prints to notes one by one to a *String* using `Note_toString(Note *n, char *s)`, incorporating "<<" and ">>" when required for polyphony.

The `Note_toString` function first converts the numerical representation of the pitch of the note to *note* and *octave* strings. If the pitch value is zero then the note is interpreted as a rest, otherwise the numbers are manipulated to return the corresponding key on the piano. If the note has a tie to the next note, "~" is added to the end of the *String* representation.

The following is then printed to the file:

```
            \bar "|."
        }
        \new Staff {
            \clef bass
            \key [key signature] \major
```

before the loop to print bars is repeated for the bass clef. Once this has finished, the file is completed with details of the output formats, depending on the original choice of the user:

```
            \bar "|."
        }
    >>
    \midi {\tempo 4=110}
    \layout { }
}
```

5.7 User Interface

The use of a high level language such as Visual Basic for the design of the user interface greatly simplifies its implementation and allows for far quicker development. The graphical design of the interface is very basic, and simply involves:

- radio buttons to choose from the templates (numerical systems);
- labels and diagrams presenting the systems themselves;
- text boxes for the specification of initial data and parameters;
- check boxes for the choice of sheet music and/or MIDI output;
- a text box for the title of the composition;
- command buttons to compose the piece or to quit.

When a radio button is clicked, the corresponding text fields are enabled and all others are disabled. This allows the user to only enter the appropriate information. When the *Compose* button is clicked, all appropriate text fields are checked for completion and a warning displayed if any are empty. If all have been completed, a final warning is displayed asking the user whether they wish to continue, after which the fields entered are written to file and the program is closed. The transferral of the information is covered in more detail in the next section.

A problem is encountered in trying to satisfy the requirement for the system to accept any discrete chaotic system as input. The ambiguity and sheer range of functions which could comprise a dynamical system would create a great deal more work to transfer information across and verify that it is interpreted correctly. As a result, the user shall have a choice between a finite number of pre-defined systems with parameters.

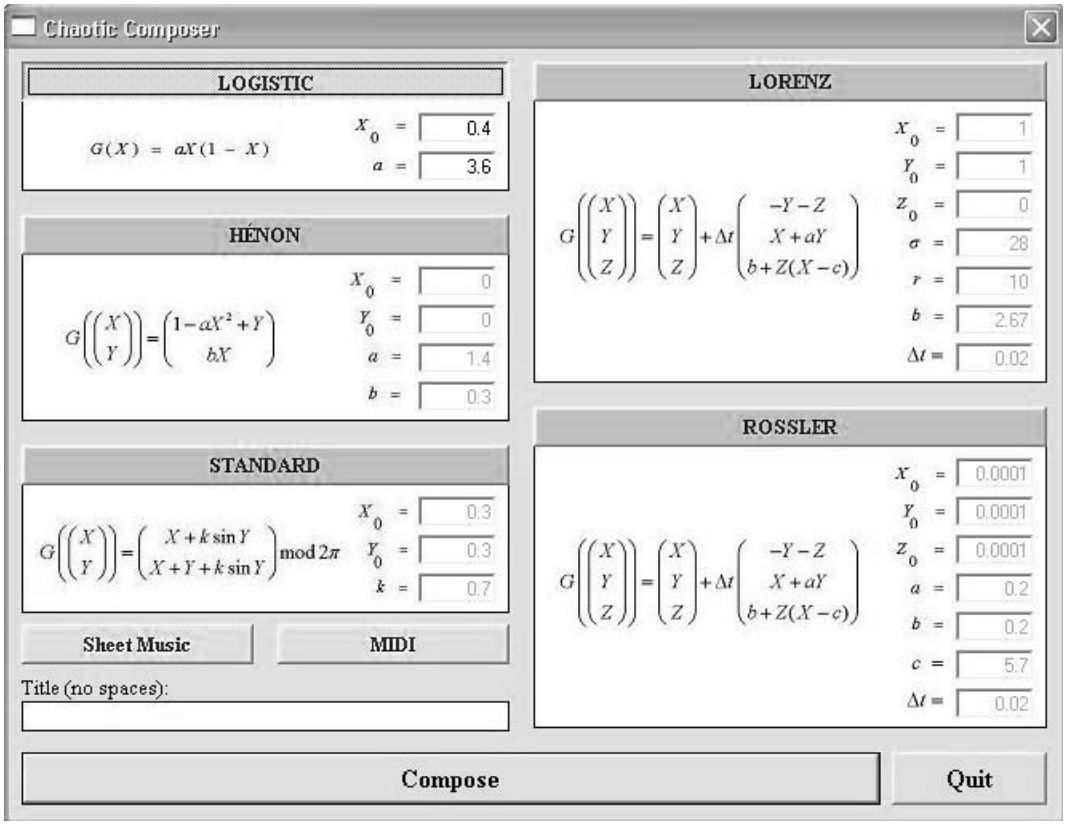

Fig. 5.4 A screenshot of the user interface.

5.8 System Integration

The variety in the software used for the system's implementation calls for special attention to the integration of the various parts. It is required that only one program must be run by the user and that all other processes be called internally. The simplest and most effective method of integration, given the circumstances, would be a single executable file written in C which calls each external process using the `System(char* s)` function. This prints the *String* `s` to the UNIX command line while the program is running and then waits for any external processes to end before continuing.

This raises an important issue, one which has been ignored up until now. The use of a Windows based language such as Visual Basic in conjunction with UNIX based LilyPond causes a platform conflict. The temporary solution comes in the

form of Windows based UNIX emulator Cygwin. This program is distributed free of charge from http://www.cygwin.com/, and also includes a GNU C compiler for programs which are to be run on this platform. It is also the platform used by LilyPond for its Windows version and so shall serve as a very useful tool

Now that the user interface can be called by the C executable, the issue arises of how to transfer data between the two distinct processes. The solution is to make use of a mutual format of communication, a simple text file. The user interface will create the file ‘info.txt’ and write to it any relevant data. The first item is a flag, and will tell the program whether or not it needs to continue. Next the title of the composition and details on sheet music and MIDI output is written, so that a directory can be created in the name of the piece. If this directory already exists, the user is prompted to choose whether or not to overwrite and previous work of the same name.

Assuming everything up to this point is satisfactory, an integer referring the template chosen is written to the file, followed by initial conditions and parameters. A *switch* statement decides which map to iterate and the other values are read in from the file and passed to the corresponding function as parameters.

An empty *Score* is initiated and the `transcribe` function called with this *Score* and the orbit generated by the iterated system (with the first 100 iterations discarded) as parameters. When this is complete, the `printLilyPond` function creates a source file, given the user defined parameters of *sheet*, *MIDI*, and *title*.

If either *sheet* or *MIDI* are non-zero (i.e. true), LilyPond is called using again the `System` function. This then generates the appropriate output formats. Following this, ‘info.txt’ is closed and destroyed.

6 Validation

6.1 Requirements Satisfaction

The requirements laid out in chapter 3 shall act as a reference for verifying that the final system performs all the necessary tasks satisfactorily. The subjective nature of this project means that many of the requirements are not immediately measurable (in particular, quality of music), and as such shall only be touched on in this section. Critical evaluation of these aspects shall feature in the subsequent section.

6.1.1 Musical Requirements

The issues of playability addressed in the requirements specification have mostly been satisfied by default in the exclusion of one-hand polyphony in the final version. Although the span of a chord is not a concern, the pitch interval between successive notes requires analysis. Since the notes performed by each hand are restricted to two octaves, it is very rare for a jump to be considered unreasonable.

The first requirement on the quality of the music produced is that of key signature. The algorithm chooses a key signature and correctly maps each note onto this scale, with a small number of exceptions. The fixed variation on the notes in the bass clef as a bar progresses means that when either the *sub-dominant* or the *leading tone* occur, a subsequent note could be outside of the scale (as with any further notes). Although by accidental means, this satisfies a further optional requirement.

As mentioned above, harmony is not addressed by the ‘final’ algorithm, although the bass structure does create a sense of chord progression.

In the area or dynamic structure, the first and most important requirement of a time signature is adhered to satisfactorily, and taken into consideration when developing the bar-by-bar structure. An attempt is made in finding near repetition in the orbit to retain an overall structure, although the success of this is not at all

clear. The short term predictability of these systems naturally creates a sense of phrasing and repetition, although this is quite dependent on the specific orbit.

The majority of the pieces produced by this system are between two and eight minutes long, with occasional overrunning, though through the development of the algorithms and the program the issue of length has been mostly sidelined to allow for more important development.

6.1.2 User Requirements

The user interface for the system satisfies most of the requirements placed upon it, except for one obvious omission. The user can not define a chaotic system of their own, but are restricted to simply submitting parameters and initial data as their only means of creative input. The system, as required, is contained in one program (from the user's perspective) and requires minimal user interaction. Any additional requests for information from the user are purely security measures.

The reflection of the behaviour of the chaotic systems in the corresponding compositions is particularly pure. The way in which the music progresses with the orbit reflects very consistently the dynamical nature of the systems, and the fact that the pitch of the notes in the treble clef is taken almost directly from the values of the orbit in the first dimension means that the shape is preserved very well.

The program, given correct input, successfully generates LilyPond source code and calls LilyPond itself (if required), which processes the composition and produces the desired output. All formats are collected together in a unique directory and have filenames as defined by the user.

6.1.3 System Requirements

As described in the previous subsection, the user interface satisfies most requirements, including the rewriting of continuous systems into discrete systems.

The algorithms which form the bulk of the system are very clear and consistent, and the structure of the *Score* object means that the data produced is adequately organised in preparation for conversion to LilyPond source code. The relatively

simple manipulation of the numerical data ensures the preservation, to a satisfactory extent, of the shape and identity of the orbit.

The LilyPond source generation is a very simple process, due to the organisation of the score as an entity. Although some desirable attributes have been left out of the compositional development, such as ornaments, dynamics and key and time changes, all elements present in the ‘final’ algorithms are catered for.

6.2 Composition

6.2.1 Quality of Music

Although this project has been successful in attaining most of its basic, measurable goals, it is fair to say that the music produced by the algorithms is of a lesser quality than desired. Many pieces, particularly those with greater time signatures, lack any audible structure, and appear to have benefited very little from the numerical manipulation their input data has undergone. Note sequences appear (appropriately) chaotic and although the near repetition of the systems creates natural phrases, they are often indiscernible. The lack of explicit consideration of structure in the algorithms means that the pieces lack real direction, and rather than ending are cut off mid-phrase. One other particularly noticeable absence is that of harmony.

Despite these numerous problems, the system does show a lot of potential. In some compositions, particularly those with shorter bars, structure is more apparent, with phrases and chord progression appearing throughout. The way in which the bass line is constructed, for the most, aids in the development of tonal structure.

Notation

Aside from the audible quality, the sheet music produced by the system is of high quality. The side of the program which writes the LilyPond source code pays particular attention to detail. All notes and rests are fitted perfectly into their bars, and care is even taken to ensure the rests are (generally) inserted in the correct order.

Two small problems are apparent, however, in the notation of the compositions. Firstly, the way in which notes are extended in the bass clef using ties is not ideal, and secondly, the note *E sharp* is not accounted for in the key of *F sharp*, instead being written as an *F natural*.

6.2.2 Simplicity of Manipulation

The manipulation of the original data has been kept extremely simple, to the extent that musical quality has been greatly compromised. Throughout the compositional algorithm there is only one step which strays from using the numerical values directly meaning that the development of structure has been left to the behaviour of the chaotic system.

6.2.3 Reflection of Chaotic Behaviour

The simplicity of manipulation of data has meant that the behaviour of the chaotic orbit has been preserved extremely well. The additional use of event trees also helps to simplify the behaviour while preserving the overall shape. The shape and dynamic progression of the system can be clearly seen and heard in the output.

6.3 Software Validation

As a prototypal program with limited development time, the explicit validation aspect has been somewhat neglected. The system has very little to no error handling, and relies completely on the discretion and integrity of the user as to whether the numerical systems submitted are indeed chaotic. Assuming everything at the user end is sound, there should be no problems in the compositional section of the program. The ongoing experimentation and concurrent development of the algorithms with their implementation has meant that implicit validation has been necessary throughout the system. The decision to limit the input to a set of predefined systems reduces the freedom of the user, thereby eliminating the need for further algorithmic validation beyond that performed in the experimentation involved in the development.

There are small amounts of useful validation in place throughout the system, including checks for completed fields in the user interface, and a warning to avoid overwriting previous work. Theoretically, the bespoke nature of this software should mean that validation is only necessary in the early stages of running, to ensure correct user input. The level to which this validation should be carried, however, is highly dependent on the assumptions placed upon the user.

6.4 Summary

To some extent, the system performs satisfactorily, meeting the majority of its basic requirements. Input is taken via a user interface, manipulated by a set of fixed algorithms and mapped to a musical score. When investigated more closely, however, several flaws are found, in particular in the quality of music produced.

7 Conclusions

The first point to be made is that which becomes immediately evident with the running of the program, that of the platform conflict. LilyPond is written to run on UNIX systems, while Visual Basic is a language written for Windows. In order to accommodate both the user interface and the musical processing, a UNIX emulator must be run on Windows (in particular, Cygwin), and this is clearly inconvenient. It is clear now that a text based interface for UNIX, probably written in C, would, although not ideal, be preferable. This would make for far greater portability and convenience, and would also greatly cut down on the need to cater for external processes. As mentioned previously, writing the entire program in C++ would perhaps have been an ideal solution, and would be seriously considered if this project was to be extended.

A second flaw comes with the limited selection of templates for dynamical systems. Although a system for the recognition of user defined equations would require far more time and depth, it would add a great deal of freedom and creativity to the composition, and would produce far more varied and interesting results.

The incompleteness of the error handling in place in the user interface and successive reliant processes is undesirable, and would be a key area for development in future work, though, for now, assuming there are no errors induced by the user's ineptitude, the user interface performs its task satisfactorily.

Although at the stage at which the development of the algorithms was halted, little success had been achieved in generating a high quality of music, the potential shown in places demonstrates that further development could lead to greater success.

One clear extension would be the reintroduction of harmony. In the case where a third dimension is available, the algorithms do not even consider this additional source of creativity which, perhaps, could be used for the development of harmony.

CONCLUSIONS

The event trees which represent local minima, as described early on in the development, are unused in the ‘final’ algorithm. There should be no reason for them to be of any less importance than those for local maxima and so should perhaps be included somewhere. Having said this, event trees (as they have been implemented so far) do very little to develop the behaviour of the chaotic systems to aid composition, and have probably been given too big a part in the algorithmic development. Perhaps further investigation into the behaviour of these systems is necessary in order to produce more useful tools for composition.

On a personal level, it is clear now that my expectations for this project were set far too high, given my expertise and the time constraints. The aim to successfully implement these algorithms while actively developing them has proved far more time consuming than expected, and the focus has been drawn away from the development of the algorithms themselves. In future, this issue would be considered with far more importance.

Over the course of this project, it has often been shown that the popular investigation into the use of chaos for composition is more than justified and has a lot of potential. The chaotic systems in question demonstrate such interesting behaviour on so many levels that artistic interpretation seems a natural progression from analysis. This project has demonstrated that automated composition, given the same level of development and investigation, can never be a substitute for the creativity of an individual, although it is very clear that given enough time and technical understanding, this approach has the potential to produce spectacular results.

8 Bibliography

Alligood, K.T., Sauer T.D. and Yorke, J.A. (1996). *Chaos - An Introduction to Dynamical Systems*, Springer-Verlag.

Alpern, A. (1995). 'Techniques for Algorithmic Composition of Music' Hampshire College, accessed at: http://alum.hampshire.edu/~adaF92/algocomp/algocomp95.html (Dec 2004).

Bidlack, R. (1992). 'Chaotic Systems as Simple (but Complex) Compositional Algorithms', *Computer Music Journal*, 16(3), MIT Press, pp. 33-47.

Burns, K.H. (1996, updated Feb 2004). 'Algorithmic Composition' Florida International University, accessed at: http://music.dartmouth.edu/~wowem/hardware/algorithmdefinition.html (Dec 2004).

Cygwin (2004). http://www.cygwin.com/ (Dec 2004).

Frøyland, J. (1992). *Introduction to Chaos and Coherence*, IOP Publishing Ltd.

Jacob, B.L. (1996). 'Algorithmic Composition as a Model of Creativity' *Organised Sound*, 1(3), Cambridge University Press, accessed at: http://www.ee.umd.edu/~blj/algorithmic_composition/algorithmicmodel.html (Dec 2004).

Järveläinen, H. (2000). 'Algorithmic Musical Composition' Helsinki University of Technology, accessed at: http://www.tml.hut.fi/Studies/Tik-111.080/2000/papers/hanna/alco.pdf (Dec 2004).

Kernighan, B.W. and Ritchie, D.M. (1988). *The C Programming Language*, 2nd edition, Prentice Hall.

Leach, J. and Fitch, J. (1995). 'Nature, Music and Algorithmic Composition' *Computer Music Journal*, 19(2), MIT Press, pp. 23-33.

BIBLIOGRAPHY

LilyPond (2004). http://www.lilypond.org/web/ (Dec 2004).

Little, D.C. (1999). ‘Composing with Chaos’ accessed at: http://vbs.ahk.nl/david/COMPwCHAOS.html (Dec 2004).

Maurer, J.A., IV (1999). ‘A Brief History of Algorithmic Composition’ Stanford University, accessed at: http://ccrma-www.stanford.edu/~blackrse/algorithm.html (Dec 2004).

Sommerville, I. (2004). *Software Engineering*, 7th edition, Addison-Wesley.

Xenakis, I. (1971). *Formalized Music*, Indiana University Press.

9 Appendices

A Dynamical Systems

A.1 Logistic Map

In $\mathbb{R}$

$$X_{n+1} = aX_n(1 - X_n)$$

$$n \geq 0$$

$$a > 0$$

A.2 Hénon Map

In $\mathbb{R}^2$

$$\begin{cases} X_{n+1} = 1 - aX_n^2 + Y_n \\ Y_{n+1} = bX_n \end{cases}$$

$$n \geq 0$$

$$a, b \in \mathbb{R}$$

A.3 Standard Map

On $\mathbb{R}^2$ modulo 2π

$$\begin{cases} X_{n+1} = X_n + k \sin Y_n \\ Y_{n+1} = X_n + Y_n + k \sin Y_n \end{cases}$$

$$n \geq 0$$

$$k \in \mathbb{R}$$

A.4 Lorenz Map

In $\mathbb{R}^3$

$$\begin{cases} X_{n+1} = X_n + \Delta t[\sigma(Y_n - X_n)] \\ Y_{n+1} = Y_n + \Delta t[X_n(r - Z_n) + Y_n] \\ Z_{n+1} = Z_n + \Delta t(X_n Y_n - bZ_n) \end{cases}$$

$$n \geq 0$$

$$\sigma, r, b > 0$$

$$\Delta t > 0$$

A.5 Rossler Map

In $\mathbb{R}^3$

$$\begin{cases} X_{n+1} = X_n + \Delta t(-Y_n - Z_n) \\ Y_{n+1} = Y_n + \Delta t(X_n + aY_n) \\ Z_{n+1} = Z_n + \Delta t[b + Z_n(X_n - c)] \end{cases}$$

$n \geq 0$

$a, b, c \in \mathbb{R}$

$\Delta t > 0$

B Input and Output Examples

These examples are taken for arbitrarily chosen parameters and initial data and have not been selected especially.

B.1 Logistic Map

Letting $a = 3.6$, the system

$$\begin{cases} X_{n+1} = 3.6 X_n (1 - X_n) \\ X_0 = 0.4 \end{cases} \qquad \textbf{(B.1.1)}$$

can be obtained.

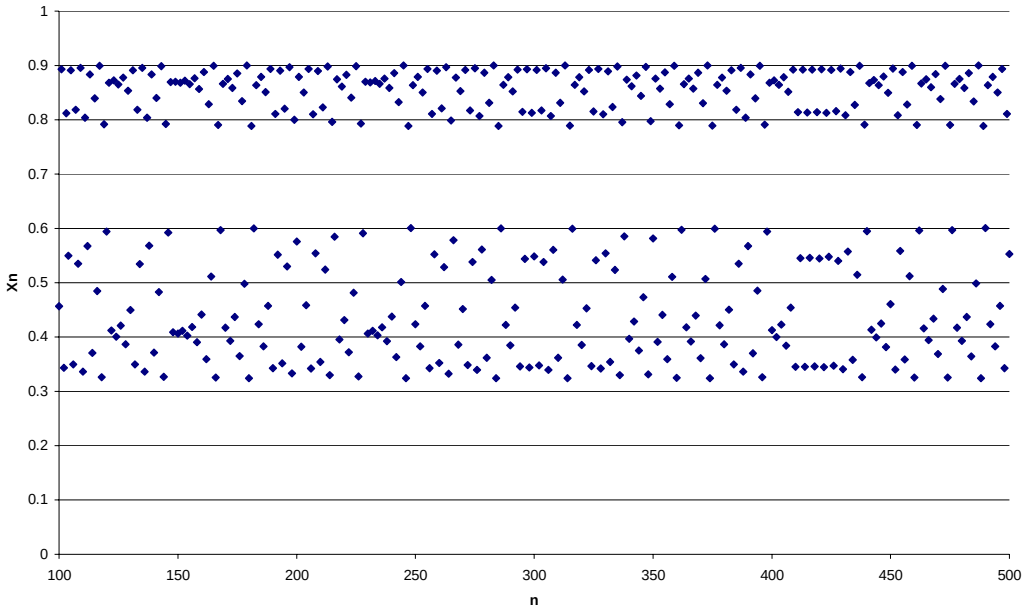

Fig. B.1 Iterations 100-500 of the system (B.1.1) on the (n, X_n) plane.

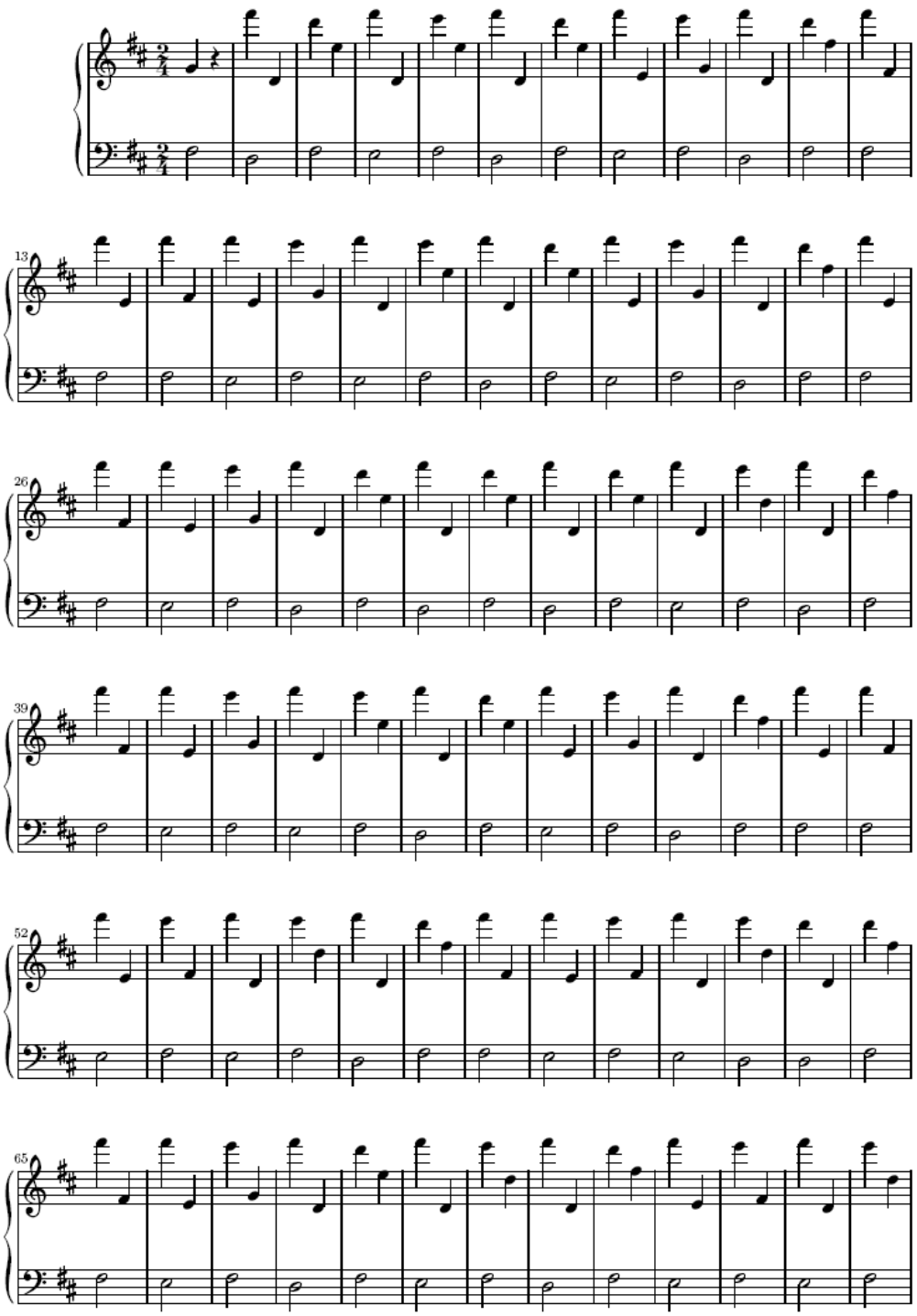

Fig. B.2 First page of the composition ‘LogEx’ generated by Chaotic Composer given system (B.1.1).

B.2 Hénon Map

Letting $a = 1.4$ and $b = 0.3$, the system

$$\begin{cases} X_{n+1} = 1 - 1.4X_n^2 + Y_n \\ Y_{n+1} = 0.3X_n \\ X_0 = 0 \\ Y_0 = 0 \end{cases} \qquad \textbf{(B.2.1)}$$

can be obtained.

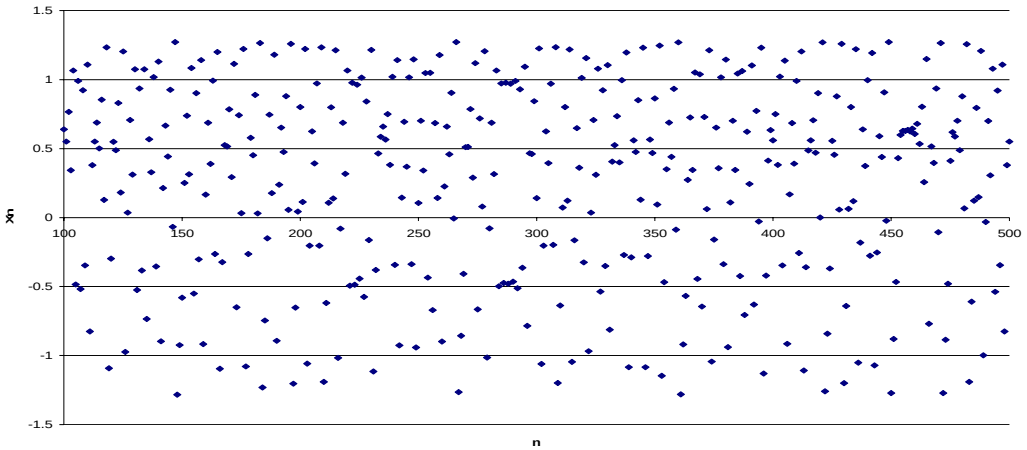

Fig. B.3 Iterations 100-500 of the system (B.2.1) on the (n, X_n) plane.

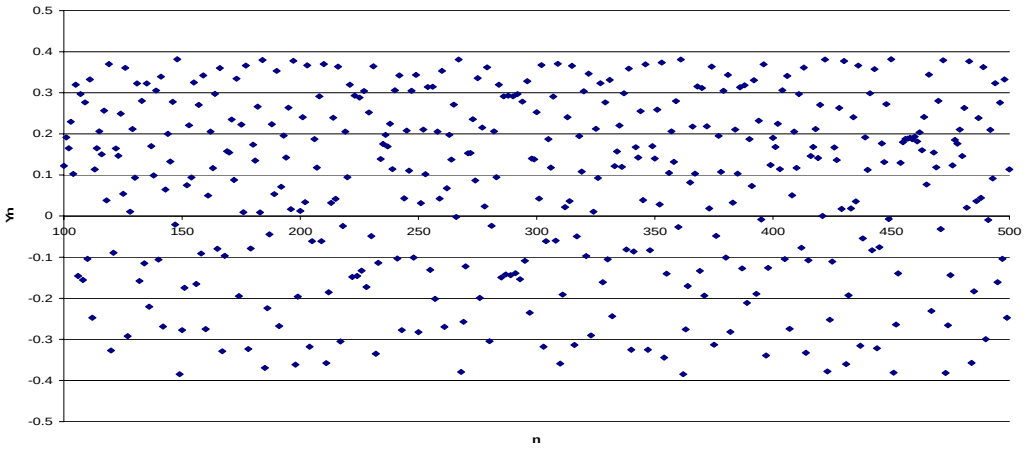

Fig. B.4 Iterations 100-500 of the system (B.2.1) on the (n, Y_n) plane.

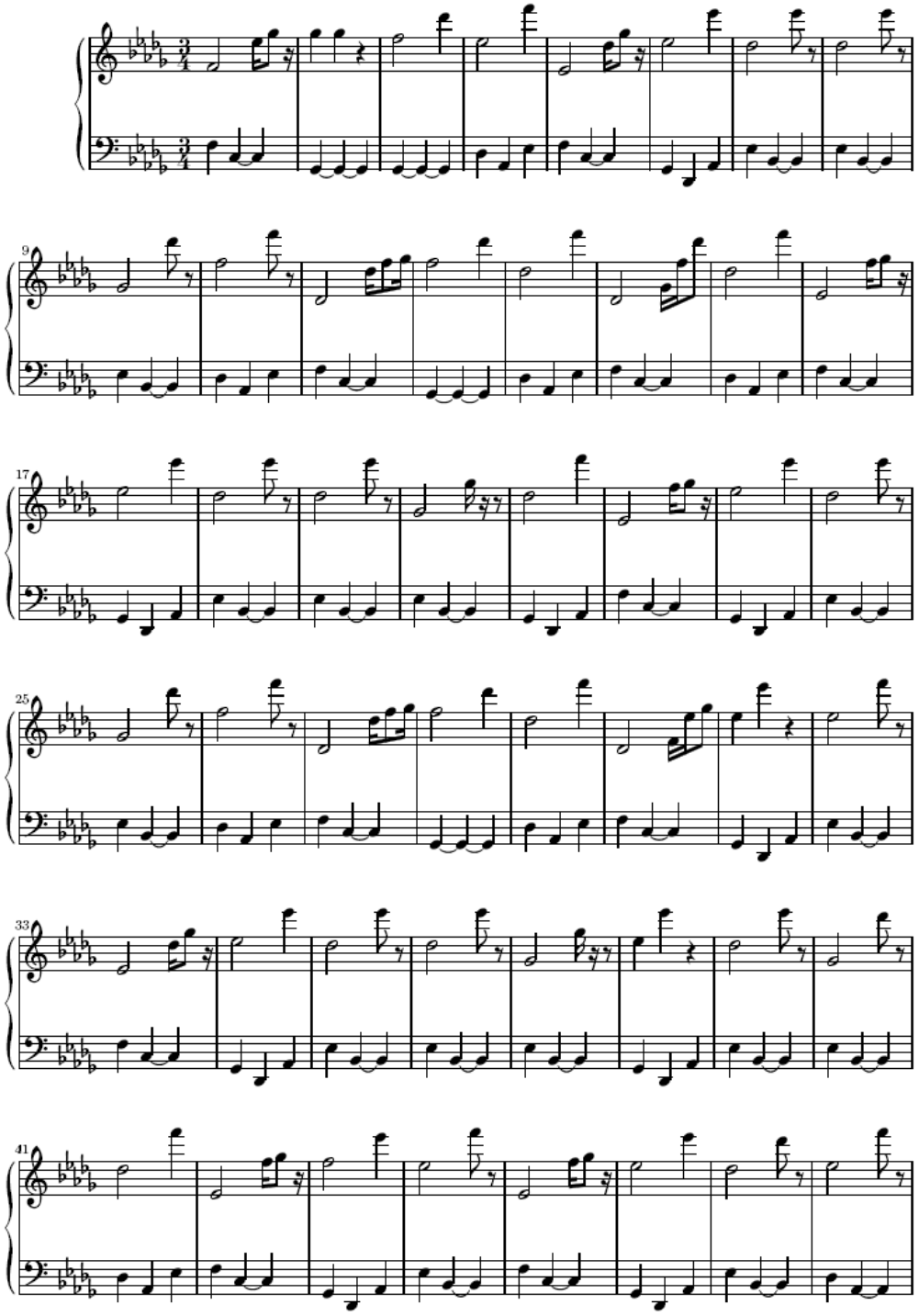

Fig. B.5 First page of the composition ‘HenEx’ generated by Chaotic Composer given system (B.2.1).

B.3 Standard Map

Letting $k = 0.7$ the system

$$\begin{cases} X_{n+1} = X_n + 0.7 \sin Y_n \\ Y_{n+1} = X_n + Y_n + 0.7 \sin Y_n \\ X_n = 0.3 \\ Y_n = 0.3 \end{cases} \qquad \textbf{(B.3.1)}$$

can be obtained.

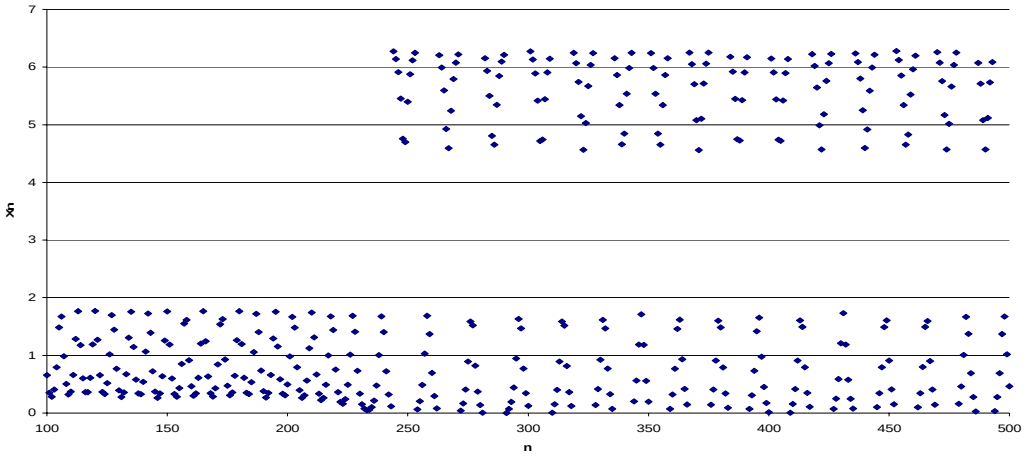

Fig. B.6 Iterations 100-500 of the system (B.3.1) on the (n, X_n) plane.

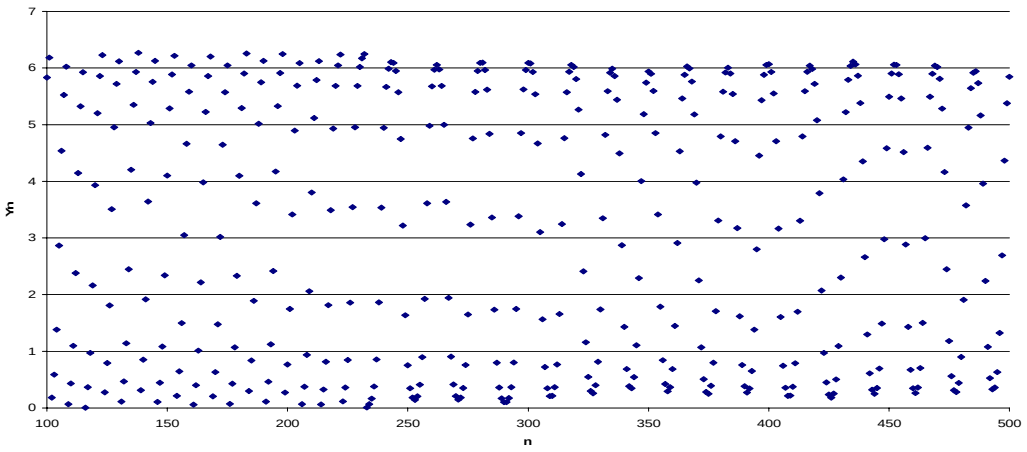

Fig. B.7 Iterations 100-500 of the system (B.3.1) on the (n, Y_n) plane.

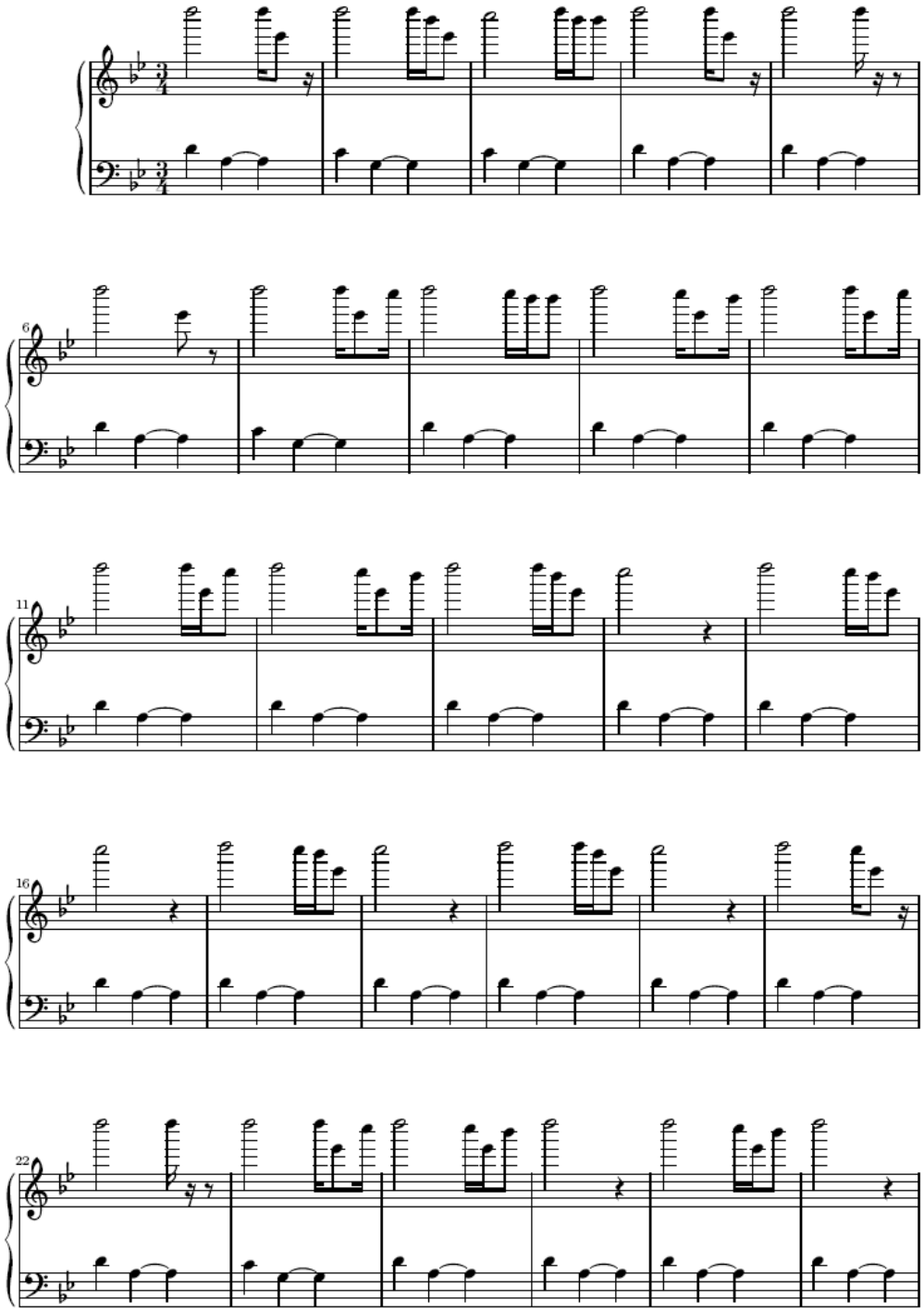

Fig. B.8 First page of the composition ‘StaEx’ generated by Chaotic Composer given system (B.3.1).

B.4 Lorenz Map

Letting $\sigma = 28$, $r = 10$, $b = \frac{8}{3}$ and $\Delta t = 0.02$ the system

$$\begin{cases} X_{n+1} = X_n + 0.02[28(Y_n - X_n)] \\ Y_{n+1} = Y_n + 0.02[X_n(10 - Z_n) + Y_n] \\ Z_{n+1} = Z_n + 0.02(X_n Y_n - \frac{8}{3} Z_n) \\ X_0 = 1 \\ Y_0 = 1 \\ Z_0 = 0 \end{cases} \qquad \textbf{(B.4.1)}$$

can be obtained.

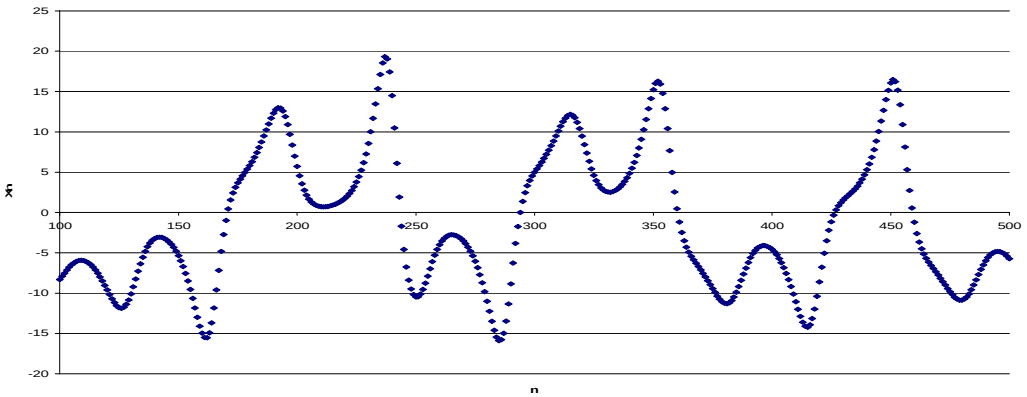

Fig. B.9 Iterations 100-500 of the system (B.4.1) on the (n, X_n) plane.

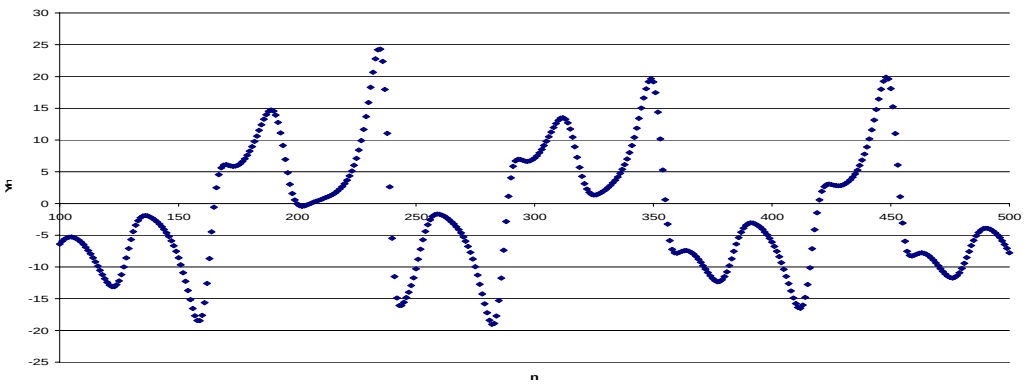

Fig. B.10 Iterations 100-500 of the system (B.4.1) on the (n, Y_n) plane.

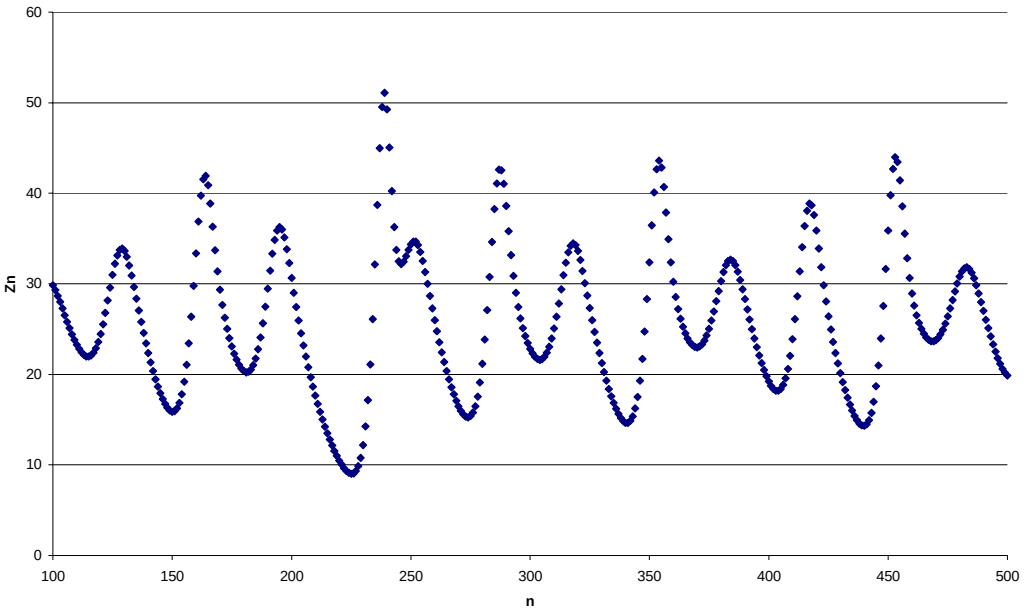

Fig. B.11 Iterations 100-500 of the system (B.4.1) on the (n, Z_n) plane.

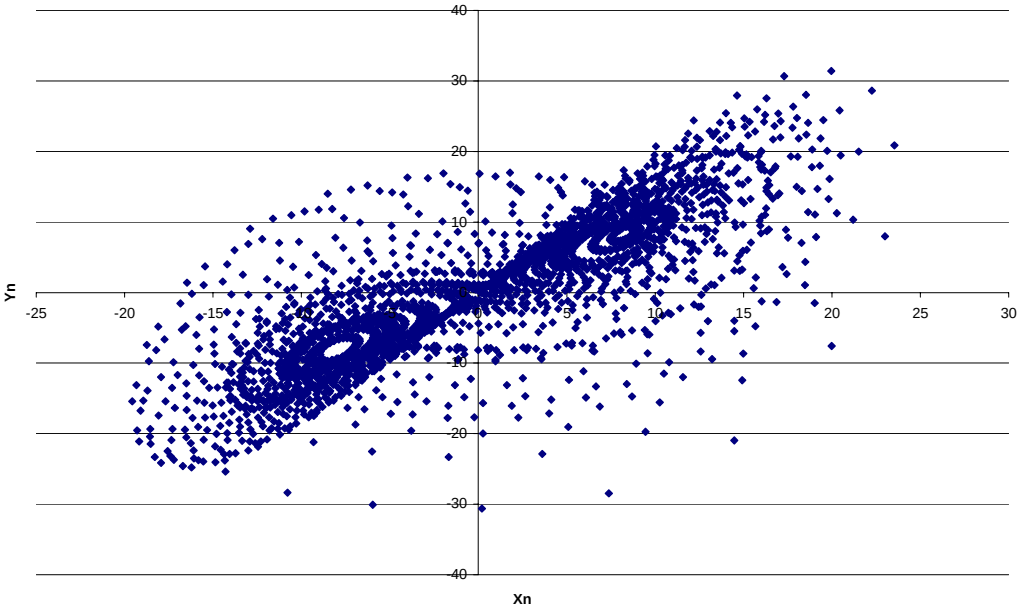

Fig. B.12 The first 1000 iterations of the system (B.4.1) on the (X_n, Y_n) plane.

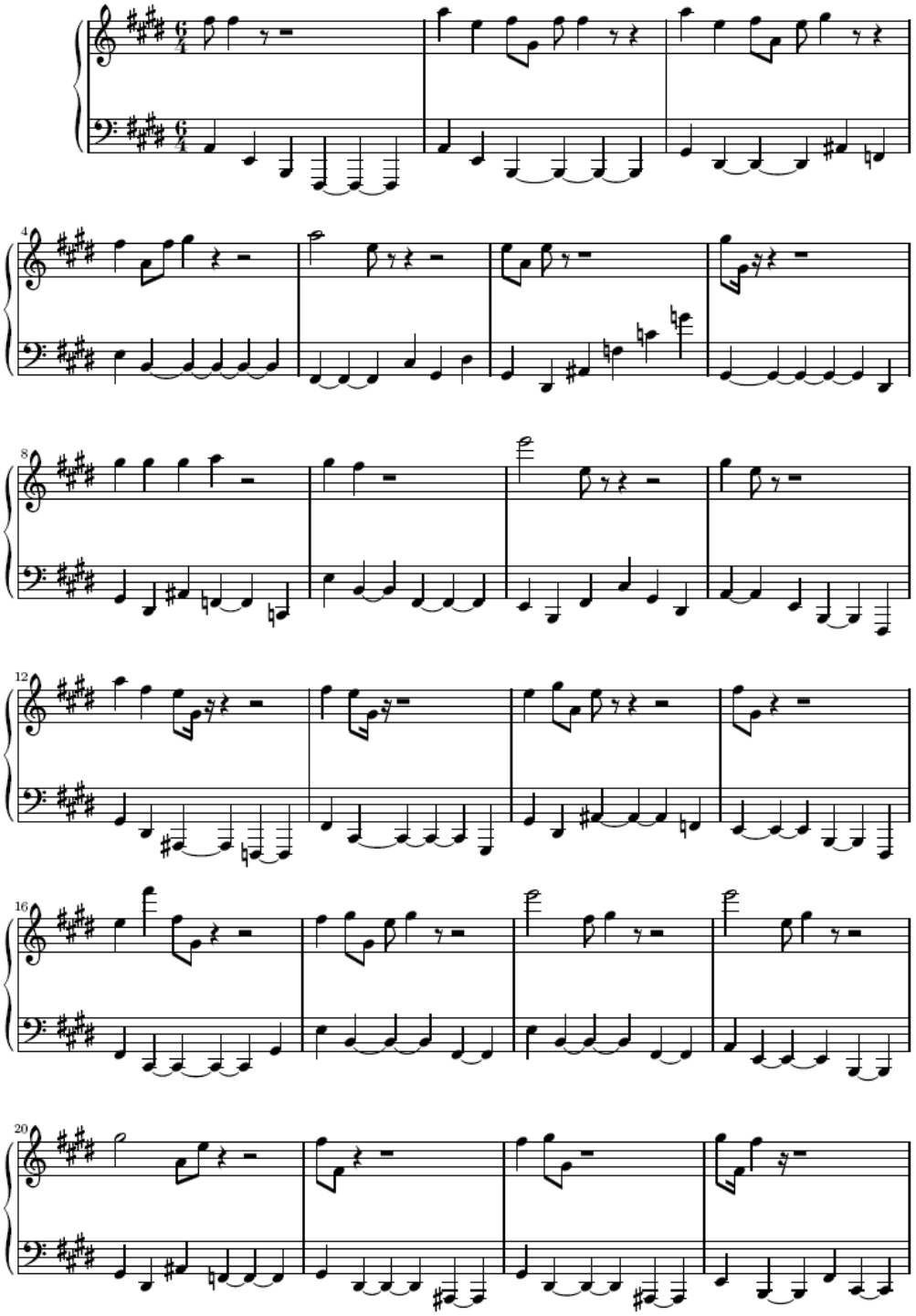

Fig. B.13 First page of the composition ‘LorEx’ generated by Chaotic Composer given system (B.4.1).

B.5 Rossler Map

Letting $a = 0.2$, $b = 0.2$, $c = 5.7$ and $\Delta t = 0.02$

$$\begin{cases} X_{n+1} = X_n + 0.02(-Y_n - Z_n) \\ Y_{n+1} = Y_n + 0.02(X_n + 0.2Y_n) \\ Z_{n+1} = Z_n + 0.02[0.2 + Z_n(X_n - 5.7)] \\ X_0 = 0.0001 \\ Y_0 = 0.0001 \\ Z_0 = 0.0001 \end{cases} \quad \textbf{(B.5.1)}$$

can be obtained.

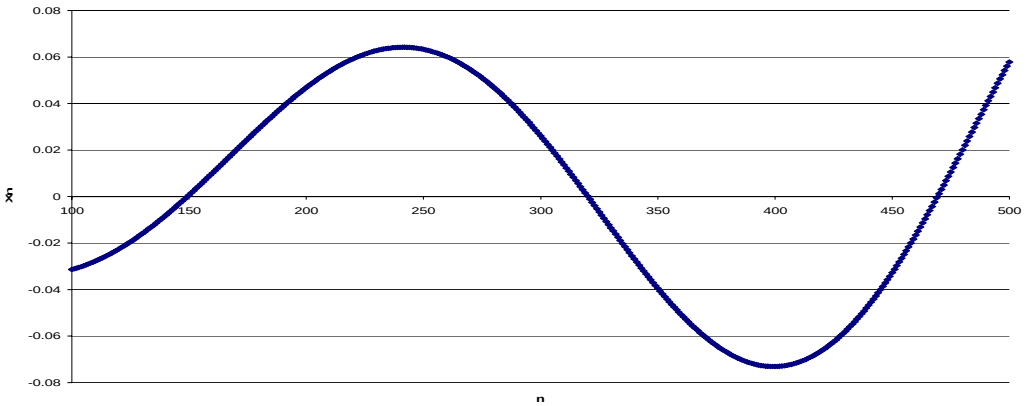

Fig. B.14 Iterations 100-500 of the system (B.5.1) on the (n, X_n) plane.

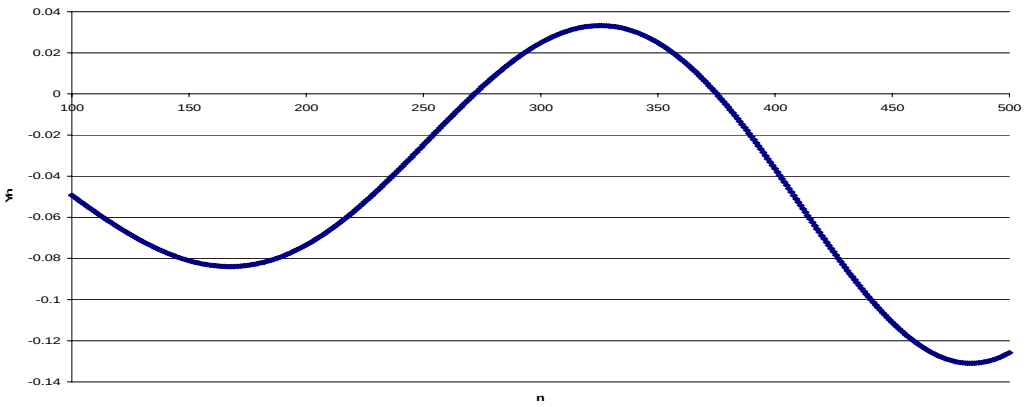

Fig. B.15 Iterations 100-500 of the system (B.5.1) on the (n, Y_n) plane.

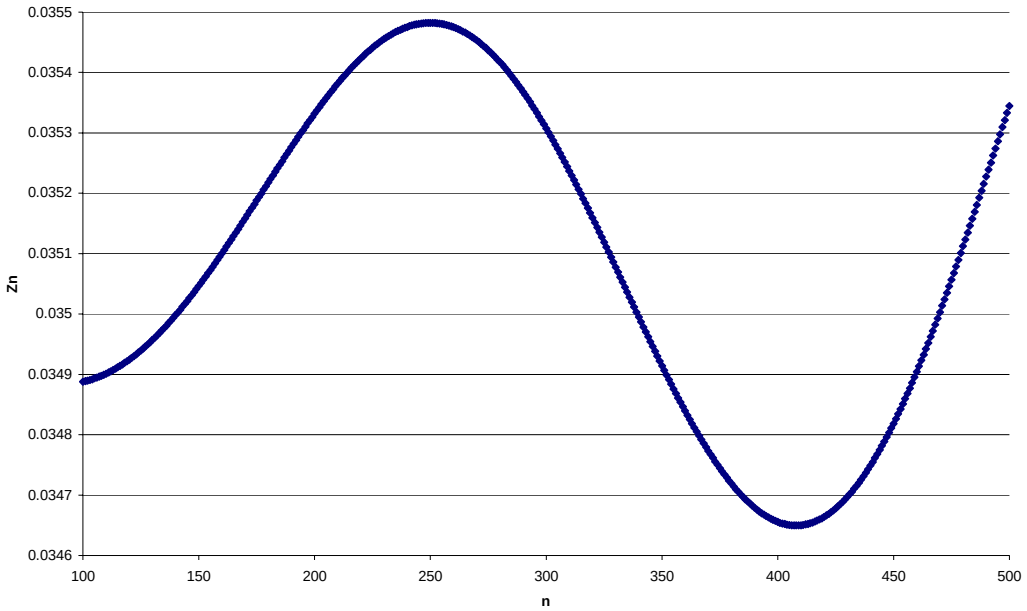

Fig. B.16 Iterations 100-500 of the system (B.5.1) on the (n, Z_n) plane.

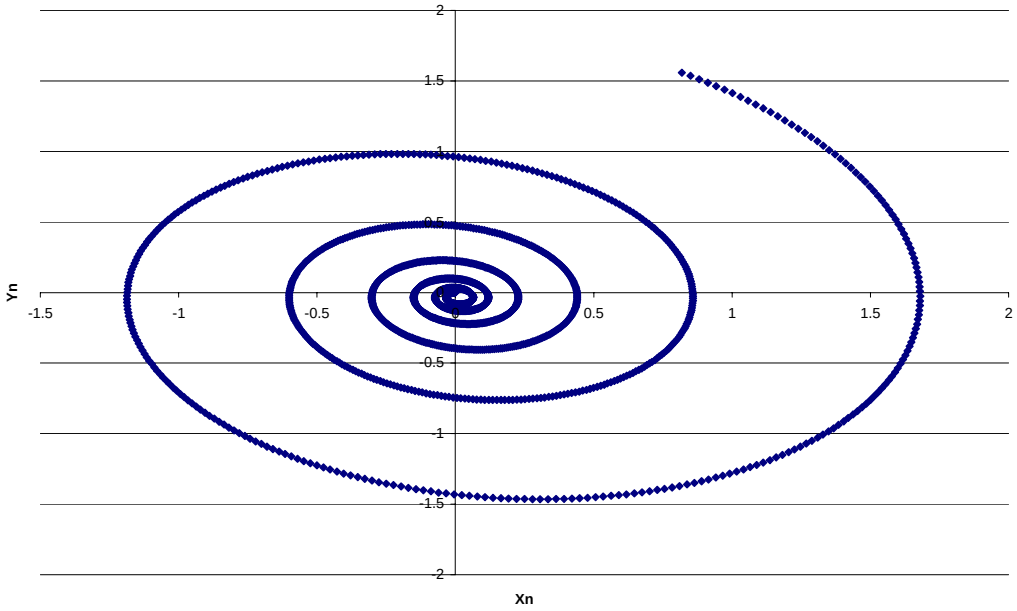

Fig. B.17 First 1000 iterations of the system (B.5.1) on the (X_n, Y_n) plane.

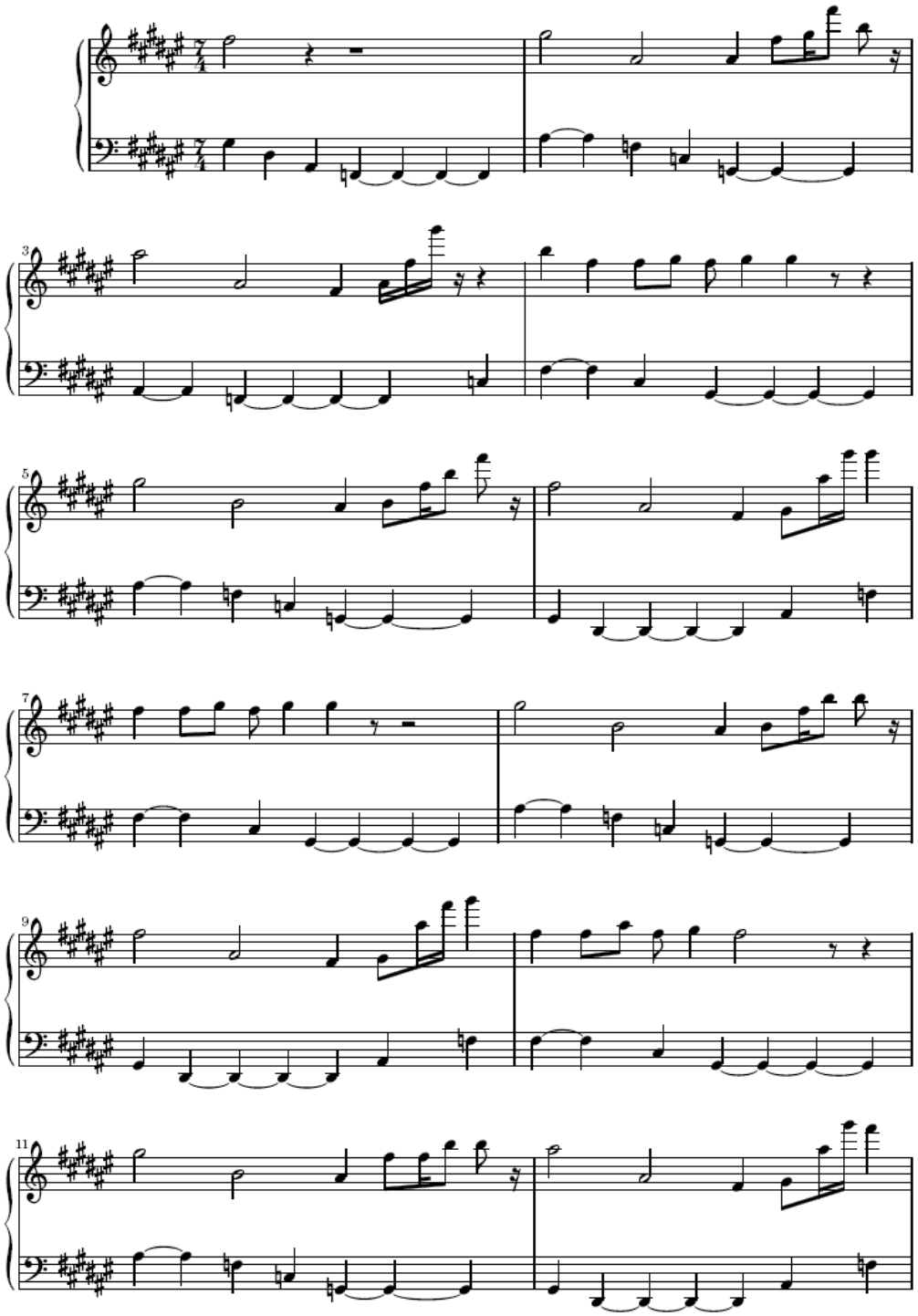

Fig. B.18 First page of the composition ‘RosEx’ generated by Chaotic Composer given system (B.5.1).

C Source Code

C.1 File List

compose.c	
auxiliary.c	auxiliary.h
chaos.c	chaos.h
Event.c	Event.h
Note.c	Note.h
Bar.c	Bar.h
Score.c	Score.h

main.frm (interface)

C.2 Code

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

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include "Score.h"
#include "Bar.h"
#include "Note.h"
#include "Event.h"
#include "chaos.h"
#define N 2000

main() {
   int flag, temp, sheet, midi, i, j, dim;
   double param[4], init[3], **orbit, fbuffer[4];
   char title[37], fileName[80], sbuffer[80], response;
   FILE *f;
   Score *s;
   if(!(system("\n./Interface/ChaoticComposer.exe\n"))) {
      f = fopen("./info.txt","r");
      fscanf(f,"%d",&flag);
      if (flag) {
         fscanf(f,"%s%d%d%d",title,&sheet,&midi,&temp);
         sprintf(fileName,"\nmkdir ./Compositions/%s\n",title);
         if(system(fileName)) {
            while (!(response == 'y' || response == 'n')) {
               fprintf(stdout,"WARNING: Directory './Compositions/%s/' (and possibly composition)
already exists. Overwrite?\ny/n: ",title);
               fscanf(stdin,"%c",&response);
            }
         } else {
            response = 'y';
         }
         if (response == 'y') {
            fprintf(stdout,"Generating orbit...\n");
            orbit = calloc(N,sizeof(double *));
            switch (temp) {
               case 1:
                  dim = 1;
                  fscanf(f,"%lf%lf",init,param);
                  orbit[0] = malloc(sizeof(double));
                  orbit[0][0] = init[0];
                  for (i=1;i<N;i++) {
                     orbit[i] = malloc(sizeof(double));
                     orbit[i][0] = Logistic(orbit[i-1][0],param[0]);
                  }
                  break;
               case 2:
                  dim = 2;
                  fscanf(f,"%lf%lf%lf%lf",init,init+1,param,param+1);
                  *orbit = calloc(dim,sizeof(double));
                  (*orbit)[0] = init[0];
                  (*orbit)[1] = init[1];
                  for (i=1;i<N;i++) {
                     *(orbit+i) = calloc(dim,sizeof(double));
                     Henon(*(orbit+i-1),param[0],param[1],*(orbit+i));
                  }
                  break;
               case 3:
                  dim = 2;
                  fscanf(f,"%lf%lf%lf",init,init+1,param);
                  *orbit = calloc(dim,sizeof(double));
                  (*orbit)[0] = init[0];
                  (*orbit)[1] = init[1];
                  for (i=1;i<N;i++) {
                     *(orbit+i) = calloc(dim,sizeof(double));
                     Standard(*(orbit+i-1),param[0],fbuffer);
                     for (j=0;j<2;j++) {
                        Standard(fbuffer,param[0],*(orbit+i));
                        Standard(*(orbit+i),param[0],fbuffer);
                     }
                     Standard(fbuffer,param[0],*(orbit+i));
                  }
                  break;
               case 4:
                  dim = 3;
                  fscanf(f,"%lf%lf%lf%lf%lf%lf%lf",init,init+1,init+2,param,param+1,param+2,param+3);
                  *orbit = calloc(dim,sizeof(double));
                  (*orbit)[0] = init[0];
                  (*orbit)[1] = init[1];
                  (*orbit)[2] = init[2];
                  for (i=1;i<N;i++) {
                     *(orbit+i) = calloc(dim,sizeof(double));
                     Lorenz(*(orbit+i-1),param[0],param[1],param[2],param[3],fbuffer);
                     for (j=0;j<10;j++) {
                        Lorenz(fbuffer,param[0],param[1],param[2],param[3],*(orbit+i));
                        Lorenz(*(orbit+i),param[0],param[1],param[2],param[3],fbuffer);
                     }
                     Lorenz(fbuffer,param[0],param[1],param[2],param[3],*(orbit+i));
                  }
                  break;
               case 5:
                  dim = 3;
                  fscanf(f,"%lf%lf%lf%lf%lf%lf%lf",init,init+1,init+2,param,param+1,param+2,param+3);
                  *orbit = calloc(dim,sizeof(double));
                  (*orbit)[0] = init[0];
                  (*orbit)[1] = init[1];
                  (*orbit)[2] = init[2];
                  for (i=1;i<N;i++) {
                     *(orbit+i) = calloc(dim,sizeof(double));
                     Rossler(*(orbit+i-1),param[0],param[1],param[2],param[3],fbuffer);
                     for (j=0;j<20;j++) {
                        Rossler(fbuffer,param[0],param[1],param[2],param[3],*(orbit+i));
                        Rossler(*(orbit+i),param[0],param[1],param[2],param[3],fbuffer);
                     }
                     Rossler(fbuffer,param[0],param[1],param[2],param[3],*(orbit+i));
                  }
                  break;
            }
            s = Score_init(NULL, 0, NULL, 0);
            fprintf(stdout,"Composing...\n");
            transcribe(orbit+100,N-100,dim,s);
            fprintf(stdout,"Writing LilyPond source...\n");
            printLilyPond(title,s,sheet,midi*110);
            Score_destroy(s);
            if (sheet || midi) {
               sprintf(fileName,"\nlilypond ./Compositions/%s/%s.ly\n",title,title);
               system(fileName);
               sprintf(sbuffer,"\nmv %s.log ./Compositions/%s \n",title,title);
               system(sbuffer);
            }
            if (sheet) {
               sprintf(sbuffer,"\nmv %s.ps ./Compositions/%s\n",title,title);
               system(sbuffer);
               sprintf(sbuffer,"\nmv %s.dvi ./Compositions/%s\n",title,title);
               system(sbuffer);
               sprintf(sbuffer,"\nmv %s.pdf ./Compositions/%s\n",title,title);
               system(sbuffer);
               sprintf(sbuffer,"\nmv %s.tex ./Compositions/%s\n",title,title);
               system(sbuffer);
```

```
         }
         if (midi) {
            sprintf(sbuffer,"\nmv %s.midi ./Compositions/%s\n",title,title);
            system(sbuffer);
         }
         fprintf(stdout,"Composition complete, see './Compositions/%s/' for output\n",title);
       }
       free(orbit);
    }
    fclose(f);
    system("rm ./info.txt");
  }
}
```

```
/********************
*                   *
*    auxiliary.h    *
*                   *
********************/

extern int normalise(double **input, int size, int dim, double **output);
extern int  divide(double **input, int size, int dim, double **top, double **bottom);
extern int  insertionSort(double *input, int size);
extern int  findRepetition(double **input, int size, int dim, int reps, int *rpts);
```

```
/********************
*                   *
*    auxiliary.c    *
*                   *
********************/

#include "auxiliary.h"

int normalise(double **input, int size, int dim, double **output) {
  if ((input) && (output)) {
    int i, j;
    for (j = 0 ; j < dim ; j++) {
      double max = input[0][j];
      double min = input[0][j];
      for (i = 0 ; i < size ; i++) {
        if (input[i][j] > max) {
          max = input[i][j];
        }
        else if (input[i][j] < min) {
          min = input[i][j];
        }
      }
      for (i = 0 ; i < size ; i++) {
        output[i][j] = (input[i][j]-min)/(max-min);
      }
    }
    return 1;
  } else {
    return 0;
  }
}

int divide(double **input, int size, int dim, double **top, double **bottom) {
  if (input && top && bottom) {
    int i, j, k, l;
    double sorted[size], pivot;
    for (i = 0 ; i < size ; i++) {
      sorted[i] = input[i][0];
    }
    insertionSort(sorted, size);
    pivot = (sorted[size/2-1] + sorted[size/2])/2;
    for (i = 0 ; i < size ; i++) {
      if (input[i][0] > pivot) {
        for (l = 0 ; l < dim ; l++) {
          top[j][l] = input[i][l];
        }
        j++;
      } else {
        for (l = 0 ; l < dim ; l++) {
          bottom[k][l] = input[i][l];
        }
        k++;
      }
    }
    return 1;
  } else {
    return 0;
  }
}
```

```
int insertionSort(double *input, int size) {
  int i, j, k;
  double buffer;
  for (i = 1 ; i < size ; i++) {
     j = 0;
     while (j < i && input[j] < input[i]) {
        j++;
     }
     buffer = input[i];
     for (k = i ; k > j ; k--) {
        input[k] = input[k-1];
     }
     input[j] = buffer;
  }
  return 1;
}

int findRepetition(double **input, int size, int dim, int reps, int *rpts) {
#define nbhd(a,b,c) ((a < (b + c)) && (a > (b - c)))
  int i, j, k, l, flag;
  rpts[0] = 0;
  k = 1;
  l = 1;
  while (k < reps) {
     k = 1;
     i = 17;
     while ((k < reps) && (i < size)) {
        j = 0;
        flag = 1;
        while ((j < dim) && (flag)) {
           flag = nbhd(input[i][j],input[rpts[k-1]][j],0.001*l);
           j++;
        }
        if (flag) {
           rpts[k] = i;
           k++;
           i += 16;
        }
        i++;
     }
     l++;
  }
  return 1;
}
```

```
/*******************
*                  *
*      chaos.h     *
*                  *
*******************/

extern double  Logistic(double X, double a);
extern int   Henon(double X[2], double a, double b, double output[2]);
extern int     Standard(double X[2], double k, double output[2]);
extern int     Lorenz(double X[3], double s, double r, double b, double dt, double output[3]);
extern int     Rossler(double X[3], double a, double b, double c, double dt, double output[3]);

/*******************
*                  *
*      chaos.c     *
*                  *
*******************/

#include <math.h>
#include "chaos.h"

double Logistic(double X, double a) {
  return a*X*(1-X);
}

int Henon(double X[2], double a, double b, double output[2]) {
  output[0] = 1-a*pow(X[0],2)+X[1];
  output[1] = b*X[0];
  return 1;
}

int Standard(double X[2], double k, double output[2]) {
  double TWO_PI = 6.283185;
  output[0] = X[0] + k * sin(X[1]);
  output[1] = X[1] + output[0];
  while (output[0] < 0)    output[0] += TWO_PI;
  while (output[0] >= TWO_PI) output[0] -= TWO_PI;
  while (output[1] < 0)    output[1] += TWO_PI;
  while (output[1] >= TWO_PI) output[1] -= TWO_PI;
  return 1;
}

int Lorenz(double X[3], double s, double r, double b, double dt, double output[3]) {
  output[0] = X[0]+dt*s*(X[1]-X[0]);
  output[1] = X[1]+dt*(X[0]*r-X[0]*X[2]-X[1]);
  output[2] = X[2]+dt*(X[0]*X[1]-b*X[2]);
  return 1;
}

Rossler(double X[3], double a, double b, double c, double dt, double output[3]) {
  output[0] = X[0]-dt*(X[1]+X[2]);
  output[1] = X[1]+dt*(X[0]+a*X[1]);
  output[2] = X[2]+dt*(b+X[0]*X[2]-c*X[2]);
  return 1;
}
```

SOURCE CODE

```
/*******************
*                  *
*       Event.h    *
*                  *
*******************/

typedef struct {
   double value;
   int importance;
} Event;

typedef struct {
   double **values;
   Event ***tree;
   int size;
   int dim;
   int direction;
} EventTree;

extern Event   *Event_init(double value, int importance);
extern Event   *Event_copy(Event *e);
extern int       Event_destroy(Event *e);
extern EventTree  *EventTree_init(double **input, int size, int dim, int direction);
extern Event   **EventTree_growUp(Event **t, int size);
extern Event   **EventTree_growDown(Event **t, int size);
extern int       EventTree_destroy(EventTree *t);

/*******************
*                  *
*       Event.c    *
*                  *
*******************/

#include <malloc.h>
#include <stdlib.h>
#include "Event.h"

Event *Event_init(double value, int importance) {
   Event *e = malloc(sizeof(Event));
   e->value = value;
   e->importance = importance;
   return e;
}

Event *Event_copy(Event *e) {
   if (e) {
      return Event_init(e->value, e->importance);
   } else {
      return NULL;
   }
}

int Event_destroy(Event *e) {
   if (e) {
      free(e);
      return 1;
   } else {
      return 0;
   }
}

EventTree *EventTree_init(double **input, int size, int dim, int direction) {
   int i, j;
   EventTree *t = malloc(sizeof(EventTree));
   t->values = calloc(size,sizeof(double *));
   for (i = 0 ; i < size ; i++) {
      (t->values)[i] = calloc(dim,sizeof(double));
      for (j = 0 ; j < dim ; j++) {
         (t->values)[i][j] = input[i][j];
      }
   }
   t->tree = calloc(size,sizeof(Event **));
   t->size = size;
   t->dim = dim;
   t->direction = direction;
   for (i = 0 ; i < size ; i++) {
      (t->tree)[i] = calloc(dim,sizeof(Event *));
      for (j = 0 ; j < dim ; j++) {
         (t->tree)[i][j] = Event_init(input[i][j],0);
      }
   }
   for (j = 0 ; j < dim ; j++) {
      Event **temp = calloc(size,sizeof(Event *));
      for (i = 0 ; i < size ; i++) {
         temp[i] = Event_copy((t->tree)[i][j]);
      }
      if (direction) {
         EventTree_growUp(temp,size);
      } else {
         EventTree_growDown(temp,size);
      }
      for (i = 0 ; i < size ; i++) {
         (t->tree)[i][j]->importance = temp[i]->importance;
         Event_destroy(temp[i]);
      }
      free(temp);
   }
   return t;
}

Event **EventTree_growUp(Event **t, int size) {
   int i, peakCounter;
   if (size > 1) {
      Event **peaks = calloc(size,sizeof(Event *));
      peakCounter = 0;
      for (i = 0 ; i < size ; i++) {
         if ((i>0 && t[i-1] && t[i+1] && (t[i]->value >= t[i-1]->value) && (t[i]->value >= t[i+1]-
>value)) || ((i==0) && (t[i]->value >= t[i+1]->value)) || (!(t[i+1]) && (t[i]->value >= t[i-1]-
>value))) {
            (t[i]->importance)++;
            peaks[peakCounter++] = t[i];
         }
      }
      EventTree_growUp(peaks,peakCounter);
      free(peaks);
   }
   return t;
}

Event **EventTree_growDown(Event **t, int size) {
   int i, troughCounter;
   if (size > 1) {
      Event **troughs = calloc(size,sizeof(Event *));
      troughCounter = 0;
      for (i = 0 ; i < size ; i++) {
         if ((i>0 && t[i-1] && t[i+1] && (t[i]->value <= t[i-1]->value) && (t[i]->value <= t[i+1]-
>value)) || ((i==0) && (t[i]->value <= t[i+1]->value)) || (!(t[i+1]) && (t[i]->value <= t[i-1]-
>value))) {
            (t[i]->importance)++;
            troughs[troughCounter++] = t[i];
         }
      }
      EventTree_growDown(troughs,troughCounter);
      free(troughs);
   }
   return t;
}

int EventTree_destroy(EventTree *t) {
   int i,j;
```

```
  if (t) {
    for (i = 0 ; i < t->size ; i++) {
      for (j = 0 ; j < t->dim; j++) {
        Event_destroy((t->tree)[i][j]);
      }
      free((t->tree)[i]);
    }
    free(t);
    return 1;
  } else {
    return 0;
  }
}
```

```
/*******************
*                  *
*       Note.h     *
*                  *
*******************/

typedef struct {
  int note;
  int duration;
  int tie;
  int chord;
  int flat;
} Note;

extern Note  *Note_init(int note, int duration, int tie, int chord, int flat);
extern Note  *Note_copy(Note* n);
extern int   Note_destroy(Note* n);
extern char  *Note_toString(Note* n, char* string);

/*******************
*                  *
*       Note.c     *
*                  *
*******************/

#include <malloc.h>
#include <string.h>
#include "Note.h"

Note *Note_init(int note, int duration, int tie, int chord, int flat) {
  Note *n = malloc(sizeof(Note));
  n->note = note;
  n->duration = duration;
  n->tie = tie;
  n->chord = chord;
  n->flat = flat;
  return n;
}

Note *Note_copy(Note *n) {
  if (n) {
    return Note_init(n->note, n->duration, n->tie, n->chord, n->flat);
  } else {
    return NULL;
  }
}

int Note_destroy(Note *n) {
  if (n) {
    free(n);
    return 1;
  } else {
    return 0;
  }
}

char *Note_toString(Note *n, char *s) {
  char *note = calloc(4,sizeof(char));
  char *octave = calloc(7,sizeof(char));
  if (n->note) {
    switch ((n->note+8)%12) {
      case 0:
        note = "c";
        break;
      case 1:
        note = n->flat ? "des" : "cis";
        break;
      case 2:
        note = "d";
        break;
```

```
        case 3:
          note = n->flat ? "ees" : "dis";
          break;
        case 4:
          note = "e";
          break;
        case 5:
          note = "f";
          break;
        case 6:
          note = n->flat ? "ges" : "fis";
          break;
        case 7:
          note = "g";
          break;
        case 8:
          note = n->flat ? "aes" : "gis";
          break;
        case 9:
          note = "a";
          break;
        case 10:
          note = n->flat ? "bes" : "ais";
          break;
        case 11:
          note = "b";
          break;
      }
      switch ((n->note+8)/12) {
        case 0:
          octave = ",,,";
          break;
        case 1:
          octave = ",,";
          break;
        case 2:
          octave = ",";
          break;
        case 3:
          octave = "\0";
          break;
        case 4:
          octave = "'";
          break;
        case 5:
          octave = "''";
          break;
        case 6:
          octave = "'''";
          break;
        case 7:
          octave = "''''";
          break;
        case 8:
          octave = "'''''";
          break;
      }
    } else {
      note = "r";
      octave = "\0";
    }
    sprintf(s,"%s%s%d",note,octave,n->duration);
    if (n->tie) {
      strcat(s,"~");
    }
    return s;
}
```

```
/*******************
*                  *
*       Bar.h      *
*                  *
*******************/

#include "Note.h"

typedef struct {
  Note **notes;
  int    size;
} Bar;

extern Bar  *Bar_init(Note **notes, int size);
extern Bar  *Bar_copy(Bar* b);
extern int  Bar_destroy(Bar* b);
extern char *Bar_toString(Bar* b, char* s);

/*******************
*                  *
*       Bar.c      *
*                  *
*******************/

#include <malloc.h>
#include <string.h>
#include "Bar.h"
#include "Note.h"

Bar *Bar_init(Note **notes, int size) {
  Bar *b = malloc(sizeof(Bar));
  int i;
  b->notes = calloc(size,sizeof(Note *));
  for (i = 0 ; i < size ; i++) {
    (b->notes)[i] = Note_copy(notes[i]);
  }
  b->size = size;
  return b;
}

Bar *Bar_copy(Bar *b) {
  if (b) {
    return Bar_init(b->notes, b->size);
  } else {
    return NULL;
  }
}

int Bar_destroy(Bar *b) {
  if (b) {
    int i = 0;;
    if (b->notes) {
      if (*(b->notes)) {
        for (i = 0 ; i < b->size ; i++) {
          Note_destroy(*(b->notes+i));
        }
      }
      free(b->notes);
    }
    free(b);
    return 1;
  } else {
    return 0;
  }
}

char *Bar_toString(Bar *b, char *s) {
  int i, chordCount;
  chordCount = 0;
  for (i = 0 ; i < b->size ; i++) {
```

```
    char buffer[12];
    int endChord = 0;
    if (!(chordCount)) {
       chordCount = (b->notes)[i]->chord;
       if (chordCount) {
          strcat(s,"<< ");
       }
    } else {
       chordCount--;
       if (!(chordCount)) {
          endChord = 1;
       }
    }
    strcat(s,Note_toString((b->notes)[i],buffer));
    strcat(s," ");
    if (endChord) strcat(s,">> ");
  }
  return s;
}
```

```
/*******************
*                  *
*       Score.h    *
*                  *
*******************/

#include "Bar.h"
#include "Event.h"

typedef struct {
   Bar ***bars;
   int time[2];
   int key;
   int size;
} Score;

extern Score *Score_init(int *time, int key, Bar ***bars, int size);
extern int    Score_destroy(Score *s);
extern int    Score_setBars(Score *s, Bar ***bars, int size);
extern int    transcribe(double **input, int size, int dim, Score *s);
extern int    printLilyPond(char *composition, Score *s, int sheet, int tempo);
extern int    convert(EventTree *t, int *time, int key, int clef, Bar ***output);

/*******************
*                  *
*       Score.c    *
*                  *
*******************/

#include <malloc.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "Score.h"
#include "Bar.h"
#include "Note.h"
#include "Event.h"
#include "auxiliary.h"

Score *Score_init(int time[2], int key, Bar *(*bars[2]), int size) {
   Score *s = malloc(sizeof(Score));
   s->size = size;
   Score_setBars(s,bars,size);
   if (time) {
      (s->time)[0] = time[0];
      (s->time)[1] = time[1];
   } else {
      (s->time)[0] = 0;
      (s->time)[1] = 0;
   }
   s->key = key;
   return s;
}

int Score_destroy(Score *s) {
   int i;
   if (s) {
      for (i = 0 ; i < s->size ; i++) {
         Bar_destroy((s->bars)[i][0]);
         Bar_destroy((s->bars)[i][1]);
         free((s->bars)[i]);
      }
      if (s->size) free(s->bars);
      free(s);
      return 1;
   } else {
      return 0;
   }
}
```

```
int Score_setBars(Score *s, Bar ***bars, int size) {
  int i;
  for (i = 0 ; i < s->size ; i++) {
    Bar_destroy((s->bars)[i][0]);
    Bar_destroy((s->bars)[i][1]);
    free((s->bars)[i]);
  }
  if (s->size) free(s->bars);
  if (size) {
    s->bars = calloc(size,sizeof(Bar **));
    for (i = 0 ; i < size ; i++) {
      (s->bars)[i] = calloc(2,sizeof(Bar *));
      (s->bars)[i][0] = Bar_copy(bars[i][0]);
      (s->bars)[i][1] = Bar_copy(bars[i][1]);
    }
    s->size = size;
  }
  return 1;
}

int transcribe(double **input, int size, int dim, Score *s) {
  if (s) {
    int i, j, flag, numBars;
    Bar ***bars;
    EventTree *tree;
    double **normal;
    normal = calloc(size,sizeof(double *));
    for (i = 0 ; i < size ; i++) {
      normal[i] = calloc(dim,sizeof(double));
    }
    normalise(input, size, dim, normal);
    s-> key = (int)(11*normal[0][0]);
    tree = EventTree_init(normal,size,dim,1);
    bars = calloc(size,sizeof(Bar **));
    for (i = 0 ; i < size ; i++) {
      bars[i] = calloc(2,sizeof(Bar *));
    }
    numBars = convert(tree,s->time,s->key,1,bars);
    Score_setBars(s,bars,numBars);
    EventTree_destroy(tree);
    for (i = 0 ; i < numBars ; i++) {
      Bar_destroy(bars[i][0]);
      Bar_destroy(bars[i][1]);
    }
    for (i = 0 ; i < size ; i++) {
      free(normal[i]);
      free(bars[i]);
    }
    free(normal);
    free(bars);
    return 1;
  } else {
    return 0;
  }
}


int printLilyPond(char *composition, Score *s, int sheet, int tempo) {
  if ((composition) && (s)) {
    int i, j, size;
    char outfile[200], *key;
    FILE *f;
    sprintf(outfile,"./Compositions/%s/%s.ly",composition,composition);
    f = fopen(outfile,"w");
    key = calloc(5,sizeof(char));
    switch (s->key) {
      case 0:
        key = "c";
        break;
      case 1:
        key = "des";
        break;
      case 2:
        key = "d";
        break;
      case 3:
        key = "ees";
        break;
      case 4:
        key = "e";
        break;
      case 5:
        key = "f";
        break;
      case 6:
        key = "fis";
        break;
      case 7:
        key = "g";
        break;
      case 8:
        key = "aes";
        break;
      case 9:
        key = "a";
        break;
      case 10:
        key = "bes";
        break;
      case 11:
        key = "b";
        break;
    }
    fprintf(f,"\\version \"2.4.3\"\n\\score {\n\t\\new PianoStaff \\with {\n\t\t\\remove
\"Note_heads_engraver\"\n\t\t\\consists \"Completion_heads_engraver\"\n\t} <<\n\t\t\\new Staff
{\n\t\t\t\\key %s \\major\n\t\t\t\\time %d/%d",key,(s->time)[0],(s->time)[1]);
    for (i = 0 ; i < s->size ; i++) {
      char *buffer = calloc(1000,sizeof(char));
      fprintf(f,"\n\t\t\t%s",Bar_toString((s->bars)[i][0],buffer));
      free(buffer);
    }
    fprintf(f,"\\bar \"|.\"\n\t\t}\n\t\t\\new Staff {\n\t\t\t\\clef bass\n\t\t\t\\key %s
\\major",key);
    for (i = 0 ; i < s->size ; i++) {
      char *buffer = calloc(1000,sizeof(char));
      fprintf(f,"\n\t\t\t%s",Bar_toString((s->bars)[i][1],buffer));
      free(buffer);
    }
    fprintf(f,"\\bar \"|.\"\n\t\t}\n\t>>\n");
    if (tempo) {
      fprintf(f,"\t\\midi {\\tempo 4=%d}\n",tempo);
    }
    if (sheet) {
      fprintf(f,"\t\\layout { }\n");
    }
    fprintf(f,"}\n");
    fclose(f);
    return 1;
  } else {
    return 0;
  }
}

int convert(EventTree *t, int *time, int key, int clef, Bar ***output) {
  int i, j, k, rpts[20], barLength, barCount, beatCount, noteCount[2];
  Note *noteSet[2][200];
  if (!(time[0])) {
    if (t->dim == 1) {
      i = 0;
```

```
      while (!((t->tree)[i][0]->importance)) i++;
      j = 1;
      while (!((t->tree)[i+j][0]->importance)) j++;
   } else {
      i = 0;
      while (!((t->tree)[i][1]->importance)) i++;
      j = 1;
      while (!((t->tree)[i+j][1]->importance)) j++;
   }
}
while (j > 7) {
   k = 2;
   while (k < 7 && j%k) k++;
   j = k < 7 ? j/k : 4;
}
time[0] = j;
time[1] = 4;
barLength = 16*time[0]/time[1];
findRepetition(t->values, t->size, t->dim, 20, rpts);
barCount = 0;
beatCount = 0;
noteCount[0] = 0;
noteCount[1] = 0;
k = 0;
while (k < 19 && barCount < 128) {
   for (i = rpts[k] ; i < rpts[k+1] ; i++) {
      int imp, flat, note, note2, duration;
      imp = t->dim > 1 ? (t->tree)[i][1]->importance : (t->tree)[i][0]->importance;
      note = (int)(14*(t->tree)[i][0]->value);
      switch (note%7) {
         case 0:
         case 1:
         case 2:
            note += note%7;
            break;
         default:
            note += note%7 - 1;
            break;
      }
      note += 5*(note/7) + key + 40;
      if (t->dim > 1) {
         note2 = (int)(14*(t->tree)[i][1]->value);
         switch (note2%7) {
            case 0:
            case 1:
            case 2:
               note2 += note2%7;
               break;
            default:
               note2 += note2%7 - 1;
               break;
         }
         note2 += 5*(note2/7) + key + 4;
      }
      flat = (key == 1 || key == 3 || key == 5 || key == 8 || key == 10);
      if (t->dim > 1) {
         duration = pow(2,4-(int)((t->tree)[i][1]->value*4));
      } else {
         duration = 4;
      }
      if (imp) {
         if (beatCount) {
            Note *rests[20];
            int restCount = 0;
            while (beatCount < barLength) {
               int restDur = 1;
               while (beatCount + 16/restDur > barLength) restDur *= 2;
               rests[restCount] = Note_init(0,restDur,0,0,0);
               restCount++;
               beatCount += 16/restDur;
            }
            for (j = restCount ; j > 0 ; j--) {
               noteSet[0][noteCount[0]] = Note_copy(rests[j-1]);
               noteCount[0]++;
               Note_destroy(rests[j-1]);
            }
            output[barCount][0] = Bar_init(noteSet[0],noteCount[0]);
            if (noteSet[1][0]) {
               output[barCount][1] = Bar_init(noteSet[1],noteCount[1]);
               for (j = 0 ; j < noteCount[1] ; j++) {
                  Note_destroy(noteSet[1][j]);
               }
            } else {
               int beats = 0;
               while (beats < barLength) {
                  int restDur = 1;
                  while (beats + 16/restDur > barLength) restDur *= 2;
                  noteSet[1][noteCount[1]] = Note_init(0,restDur,0,0,0);
                  noteCount[1]++;
                  beats += 16/restDur;
               }
               output[barCount][1] = Bar_init(noteSet[1],noteCount[1]);
               for (j = 0 ; j < noteCount[1] ; j++) {
                  Note_destroy(noteSet[1][j]);
               }
            }
            for (j = 0 ; j < noteCount[0] ; j++) {
               Note_destroy(noteSet[0][j]);
            }
            barCount++;
            noteCount[0] = 0;
            noteCount[1] = 0;
            beatCount = 0;
         }
         while (16/duration > barLength) duration *= 2;
         noteSet[0][0] = Note_init(note,duration,0,0,flat);
         noteCount[0]++;
         beatCount += 16/duration;
         if (t->dim > 1) {
            noteSet[1][0] = Note_init(note2,4,0,0,flat);
            noteCount[1]++;
            for (j = 1 ; j < barLength/4 ; j++) {
               if (i+j < t->size) {
                  int prev = noteSet[1][j-1]->note;
                  if ((t->tree)[i+j][1]->value > (t->tree)[i][1]->value + 0.1) {
                     noteSet[1][j] = Note_init(prev+7,4,0,0,flat);
                  } else if ((t->tree)[i+j][1]->value < (t->tree)[i+j-1][1]->value - 0.1) {
                     noteSet[1][j] = Note_init(prev-5,4,0,0,flat);
                  } else {
                     noteSet[1][j] = Note_copy(noteSet[1][j-1]);
                     noteSet[1][j-1]->tie = 1;
                  }
               } else {
                  noteSet[1][j] = Note_copy(noteSet[1][j-1]);
                  noteSet[1][j-1]->tie = 1;
               }
               noteCount[1]++;
            }
         } else {
            int susDur, beats;
            beats = 0;
            while (beats < barLength) {
               susDur = 1;
               while (beats + 16/susDur > barLength) susDur *= 2;
               noteSet[1][noteCount[1]] = Note_init(note-36,susDur,1,0,flat);
               beats += 16/susDur;
               noteCount[1]++;
            }
            noteSet[1][noteCount[1]-1]->tie = 0;
         }
```

```
      } else {
         while (beatCount+16/duration > barLength) duration *= 2;
         if (!(beatCount) && barCount) {
            for (j = 0 ; j < output[barCount-1][1]->size ; j++) {
               noteSet[1][j] = Note_copy((output[barCount-1][1]->notes)[j]);
               noteCount[1]++;
            }
            (output[barCount-1][1]->notes)[j-1]->tie = (noteSet[1][0]->note == noteSet[1][j-1]-
>note);
         }
         noteSet[0][noteCount[0]] = Note_init(note,duration,0,0,flat);
         noteCount[0]++;
         beatCount += 16/duration;
      }
      if (beatCount == barLength) {
         output[barCount][0] = Bar_init(noteSet[0],noteCount[0]);
         output[barCount][1] = Bar_init(noteSet[1],noteCount[1]);
         for (j = 0 ; j < noteCount[0] ; j++) {
            Note_destroy(noteSet[0][j]);
         }
         for (j = 0 ; j < noteCount[1] ; j++) {
            Note_destroy(noteSet[1][j]);
         }
         beatCount = 0;
         noteCount[0] = 0;
         noteCount[1] = 0;
         barCount++;
      }
    }
    k++;
  }
  for (j = 0 ; j < noteCount[0] ; j++) {
    Note_destroy(noteSet[0][j]);
  }
  for (j = 0 ; j < noteCount[1] ; j++) {
    Note_destroy(noteSet[1][j]);
  }
  return barCount;
}
```

```
''''''''''''''''
'              '
'    main.frm  '
'              '
''''''''''''''''

VERSION 5.00
Begin VB.Form main
   BackColor       =   &H00FFFFC0&
   BorderStyle     =   1  'Fixed Single
   Caption         =   "Chaotic Composer"
   ClientHeight    =   7230
   ClientLeft      =   45
   ClientTop       =   435
   ClientWidth     =   9870
   Icon            =   "main.frx":0000
   LinkTopic       =   "Form1"
   MaxButton       =   0   'False
   MinButton       =   0   'False
   ScaleHeight     =   7230
   ScaleWidth      =   9870
   StartUpPosition =   1  'CenterOwner
   Begin VB.OptionButton Template
      BackColor       =   &H0080C0FF&
      Caption         =   "ROSSLER"
      BeginProperty Font
         Name            =   "Times New Roman"
         Size            =   9.75
         Charset         =   0
         Weight          =   700
         Underline       =   0   'False
         Italic          =   0   'False
         Strikethrough   =   0   'False
      EndProperty
      ForeColor       =   &H00800000&
      Height          =   375
      Index           =   5
      Left            =   4920
      Style           =   1  'Graphical
      TabIndex        =   20
      Top             =   3360
      Width           =   4815
   End
   Begin VB.OptionButton Template
      BackColor       =   &H0080C0FF&
      Caption         =   "LORENZ"
      BeginProperty Font
         Name            =   "Times New Roman"
         Size            =   9.75
         Charset         =   0
         Weight          =   700
         Underline       =   0   'False
         Italic          =   0   'False
         Strikethrough   =   0   'False
      EndProperty
      ForeColor       =   &H00800000&
      Height          =   375
      Index           =   4
      Left            =   4920
      Style           =   1  'Graphical
      TabIndex        =   12
      Top             =   120
      Width           =   4815
   End
   Begin VB.OptionButton Template
      BackColor       =   &H0080C0FF&
      Caption         =   "STANDARD"
      BeginProperty Font
         Name            =   "Times New Roman"
         Size            =   9.75
```

```
      Charset         =   0
      Weight          =   700
      Underline       =   0   'False
      Italic          =   0   'False
      Strikethrough   =   0   'False
   EndProperty
   ForeColor       =   &H00800000&
   Height          =   375
   Index           =   3
   Left            =   120
   Style           =   1  'Graphical
   TabIndex        =   8
   Top             =   3720
   Width           =   4575
End
Begin VB.OptionButton Template
   BackColor       =   &H0080C0FF&
   Caption         =   "HÉNON"
   BeginProperty Font
      Name            =   "Times New Roman"
      Size            =   9.75
      Charset         =   0
      Weight          =   700
      Underline       =   0   'False
      Italic          =   0   'False
      Strikethrough   =   0   'False
   EndProperty
   ForeColor       =   &H00800000&
   Height          =   375
   Index           =   2
   Left            =   120
   Style           =   1  'Graphical
   TabIndex        =   3
   Top             =   1560
   Width           =   4575
End
Begin VB.OptionButton Template
   BackColor       =   &H0080C0FF&
   Caption         =   "LOGISTIC"
   BeginProperty Font
      Name            =   "Times New Roman"
      Size            =   9.75
      Charset         =   0
      Weight          =   700
      Underline       =   0   'False
      Italic          =   0   'False
      Strikethrough   =   0   'False
   EndProperty
   ForeColor       =   &H00800000&
   Height          =   375
   Index           =   1
   Left            =   120
   MaskColor       =   &H80000000&
   Style           =   1  'Graphical
   TabIndex        =   0
   Top             =   120
   Width           =   4575
End
Begin VB.Frame Rossler
   Appearance      =   0  'Flat
   BackColor       =   &H80000005&
   ForeColor       =   &H80000008&
   Height          =   2775
   Left            =   4920
   TabIndex        =   38
   Top             =   3600
   Width           =   4815
   Begin VB.PictureBox RosPic
      Appearance      =   0  'Flat
      BackColor       =   &H80000005&
      BorderStyle     =   0  'None
      ForeColor       =   &H80000008&
      Height          =   2415
      Left            =   120
      Picture         =   "main.frx":08CA
      ScaleHeight     =   2415
      ScaleWidth      =   3735
      TabIndex        =   42
      TabStop         =   0   'False
      Top             =   240
      Width           =   3735
   End
   Begin VB.TextBox RosText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   6
      Left            =   3960
      TabIndex        =   27
      Text            =   "0.02"
      Top             =   2400
      Width           =   735
   End
   Begin VB.TextBox RosText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   5
      Left            =   3960
      TabIndex        =   26
      Text            =   "5.7"
      Top             =   2040
      Width           =   735
   End
   Begin VB.TextBox RosText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   4
      Left            =   3960
      TabIndex        =   25
      Text            =   "0.2"
      Top             =   1680
      Width           =   735
   End
   Begin VB.TextBox RosText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   3
      Left            =   3960
      TabIndex        =   24
      Text            =   "0.2"
      Top             =   1320
      Width           =   735
   End
   Begin VB.TextBox RosText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   2
      Left            =   3960
      TabIndex        =   23
      Text            =   "0.0001"
      Top             =   960
      Width           =   735
   End
   Begin VB.TextBox RosText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   1
      Left            =   3960
      TabIndex        =   22
      Text            =   "0.0001"
      Top             =   600
```

```
      Width          =   735
   End
   Begin VB.TextBox RosText
      Alignment      =   1  'Right Justify
      Height         =   285
      Index          =   0
      Left           =   3960
      TabIndex       =   21
      Text           =   "0.0001"
      Top            =   240
      Width          =   735
   End
End
Begin VB.Frame Lornez
   Appearance      =   0  'Flat
   BackColor       =   &H80000005&
   ForeColor       =   &H80000008&
   Height          =   2775
   Left            =   4920
   TabIndex        =   37
   Top             =   360
   Width           =   4815
   Begin VB.PictureBox LorPic
      Appearance      =   0  'Flat
      BackColor       =   &H80000005&
      BorderStyle     =   0  'None
      ForeColor       =   &H80000008&
      Height          =   2415
      Left            =   120
      Picture         =   "main.frx":5A50
      ScaleHeight     =   2415
      ScaleWidth      =   3735
      TabIndex        =   41
      TabStop         =   0   'False
      Top             =   240
      Width           =   3735
   End
   Begin VB.TextBox LorText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   0
      Left            =   3960
      TabIndex        =   13
      Text            =   "1"
      Top             =   240
      Width           =   735
   End
   Begin VB.TextBox LorText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   6
      Left            =   3960
      TabIndex        =   19
      Text            =   "0.02"
      Top             =   2400
      Width           =   735
   End
   Begin VB.TextBox LorText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   5
      Left            =   3960
      TabIndex        =   18
      Text            =   "2.67"
      Top             =   2040
      Width           =   735
   End
   Begin VB.TextBox LorText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   4
      Left            =   3960
      TabIndex        =   16
      Text            =   "28"
      Top             =   1320
      Width           =   735
   End
   Begin VB.TextBox LorText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   3
      Left            =   3960
      TabIndex        =   17
      Text            =   "10"
      Top             =   1680
      Width           =   735
   End
   Begin VB.TextBox LorText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   2
      Left            =   3960
      TabIndex        =   15
      Text            =   "0"
      Top             =   960
      Width           =   735
   End
   Begin VB.TextBox LorText
      Alignment       =   1  'Right Justify
      Height          =   285
      Index           =   1
      Left            =   3960
      TabIndex        =   14
      Text            =   "1"
      Top             =   600
      Width           =   735
   End
End
Begin VB.TextBox Title
   Appearance      =   0  'Flat
   Height          =   285
   Left            =   120
   TabIndex        =   30
   Top             =   6120
   Width           =   4575
End
Begin VB.CommandButton Quit
   BackColor       =   &H00C0FFC0&
   Caption         =   "Quit"
   BeginProperty Font
      Name            =   "Times New Roman"
      Size            =   12
      Charset         =   0
      Weight          =   700
      Underline       =   0   'False
      Italic          =   0   'False
      Strikethrough   =   0   'False
   EndProperty
   Height          =   495
   Left            =   8280
   MaskColor       =   &H000000FF&
   Style           =   1  'Graphical
   TabIndex        =   32
   Top             =   6600
   Width           =   1455
End
Begin VB.Frame Henon
   Appearance      =   0  'Flat
   BackColor       =   &H80000005&
   ForeColor       =   &H80000008&
```

```
      Height          =   1695
      Left            =   120
      TabIndex        =   35
      Top             =   1800
      Width           =   4575
      Begin VB.PictureBox HenPic
         Appearance      =   0  'Flat
         BackColor       =   &H80000005&
         BorderStyle     =   0  'None
         ForeColor       =   &H80000008&
         Height          =   1335
         Left            =   240
         Picture         =   "main.frx":7390
         ScaleHeight     =   1335
         ScaleWidth      =   3375
         TabIndex        =   40
         TabStop         =   0   'False
         Top             =   240
         Width           =   3375
      End
      Begin VB.TextBox HenText
         Alignment       =   1  'Right Justify
         Enabled         =   0   'False
         Height          =   285
         Index           =   2
         Left            =   3720
         TabIndex        =   6
         Text            =   "1.4"
         Top             =   960
         Width           =   735
      End
      Begin VB.TextBox HenText
         Alignment       =   1  'Right Justify
         Enabled         =   0   'False
         Height          =   285
         Index           =   1
         Left            =   3720
         TabIndex        =   5
         Text            =   "0"
         Top             =   600
         Width           =   735
      End
      Begin VB.TextBox HenText
         Alignment       =   1  'Right Justify
         Enabled         =   0   'False
         Height          =   285
         Index           =   0
         Left            =   3720
         TabIndex        =   4
         Text            =   "0"
         Top             =   240
         Width           =   735
      End
      Begin VB.TextBox HenText
         Alignment       =   1  'Right Justify
         Enabled         =   0   'False
         Height          =   285
         Index           =   3
         Left            =   3720
         TabIndex        =   7
         Text            =   "0.3"
         Top             =   1320
         Width           =   735
      End
   End
   Begin VB.Frame Standard
      Appearance      =   0  'Flat
      BackColor       =   &H80000005&
      ForeColor       =   &H80000008&
      Height          =   1335
      Left            =   120
      TabIndex        =   34
      Top             =   3960
      Width           =   4575
      Begin VB.PictureBox StaPic
         Appearance      =   0  'Flat
         BackColor       =   &H80000005&
         BorderStyle     =   0  'None
         ForeColor       =   &H80000008&
         Height          =   975
         Left            =   120
         Picture         =   "main.frx":8C92
         ScaleHeight     =   975
         ScaleWidth      =   3495
         TabIndex        =   43
         TabStop         =   0   'False
         Top             =   240
         Width           =   3495
      End
      Begin VB.TextBox StaText
         Alignment       =   1  'Right Justify
         Enabled         =   0   'False
         Height          =   285
         Index           =   2
         Left            =   3720
         TabIndex        =   11
         Text            =   "0.7"
         Top             =   960
         Width           =   735
      End
      Begin VB.TextBox StaText
         Alignment       =   1  'Right Justify
         Enabled         =   0   'False
         Height          =   285
         Index           =   1
         Left            =   3720
         TabIndex        =   10
         Text            =   "0.3"
         Top             =   600
         Width           =   735
      End
      Begin VB.TextBox StaText
         Alignment       =   1  'Right Justify
         Enabled         =   0   'False
         Height          =   285
         Index           =   0
         Left            =   3720
         TabIndex        =   9
         Text            =   "0.3"
         Top             =   240
         Width           =   735
      End
   End
   Begin VB.Frame Logistic
      Appearance      =   0  'Flat
      BackColor       =   &H00FFFFFF&
      ForeColor       =   &H80000008&
      Height          =   975
      Left            =   120
      TabIndex        =   33
      Top             =   360
      Width           =   4575
      Begin VB.PictureBox LogPic
         BorderStyle     =   0  'None
         FillColor       =   &H80000000&
         ForeColor       =   &H80000000&
         Height          =   615
         Left            =   600
         Picture         =   "main.frx":A2CC
         ScaleHeight     =   615
```

```
      ScaleWidth      =   3015
      TabIndex        =   39
      TabStop         =   0   'False
      Top             =   240
      Width           =   3015
   End
   Begin VB.TextBox LogText
      Alignment       =   1  'Right Justify
      Enabled         =   0   'False
      Height          =   285
      Index           =   0
      Left            =   3720
      TabIndex        =   1
      Text            =   "0.4"
      Top             =   240
      Width           =   735
   End
   Begin VB.TextBox LogText
      Alignment       =   1  'Right Justify
      Enabled         =   0   'False
      Height          =   285
      Index           =   1
      Left            =   3720
      TabIndex        =   2
      Text            =   "3.6"
      Top             =   600
      Width           =   735
   End
End
Begin VB.CheckBox MIDI
   Appearance      =   0  'Flat
   BackColor       =   &H00C0FFFF&
   Caption         =   "MIDI"
   BeginProperty Font
      Name            =   "Times New Roman"
      Size            =   9.75
      Charset         =   0
      Weight          =   700
      Underline       =   0   'False
      Italic          =   0   'False
      Strikethrough   =   0   'False
   EndProperty
   ForeColor       =   &H00800000&
   Height          =   375
   Left            =   2520
   Style           =   1  'Graphical
   TabIndex        =   29
   Top             =   5400
   Width           =   2175
End
Begin VB.CheckBox SheetMusic
   Appearance      =   0  'Flat
   BackColor       =   &H00C0FFFF&
   Caption         =   "Sheet Music"
   BeginProperty Font
      Name            =   "Times New Roman"
      Size            =   9.75
      Charset         =   0
      Weight          =   700
      Underline       =   0   'False
      Italic          =   0   'False
      Strikethrough   =   0   'False
   EndProperty
   ForeColor       =   &H00800000&
   Height          =   375
   Left            =   120
   Style           =   1  'Graphical
   TabIndex        =   28
   Top             =   5400
   Width           =   2175
```

```
   End
   Begin VB.CommandButton Compose
      BackColor       =   &H00C0FFC0&
      Caption         =   "Compose"
      Default         =   -1  'True
      BeginProperty Font
         Name            =   "Times New Roman"
         Size            =   12
         Charset         =   0
         Weight          =   700
         Underline       =   0   'False
         Italic          =   0   'False
         Strikethrough   =   0   'False
      EndProperty
      Height          =   495
      Left            =   120
      MaskColor       =   &H000000FF&
      Style           =   1  'Graphical
      TabIndex        =   31
      Top             =   6600
      Width           =   8055
   End
   Begin VB.Label TitleLab
      Appearance      =   0  'Flat
      BackColor       =   &H80000005&
      BackStyle       =   0  'Transparent
      Caption         =   "Title (no spaces):"
      BeginProperty Font
         Name            =   "Times New Roman"
         Size            =   9.75
         Charset         =   0
         Weight          =   400
         Underline       =   0   'False
         Italic          =   0   'False
         Strikethrough   =   0   'False
      EndProperty
      ForeColor       =   &H80000008&
      Height          =   255
      Left            =   120
      TabIndex        =   36
      Top             =   5880
      Width           =   1455
   End
End
Attribute VB_Name = "main"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False

Private Sub Compose_Click()
  Dim t, d, flag, response
  flag = False
  For i = 0 To 1
     If LogText(i).Enabled And LogText(i).Text = "" Then flag = True
  Next
  For i = 0 To 3
     If HenText(i).Enabled And HenText(i).Text = "" Then flag = True
  Next
  For i = 0 To 2
     If StaText(i).Enabled And StaText(i).Text = "" Then flag = True
  Next
  For i = 0 To 6
     If LorText(i).Enabled And LorText(i).Text = "" Then flag = True
  Next
  For i = 0 To 6
     If RosText(i).Enabled And RosText(i).Text = "" Then flag = True
  Next
  If Title.Text = "" Then flag = True
  If flag Then
```

```
      response = MsgBox("Missing information", vbInformation, "Error")
   Else
      response = MsgBox("Compose '" + Title.Text + "'?", vbOKCancel, "Continue")
      If response = vbOK Then
         Open "info.txt" For Output As #1

         Print #1, "1 "; Title; SheetMusic.Value; MIDI.Value
         t = 1
         While Not Template(t).Value
            t = t + 1
         Wend
         Print #1, t
         Select Case t
            Case 1
               For i = 0 To 1
                  Print #1, LogText(i).Text
               Next
            Case 2
               For i = 0 To 3
                  Print #1, HenText(i).Text
               Next
            Case 3
               For i = 0 To 2
                  Print #1, StaText(i).Text
               Next
            Case 4
               For i = 0 To 6
                  Print #1, LorText(i).Text
               Next
            Case 5
               For i = 0 To 6
                  Print #1, RosText(i).Text
               Next
         End Select
         Close #1
         End
      End If
   End If
End Sub

Private Sub Quit_Click()
   Open "info.txt" For Output As #1
   Print #1, "0"
   Close #1
   End
End Sub

Private Sub Template_Click(Index As Integer)
   Select Case Index
      Case 1
         For i = 0 To 1
            LogText(i).Enabled = True
         Next
         For i = 0 To 3
            HenText(i).Enabled = False
         Next
         For i = 0 To 2
            StaText(i).Enabled = False
         Next
         For i = 0 To 6
            LorText(i).Enabled = False
         Next
         For i = 0 To 6
            RosText(i).Enabled = False
         Next
      Case 2
         For i = 0 To 1
            LogText(i).Enabled = False
         Next
         For i = 0 To 3
            HenText(i).Enabled = True
         Next
         For i = 0 To 2
            StaText(i).Enabled = False
         Next
         For i = 0 To 6
            LorText(i).Enabled = False
         Next
         For i = 0 To 6
            RosText(i).Enabled = False
         Next
      Case 3
         For i = 0 To 1
            LogText(i).Enabled = False
         Next
         For i = 0 To 3
            HenText(i).Enabled = False
         Next
         For i = 0 To 2
            StaText(i).Enabled = True
         Next
         For i = 0 To 6
            LorText(i).Enabled = False
         Next
         For i = 0 To 6
            RosText(i).Enabled = False
         Next
      Case 4
         For i = 0 To 1
            LogText(i).Enabled = False
         Next
         For i = 0 To 3
            HenText(i).Enabled = False
         Next
         For i = 0 To 2
            StaText(i).Enabled = False
         Next
         For i = 0 To 6
            LorText(i).Enabled = True
         Next
         For i = 0 To 6
            RosText(i).Enabled = False
         Next
      Case 5
         For i = 0 To 1
            LogText(i).Enabled = False
         Next
         For i = 0 To 3
            HenText(i).Enabled = False
         Next
         For i = 0 To 2
            StaText(i).Enabled = False
         Next
         For i = 0 To 6
            LorText(i).Enabled = False
         Next
         For i = 0 To 6
            RosText(i).Enabled = True
         Next
   End Select
End Sub
```