



SCHOOL OF MATHEMATICAL AND COMPUTER SCIENCES
HERIOT-WATT UNIVERSITY

FINAL YEAR DISSERTATION

Financial Music

Author:

DANIEL DEMBY

Supervisor:

Prof. DAVID CORNE

June 2, 2008

Declaration:

I, Daniel Demby, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form eg: ideas, figures, text, tables, programs are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

Date:

Abstract

In Douglas Adams' novel 'Dirk Gently's Holistic Detective Agency', he describes software capable of generating music from corporate accounts. The music generated was capable of conveying the state of an account, to the point where a listener was able to make an investment decision.

Over the course of this dissertation, we will design, develop, implement and evaluate two approaches towards achieving this. We will focus on both conveying the account's nature, and on making the audio output as musical as possible. The first approach explored uses a mathematical approach in which we generate 'signals' from the accounts. These signals to appropriate musical sequences. The second approach uses L-Systems to generate music from these same signals.

We will discover in the evaluation that both approaches produce audio output which is considered musical. We will also discover that the L-System approach shows the highest level of success when conveying the nature of an account.

Finally, we will go forward to design a novel idea called *The Financial Genome*, in which we attempt to use a biologically inspired approach to identify patterns in accounts, find an optimum genome.

Acknowledgments

David Corne, who presented such a compelling dissertation topic, and agreed to take me on.

Gerard Briscoe, for proof-reading the chapters and explaining many of the intricacies of LaTeX to me.

David Demby, for analysing the accounts for me and providing an insight into accountancy and finance that I lacked.

Mike Chantler, for providing useful and constructive feedback.

Finally, the late Douglas Adams, who came up with the original idea of generating music from accounts.

This document was composed in L^AT_EX and processed using MiKTeX 2.7.

Contents

1	Introduction	1
	Motivation and Objectives	2
	Dissertation Outline	3
2	The Theory	4
	Company Accounts Analysis	4
	Music Theory	5
	Music Cognition	5
	The MIDI Standard	6
	First Case Study: Playing the Market	6
	Summary	7
3	Software Architecture	8
	Requirements of the Framework	8
	The Shell and the Core	9
	Languages of Implementation	9
	Modules of the Shell	11
	Modules of the Core	11
	Module Interactions and Data Flow	11
	Platform Requirements	11
	Summary	12
4	Approach 1: Signal Mapping	14
	Notation and Definitions	14
	Problem Discussion	14
	Processor Isolation: A Sequential Approach	15
	Processor Interaction: A Parallel Approach	16
	From Signals to Music	18
	Signal Generation	20
	Generating a Compound Signal	21
	Mapping Signals to Music	21
	Programming the Functions for Signal Generation	23
	Summary	26
5	Approach 2: L-System Music Generation	27
	Definitions	27
	An Introduction to L-Systems	27
	Design	29
	Variation 1: Dynamic Rule Generation	29
	Variation 2: Dynamic Axiom Generation	29

Defining a Grammar	31
Adding Stacks to the Grammar	31
Interpreting the Grammar	32
An Example in Action	32
Selecting the Rules	33
Selecting the Number of Iterations	35
Programming Functions for an L-System	35
Programming the Axiom Generator	36
Programming the Axiom Parser	37
Preliminary Evaluation	40
Summary	41
6 Evaluation of the First Two Implementations	42
How Can an Objective Evaluation be made of a Subjective Concept?	42
Human Factors	43
Preparation of Test Data	44
Designing an Evaluation Strategy	44
Results Analysis	46
Summary and Conclusions	50
7 Approach 3: The Financial Genome	53
Second Case Study: The Music Genome Project	53
Notation and Definitions	54
Background	54
Defining Genes	55
Rules for Defining a Gene	56
Invalid Genes	57
Gene Expression	58
Implementation as a Prototype	58
Gene Encoding	59
Gene Weightings	59
Functions of the Prototype	60
Beyond the Prototype	61
Summary	61
8 Conclusions	62
Discoveries and Achievements	62
Project Evaluation	62
Future Work	63
Final Conclusion: Is There a Real-World Application for Financial Music?	66
Appendix	I
Appendix A: Test Results	II
Appendix B: Interfaces	VI
Appendix C: Evalutaion Results and Supplementary Graphs	IX
Appendix D: Questionnaire	XI
Appendix E: User Guide	XVII
Appendix F: Financial Genome Example	XVIII
Appendix G: Progress Log	XXII

List of Figures

3.1	Use case diagram, demonstrating the relationship between the user and the software.	8
3.2	The relationship between the Shell and the Core in the software's architecture. . .	10
3.3	A UML representation of the interaction between modules in Financial Music. . .	13
4.1	Mapping of account information to music. Each item of the account is independently mapped to a musical sequence.	15
4.2	Mapping with a compound signal to control tempo and key. The compound signal is derived from the set of individual signals.	16
4.3	Mapping with sequences in parallel.	17
4.4	Showing how a processor can be skinned from a selection of choices.	22
4.5	Attributes of a scale, which can be set to values in order to produce a unique sound.	22
4.6	Diagram showing how an input signal will result in an output musical sequence by passing a threshold level. In this instance, the signal results in a scale.	23
4.7	Diagram showing how an input signal will result in an output musical sequence by passing a threshold level. In this instance, the signal results in an arpeggio. . .	23
4.8	Diagram showing how an input signal will result in an output musical sequence by passing a threshold level. In this instance, the signal results in a broken chord. . .	23
5.1	A Finite State Machine showing a possible application of replaceable symbols within the rules of an L-System.	34
5.2	A Finite State Machine showing a more sensible application of replaceable symbols within the rules of an L-System.	35
5.3	A sliding scale to show how as musicality increases, ability to accurately predict the account's health decreases. The reverse is also true.	40
5.4	The triangle shows how increasing musicality or an increase in the need for concentration makes it more difficult to accurately predict the account.	41
6.1	This word cloud shows how frequently some buzzwords occur in financial articles. The larger the font size, the greater the word's frequency.	43
6.2	A diagram showing how the evaluation process is organised.	44
6.3	A graph showing the average amount that testers agreed with the expert's decision whether to invest in an account, not invest in an account or to remain undecided. (note that 'strings' and 'piano' are L-System generated sequences) . .	46
6.4	A graph showing how many times testers selected buzzwords for each account in the Signal Mapping Sequential output. A star designates that this option was selected by the expert.	47
6.5	A graph showing how many times testers selected buzzwords for each account in the Signal Mapping Parallel output. A star designates that this option was selected by the expert.	48

6.6	A graph showing how many times testers selected buzzwords for each account in the L-System Strings output. A star designates that this option was selected by the expert.	49
6.7	A graph showing how many times testers selected buzzwords for each account in the L-System Strings output. A star designates that this option was selected by the expert.	50
6.8	A graph showing the average number of listens needed for each output type. (note that ‘strings’ and ‘piano’ are L-System generated sequences)	51
6.9	A graph showing the correlation between the length of a musical sequence (in seconds) and the average number of listens that the tester needed.	52
6.10	A graph showing the average spread of the musicality ratings given to each output type.	52
7.1	Example generated from a simple genome. The genes represent both the features of an account, and also a corresponding musical feature.	55
8.1	Sheet 1 of the results.	III
8.2	Sheet 2 of the results.	IV
8.3	Sheet 3 of the results.	V
8.4	Importing CSV files into the Java application.	VII
8.5	Flow of account data from Google Finance to the Java applet.	VIII
8.6	A graph showing how many times on average testers listened to each output sequence.	IX
8.7	A graph showing the average musicality score between 1 and 5 that testers gave each output sequence.	X
8.8	Page 1 of the expert’s questionnaire.	XII
8.9	Page 2 of the expert’s questionnaire.	XIII
8.10	Page 3 of the expert’s questionnaire. Future pages continue in the same fashion.	XIV
8.11	Page 1 of the tester’s questionnaire.	XV
8.12	Page 2 of the tester’s questionnaire. Future pages continue in the same fashion.	XVI

Chapter 1

Introduction

“By the time you’ve sorted out a complicated idea into little steps that even a stupid machine can deal with, you’ve certainly learned something about it yourself.”

– Douglas Adams

In Douglas Adams’ book **Dirk Gently’s Holistic Detection Agency**, a talented software developer named Richard Macduff develops a spreadsheet programme named *Anthem* with a unique feature; It can turn account data into music.

The book was written in a year when the *Apple Macintosh Plus* was considered state-of-the-art technology, and Douglas Adams was clearly captivated by the potential of what kinds of programs could be created. Would the computer limit the bounds of a programmer’s imagination, or allow its programmer to realise ideas that appear seemingly impossible?

Nowadays, Digital on-the-fly music generation is becoming increasingly popular for many different applications, so perhaps Adams’ idea doesn’t seem so far-fetched anymore. In the entertainment industry, games such as *Electroplankton* allow the dynamic generation of music based on the actions of fictional organisms.¹ Conversely, some games work the other way round, generating levels from music. *Audiosurf* is one such example, generating racing tracks from the user’s MP3 collection.²

However, to my knowledge, no one has attempted to implement Adams’ idea... until now.

¹<http://electroplankton.nintendods.com>

²<http://www.audio-surf.com>

Motivation and Objectives

In theoretical computer science (computability theory), we talk about a problem having a certain level of complexity. In other words, some problems can be solved by a computer, some can't, and for some we just don't know. Whilst this concept doesn't directly apply to this dissertation topic (we're not looking at directly solving a mathematical problem per se), it may make us wonder about solving a problem with a computer where the end result is dependant on human opinion.³

This is one of the fundamental challenges posed by this project; Can we solve a problem (generating music from accounts) whereby our success measure (that the music represents the account's nature) is dependant on human subjectivity?

The analysis of a company's account by an expert (such as an accountant) is a logical process, and one that may take some considerable skill. Therefore, an account given to any number of experts to analyse will almost always conclude with a unanimous opinion as to the account's condition.

Music is different. When a piece of music is played to a group of people, there often will *not* be a consensus of opinion as to the meaning of a piece of music. This in its self provides the challenge of taking something of an objective perception (raw account data), and presenting its meaning using something of a subjective perception (music).

An equally fundamental challenge that we will encounter is how to take two disparate concepts with seemingly little in common (accounts and music), and then link them together with tangible ideas which can be implemented and evaluated.

From these challenges, the ultimate aim of this project (this project's 'holy grail') is to produce note sequences from accounts which both **convey the state of the account** and also **sound like well structured pieces of music**. Along the way, we will hope to learn some interesting and surprising things from the issues we encounter.

³Donald E. Knuth published a surreal paper entitled "The Compexity of Songs" in 1984, implying that songs had complexity levels similar to those of problems in the computability theory. http://www.cs.utexas.edu/arvindn/misc/knuth_song_complexity.pdf

Dissertation Outline

The structure of this dissertation follows a natural flow from one idea to another. Where appropriate, we discuss the origin, design and implementation of a single idea in the same chapter.

In **Chapter 2**, we will begin by looking at the theory underpinning the project. What is a financial statement? What are the fundamentals of music cognition, and why are they so important in this dissertation?

With **Chapter 3**, we will look the design and implementation of a central software architecture capable of reading accounts, and playing music. We will also look at how implementations of ideas discussed in this project will connect to this crucial framework.

In **Chapter 4** and **Chapter 5**, we will consider in detail two approaches to generating music from accounts. The first, **Signal Mapping**, takes a mathematical approach. The second, **L-System Music Generation**, comes from a Biologically-Inspired direction. Both of these approaches will be followed through from conception, to design and finally to an implementation, using the architecture discussed in the second chapter.

In **Chapter 6**, we will evaluate these two approaches to determine how successful they have been. The evaluation will use human testers, and the evaluation technique will especially consider the subjective nature of music interpretation. The results of the evaluation will be carefully analysed, and some interesting conclusions will be drawn.

In **Chapter 7**, we will propose a more ambitious approach called **The Financial Genome**. This idea presents a method by which unique features of an account can be indentified through a combination of supervised machine learning and evolutionary algorithms. We will see how the approach used for evaluating the first two implementations naturally leads to this approach.

Finally, in **Chapter 8**, we will conclude and look at avenues in which the project could potentially be expanded on beyond its current scope in the future. We will look at questions raised during the implementation process, and evaluate the project as a whole.

Chapter 2

The Theory

In this chapter, we will look at some of the theory underpinning this project. We will discover how company accounts are formed, and we will look at how music cognition relates to how people perceive music.

Company Accounts Analysis

A company's account consists of three main statements. They are the **Balance Sheet**, the **Income Statement** and the **Cash Flow Statement**.¹

The **balance sheet** gives an impression of a company's situation at a specific point in time. If we compare two balance sheets that are a year apart, we can determine useful details about the company's performance over the year, and the current direction that it is heading. Here is an example of what a company's balance sheet looks like (amounts are given in millions of US Dollars):

	Current Assets	Total Assets	Current Liabilities	Total Liabilities	Total Equity
2006	4076.71	9251.8	3150.2	8401.05	850.13
2007	3766.27	8342.6	5912.8	7821.86	520.74

Of the remaining two statements, The **profit and loss statement** records how the company's profit and losses were reached over the course of a year, and the **cash flow statement** shows movements in cash and cash equivalents (assets). As we are going to be using snapshots of a company's state at points in time, we will keep our focus on the balance sheet as the source for deriving music.

¹<http://www.flexinvest.co.uk/secrets.htm>

Music Theory

Music Theory is the field of study which explains the mechanics of music, and there are some specific ideas that we will need to be very familiar with in order to develop ideas for Financial Music.

Melody

A melody is a successive sequence of notes. If the sequence is well structured within a scale, it will sound musical and pleasant to the ear.

Key

A key has a signature (major or minor) and a tonic (the root note).

Octave

An octave is a distance of twelve successive notes between two successive musical pitches.

Scales

A scale is a sequence of notes within a key. For example, a major scale is given as the following notes in an octave: [1, 3, 5, 6, 8, 10, 12].

Chords

Several notes played concurrently within the same key. Chords add depth to music when complementing a melody.

Clashing Notes

Two or more notes played concurrently, of which at least two are in different keys. The sound is discordant.

Music Cognition

Music cognition is a field of study which concerns its self with how the human mind perceives music. This is a crucial area to understand before beginning the design process, as we need to know what kind of music to generate to create a certain impression in the listener.

Music cognition consists of several sub-topics, and in this project, we are particularly concerned with the topic of music perception.

Music perception is the process by which the **past experience** of a listener processes **sounds** in to **music**.² Musical elements that the listener's brain is capable of perceiving include pitch,

²Note that as the experience of the listener will determine the impression given by a piece of music, it may

rhythm and tonality.

Tonality is an important musical element to consider, as it can convey **emotion**, triggering different areas of the brain depending on the emotion being conveyed. This is a feature that we can apply in designing Financial Music, as we can use tonality to convey the nature of an account in this way.

The MIDI Standard

MIDI is an acronym for **Musical Instrument Digital Interface**. Developed in 1983, it is a versatile protocol for communicating musical sequences and instrumentation. MIDI data consists of a number of **channels**, which contain lists of integer values to represent notes. Channel contents can be played (or transmitted) concurrently, resulting in music.

The most commonly used implementation of MIDI is known as **Genral MIDI**, and it is this version that we will be utilising. MIDI is a good choice for representing music in this project, as we can easily write algorithms to generate the sequences of integers needed to play music using this standard. MIDI is also very well supported, and class libraries are available for many programming languages which allow the playing of MIDI sequences with ease.

First Case Study: Playing the Market

An experimental music project named *Emerald Suspension*³ produced an album titled “Playing the Market”, which uses patterns derived from stock market movements to inspire interesting music, which was then further arranged by the musicians. A soundbite from their website declares the following:

“Conceptual audio arrangements by Emerald Suspension are structured based on patterns created by the stock market, economic indicators, algorithms, and other data sources.”

The *Playing the Market* project differs somewhat from Financial Music in that *Emerald Suspension* are using movements in the *stock market* as a template to produce music for artistic

be that a listener’s cultural musical background may effect how well we can convey an account’s nature to them through the music. This concern is beyond the scope of this project, and is further discussed in the *Further Work* section in the final chapter.

³The homepage of Emerald Suspension can be found at: <http://www.emeraldsuspension.com>

reasons. The data they used to generate the music was specifically chosen because it resulted in good music. The music produced was then refined by the musicians to a high standard.

In Financial Music, we will be using *company accounts* to generate music for *any* account. As there will be no artistic selection of musical output, we will need to develop a way of generating music for any given account in real time. There will also be no refining of the output; the music produced must stand on its own merits.

We can draw inspiration from Emerald Suspension's project. Their album demonstrates that there *are* distinct patterns in the financial world, which can be used to generate music. They also demonstrate that the music generated from these patterns can be perceived by listeners to have a meaning which represents the original patterns.

Summary

So far, we have seen how a company's accounts are represented. We have looked at issues of music cognition, and seen how these issues will play a part in our approach towards a design.

With a basic understanding of the issues of the financial statement and music cognition, we are now in a position to think about drawing connections between these two eclectic concepts. But, before we do this, we need to design and implement an appropriate software architecture, and it is this that we will look at in the next chapter.

Chapter 3

Software Architecture

In this chapter, we will consider the **general requirements** of a program which will turn accounts into music. At this stage we do not know the specifics of how this will be achieved, therefore the architecture needs to be as versatile as possible.

We will see how we can split the software into two distinct sections. One will handles the input and output operations. the other will do the actual processing to produce music from accounts.

Additionally, the architecture will support a “plug-in” framework, allowing many different implementations to be attached with ease.

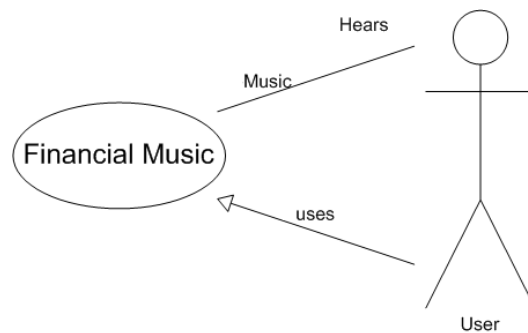


Figure 3.1: Use case diagram, demonstrating the relationship between the user and the software.

Requirements of the Framework

The framework must meet the following requirements:

1. Input:

- (a) Ability to import accounts from a standard format.
 - (b) Simplicity to add additional methods of importing accounts.
2. Output:
 - (a) Output in a common music format.
 - (b) Ability to play music.
 3. Easy addition of core modules for different methods of generating music from accounts.
 4. Versatile framework to allow for high levels of experimentation during development.

The Shell and the Core

Recall that we mentioned that parts of the software can will fall into two distinct categories. We will term these categories the **Shell** and the **Core**. The Shell deals with **input and output** (I/O) operations such as reading in accounts (input), dealing with file operations (input and output) and playing the music (output).

The Core is where the process of turning accounts into music takes place, and therefore performs the **processing**.

Looking at the software in this way is essential, as we do not wish to be concerned with issues of input and output (worrying about where the data is coming from or going to) while we are designing processing strategies. Therefore, we should set things up so that the Core doesn't need to be concerned with where the account data is coming from, or what to do with the music that is produced (*figure 3.2*).

Languages of Implementation

Given the differing natures of the Shell and the Core, it is necessary to carefully select a language of implementation that will best suit each task.

The language chosen for implementation of the Shell is **Java**. Java is a popular high-level programming language. It also has the advantage of being cross-platform, and web capable.

Java is a suitable language for the tasks the Shell will have to perform, as it has a good selection of class libraries at its disposal for dealing with all the I/O operations that will be required by this project. It has libraries for reading and parsing files. It also has excellent MIDI capabilities.

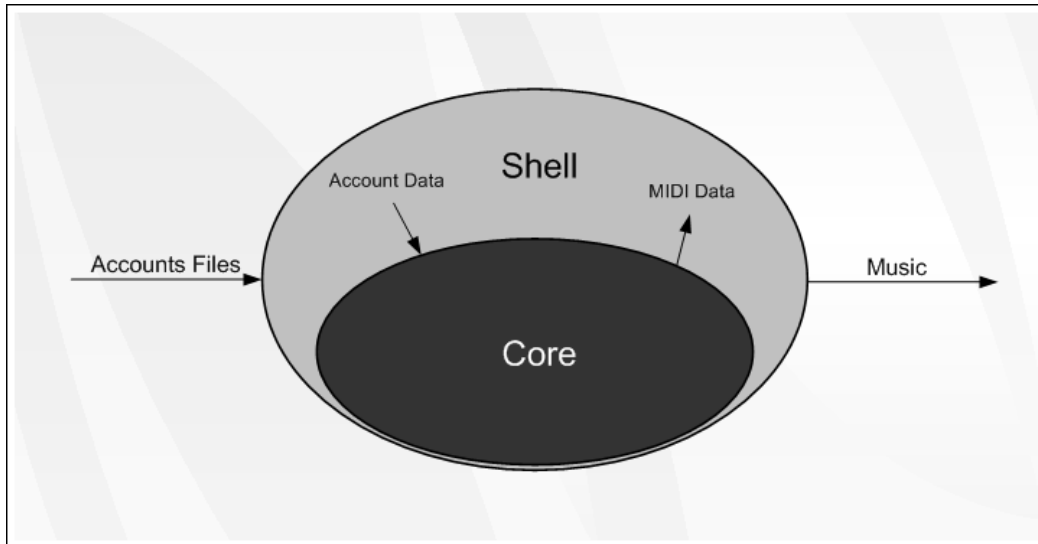


Figure 3.2: The relationship between the Shell and the Core in the software's architecture.

The Core is concerned purely with mathematical and logical operations which will turn accounts into music, and therefore requires a language which has a syntax oriented towards this end. A functional programming language is a good choice for this, and **Python** was chosen for this purpose. Python is a high level dynamically typed language, with support for lists, sets and tuples. It supports a functional programming paradigm, inspired by languages such as SML and Haskell. As we will be presenting designs which will be expressed formally, this paradigm will allow focus on the actual processes involved in music generation.

Taking a functional programming approach to writing the core modules is useful, as it allows us to abstract a problem down to a state where each element of a problem has its own function. This in turn allows the sharing of functions across problems which share many of the same elements.

It also allows functions to be tested individually as they are written. This way, debugging the program is a simpler process; If the integrity of the modules can be verified, then the interactions between modules can be studied in isolation.

(It should be noted however, that the functions developed on the coming pages often display *side-effects* such as displaying text on screen or accessing a global variable, therefore some may not consider the approach truly functional)

As a final point, we need the language used for the core to be able to communicate transparently with the shell. To this end, the core is really implemented in **Jython**, which is an

implementation of the Python language using the **Java Virtual Machine**.¹

Modules of the Shell

Recall that modules in the shell are responsible mainly for I/O operations, and will be implemented in Java. There are three main classes involved in this:

- `AccountReader.class` – Reads in the accounts from source files.
- `PlayMusic.class` – Plays music when given a 2 dimensional integer array of MIDI values.
- `MusicReader.class` – Looks for a CSV file with MIDI values and pipes it to `PlayMusic.class`.

Modules of the Core

Core modules deal with the actual processing. They are as follows:

- `Shared.py` – Contains functions shared across modules in the Core.
- `Settings.py` – Contains global settings in one location, for easy access.
- `Mapping.py` – Contains function for the Signal Mapping implementation (Chapter 4).
- `LSS.py` – Contains function for the L-System Music Generation implementation (Chapter 5).
- `Linden.py` – Contains functions to implement a generic L-System (also Chapter 5).
- `Genome.py` – Contains functions for the Financial Genome approach (Chapter 7).

Module Interactions and Data Flow

The interactions between modules can be observed in *figure 3.3*.

Platform Requirements

A system running *Financial Music* will need to meet the following minimum requirements:

1. Hardware
 - (a) MIDI Capabilities

¹The Jython website can be found at: <http://www.jython.org>

2. Software

- (a) Java 1.6.0
- (b) Jython 2.2.1

Summary

In this chapter, we have designed and constructed a software architecture to support Financial Music. With a Shell (implemented in Java) and a Core (Implemented in Jython), it will be capable of reading in accounts from an external source, and of playing music using the MIDI standard.

As we develop Python code in the future chapters, we will see an evolution of the Core. However, the Java code of the Shell will remain unchanged, and unaffected by development in the Core.

With a framework in place, we are now ready to begin developing approaches to turning accounts into music, and the next couple of chapters will explore two approaches towards achieving this.

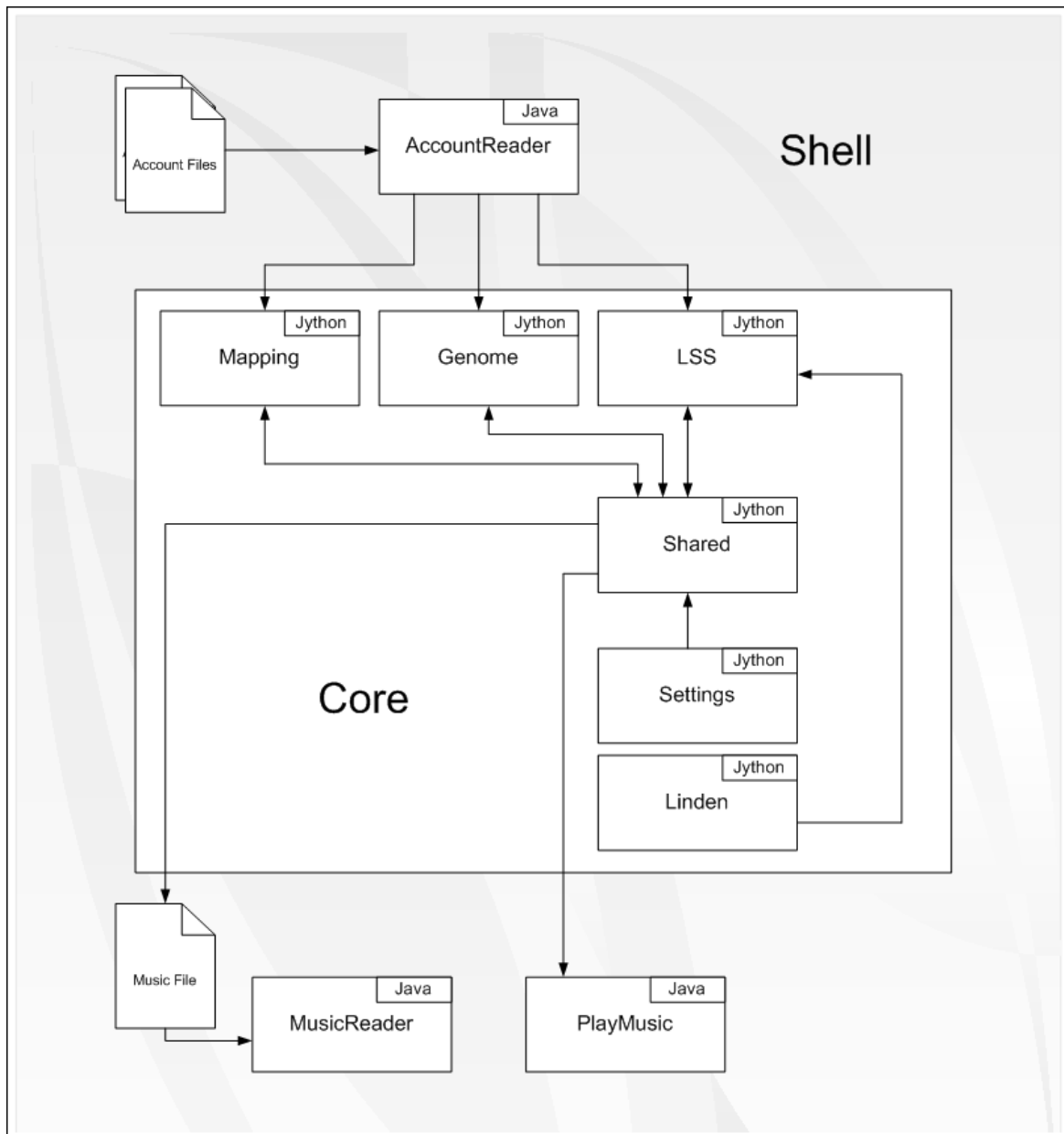


Figure 3.3: A UML representation of the interaction between modules in Financial Music.

Chapter 4

Approach 1: Signal Mapping

The first approach we will develop is an idea called **Signal Mapping**. Accounts have attributes, and when we look at the accounts for two subsequent years, we can observe changes in these attributes. We will see that from these changes, we can derive signals, which can be mapped to appropriate musical constructs. These constructs can be arranged together to form music. To do this, we will be working with the account **balance sheet**.

Notation and Definitions

$\{ \dots \}$	A set of items
Sequence	An ordered set which can contain repeating elements
List	Another name for a Sequence
$\langle \dots \rangle$	A sequence of items
Musical sequence:	A finite sequence of notes.
Set of musical sequences:	Multiple musical sequences whose feel is governed by the set of musical attributes.
Musical attribute:	A static value which pre-determines the feel of a musical sequence. For example: tempo, time signature or key signature.
Set of musical attributes:	Multiple non-contradicting musical attributes.
Musical movement:	The set of musical sequences as governed by the set of musical attributes (<i>i.e. the complete music generated by the accounts</i>).

Problem Discussion

Recall that an account's balance sheet consists of five attributes. Recall also that to assess the health of a company, we can compare the balance sheets between two subsequent years.

Now consider the diagram given in *figure 4.1*. We can connect two parameters together to produce a unique **signal**. This signal can then be routed to a **processor**, which converts it into

a musical sequence. Continuing onwards by piping the output signals of multiple processors produces a complete movement of music. By tweaking and fine tuning the settings of these processors, we can harness the sound so that it represents the account's data.

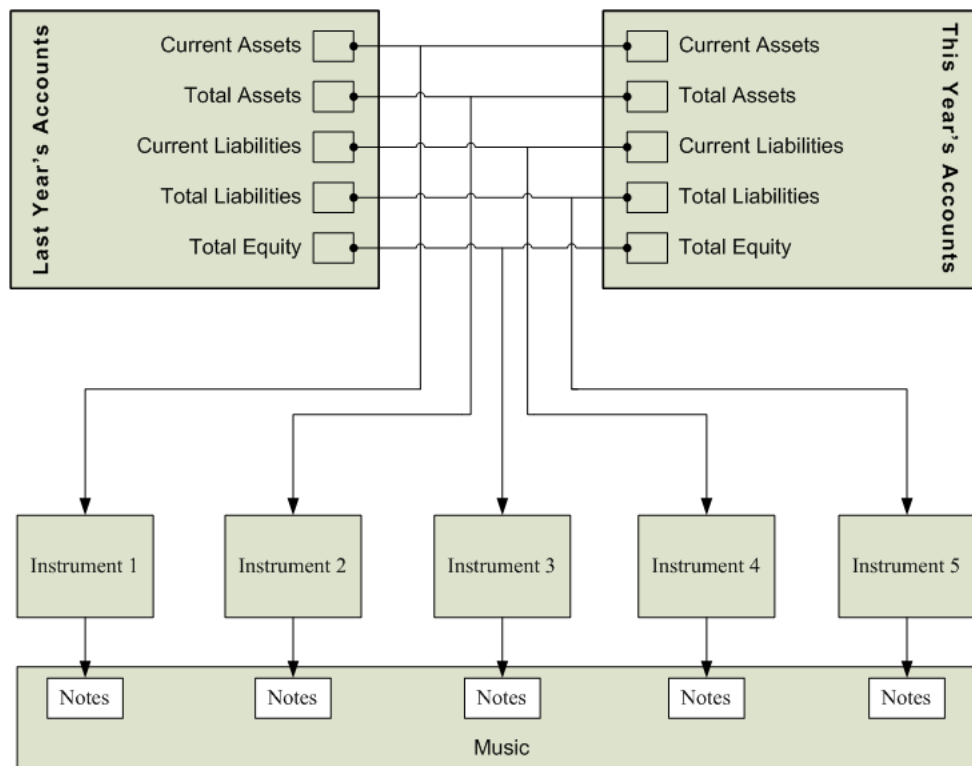


Figure 4.1: Mapping of account information to music. Each item of the account is independently mapped to a musical sequence.

Essentially, we are mapping the difference between two parameters to a musical sequence or musical attribute. The processors are ‘black boxes’, and the overall picture is only concerned with the signals going in and out of the processors. In other words, each processor will correspond to a function in the program.

Looking inside one of these processors, there would be an algorithm to take the account values and turn them into a suitable musical sequence. The processors will have ‘dials’ which when modified change variables within the algorithm. It is these settings that we will tweak to fine tune the system, so that it produces the musical output we are looking for.

Processor Isolation: A Sequential Approach

The most crucial issue we are concerned with is that the overall sound produced must reflect the overall state of the accounts. Although we are mapping individual account details to individual

musical elements, we have to ensure that the overall feel of the sound fits the overall state of the accounts.

The first approach we will try is to have each signal routed to an isolated processor. This processor generates a musical sequence. These sequences are played sequentially, one after the other. The result is a musical sequence in the form of a melody, which gives an impression of the account.

As some changes are more significant than other changes (bigger change in values of account attribute between years), we order the signals in terms of **magnitude**. This way, the individual musical sequences are joined up in order of importance.

We also need a way of deriving musical attributes which affect the *whole* musical sequence. To do this, we calculate a **compound signal** which represents the overall signal spread.

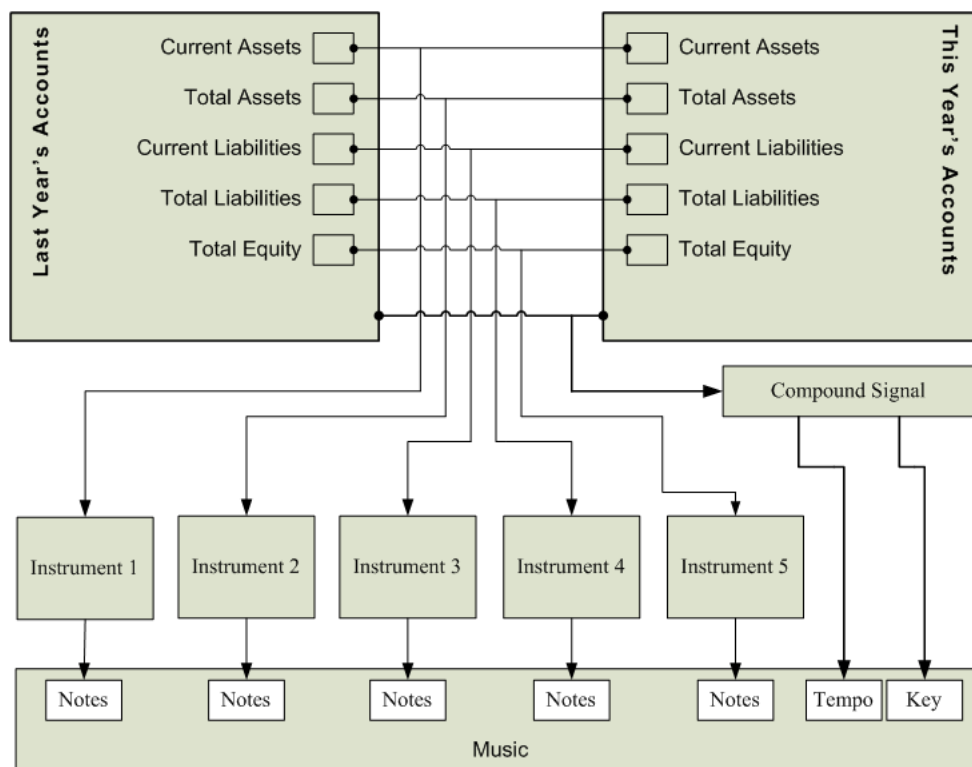


Figure 4.2: Mapping with a compound signal to control tempo and key. The compound signal is derived from the set of individual signals.

Processor Interaction: A Parallel Approach

With the ability to play several musical sequences in sequence, we can modify this approach to produce music where the sequences play concurrently (*figure 4.3*). Taking this approach has

the advantage of creating a completely different sound to the sequential approach, which we can compare with later.

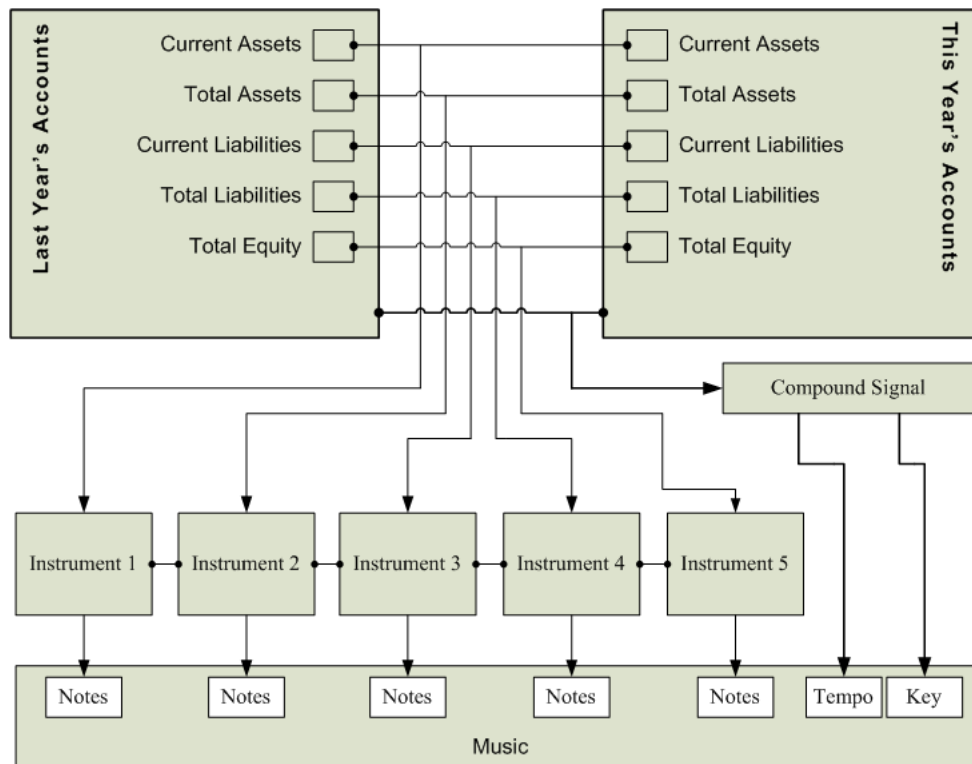


Figure 4.3: Mapping with sequences in parallel.

However doing this presents us with a new problem. As each of the note sequences generated by the processors are in their own key, if we attempt to play them together, there will be note clashes and general discordance of notes. Seeing as we are aiming to generate *music*, this situation is clearly undesirable, and so we need a strategy to account for this. As an example, let's take the following two arbitrary note sequences:

$$Seq_1 = \langle C\#, F, A\# \rangle$$

$$Seq_2 = \langle E\#, F\#, C \rangle$$

Some of these notes will clash if the sequences are played in parallel (for example, the 2nd item in each list). Additionally, if we are to adhere to a proper musical structure, we need an overall key to which all sequences should reside in.

To solve this problem, we invent the concept of a **keymap**. A keymap is a list of allowed notes, and will usually specify a scale of notes in the overall key. For example, consider the

scale of C major:

$$\text{KeyMap} = \langle C, D, E, F, G, A, B \rangle$$

We map the notes of Seq_1 and Seq_2 to the notes in KeyMap via the following algorithm:

- For each Note in Seq:
 - If Note not in KeyMap then:
 - If Note - 1 in KeyMap then:
 - Note = Note - 1
 - else:
 - Repeat until Note in KeyMap:
 - Note = Note + 1

Running the algorithm would re-map Seq_1 and Seq_2 as follows:

$$\begin{aligned} \text{Seq}_1 &= \langle C, F, A \rangle \\ \text{Seq}_2 &= \langle E, F, C \rangle \end{aligned}$$

From Signals to Music

So far, we have looked at the overall picture; individual elements of an account are mapped to individual elements of music. The next step is to discuss the activity within the processors, so that an individual account attribute is mapped to a *representative musical element*.

Each processor will receive two values; one from the same attribute in each of the two account sheets. The output musical sequence could be attributed to a member of a set of pre-set sequences. This musical sequence can then be ‘stretched’ or ‘squashed’ according to the spread of the values from the accounts, and its starting note may also be set independently of the default. For mapping account data to musical attributes, output may be as simple as a single value (such as a tempo). For example, let S be the set of types of musical sequences given by:

$$S \rightarrow \{scale_ascend, scale_descend, broken_chord_ascend, broken_chord_descend, arpeggio_ascend, arpeggio_descend\}$$

Each of these musical sequences has an attribute associated with it, which will determine how far the spread is between the start and end note. The sound of the sequence will depend on the musical attribute generated for the final movement (a minor key signature will mean that *scale_ascend* will be in a minor key, as will all members of *S*).

If we take an arbitrary account, we may end up with a musical output as described below.

Let *M* be the full musical movement (consisting of the set of musical sequences and the set of musical attributes):

$$M \rightarrow \{I, A\}$$

Let us take a set *I* to be the set of musical sequences for four fields in the account sheet:

$$I \rightarrow \{I_1, I_2, I_3, I_4\}$$

Let *start* (the starting note) and *spread* (the amount of stretching) be attributes of *I*. Let the members of *I* in our arbitrary example be defined as follows:

$$\begin{aligned} I_1 &= scale_ascend(root = A, tonic = major) \\ I_2 &= scale_ascend(root = C, tonic = major) \\ I_3 &= arpeggio_descend(root = A, tonic = minor) \\ I_4 &= broken_chord_descend(root = F, tonic = minor) \end{aligned}$$

Let *A* be the set of musical attributes for *M*:

$$A \rightarrow \{A_{tempo}, A_{key_signature}\}$$

Let the members of *A* be defined as follows:

$$\begin{aligned} A_{tempo} &= 100bpm \\ A_{key_signature} &= Bminor \end{aligned}$$

The above representation would define a musical movement consisting of four instruments. Two of these instruments are playing ascending scales (beginning at A and C respectively, and the

second jumping two tones each beat of the bar). The third will play a descending arpeggio beginning on A, and the fourth will play a descending scale beginning with F. As the key signature is defined as B minor, the notes of the scales and arpeggios will correspond to the notes in this scale. The speed this movement will be played at is 100 beats per minute.

Signal Generation

At this stage, we must step back and consider the important issue of signal generation. How exactly can we generate a signal from account attributes? To do this, we first need to define what a signal is, in the context of Financial Music.

Definition: A signal generated for an account attribute is a **ratio** of the attributes between two years with respect to direction of change.

It is easier to understand what is meant by the above definition through the use of an example.

Consider the following account:

	Current Assets	Total Assets	Current Liabilities	Total Liabilities	Total Equity
Year 1	4546723	11716362	3551852	8345658	3370704
Year 2	3769524	10607753	3200228	7403901	3203852

A ratio for an account attribute i is derived as follows:

$$S = \frac{year1_i}{year2_i}$$

In the case of the example account, we would generate a list of ratios, which would appear as the following list:

$\langle 1.2061796131288725, 1.1045093150264718, 1.1098746714296606, 1.1271974057999965, 1.0520785604328788 \rangle$

The signals can be interpreted as follows:

$S < 1 \Rightarrow$ *decreasing*
 $S > 1 \Rightarrow$ *increasing*
 $S = 1 \Rightarrow$ *unchanged*

At this point, there is a serious issue that needs to be addressed. In forming the ratio by dividing the account attribute from the first year by the same attribute from the second year, we have made the assumption that an increase between years is always desirable.

This is not always the case, and we need to look at which attributes this applies to. For these attributes, an increase in liabilities over the course of a year is considered bad, and we should adjust the ratios of *current liabilities* and *total liabilities* to reflect this. This can be done by defining a list as follows:

$$R = \langle false, false, true, true, true \rangle$$

If we using this list as we are generate the ratios, if the attribute's position in R is *true*, then we reverse the ratio such that for attribute i :

$$R_i = false \Rightarrow S_i = \frac{year1_i}{year2_i}, \quad R_i = true \Rightarrow S_i = \frac{year2_i}{year1_i}$$

Doing this now results in a correct list of ratios (the altered values are underlined):

$$\langle 1.2061796131288725, 1.1045093150264718, \underline{0.90100263186641782}, \underline{0.88715605168579881}, 1.0520785604328788 \rangle$$

These ratios are the representation of our **signals**, and will be referred to by the term 'signals' from this point onwards.

Generating a Compound Signal

In order to set a tempo and initial key, we need to generate a signal which represents the overall state of the account. This is the compound signal, and is generated by taking an average of the five signals generated from the account attributes. With a set of signals at our disposal, we can now map these to musical sequences.

Mapping Signals to Music

In this Signal Mapping approach, we have formulated the idea of 'processors' which turn a signal into a music sequence. But, how is this done?

In professional music sequencing software, a series of effects may be added to each channel to process the sound before it's heard. These effect boxes process the sound as it travels through. These processors are in effect empty boxes which can be 'skinned' with a chosen algorithm. The algorithmic skins are stored as entities separate to the processor (figure: 4.4).

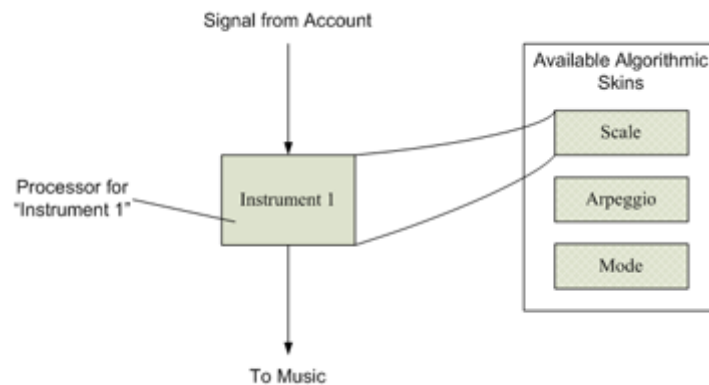


Figure 4.4: Showing how a processor can be skinned from a selection of choices.

The skins themselves have a number of field values which can be set to make the processor's sound differ from other processors with the same skin (figure: 4.5).

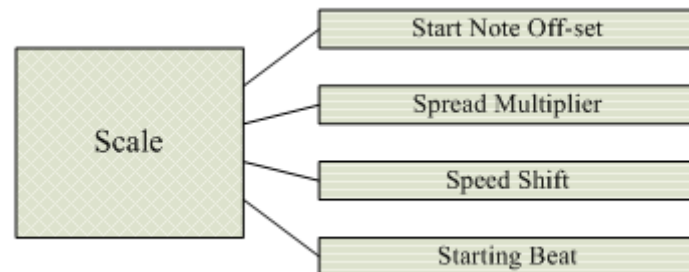


Figure 4.5: Attributes of a scale, which can be set to values in order to produce a unique sound.

This modular approach allows for each processor to choose its skin based on the nature of the input signal. For simplicity's sake, we will allow each processor to have only one skin applied to it. The processors work by having the signals 'trigger' musical sequences by passing **threshold levels**. To understand this concept, consider the following proposition for a signal S which produces musical sequence M :

$S \in \mathbb{R}^+$ $S < 0.15 \Rightarrow M = Scale$ $S < 0.3 \Rightarrow M = Arpeggio$ $S \geq 0.3 \Rightarrow M = BrokenChord$
--

Therefore, if we have an input signal strength of 0.14, we get a scale returned, as this signal strength is below the threshold for a scale (*figure 4.6*).

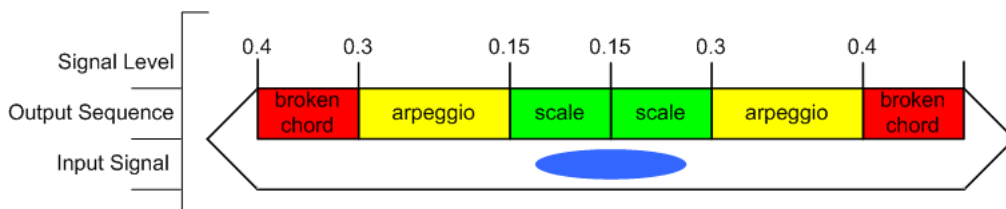


Figure 4.6: Diagram showing how an input signal will result in an output musical sequence by passing a threshold level. In this instance, the signal results in a scale.

If we have a signal strength of 0.25, this *is* enough to exceed the threshold for a scale, and therefore results in an arpeggio (*figure 4.7*).

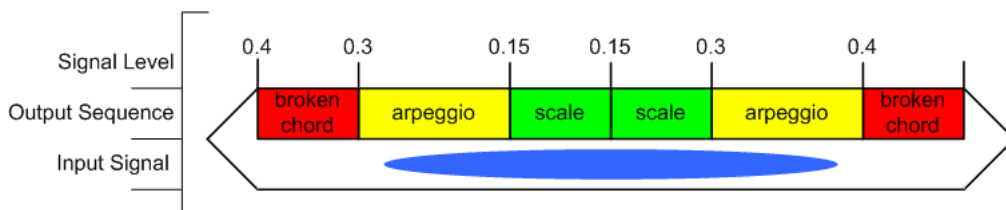


Figure 4.7: Diagram showing how an input signal will result in an output musical sequence by passing a threshold level. In this instance, the signal results in an arpeggio.

If the signal strength exceeds 0.3 threshold, we get a broken chord returned (*figure 4.8*).

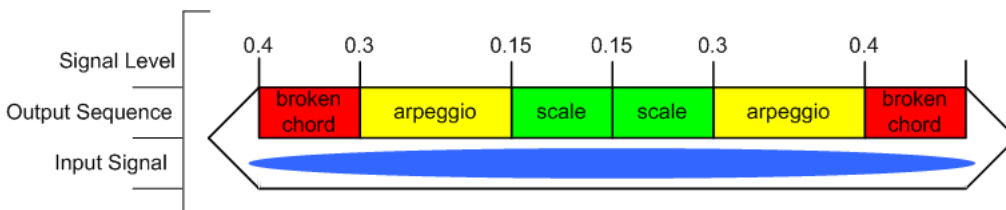


Figure 4.8: Diagram showing how an input signal will result in an output musical sequence by passing a threshold level. In this instance, the signal results in a broken chord.

Programming the Functions for Signal Generation

The following functions make up the Signal Mapping implementation:

`prepareAccounts()`: Function to prepare the accounts and return a list of pairs.

`generateSignals(accounts)`: This function generates a list of signals from two accounts. The accounts are expected to be inputted as a list of pairs.

`orderSignals(signals, referenceSignal)`: Function to order the signals from most relevant to least relevant.

`signalProcessorTempo(signal)`: Tempo signal processor.

`getOverallKey(signal)`: Derives a key signature from a signal.

`signalProcessorKey(signal, referenceSignal)`: Key signal processor. Chooses the key based on the signal. As there are 12 notes in an octave (including accidentals), our output value will be between 0 and 11.

`signalCombinator(signals)`: This combines a set of signals into one.

`signalProcessorSequence(signal, referenceSignal, signalPriority)`: Produces a sequence of notes from a signal.

`getStartingNote(signalVariance, cutOff, signalPriority)`: Function to choose a start note within a max and min bounds.

`getStartingOctave(signalPriority)`: Derives a starting octave from a signal.

`signalProcessorScale(signal, signalVariance, signalPriority, referenceSignal)`: Produces a scale from a signal.

`signalProcessorArpeggio(signal, signalVariance, signalPriority, referenceSignal)`: Produces an arpeggio from a signal.

`signalProcessorBrokenChord(signal, signalVariance, signalPriority, referenceSignal)`: Produces a broken chord from a signal.

`createFullScale(octaveTemplate)`: Function to produce a complete scale for 12 octaves

`shiftKey(octaveTemplate, keyShift)`: Function to shift the key.

`getKeyMap(octaveTemplate, keyShift)`: Function to produce a map of notes for a given key.

`mapToKey(musicalSequence, keyMap)`: Function to force a note sequence to map its self to a keyMap.

There is a `main()` function which is invoked when the program is first run:

`main(argv)`: The `main(argv)` function which is invoked when the program is run. It reads the accounts, and then outputs the music.

By looking at the `main()` function, we can get a good idea of the stages that the program goes through to derive music. The pseudocode for this is given below:

- `main(argv)`:
 - Read in accounts
 - Generate signals from accounts
 - Generate `referenceSignal` from signals
 - Calculate tempo from `referenceSignal`
 - Calculate key from `referenceSignal`
 - Re-order signals according to how far they deviate from `referenceSignal`
 - Initialise `musicalSequences` to empty list
 - For each signal in `signals`:
 - Produce a `musicalSequence` from signal
 - Append `musicalSequence` to `musicalSequences`
 - If parallel approach:
 - derive `overallKey` from `referenceSignal`
 - for each `musicalSequence` in `musicalSequences`:
 - Shift `musicalSequence` into `overallKey`
 - `outputMusic` = `musicalSequences`
 - Else if sequential approach:
 - Initialise `outputMusic`
 - for each `musicalSequence` in `musicalSequences`:
 - Append `musicalSequence` to `musicalSequences`
 - Send `outputMusic` to be played

The above algorithm shows that the `main()` function derives signals from the attributes. It then derives a reference signal (which is the same as the compound signal described earlier). This reference signal is used to calculate the tempo and overall key signature.

Next, we iterate through each signal and produce a musical sequence. Each musical sequence is appended to a list. We then consider whether we are using the sequential or parallel approach.

If we are using the parallel approach, we iterate through each music sequence and shift them into the overall key. If we are using the sequential approach, we keep each musical sequence in

its current key, but we append them to each other so they play in sequence. Finally, we play (or output) the music sequence or sequences.

Summary

In this chapter we have developed a primary approach towards generating music from accounts. We have seen how we can derive signals from account attribute, and match these to musical sequences.

In the next chapter, we will attempt another approach to music generation by using biological inspiration.

Chapter 5

Approach 2: L-System Music Generation

One of the biggest challenges in producing music from accounts is simply that we're using a small amount of simple data (amounts of money) to derive music, which is far more complex (sequences of notes, chords, etc). To meet this challenge, an approach is needed in which we can encourage music to "emerge" from something as simple as a company's balance sheet. To this end, we will turn to the field of **Biologically-Inspired Computing**.

Biologically inspired techniques tend to be good for solving complex problems with no obvious solution. Applying these techniques often leads to emergent properties, which is exactly what we're looking for. One such technique is that of the **L-System**.

In this chapter, we will look at using L-Systems to generate music from accounts. Two possible variations to achieving this will be looked at, with the most promising approach chosen for implementation and evaluation. With a working implementation, we will perform a preliminary evaluation, and then experiment with the rule generation.

Definitions

L-System:	A Biologically-Inspired parallel re-writing system.
Grammar:	Set of symbols unique to a specific L-System.
Rules:	Part of an L-System. Specifies re-writing strategies.
Axiom:	A string of characters on which an L-System operates.

An Introduction to L-Systems

L-Systems (also known as Lindenmayer Systems) were developed in 1968 by Aristid Lindenmayer. Lindenmayer was a biologist, who was trying to understand the behaviour of plant cells.

During the process of his research, he developed the idea of “an axiomatic theory of biological development”¹, which became known as the L-System. Nowadays, L-Systems have applications ranging from modeling the physical structure of plants to computer graphics. They can also be used to generate music.

L-Systems allow the re-writing of a string of characters based on a set of rules. An L-System has a **grammar**, **rules** and a **start axiom**. An L-System G is formally defined as follows:

$$G = \{V, S, \omega, P\}$$

V is a set of **variables** (replaceable symbols), and S is a set of **constants** (non-replaceable symbols). P are the **re-writing rules** of the L-System, and ω is the **start axiom**. Rules given in P will be applied to ω each time the L-System is run.

A common example of an L-System in action is one which generates the **Fibonacci Sequence**. The Fibonacci Sequence is a sequence of numbers with very special properties. The first Fibonacci number is always 0, and the second Fibonacci number is always 1. After that, each subsequent number is defined as the sum of its two predecessors. Below is the definition of this L-System:

$$\begin{aligned} V &= \{A, B\} \\ S &= \{\} \\ \omega &= \langle A \rangle \\ P &= \{(A \rightarrow B), (B \rightarrow AB)\} \end{aligned}$$

Each time we run this L-System, we work through all the symbols in ω , turning each ‘A’ into a ‘B’, and each ‘B’ into an ‘AB’. If we count the length of the axiom ω , we get the next number in the Fibonacci Sequence:

start: A (axiom length = 1)
 1st run : B (axiom length = 1)
 2nd run : AB (axiom length = 2)
 3rd run : BAB (axiom length = 3)
 4th run : ABBAB (axiom length = 5)
 5th run : BABABBAB (axiom length = 8)
 6th run : ABBABBABABBAB (axiom length = 13)

¹http://www.biologie.uni-hamburg.de/b-online/e28_3/lsys.html

From these simple rules, we witness a more complex property emerging. But, what happens if we define our grammar as something that can be interpreted musically?

Design

Recall that with the **Signal Mapping**, we derived signals from an account. These signals represented changes. We mapped then these signals directly to musical sequences. This time, we will instead use these signals to drive an L-System to generate music.

Two options are open for consideration. One option, is to use the accounts to generate the *rules* of the L-System. The alternative is to use the accounts to generate the *starting axiom*, and apply a manually crafted set of rules. Let's consider the merits of these two approaches.

Variation 1: Dynamic Rule Generation

If we decide to use the account to generate the rules, we are having to generate both the variables and the constants. Therefore, the process of generating the rules from the account must consider both musical structure (the constants in S) and reduction strategies (the variables in V). As we are generating music, we should therefore attempt to hold as much control over the constant symbols of S as possible, as it is these elements that will directly translate into music.

We could be more efficient by simply using the accounts to generate the variable symbols, and define the constant symbols ourselves so that they conform to musical structures. In doing so, we may as well separate the constants and variables from one another, placing these variables in the start axiom. Doing this has the added advantage of resulting in a dynamic start axiom that will be unique for each account.

By this logic, we can conclude that Dynamic Rule Generation is *not* the most efficient approach to use. But, by simplifying this idea we have derived a better approach; that of **Dynamic Axiom Generation**.

Variation 2: Dynamic Axiom Generation

Recall that instead of generating the rules from the accounts, we have chosen to manually define the rules to conform to sensible musical sequences. This way we can be assured that the L-System will produce musical sequences that will conform to what we understand to be *music*

rather than a chaotic collection of notes.

Now, consider that we now use the accounts to directly generate the start axiom. This will result in a unique axiom for each account, which when operated on by the L-System will produce an equally unique piece of music. The question then becomes, how can we use data in the account to generate an appropriate starting axiom?

The signal generation from the *Signal Mapping* implementation presents us with a way of evaluating the health of an account, and we can build on this idea with an L-System. We can divide the potential spread of signal values into discrete grades (remember that a signal strength above 1.0 indicates an increase, and below 1.0 indicates a decrease). For example, let's say we try a six grade system:

$$\begin{aligned} A &> 1.25 \\ 1.25 &\geq B > 1.15 \\ 1.15 &\geq C > 1.05 \\ 1.05 &\geq D > 0.95 \\ 0.95 &\geq E > 0.85 \\ F &\leq 0.85 \end{aligned}$$

By doing this, we map signal ranges to symbols. If we include these characters in the grammar of our L-System as replaceable symbols, we can define the set of variables V as:

$$V = \{A, B, C, D, E, F\}$$

An arbitrary account may therefore result in the following starting axiom²:

$$\omega = \langle C, D, A, B, B \rangle$$

At this stage, we have successfully defined two parts of an L-System: The set V of variables, and we can derive the start axiom ω from an account. The next step is to carefully choose a set of symbols which can be interpreted as music. These symbols will form the constants of S in our L-System. It is at this stage that we need to decide on an appropriate grammar.

²Note that although usually L-Systems uses a string to represent its axiom, I have chosen to use a **list** of **characters**. In the implementation, we will see that a list is easier to manipulate in Python.

Defining a Grammar

If we look at music from the point of view of how a note in a sequence relates to its predecessor, then a note can be in one of three states; higher, lower or unchanged. We can use this to begin defining the set of variables V .

Let 'u' = raise note by a tone Let 'd' = lower note by a tone Let 's' = sustain note
--

Given a starting note, from these three symbols we can generate a sequence of notes. However, this grammar is quite limited, and we would like to expand it so it can better represent many more musical aspects such as chords, and key changes. To this end, we can expand the grammar with the following additions:

Let '/' = Raise note by two tones Let '-' = Lower note by two tones Let 'r' = Don't play anything (rest) Let '.' = Increase overall key by a semi-tone Let ',' = Decrease overall key by a semi-tone Let 'j' = Shift into a major key Let 'n' = Shift into a minor key
--

It would also be nice to add a harmony to complement a melody (a harmony in this case is a note played n semi-tones above the note of the melody):

Let '+' = Turn on harmony Let '-' = Turn off harmony

With this, we have a full grammar for the set S of variables:

$S = \{ 'u', 'd', 's', '/', '-', 'r', '.', ',', 'j', 'n' \}$
--

Adding Stacks to the Grammar

In computer science, a **stack** is a primitive (yet invaluable) data structure which only allows items to be added and removed from its top. Therefore the first item into the stack will be the last item removed, and vice-versa.

Why would we need to make use of stacks in *this* L-System? Without a mechanism to jump back to earlier points in the sequence, the music would meander up and down, with no specific points to break it up. Adding a stack brings more musical structure, and allows different account attributes to affect parts of the music independently.

Many L-System implementations have built in support for the use of stacks. So, when we play the music from an axiom produced by this L-System, if we encounter a stack “push” symbol (which looks like '['), the program will record the current note, chord and key signature. It then continues playing the music until it reaches a stack “pop” symbol (which looks like ']'). When this happens, the program resets the note, chord and key signature to its earlier state, and continues playing the axiom.

Let '[' = Push current state onto stack
 Let ']' = Pop current state from stack

Interpreting the Grammar

Turning the grammar in the axiom into music is a simple matter. Let *note* be the current note, *tonic* be the tonic (root note of the current key) and *scale* be the scale (major or minor) of the current key. Let *harmony* be whether a harmony note is being played or not.

$note \rightarrow \{21 \leq note \leq 108, note \in \mathbb{N}\}$
 $tonic \rightarrow \{21 \leq tonic \leq 108, tonic \in \mathbb{N}\}$
 $scale \rightarrow \{major, minor\}$
 $harmony \rightarrow \{true, false\}$

We initialise *note*, *tonic*, *scale* and *harmony* to values of our choosing, and begin to process the axiom into *MIDI* values. Chords are calculated by playing a triad from the current key (notes 1, 3 and 5 of the scale)

An Example in Action

Let's look at a simple parsing example. Consider the following axiom ω generated by our L-System after a few runs:

$$\omega = \langle 's', 'u', 'u', 'd', '[', 'u', 'u', 'u', 'u', ']', 's', 'd', 'd', 'd' \rangle$$

If we set our starting note to Middle C (MIDI value of 48), the above would transcribe the following sequence of notes:

48, 50, 52, 50, 52, 54, 56, 58, 50, 48, 46, 44

Selecting the Rules

With the axiom being generated by the account, the rules of the L-System must be defined manually. As we want our generated sequences to sound as musical as possible, we should ensure that our rules define short sequences with proper musical structure. To do this, we enforce some constraints when defining these short sequences.

To begin with, we can choose a **time signature**. Doing so means that we should have a consistent amount of notes in each sequence. For example, a time signature of 4 beats per bar would mean that the number of notes in each sequence must be divisible by 4.

So, we as a result of this, we might define the rules for our six-grade system as follows:

($A \rightarrow j..uuuu$)
 ($B \rightarrow [..suud]$)
 ($C \rightarrow uuu_uuu_$)
 ($D \rightarrow ddd/ddd/$)
 ($E \rightarrow [, , sddu]$)
 ($F \rightarrow n, , dddd$)

This defines some suitable musical sequences, which are appropriate for the account signal ranges they're representing. However, you will notice that after one iteration of the L-System, the axiom will be fully reduced (ie, will consist entirely of non-replaceable symbols).

To complete the rule set, we need to keep producing replaceable symbols at each iteration. This might be done by modifying the rules to appear as follows:

$(A \rightarrow j..uuuuB)$ $(B \rightarrow [..suud]D)$ $(C \rightarrow uuu_uuu_A)$ $(D \rightarrow ddd/ddd/F)$ $(E \rightarrow [, , sddu]C)$ $(F \rightarrow n, , ddddE)$
--

The way in which the L-System produces music is often surprising, and it is not immediately obvious how to place the variables within the rules. On one hand, we want to have the music mimic the account as closely as possible. On the other hand, we want to encourage the emergence of interesting music from the L-System.

To help us choose which replaceable symbols to use, I propose that we can visualise members of V occurring in P as a **Finite State Machine** (FSM). In this way, when we define the set of rules P , we can see how the L-System will reduce the axiom ω (*figure 5.1*).

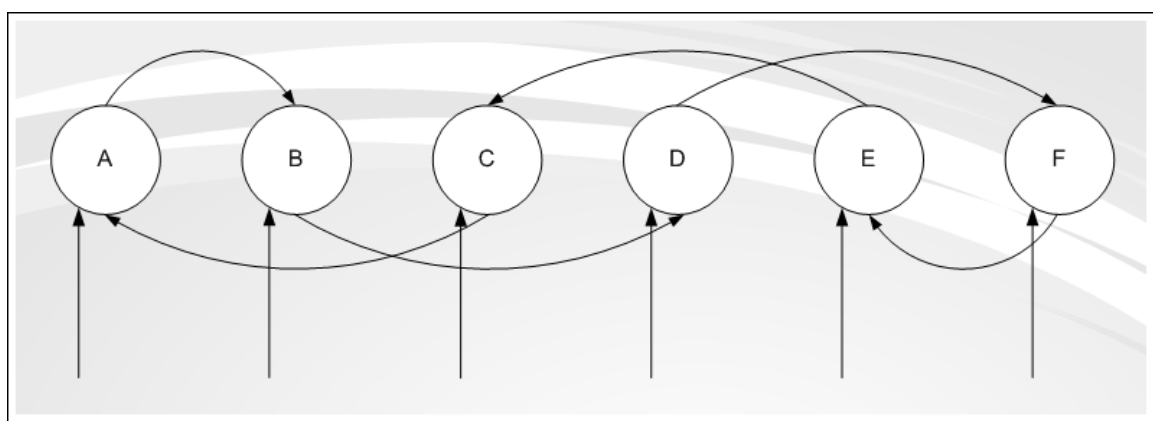


Figure 5.1: A Finite State Machine showing a possible application of replaceable symbols within the rules of an L-System.

From *figure 5.1*, we can see that our rules are probably not all that sensible. For example, A reduces to a string containing D . This leads to more varied music being generated, but prevents a true impression being given of an account. A more sensible strategy is given in the *figure 5.2*.

This gives us the following rules:

$(A \rightarrow j..uuuuB)$ $(B \rightarrow [..suud]C)$ $(C \rightarrow uuu_uuu_C)$ $(D \rightarrow ddd/ddd/D)$ $(E \rightarrow [, , sddu]D)$ $(F \rightarrow n, , ddddE)$
--

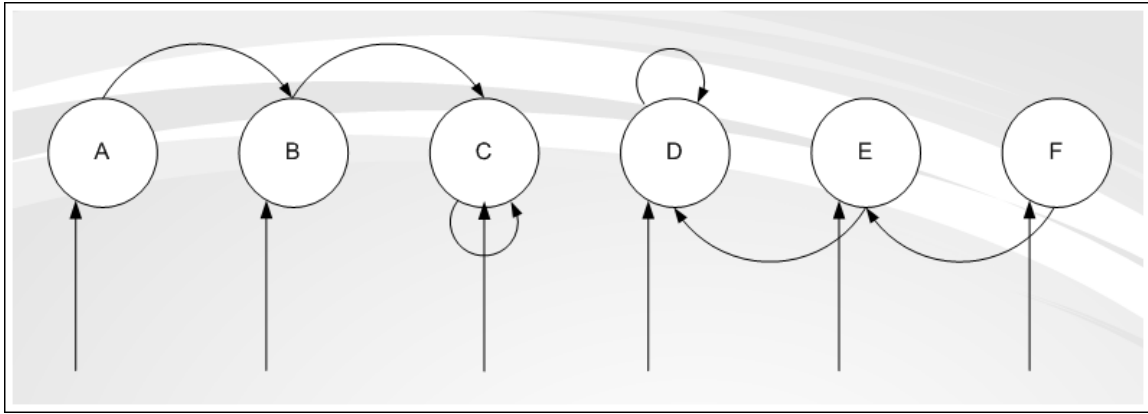


Figure 5.2: A Finite State Machine showing a more sensible application of replaceable symbols within the rules of an L-System.

Selecting the Number of Iterations

With all parts of the L-System now defined, there remains one important question: How many iterations should the L-System run for?

To help answer this question, we might choose to think about the overall length of the final piece of music. If we can make a decision on roughly how many beats in total the music should play for, then we can arrange the program to keep running the L-System until the axiom length meets this requirement.

There is a good reason for taking this approach. As the listener will be paying close attention to the music, they may suffer from listening fatigue³, so the music should not play for too long.

Programming Functions for an L-System

The first step of implementation is to program a set of functions which can represent an L-System. By keeping as close to the formal definition as possible, and by making use of *Python*'s lists and tuples, we define the following functions:

³Listening Fatigue is a lapse in concentration caused when the brain has to work hard to interpret what it's hearing. During the evaluation, testers will have to listen to lots of these musical sequences. Therefore, listening fatigue is a real concern.

`getLSystem(V, S, w, P)`: Takes in the elements of an L-System, and validates them to ensure that the L-System is properly defined. If it is, it returns a tuple representing the L-System. If not, it returns an empty tuple.

`runLSystem(lSystem)`: Takes in a valid L-System tuple and applies the rules against the start axiom. It then returns the modified L-System tuple.

`getAxiom(lSystem)`: Takes in an L-System tuple, and extracts the axiom, returning it as a list of elements.

Recall that it is common for many L-Systems to make use of stacks. Therefore, this implementation implicitly includes the stack operators “[” and “]” in the set of constants.

The L-System functions are kept separate from the rest of the L-System implementation, and reside in a function library `Linden.py`, which is imported into the main implementation.

Programming the Axiom Generator

The bulk of the implementation for the L-System generator resides in a new function library called `LSS.py`. Once again, we import `Settings.py` (which contains all the settings) and `Shared.py` (which contains functions shared across implementations).

For the most part, the implementation directly mimics the design strategy discussed so far in this chapter. We will begin by looking at the part of the program which generates the start axiom. Consider the following functions:

`generateAxiomSixGrade(signals)`: Returns a start axiom generated by splitting signal ranges into six grades.

`generateAxiomTenGrade(signals)`: Returns a start axiom generated by splitting signal ranges into ten grades.

For example, we feed the function `generateAxiomSixGrade()` the following signal list: `[1.513267626990144, 1.4732345248474281, 0.6928702010968921, 0.6676837725381415, 1.4555288461538463]`. The axiom returned might be: `['A', 'A', 'F', 'F', 'A']`. (The values which specify the grade boundaries are declared in `Settings.py`).

Programming the Axiom Parser

Interpreting the final axiom generated by the L-System is more involved than generating the start axiom. Given that we have a melody, a harmony and chords (chords are made up of *three* notes), this gives us a total of **five concurrent note sequences** which are generated from the axiom. To this end, we represent each note sequence as a list of MIDI values (integers), and these lists are returned in a tuple of order 5.

The following functions are used when parsing the axiom, and are called depending on the nature of the current symbol (note that variables of set V in the axiom are simply ignored during parsing):

```
appendHarmony( outputMusicHarmony, currentNote, harmonize, harmonyKey, keyMap
): Takes in a list of harmony notes (outputHarmony) and uses the other parameters to append
the harmony note to this list. It then returns it.

getDefaultKeyMap( keyMap ): Returns a keyMap which is used by functions such as
pitchUp() to determine where adjacent notes are in the scale.

pitchUp( note, keyMap ): Increases note to the next note above in the scale.

pitchDown( note, keyMap ): Decreases note to the next note below in the scale.

doublePitchUp( note, keyMap ): Increases note two note up in the scale.

doublePitchDown( note, keyMap ): Decreases note two note up in the scale.

keyShiftUp( keyMap, shiftDistance, note ): Shifts keyMap up by shiftDistance.
Returns a shifted keyMap and a shifted note as a pair.

keyShiftDown( keyMap, shiftDistance, note ): Shifts keyMap down by shiftDistance.
Returns a shifted keyMap and a shifted note as a pair.

getChord( keyMap ): Returns a tuple consisting of the 1st, 3rd and 5th note of the
scale.

chordComparison( currentChord, lastChord ): Compares two chords. If the two
chords are the same, it sustains the chord. Otehrwise, it plays the new chord. This is used to
stop the chord being re-played each beat.

keySwitchMinor( keyMap ): Shifts keyMap into a minor key.

keySwitchMajor( keyMap ): Shifts keyMap into a major key.
```

With these functions now written and successfully tested for robustness, we have two more functions to define which complete the program:

```
produceMusic( axiom, startNote, keyMap ): Takes in an axiom, a start note and a key signature. Returns a tuple with lists of MIDI values.
```

```
main( argv ): The main() function which is invoked when the program is run. It reads the accounts, generates and runs the L-System, calls produceMusic() and then outputs the music.
```

As these two functions are so important, their pseudocode is given below. Firstly we'll look at `produceMusic()`:

- produceMusic(axiom, startNote, keyMap):
 - Initialise currentNote = startNote
 - Initialise currentChord = NULL
 - Initialise outputMusic = $\langle \rangle$
 - Initialise harmony = NULL
 - Initialise harmonize = false
 - Initialise noteStack = $\langle \rangle$
 - For each character in axiom:
 - if $c == 'u'$ then increase currentNote by 1 tone
 - if $c == 'd'$ then decrease currentNote by 1 tone
 - if $c == '/'$ then increase currentNote by 2 tones
 - if $c == '_'$ then decrease currentNote by 2 tones
 - if $c == 's'$ then sustain currentNote
 - if $c == 'r'$ then currentNote = NULL
 - if $c == '.'$ then increase keyMap's tonic by 1
 - if $c == ','$ then decrease keyMap's tonic by 1
 - if $c == 'j'$ then shift keyMap into a major key
 - if $c == 'n'$ then shift keyMap into a minor key
 - if $c == '+'$ then harmonize = true
 - if $c == '-'$ then harmonize = false
 - if $c == '['$ then push currentNote, harmonize & keyMap onto noteStack
 - if $c == ']'$ then pop currentNote, harmonize & keyMap from noteStack
 - currentChord = 1st, 3rd and 5th note of keyMap
 - if harmonize == true && keyMap in major then harmony = currentNote + 4 semitones
 - else if harmonize == true && keyMap in minor then harmony = currentNote + 3 semitones
 - else harmony = NULL
 - Append currentNote, harmony & currentChord to outputMusic
- return outputMusic

From the above pseudocode, we can see that the algorithm begins with a starting note, and a key signature (consisting of a tonic and scale). Then, for each character in the axiom, it makes adjustments to these attributes.

For each axiom processed, we count that as one beat of the bar. Therefore, after processing each character, it records the current state of the attributes to outputMusic.

The `main()` function works in a similar fashion to the Signal Mapping's implementation. Here is the pseudocode:

- `main(argv):`
 - Read in accounts
 - Generate signals from accounts
 - Generate compoundSignal from signals
 - if `compoundSignal > 1.0` then keyMap in major
 - else keyMap in minor
 - `l_system = new L-System` generated from specification in `Settings.py`
 - run `l_system` until axiom length \geq minimum specified in `Settings.py`
 - `outputMusic = produceMusic(l_system.axiom)`
 - Send `outputMusic` to be played

Preliminary Evaluation

A preliminary evaluation of this approach was performed consisting of just a few testers. The objective was to try out several rule sets and decide which one would be used in the full evaluation.

The discovery was that the sequences produced were much more musical (more complex) than the Signal Mapping approach. From this, we can formulate a testable hypothesis; That the more complex the music, the harder it will be to accurately determine the true nature of the account. (*figure 5.3*).

In arriving at this conclusion, we would use a simple rule set (such as the ones given in this chapter) for the coming evaluation.

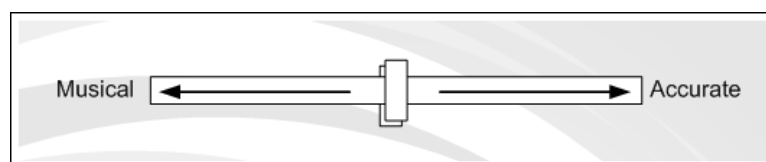


Figure 5.3: A sliding scale to show how as musicality increases, ability to accurately predict the account's health decreases. The reverse is also true.

If we also take into account issues of listening fatigue, we may also conclude that the more concentration is required to analyse the music, the harder it is to predict the account's health

(figure 5.4). Therefore, the maximum length of a musical sequence will be limited to under 20 seconds.

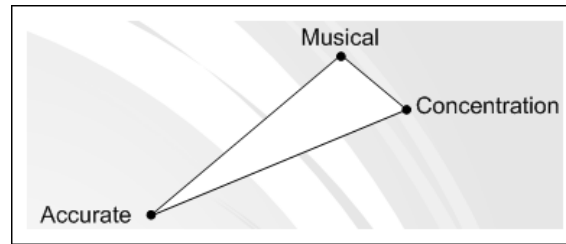


Figure 5.4: The triangle shows how increasing musicality or an increase in the need for concentration makes it more difficult to accurately predict the account.

Summary

In this chapter, we have looked at L-Systems as a way of generating music from accounts. The emergent nature of L-Systems makes them well suited to an application such as music generation.

We have also seen some techniques for generating the L-System from an account, and we have settled on a way to dynamically generate the axiom using a grading system.

Finally, we also formed a hypothesis that the more complex the music, the harder it will be to determine the true nature of the account. As we will soon see, the results of the evaluation will yield some surprises.

In the next chapter, we will fully evaluate this implementation, along with the Signal Mapping approach of the previous chapter.

Chapter 6

Evaluation of the First Two Implementations

“Inspiration may be a form of super-consciousness, or perhaps of sub consciousness, I wouldn’t know. But I am sure it is the antithesis of self-consciousness.” - Aaron Copland

In this chapter, we will evaluate the Mapping and L-System implementation with a carefully considered strategy. This strategy will place special emphasis on the main problem posed by this project; that **music is subjective**.

How Can an Objective Evaluation be made of a Subjective Concept?

Recall that we looked briefly at the concepts of **music cognition**. From these concepts, we understood that sound is “processed” into music by a human based on their personal experience. In other words; music is subjective.

How can we describe aspects of an account that will present themselves through the generated music? How can we ensure that these aspects are described in a way that *both* represent the aspects of the account *and* corresponding aspects of the music that relate to the account?

To help answer this, a simple analysis of publications such as *The Financial Times* is called for. A browse through will reveal a surplus of buzz-words which can be used to describe aspects of an account (figure: 6.1). An accountant who is looking at an accountant will develop an overall impression of an account. They can then describe it to a novice using a selection of these words. A novice, with little understanding of accountancy, can then understand the overall state of the account. These buzz-words are both accessible to an expert accountant, and a novice.



Figure 6.1: This word cloud shows how frequently some buzzwords occur in financial articles. The larger the font size, the greater the word’s frequency.

As an example, consider a company with vastly increasing debts and vastly decreasing profit. We give this company’s balance sheet to an expert to analyse. Their experience allows them to perceive the dire state of this company, and choose the word “plunge” to describe the company’s state.

Now consider that we generate music from this account using one of the strategies developed in previous chapters. This music is played to someone who has not seen the balance sheet. What they might hear is a lot of descending scales in a minor key quickly moving downwards. They may well also choose to describe this as a “plunge”. It is this **synchronisation** of descriptions that we are looking for when we evaluate the approaches.

Therefore it is these buzz-words that will be the key to bridging the descriptive gap between the accounts and the generated music.

Human Factors

In the previous chapter, we briefly mentioned the issue of **listening fatigue**. If the tester’s concentration lapses during the testing, they will have the opportunity to play a sequence again. They will also have the opportunity to take breaks during the testing process. Additionally, sequence length are limited to a maximum of 20 seconds, and the whole testing process is intended to take no longer than 20 minutes.

Preparation of Test Data

Fifteen accounts were sent to the expert to be evaluated. From these, **six were chosen**. Of these, **Three** were selected which displayed mostly **desirable aspects** (good investment proposition), and **Three** were selected which displayed mostly **undesirable aspects** (bad investment proposition).

From the approaches, **four methods** of generating music were chosen. From the **Signal Mapping** approach, music was generated both in **sequence** and in **parallel**. And, from the **L-System** approach, music was generated firstly with **strings and piano** and then with **piano only in staccato**.

This results in a total of 24 generated musical sequences which were prepared for evaluation. These musical sequences were then **shuffled randomly**.

Designing an Evaluation Strategy

With most testing strategies, a baseline is needed which can be used as a fixed point for comparing data. For this, an accountant was asked to analyse 16 accounts (A sample of the questionnaire that the accountant received can be seen in *figures 8.8, 8.9 and 8.10* of the appendix).

The accountant was asked to choose a selection of terms to describe the *account*, and were also asked to state whether they would invest in the company based on what they'd seen (they could select 'yes', 'no' or 'unsure').

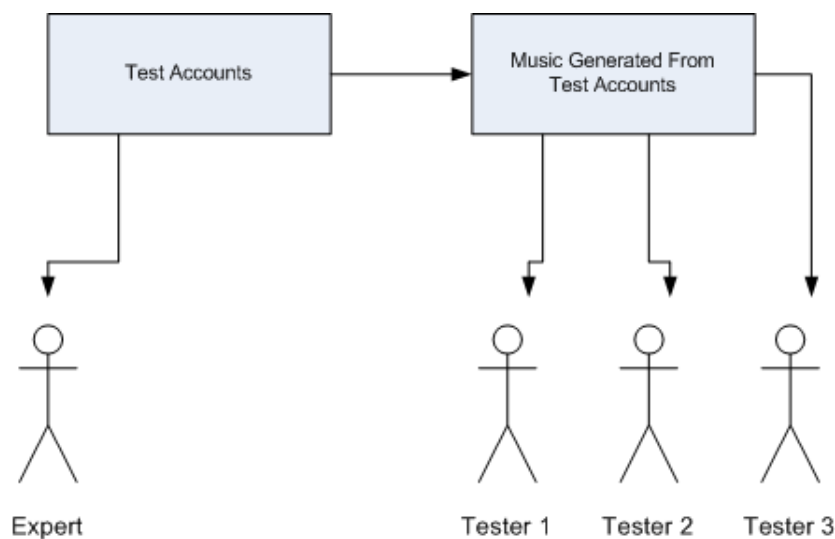


Figure 6.2: A diagram showing how the evaluation process is organised.

For the testers who would be listening to the music, they will have to choose a selection of terms to describe the *music*, and also make an assessment as to whether they'd invest in the company or not (A sample of the questionnaire that the testers received can be seen in *figures 8.11 and 8.12* of the appendix). The testers were all students, but from a variety of academic disciplines.

The testers were also asked to assess the 'musicality' of each music sequence they listened to, giving it a rating from 1 (not musical) to 5 (very musical).

The motivation behind this strategy is that both the accountant and testers are using the same terminology to evaluate the account.

The tester was provided with the following information before testing:

1. That the music is generated from company accounts by a computer.
2. That the music is intended to reflect the state of the account.
3. That the test has no 'right' or 'wrong' answers.

The testers are *not* aware of the following details:

1. That the music they are listening to is generated by four different methods
2. That there is actually only a total of six accounts, not twenty-four.

The testers are left to assume that each piece of music represents a unique account (although the more perceptive testers may well have guessed that this was not the case) The above approach attempts to counter the issue of subjectivity by doing the following:

1. Taking a baseline analysis of the account by asking the opinion of an 'expert' (an accountant). This baseline will be used as the origin for comparison with other test results.
2. The tester is intended to be kept in the dark that they are listening to each account four times over.
3. The random ordering help remove any bias if the tester does suspect that they're going over the same accounts twice.
4. The random ordering help evenly disperse statistical noise caused by listening fatigue.

Results Analysis

Before beginning analysis of the results, we need to clarify the terms used in this section:

The Expert	A practicing chartered accountant who analysed the accounts.
The Tester	One of several individuals who evaluated the musical output.
Output type	One of the four music generation strategies.

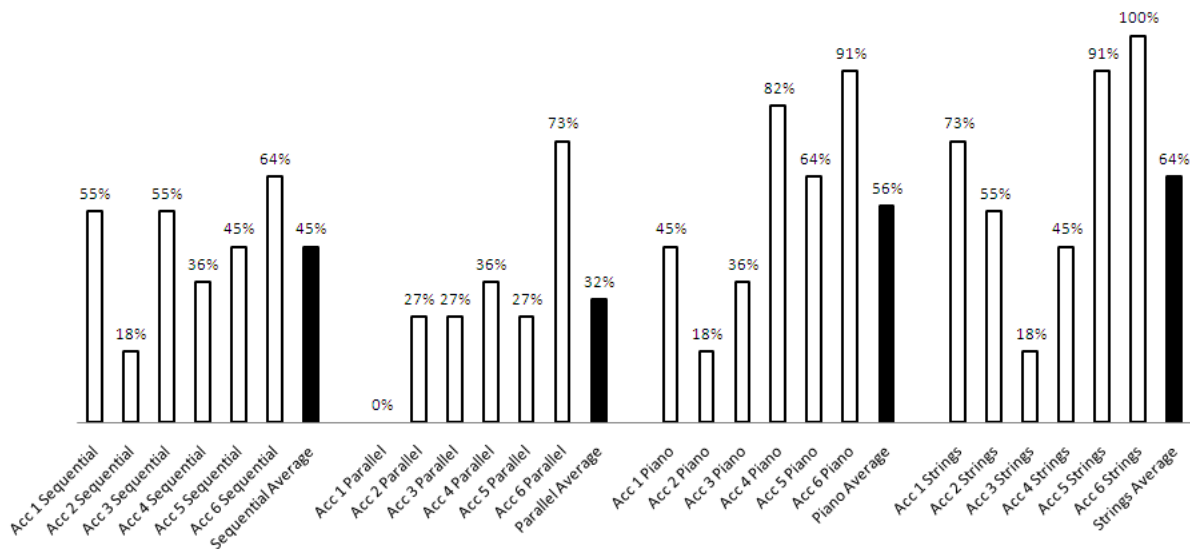


Figure 6.3: A graph showing the average amount that testers agreed with the expert's decision whether to invest in an account, not invest in an account or to remain undecided. (note that 'strings' and 'piano' are L-System generated sequences)

The challenge of the evaluation was to deduce how often the testers agreed with the investment decisions of the expert. *Figure 6.3* shows on average how much agreement there was (recall that the choice was one of *three categories*: Invest, don't invest or undecided).

Looking at the averages for each of the four output types, we see that the two L-System implementations (Strings & Piano) were more successful than the two Signal Mapping implementations (Sequential & Parallel).

We can gain more insight if we look at how spread out the opinions of the testers were for each of the output types. This can be done by calculating their standard deviations:

$$\begin{aligned} \sigma_{Signal_Mapping_Sequential} &= 0.162623126 \\ \sigma_{Signal_Mapping_Parallel} &= 0.235312347 \\ \sigma_{L-System_Piano} &= 0.305595206 \\ \sigma_{L-System_Strings} &= 0.30424001 \end{aligned}$$

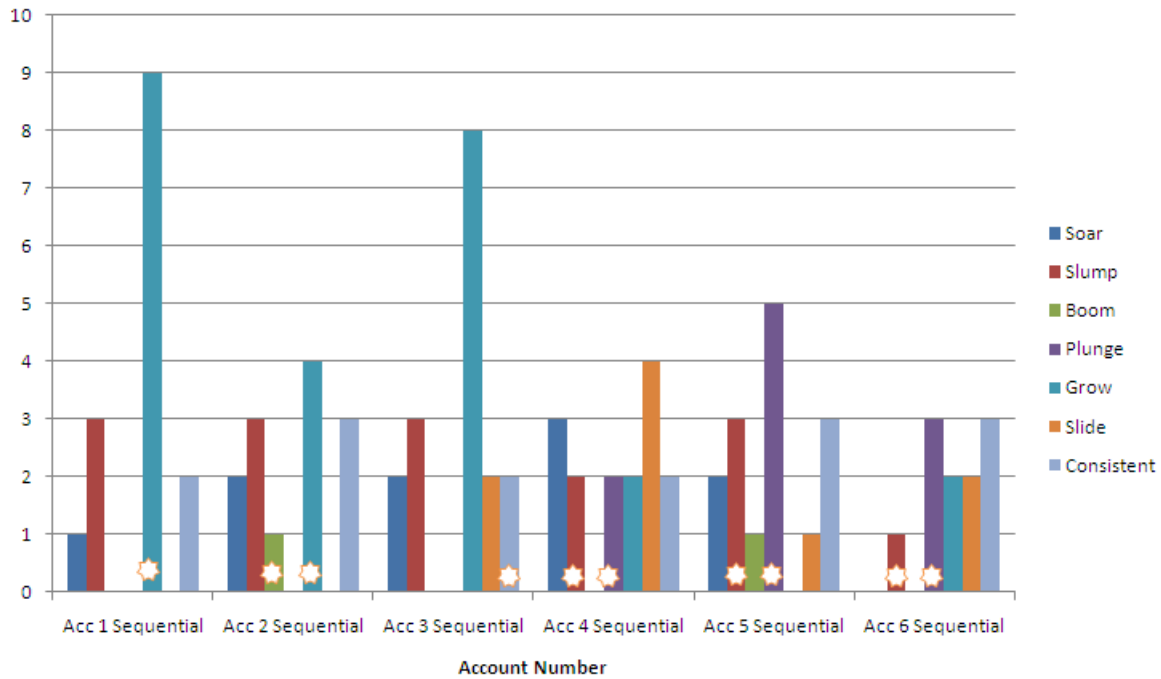


Figure 6.4: A graph showing how many times testers selected buzzwords for each account in the Signal Mapping Sequential output. A star designates that this option was selected by the expert.

We can see that the Signal Mapping outputs show less of a spread than the L-System outputs. From this we might deduce that the L-System output types were more ambiguous than the Signal Mapping output types, and therefore the tester’s investment choices were more varied. Why might this be? Consider that the Signal Mapping output types were composed of simple musical sequences (scales going up, etc). These movements are clear and were easier to interpret by the testers.

The L-System output types were generated from rules, and therefore produced sequences which were not as clear cut as an ascending scale might have been. This would mean that it was more difficult to arrive at a conclusion as to the state of the account.

That said, if the L-System output types were more ambiguous than the Signal Mapping output types, then why would it be that the L-System approach would fare better? Intuitively, we might assume that simpler musical sequences would be easier to interpret. In order to solve this mystery, we need to investigate further.

Recall that we used descriptive ‘buzzwords’ as a way of evaluating the state of an account. Both the expert and the testers selected one or more words to describe each account (or account’s

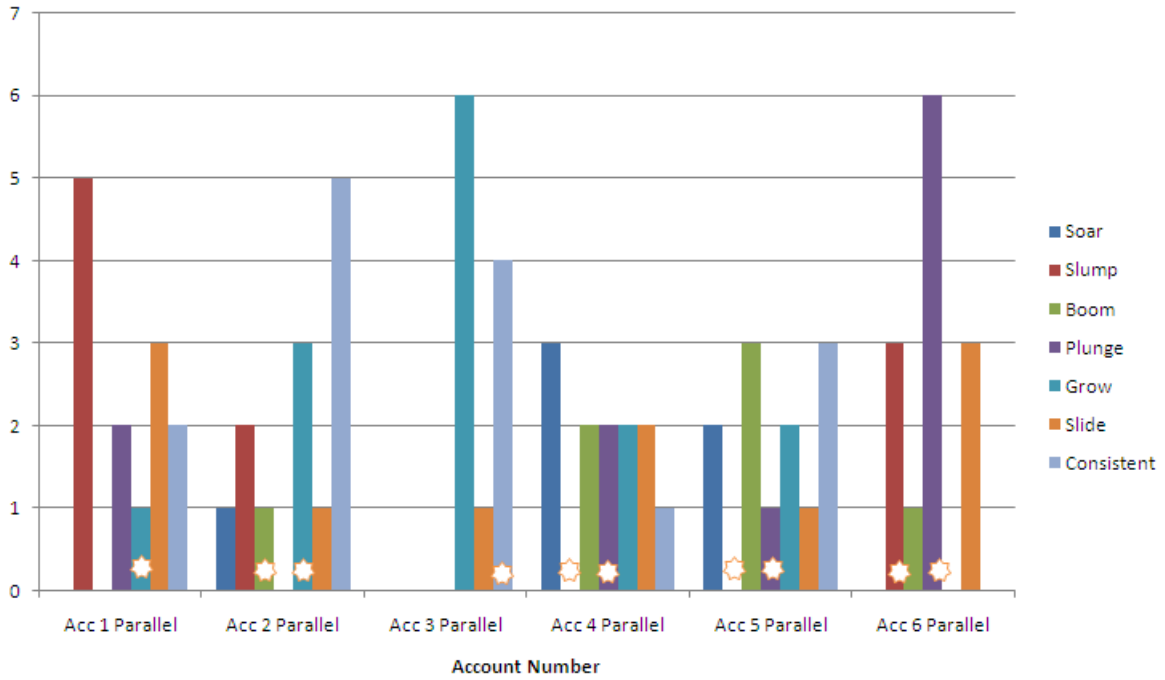


Figure 6.5: A graph showing how many times testers selected buzzwords for each account in the Signal Mapping Parallel output. A star designates that this option was selected by the expert.

output sequence) based on their perception. From *figures 6.4, 6.5, 6.6 and 6.7* we can tell how often the testers chose the same descriptive words as the expert.

What we find with the two Signal Mapping implementations is that the testers were poor at correctly picking the same buzzwords as the expert. However, with the L-System implementation, the testers were better at selecting the same word.

As *both* of the two implementations are based on the same derived signals, we can trace the difference down to the L-System processing. With the Signal Mapping implementation, musical sequences are produced mechanically by performing calculations. With the L-System implementation, we have a more organic process. This process appears on the surface to produce more complex music, but in fact we would suggest that biologically-inspired techniques are better able to tune the music towards the human ear.

In terms of the average number of listens needed by each tester for each output type (*figure 6.8*), there is an approximate correlation between the length of the sequence, and the number of listens needed (*figure 6.9*). For example, a very short sequence (3 seconds) sometimes requires more listens than a longer piece (20 seconds). This is of interest to us, because the tempo reflects the amount of change in an account.

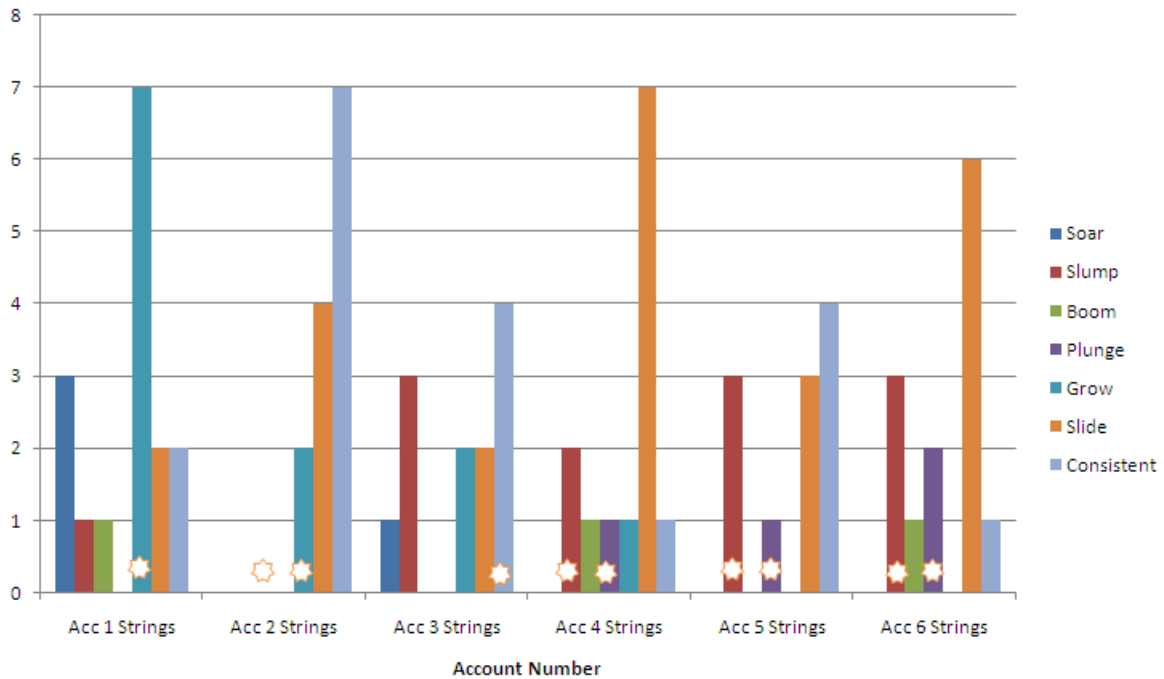


Figure 6.6: A graph showing how many times testers selected buzzwords for each account in the L-System Strings output. A star designates that this option was selected by the expert.

This is also an unusual observation, because we might intuit that a short sequence would not need a further listen because it is simpler to analyse¹. As a consequence of this, we may wish to take into account that the tempo setting may have a side effect in the way the music was perceived by the tester.

But, what about the musicality of each approach? *Figure 6.10* shows how ‘musical’ the tester considered each of the four types of output sequence to be, on a scale of 1 to 5. The two best achievers in this instance were the Sequential Signal Mapping and the Piano L-System.

Why was the prediction accuracy lower for the Signal Mapping Parallel output than it was for its Sequential counterpart? I would suggest that this is for three reasons. Primarily, we have several musical sequences playing concurrently in the Parallel output, which could make them difficult to tell apart. In the Sequential output, these sequences are played one after the other, in order of importance.

The second reason might be to do with the key shift which occurs in the Parallel output. Remember, that in order to avoid discordancy, all sequences are shifted into the same key for

¹The short-term memory model describes the way the human brain can record a small number of items in detail for up to 30 seconds. This would suggest that a shorter piece of music would be easy to hold in the mind than a longer one.

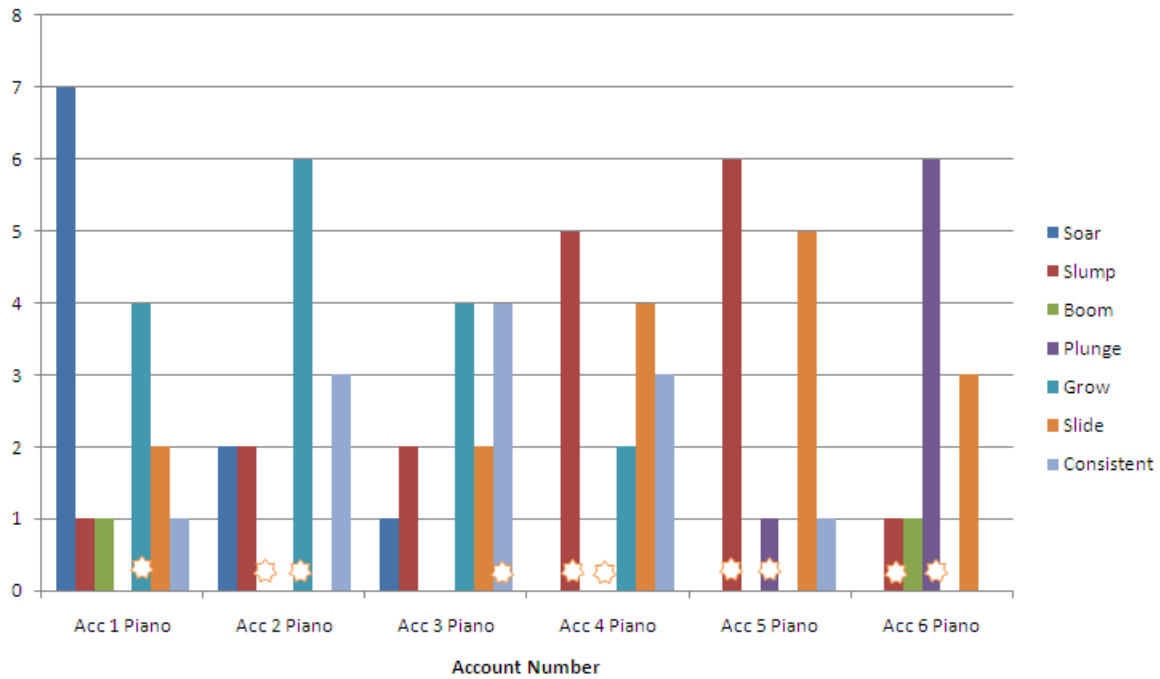


Figure 6.7: A graph showing how many times testers selected buzzwords for each account in the L-System Strings output. A star designates that this option was selected by the expert.

the Parallel output. In the Sequential output, sequences retain their own key, therefore giving them their own ‘flavour’.

The third reason I propose is that as the sequences are played one after another in the Sequential output, the tester was able to look at their relations to each other in a different way to having them played in parallel. For example, key changes stand out between parts of the sequence. This perspective is lost in the Parallel output.

Summary and Conclusions

By looking at the results, we can see that **the music does to an extent represent the accounts they’re generated from**. Additionally, we witness that the **L-System approach performs the best**, although both approaches do have their merits. We discovered that under ideal circumstances, testers will only correctly analyse the account from the music 64% of the time.

Testers considered the sequences they heard as music (ie, have a high level of ‘musicality’). This is important, as one of the objectives of this project is to generate music

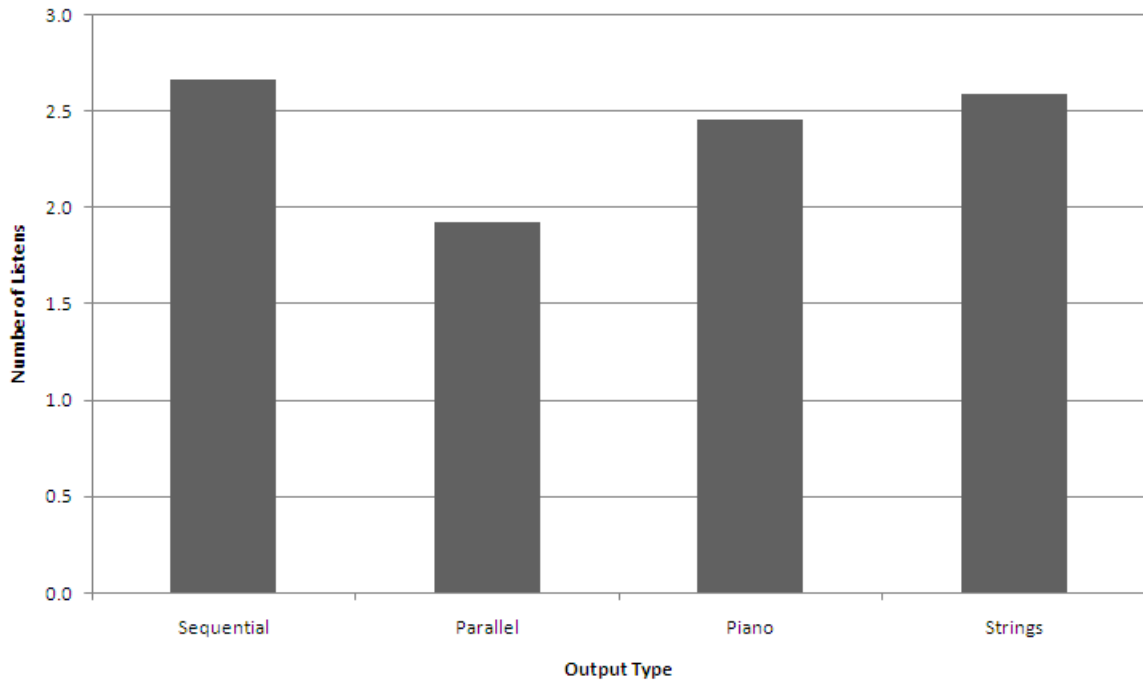


Figure 6.8: A graph showing the average number of listens needed for each output type. (note that ‘strings’ and ‘piano’ are L-System generated sequences)

from accounts. In this area, we can say that we have been successful.

We discovered that, paradoxically, adding an extra level of complexity to the music (via the L-System implementation) actually improves a person’s ability to assess the account. This implies that the emergent properties displayed by the L-System make the nature of the accounts more clear. Perhaps it shouldn’t be surprising that a biologically-inspired approach should work well when the subjects are biological!

What is really needed now is a way of formally defining the attributes in music in the same way that we can define attributes in the accounts. Achieving this would open up new avenues which would help us generate music which more closely represents the account, and it is this challenge we will attempt to meet in the next chapter.

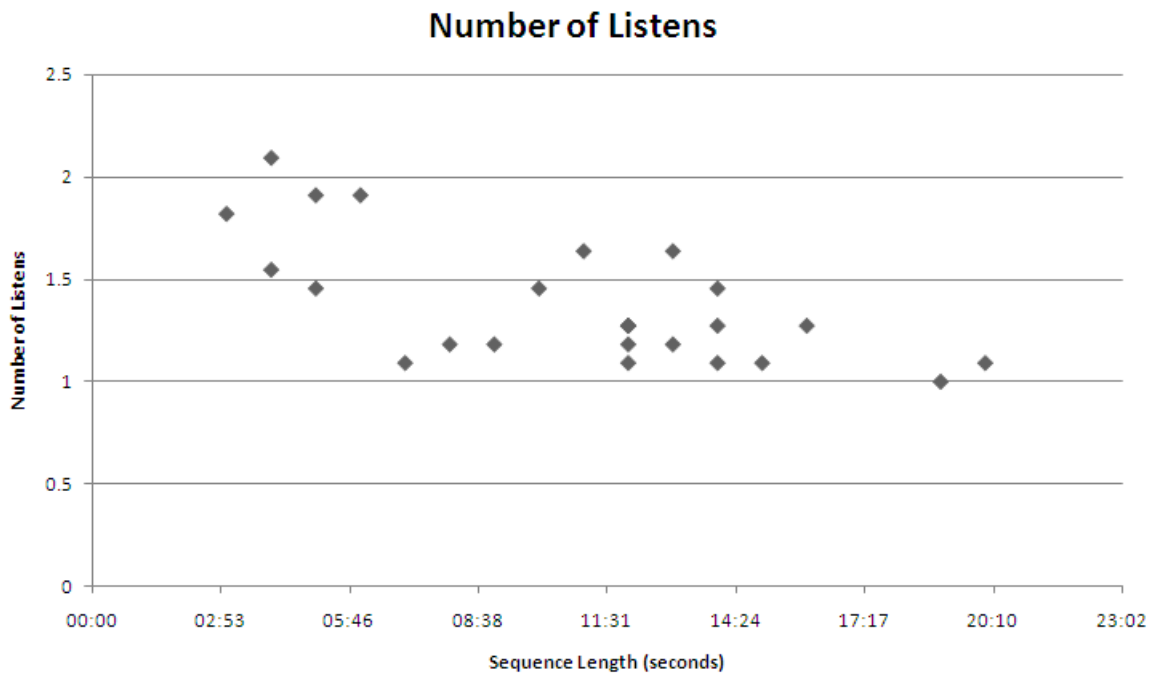


Figure 6.9: A graph showing the correlation between the length of a musical sequence (in seconds) and the average number of listens that the tester needed.

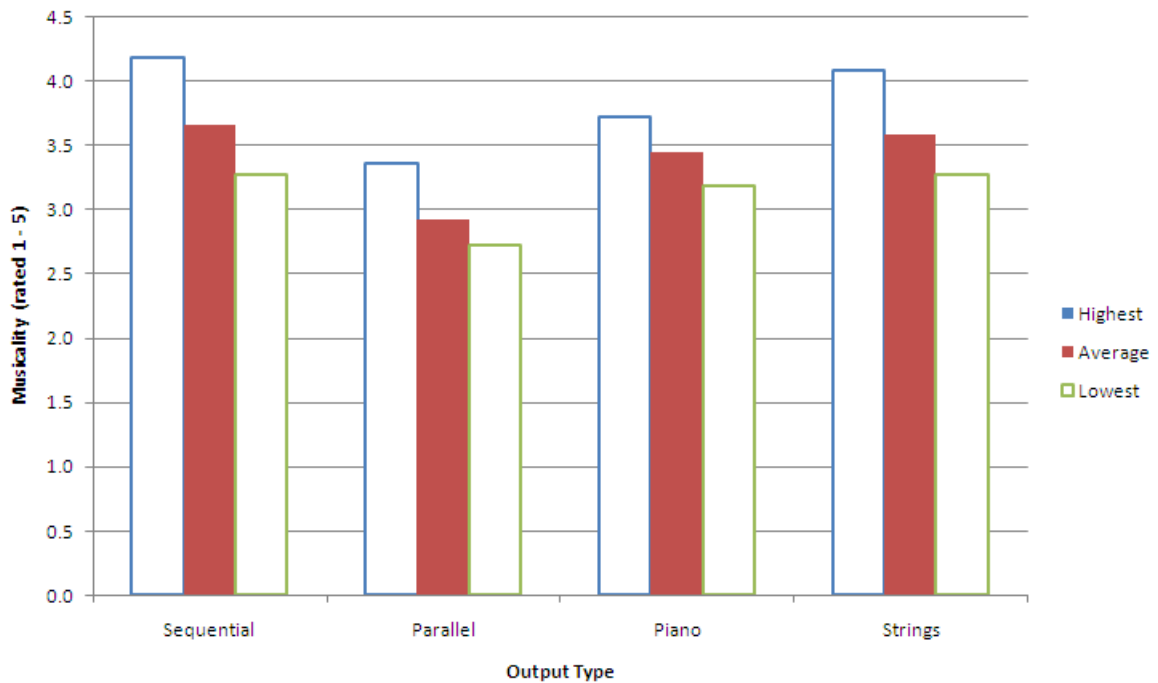


Figure 6.10: A graph showing the average spread of the musicality ratings given to each output type.

Chapter 7

Approach 3: The Financial Genome

In the previous chapter, we used a set of descriptive words to simultaneously describe features of an account and musical patterns. This technique was used to evaluate the success of the first two implementations.

In this section, we will see how these words can be used to generate a genome which represents an account. We will only go so far as to specify a way of representing this genome, and we will then define a very small genome as a way of demonstrating how this approach will work, and produce a prototype which allows some basic manipulation.

Second Case Study: The Music Genome Project

In 2006, a group of musicians developed a new way of analysing music. Their aim was to use biological inspiration to classify discrete attributes of music. They called this **The Music Genome Project**, and used this to produce the successful internet based *Pandora Music* service.¹

The idea they proposed was to have musicians listen to various pieces of music. They would analyse them by ear, and decide which musical elements (genes) made up the song.

Their ultimate aim was to be able to take the abstract ‘essences’ or ‘moods’ which make up a piece of music, and identify them as something tangible that can be used for classification. This way, when a user likes a specific song, the system can make recommendations to them based on songs with a similar genome.

¹<http://www.pandora.com/mgp.shtml>, 21st Feb 2008

Notation and Definitions

$ S $	The cardinality of S
\mapsto	Maplet (maps two elements together)
Gene	An arbitrary unit which determines a characteristic of an account or piece of music

Background

Consider that the approach developed for the *Music Genome Project* can be reversed to generate *music* from *accounts*. If we define a set of genomes, we can then use this as a midway point to map accounts to music. For example, each gene (we will assume a gene has only a single allele) will represent a particular part of some arbitrary music. It will also represent a specific feature in a given account. Added to this, is a set of rules which decide expression rules for sets of genes (transcription rules).

Let G be a single gene Let A be an account consisting of many genes
--

Using the idea of using words to represent the state of an account, we can set up a financial genome for an account as follows:

Let N_f be a sequence Let N_f be the genome for an account

By the same token, we can also have a musical genome given as follows:

Let N_m be a sequence Let N_m be the genome for music
--

As N_f and N_m are sequences, the order is important. Take for example the relationship seen in figure 7.1. As elements N_f and N_m are mapped to specific genes, both sequences need to be specified in the order such that a gene n will be expressed correctly its financial nature in N_f and its musical nature in N_m . The genome from figure 7.1 might therefore be sequenced as follows:

$$F = \{0 \mapsto \text{null}, 1 \mapsto \text{slump}, 2 \mapsto \text{soar}, 3 \mapsto \text{plunge}, 4 \mapsto \text{boom}\}$$

$$N_a = \{0 \mapsto \text{null}, 1 \mapsto \text{descending}, 2 \mapsto \text{ascending}\}$$

$$N_b = \{0 \mapsto \text{null}, 1 \mapsto \text{major}, 2 \mapsto \text{minor}\}$$

$$N_c = \{0 \mapsto \text{null}, 1 \mapsto \text{scale}, 2 \mapsto \text{crescendo}, 3 \mapsto \text{chord}\}$$

The intersection of these two sequences' enumeration produces a sequence N , which is the general genome. N_f and N_m are simply expressions (phenotypes) of $\{F, N\}$.

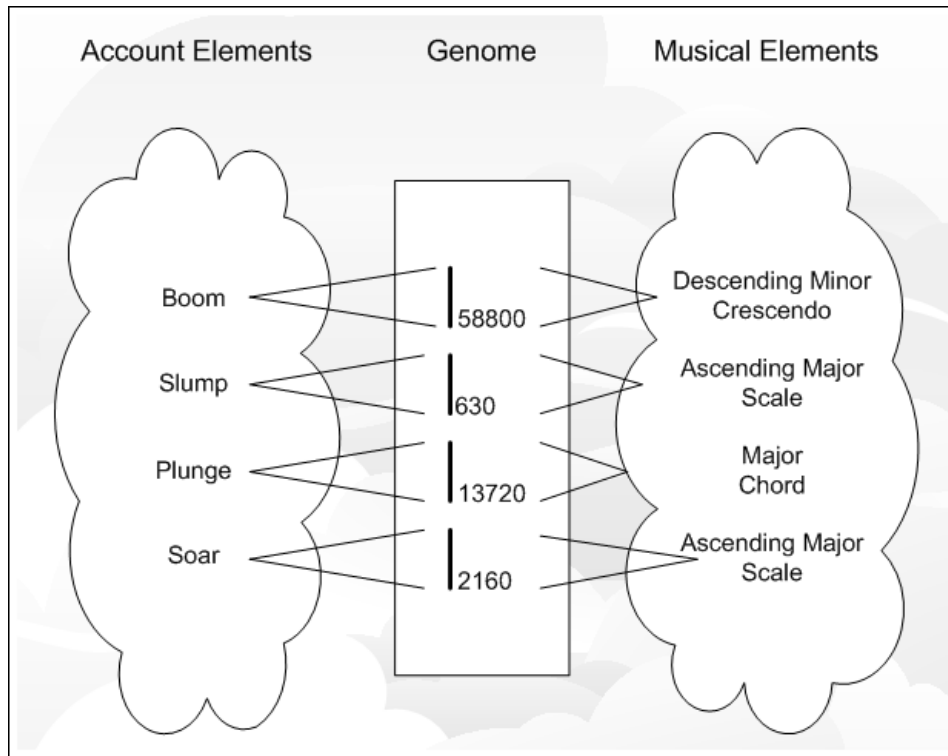


Figure 7.1: Example generated from a simple genome. The genes represent both the features of an account, and also a corresponding musical feature.

Defining Genes

At the moment, we have a limited system where by an account feature is directly mapped to a musical feature. We can now propose that this system be extended so that an account feature can be mapped to a combination of musical elements.

The first thing that needs to be done is to define rules that take raw numbers, and reduce them to a simple **account** keyword which describes its overall state.

For example, a decrease in yearly profit from 100,000 to 90,000 is a decrease of 10%. If we look this up in a 'word table', we may find that this corresponds to the word 'slump'.

We then define musical sequences in terms of words as well. In this case, I have elected to use two words to describe a musical sequence. For example, the above 10% decrease in profit corresponds to the word ‘slump’ which is linked to the same genome as the description ‘descending scale’. This fully defines a gene.

Rules for Defining a Gene

Below is a table for a specific account (and its generated music), where by each gene has a unique number:

Gene #	Account Wrđ	Music Wrđ 1	Music Wrđ 2	Music Wrđ 3
58800	Boom	Descending	Minor	Crescendo
630	Slump	Ascending	Major	Scale
13720	Plunge	NULL	Major	Chord
2160	Soar	Ascending	Major	Scale

The gene numbering may seem random upon first glance, but I will show in due course that there is a logical method, and that each combination of words results in a unique gene value. We will now go forward to explain how this genome was generated. We will use the following notation:

Let A be last years account
 Let B be this years account
 Let F be a financial buzz-phrase
 Let $W1$ be music word 1
 Let $W2$ be music word 2
 Let $W3$ be music word 3
 Let G be a gene
 Let GN the set of all possible genes for an account

We now enumerate F as follows:

Financial Phrase	Enumeration
NULL	0
Slump	1
Soar	2
Plunge	3
Boom	4

We also enumerate $W1$ and $W2$ as follows:

Musical Word	Word ID	Enumeration
NULL	W1	0
Descending	W1	1
Ascending	W1	2
NULL	W2	0
Major	W2	1
Minor	W2	2
NULL	W3	0
Scale	W3	1
Crescendo	W3	2
Chord	W3	3

We then generate a rule table, a part of which may appear as follows:

ID	Rule
1	if $A > B$ then $W1 = \text{'Descending'}$
2	if $A \leq B$ then $W1 = \text{'Ascending'}$
3	if difference between A and $B \leq 10\%$ then $W2 = \text{'Scale'}$
4	if difference between A and $B > 10\%$ and $\leq 70\%$ then $W2 = \text{'Crescendo'}$
5	if difference between A and $B > 70\%$ then $W2 = \text{'Chord'}$
...	...

From this table, $W1$ and $W2$ now have values (note, all rules must be processed until both $W1$ and $W2$ have values). We can then assign a unique identity to each gene in our genome. A **Gödel Numbering Function** is a suitable way to achieve this, and shows how each combination of words results in a unique identifier:

$$G = 2^F \cdot 3^{W_1} \cdot 5^{W_2} \cdot 7^{W_3}$$

The reason for assigning a gene's identity in this way is so that over the processing of many different accounts, common genes can be identified. These genes will always have a unique identifier, and this feature will become incredibly useful if we wish to implement machine learning based on human feedback to decide which genes are expressing themselves the best.

Invalid Genes

Clearly, some of these genes will be invalid. In the first case, situations such as 'ascending chord' simply would not make sense. You will notice that there are several *NULL* values which can be attributed to a gene. These exist simply so that we can remove any words from a musical

description without having to change the structure.²

Also, due to the nature of the numbering function used, most values will produce genes that cannot be expressed. However, by using a rule table to detect genes, we can be assured that the genome will consist only of valid genes. Any gene enumerations not present in GN cannot be generated from an account by accident.

Gene Expression

At this point, each set of accounts has a unique genome:

Let GN be the set of all possible genes in the genome (the genome)
Let GN_n be the genome for account n
 $GN_n \subset GN$

Therefore, the genome for the example in 7.1 is given as the set $GN_{example} = \{58800, 630, 13720, 2160\}$.

Implementation as a Prototype

Recall that we have a unique identifier for each gene, generated by enumerating the words and applying the enumeration to a Gödel numbering function. Because of this, we can make use of Python's **dictionary** feature. Dictionaries store data in a similar way to lists, but instead of returning an entry via it's *position* in the list, we use a key. This key will be the gene's identifier.

Here is the syntax for a gene's dictionary entry:

```
{ geneID: [ 'Financial Keyword', [ 'Direction', 'Key', 'SequenceType' ] ] }
```

Therefore, we might have a gene which is defined as:

```
{ 88200 [ 'Plunge', [ 'Ascending', 'Minor', 'Crescendo' ] ] }
```

We also need to specify which word combinations for valid genes. To do this, we define `W_VALID` as follows:

²A Survey of Evolutionary Algorithms for Data Mining and Knowledge Discovery, Alex A. Freitas, section 3.1.2: <http://www.macs.hw.ac.uk/~dwcorne/RSR/freitas01survey.pdf>

```
W_VALID = [ ( [ 'Descending', 'Ascending' ], [ 'Major', 'Minor' ],  
[ 'Scale', 'Crescendo' ] ), ( [ 'NULL' ], [ 'Major', 'Minor' ], [ 'Chord' ] ) ]
```

In `W_VALID`, each tuple consists of three items. Each of these three items is a list of words. Within each tuple, the valid genes are the set produced by the cartesian product of these lists.

We then call the `getValidGenes(GN, W_VALID)` function, which takes in our specification of valid genes, as well as the full genome (`GN`). It returns a dictionary of just the valid genes.

Gene Encoding

Encoding the genes requires a simple application of the enumeration we described above. For example, consider the following gene:

```
{ 1260: [ 'Soar' , [ 'Ascending', 'Major', 'Scale' ] ] }
```

Using the encoding we describe, this would result in the following list:

```
[ 2, 2, 1, 1 ]
```

It then becomes a straight forward matter to perform mutations on single genes, and evolutionary algorithms on gene sets. Seeing as each gene always results in a unique identifier, it is also straight forward to generate a Gödel number and locate its position in the dictionary. In this way, we can determine whether the mutant is a valid gene or not.

Gene Weightings

In addition to the current setup, we assign each gene a weight based on its fitness:

```
Let  $W_n$  be a sequence containing the weights of genes in account  $n$   
 $|W_n| = |GN_n|$   
Let  $m$  be the weight multiplier
```

So, as the set GN is built whilst processing the account, a sequence (represented by a list in Python) is built up. The sequence W_1 corresponding to GN_1 would be $\langle 1.0, 1.0, 1.0, 1.0 \rangle$ as each of these genes is detected only once.

Lets say for example, that an account GN_2 has the same selection of genes as GN_1 , but gene 900 is detected twice, and gene 54000 is detected three times. We set m to a constant value of 1.5. Each weight value is calculated as follows:

Let G_n be a gene
Let c be the number of times G_n is detected
The weight of G_n is given by $mc - 1$

In the case of GN_2 , this gives us $W_2 = \langle 1.0, 1.0, 1.5, 2.25 \rangle$

Functions of the Prototype

Below is a full list of functions employed in the prototype:

`godel(a, b, c, d)`: Gödel numbering function.

`displayGenome(GN)`: Function to display the genome.

`generateGenes(F_SET, W1_SET, W2_SET, W3_SET)`: Generate the set of all possible genes.

`getValidGenes(GN, W_VALID)`: Generate the set of all valid genes (ie, those with valid musical sequences).

`getRandomGeneSet(n, GN_VALID)`: Produce a random set of genes with n members.

`generateEncoding(gene)`: Generates an encoding for a gene.

`generateEncodings(genes)`: Generates encodings for a set of genes.

`decodeGene(encoding)`: Decodes a gene.

`decodeGenes(encodings)`: Decodes several genes.

`mutateGene(gene, validGenes)`: Performs an single-random-gene new-allele mutation given an encoding.

`mutateGeneSet(genes, m, validGenes)`: Performs an m-random-gene new-allele mutation on a set of genes.

`generateGenomeWeights(genome)`: Generate evenly distributed weights for a genome.

`increaseWeight(encodedGene, weights)`: Increase the weight of a gene.

The prototype operates by calling functions from within the Jython interpreter in order to

perform the operations.

Beyond the Prototype

Due to time constraints, the prototype was not developed beyond its current state. The next stage would have involved plugging the genome into one of the first two implementations. A population would have been generated for an account, and its fitness evaluated in relation to the music produced. A number of experts would have been required to make this a reality.

Summary

In this, the penultimate chapter, we have seen how we can use words to describe both accounts and music. By using the cartesian product, we can produce a genome set which covers all mapping possibilities of accounts to music. The evaluation of the fitness of these genes is the domain of financial experts and musicians, and these resources lie outside the scope of this project.

Over the previous chapters, we have looked at two full implementations and one prototype. In the next chapter, we will sum up and see what conclusions have been reached from this project.

Chapter 8

Conclusions

We now arrive at the end of the project, and in this final chapter we will reflect upon what we've discovered, and how future projects could continue the work.

Discoveries and Achievements

At this point, it is pertinent to ask the question “What have we discovered from this process?”. To begin with, we discovered that it *is* possible to generate music from accounts. More so, we found that by refining techniques, we could get the music to approximate the account's features, so that a listener could make a guess at the state of the account. We also discovered that a biologically-inspired approach worked much better than one which came from a purely mathematical direction.

With the conclusion of this dissertation, we have successfully managed to generate music from accounts. The evaluation implies that one of the approaches is over 60% effective in representing the account's true nature. Recall that these were the two main aims of the project.

Project Evaluation

The purpose of this section is to clear up any ambiguities for those readers who followed this project through its first and second deliverables (The final document you are reading now would be classed as the third deliverable).

By Deliverable 2, two implementations were proposed, namely the Signal Mapping and the Financial Genome. During the course of the implementation, the L-System Music Generation idea was formulated, and showed enough promise to warrant further exploration.

In the end, a full implementation of the L-System idea was produced, and evaluated alongside the Signal Mapping. Results of the evaluation showed that developing the approach was worth the time and effort put into it.

Meanwhile, the Financial Genome approach was showing signs of being too ambitious for an undergraduate dissertation. The design process continued to progress from the second deliverable, but a fully developed prototype was not able to be produced in the time available. Moreso, the ambitiousness of the idea would have made evaluating such a prototype very difficult without greater resources. In the end L-System implementation was pursued at the expense of the Financial Genome approach.

However, the Financial Genome idea was not abandoned by any means. This resulted in a greater overall quantity of work being performed than was originally proposed at the time of Deliverable 2 (three ideas instead of two). Unfortunately, the Financial Genome was not as fully developed as it was intended to be.

Future Work

The basic idea proposed of generating music from accounts is a vast one, and one with huge potential. During the course of this project, we have developed some preliminary ideas, but there is so much more work that could be done in this area. In this section, various avenues for future work are proposed. Approximations of time scales are given with each proposal. (A **medium** timescale should be considered to be the same amount of time spent on this dissertation.)

User Interface

- 1) Development of a user interface in Java (*see appendix B, page VI*). *Time scale: short*
- 2) Development of a web based version using a Java applet (*see appendix B, page VI*). *Time scale: short*
- 3) Ability to read accounts from websites such as *Google Finance* via the use of data scraping techniques (*see appendix B, page VI*). *Time scale: medium*

4) Full integration into a spreadsheet application, as described in Douglas Adams' book. *Time scale: medium*

Signal Mapping

1) In this dissertation, only the balance sheet is used to generate signals. The implementation could be expanded by also using the Income and Cash Flow statements. *Time scale: medium*

2) The concept of 'skinning' the processors to produce different types of musical output can be applied. *Time scale: medium*

L-System Music Generation

1) Rule definitions: Research into how to structure the rules of the L-System so that they better reflect the account's nature. This kind of research would probably be suited to someone with a musical background. *Time scale: medium to long*

2) What happens if we have **longer strings** in the replacement rules? This leads to more control over the musical sequences used, but does it come at a cost? *Time scale: short*

3) We could use is to have additional variables in V of the L-System that produce 'generic' musical sequences not tied to any account features. These can be used to pad out the music, and are included simply to provide colour to the music. This will increase the musical complexity, but will it dilute the signs which point to account features? *Time scale: medium*

4) We could consider including more than one variable in a replacement rule. This way, the axiom reduces in a less predictable manner, leading to more emergent properties. For example, we could have a rule: $(A \rightarrow BuuC)$. *Time scale: short to medium*

5) A ten grade system could be applied (and indeed, the implementation already supports this). This would mean that a greater variety of music is produced, as there are more grade boundaries for signals. However, the rules for each grade need careful planning, and many more accounts are needed for testing a ten grade system than a six grade system, simply because of the greater variety of music. *Time scale: short to medium*

6) Each account attribute has its own set of replacement rules. This would perhaps make it easier to hear the movement of specific attributes within an account, as they could generate their own sequences. *Time scale: short to medium*

7) We could define musical genres, allowing the user to choose a genre of their preference before the music is generated. For example, the rules we defined in this chapter produce music which is split into four beats per bar. If we reduced this to three beats, we could set this alternative rule set up as its own 'waltz' genre. *Time scale: medium to long*

Financial Genome

1) Research into developing a fuller financial genome. The genome given in Chapter 7 only defines a few simple characteristics. A fuller genome could lead to better music generation. *Time scale: long*

2) Genes can be given the ability to be dominant or recessive. They can also be developed so combining two genes produces a unique feature not found in any other configuration. *Time scale: medium*

General Research

1) Music cognition tells us that a person's perception of music is based on their experience. Research could be conducted into how cultural background affects the perception of the accounts when heard through the music. *Time scale: long*

2) There may be many other (and possibly better) approaches to generating music from accounts. Some of these could be investigated. *Time scale: medium to long*

3) As the generated music becomes more complex, what is the ‘cost’ associated with this in terms of ability to accurately assess the account’s true nature? *Time scale: medium*

Final Conclusion: Is There a Real-World Application for Financial Music?

After reading through this document, the reader may conclude that whilst Financial Music is an interesting topic for research, it lacks any value as a real-world application, perhaps doomed to reside with thousands of other programming curiosities which can be found the internet.

But, if the amount of accounts that a novice can correctly analyse through the music *exceeds* the amount that they can correctly analyse just from looking at the numbers, then Financial Music can serve a useful purpose. On top of this, if the music provides a *quicker* way to analyse accounts than traditional methods, then we have found a real-world use for Financial Music.

For example, an investment portfolio manager could batch convert a series of accounts to music, and then listen to the tracks on their iPod whilst out jogging during their lunch break. When returning to their office, they can then choose which accounts to investigate further. Therefore, the main purpose Financial Music could serve is to help filter out the worst investment propositions.

Finally, It is my hope that the development of Financial Music idea will continue.

Appendices

Appendix A: Test Results

	Soar	Slump	Boom	Plunge	Grow	Slide	Consistent
Acc 1 Sequential	1	3	0	0	9	0	2
Acc 2 Sequential	2	3	1	0	4	0	3
Acc 3 Sequential	2	3	0	0	8	2	2
Acc 4 Sequential	3	2	0	2	2	4	2
Acc 5 Sequential	2	3	1	5	0	1	3
Acc 6 Sequential	0	1	0	3	2	2	3
Acc 1 Parallel	0	5	0	2	1	3	2
Acc 2 Parallel	1	2	1	0	3	1	5
Acc 3 Parallel	0	0	0	0	6	1	4
Acc 4 Parallel	3	0	2	2	2	2	1
Acc 5 Parallel	2	0	3	1	2	1	3
Acc 6 Parallel	0	3	1	6	0	3	0
Acc 1 Piano	3	1	1	0	7	2	2
Acc 2 Piano	0	0	0	0	2	4	7
Acc 3 Piano	1	3	0	0	2	2	4
Acc 4 Piano	0	2	1	1	1	7	1
Acc 5 Piano	0	3	0	1	0	3	4
Acc 6 Piano	0	3	1	2	0	6	1
Acc 1 Strings	7	1	1	0	4	2	1
Acc 2 Strings	2	2	0	0	6	0	3
Acc 3 Strings	1	2	0	0	4	2	4
Acc 4 Strings	0	5	0	0	2	4	3
Acc 5 Strings	0	6	0	1	0	5	1
Acc 6 Strings	0	1	1	6	0	3	0

Figure 8.1: Sheet 1 of the results.

Listens	NOTUSE	Musicality		Highest	Average
1	2.181818182	3.818181818	Sequential	4.2	3.7
1.636364	2.727272727	3.272727273	Parallel	3.4	2.9
1.090909	2.545454545	3.454545455	Piano	3.7	3.5
1.454545	2.727272727	3.272727273	Strings	4.1	3.6
1.181818	1.818181818	4.181818182			
1.454545	2	4			
1.454545	3.181818182	2.818181818			
2.090909	3.272727273	2.727272727			
1.909091	3.272727273	2.727272727			
1.909091	3	3			
1.545455	2.636363636	3.363636364			
1.818182	3.090909091	2.909090909			
1.090909	2.272727273	3.727272727			
1.272727	2.727272727	3.272727273			
1.272727	2.818181818	3.181818182			
1.090909	2.545454545	3.454545455			
1.181818	2.545454545	3.454545455			
1.272727	2.363636364	3.636363636			
1.181818	2.272727273	3.727272727			
1.181818	1.909090909	4.090909091			
1.636364	2.363636364	3.636363636			
1.272727	2.454545455	3.545454545			
1.090909	2.727272727	3.272727273			
1.090909	2.727272727	3.272727273			

Figure 8.2: Sheet 2 of the results.

Lowest
3.3
2.7
3.2
3.3

Figure 8.3: Sheet 3 of the results.

Appendix B: Interfaces

In this section, ways that account data can be imported into the Financial Music software will be explored. An ideal outcome would be that accounts can be imported directly from a website using an API, RSS feed, or other XML based protocol. However, other options will also be considered.

A User Interface

The most basic way of having the program attain account information would be to provide a simple user interface.

This interface would consist of fields which can be filled in, and a button to submit their contents for analysis.

This kind of interface, whilst crude, would provide a very nice way to test the software. Values could be tweaked and altered on the fly to generate music. Feedback would be instant, providing a quick mental picture of what kind of data produces what kind of music.

Local Sources

My second reader suggested to me that I use **CSV** as a commonly used format for importing accounts. This would allow for the easy importing of accounts stored in spreadsheet format (figure: 8.4).

The system I have implemented requires that two years worth of accounts be set up in individual spreadsheets. These spreadsheets consist of two rows; the top row contains the headings, and the bottom row contains the numerical values.

The importing procedure runs checks to ensure that both spreadsheets have the same number of columns in the first two rows as each other, and the headings in both spreadsheets are identical. These two clauses provide reasonable assurance that the two spreadsheets are from the same template.

Internet Sources

The primary candidate for sourcing account information is **Google Finance**, which has recently launched a localised UK version of this service¹. Google Gadgets provides a JavaScript API for accessing data from Google Finance², but due to licensing restrictions, Google have not opened up this API for use on any other platform³. **Yahoo Finance** suffers from a similar restriction, and will only allow the manual generation of HTML as a badge that can be placed on a web log (blog) or website⁴. It also does not provide the complete balance sheet that is needed.

A Workaround

The option I'm considering is the inclusion of a JavaScript, which can be added as a **bookmarklet** (a small script or applet stored as a browser bookmark) in a web browser. This JavaScript would be able to scan a web page for specific elements (known as **data scraping**), and extract them (figure: 8.5). This can be done by accessing elements of the **document object model (DOM)**. Then, the user views a companys account on Google Finance. They then select the bookmarklet, which runs the JavaScript. The JavaScript could pass the relevant data

¹<http://finance.google.co.uk>, *Google Finance UK*, 01/02/2008

²<http://code.google.com/apis/gadgets/docs/finance.html>, *Financial Gadgets*, 01/02/2008

³<http://googlefinanceblog.blogspot.com/2007/10/api-gadgets-and-tabs-oh-my.html>, 01/02/2008

⁴<http://finance.yahoo.com/badges>, 01/02/2008

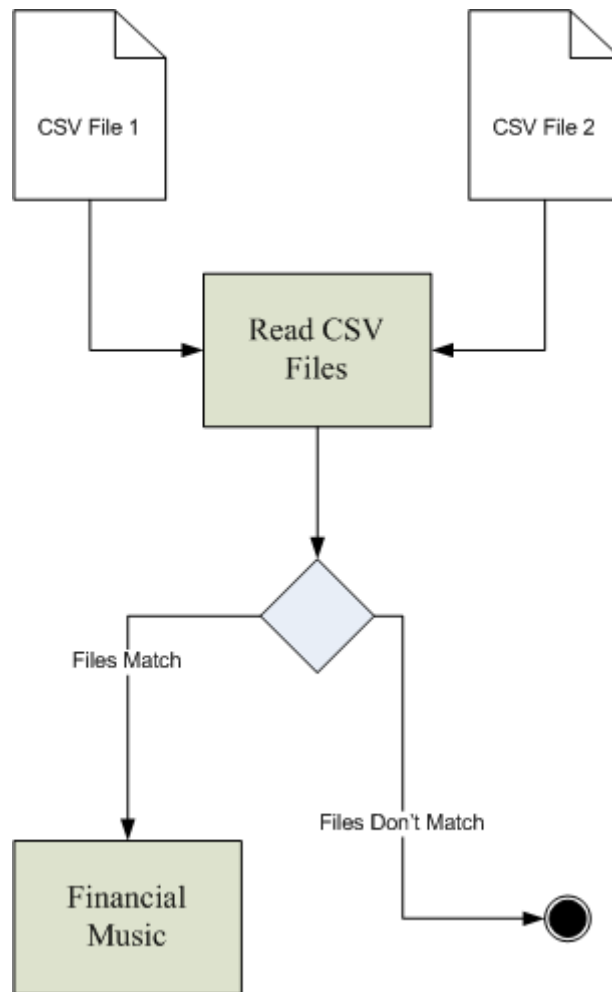


Figure 8.4: Importing CSV files into the Java application.

to another page containing **Financial Music** as a Java applet⁵. This would allow an account to be played with a single click. (The idea of disguising a JavaScript as a bookmark is not a new one, as it is used by Google to automate the adding of RSS feeds to Google Reader, and also by *tinyurl.com* to minimise the URL of the currently viewed web page)

There are limitations to this compromise: (a) This is all be dependent on Google not changing the setup of their web pages; otherwise the script may no longer function. (b) The account information to be extracted is chosen by the JavaScript. Therefore, if we wish to use different account attributes, we need to alter both the JavaScript and the Java Applet to be synchronised with each other.

In Practice

The reality of the situation, is that Google Finance isn't set up for easy data extraction (in an ideal situation, the relevant HTML elements would be nicely tagged with the *ID* element). What we do know from looking at the HTML source, is that the value for an attribute is located directly after its label. Therefore, we need to search for the name of the account attribute we want to extract. Once this is done, the next item of data should be the value that we need to

⁵<http://www.devdaily.com/java/edu/pj/pj010003/pj010003.shtml>, 01/02/2008

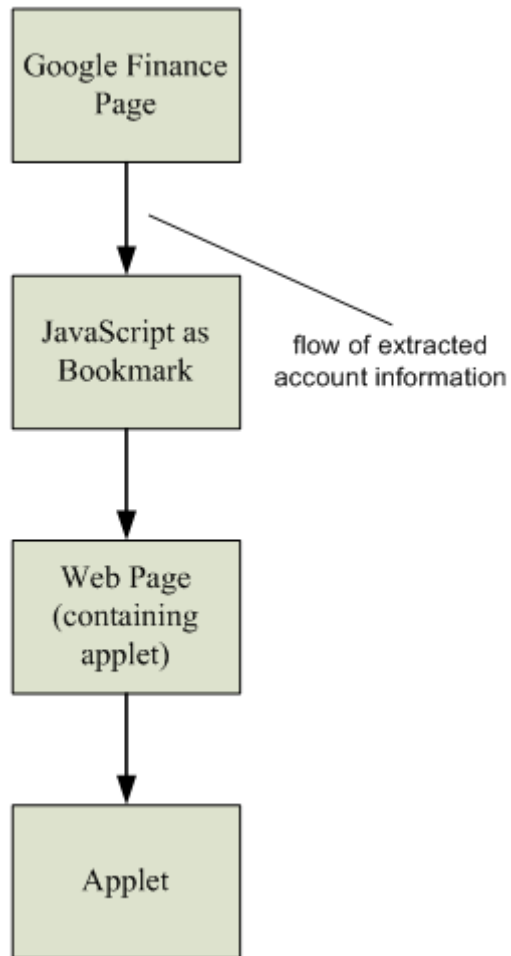


Figure 8.5: Flow of account data from Google Finance to the Java applet.

extract.

The Java applet should also have a default state, whereby if no information is passed to it via its containing web page, it will allow the user to enter account information manually.

Appendix C: Evaluation Results and Supplementary Graphs

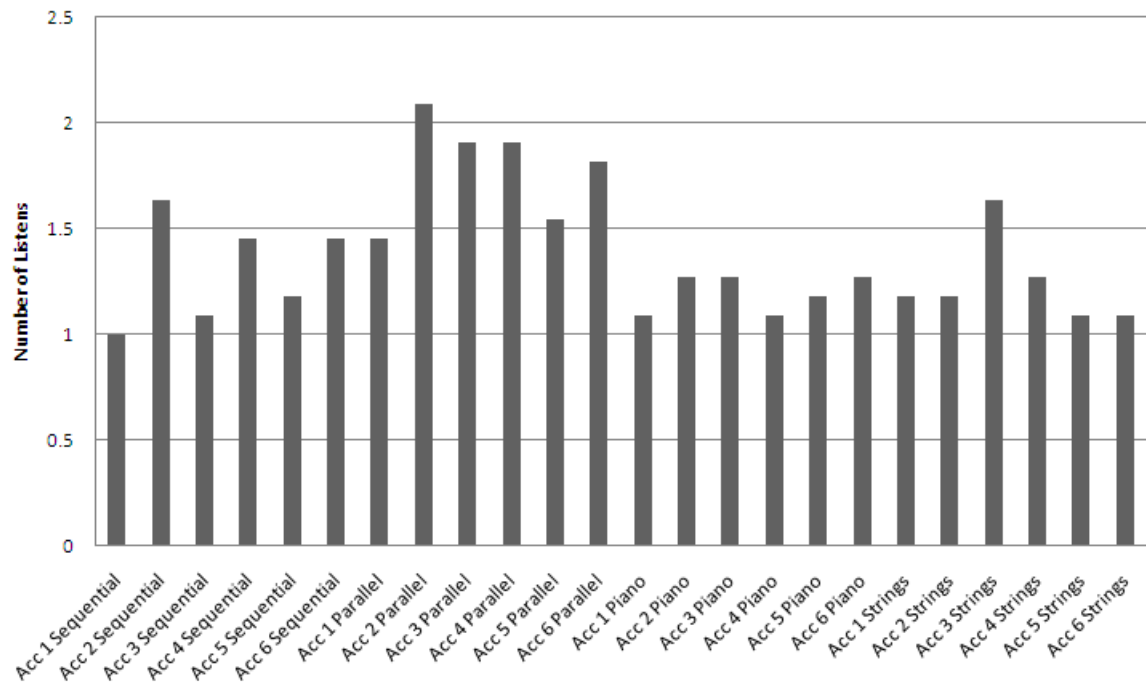


Figure 8.6: A graph showing how many times on average testers listened to each output sequence.

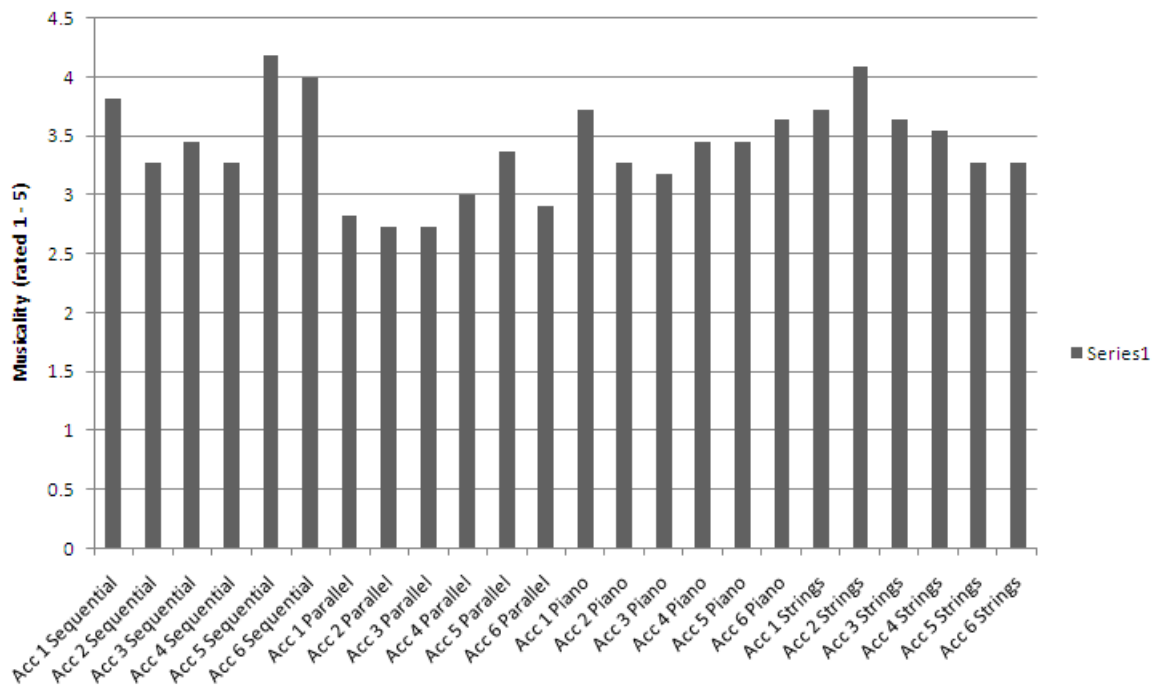


Figure 8.7: A graph showing the average musicality score between 1 and 5 that testers gave each output sequence.

Appendix D: Questionnaire

In this section, I have included the questionnaires that were used by the expert and the testers to provide feedback for the evaluation. These questionnaires were provided to the testers as Microsoft Word templates. The boxes could be ticked directly, and the music sequences were embedded in the document. Once the questionnaires were filled out, they could simply be saved under a different file name, and were then archived for later analysis.

Financial Music
- Expert Evaluation -

Hi there, and thanks for taking the time to help me evaluate my dissertation project. Please take the time to read the **Background** and **Instructions** before starting.

Background

Financial Music is software which generates music from financial accounts. The aim is that you can tell whether a company is making money just from the music; there's no need to look at the numbers in the accounts.

In order to evaluate whether the music derived from accounts presents an accurate picture of the account's actual meaning, it is necessary to have an expert inspect each account and give their analysis.

This analysis will be used as a baseline to compare with the results of other testers who will be listening to the music.

To keep things simple, accounts will be presented as follows:

1. Only the **balance sheet** will be given.
2. You may assume that the amounts shown are in multiples of **millions of US Dollars**.
3. The name of the company is kept hidden.
4. There will be 15 accounts to analyse.

Instructions

1. First, fill out the preliminary information on the next page.
2. You'll answer questions by ticking boxes like this: . To **tick a box**, just click it. To un-tick it if you change your mind, click it again.
3. The balance sheet for **two subsequent years** will be shown. Look at these with an eye to the perceived **changes over the course of the year**.
4. Answer the questions for each account you look at.
5. If you want to take a break, then you can simply save this document and come back to it later.
6. When you're done, be sure to **save this file** and **exit Microsoft Word**.
7. Finally, please e-mail this file to danieldomby@googlemail.com.

Figure 8.8: Page 1 of the expert's questionnaire.

- Preliminary Information -
Please rate your understanding of account mechanics from 1 (experienced) to 5 (novice) 1 > <input type="checkbox"/> 2 > <input type="checkbox"/> 3 > <input type="checkbox"/> 4 > <input type="checkbox"/> 5 > <input type="checkbox"/>
Would you like to be informed of the results of the testing? Yes > <input type="checkbox"/> No > <input type="checkbox"/>

Figure 8.9: Page 2 of the expert's questionnaire.

- Account 1 -					
	Current Assets	Total Assets	Current Liabilities	Total Liabilities	Total Equity
Last Year	14509	17205	6443	7221	9984
This Year	21956	25347	9299	10815	14532
Please choose one or more words that you think represent the account:					
	Soar	>	<input type="checkbox"/>		
	Slump	>	<input type="checkbox"/>		
	Boom	>	<input type="checkbox"/>		
	Plunge	>	<input type="checkbox"/>		
	Grow	>	<input type="checkbox"/>		
	Slide	>	<input type="checkbox"/>		
	Consistent	>	<input type="checkbox"/>		
Do you think this company is doing well enough to invest money in?					
	Yes >	<input type="checkbox"/>		No >	<input type="checkbox"/>
		<input type="checkbox"/>		Unsure >	<input type="checkbox"/>
Please type any useful comments in the box below:					
<input style="width: 100%; height: 20px;" type="text"/>					

Figure 8.10: Page 3 of the expert's questionnaire. Future pages continue in the same fashion.

Financial Music
- Music Evaluation -

Hi there, and thanks for taking the time to help me evaluate my dissertation project. This questionnaire should take around 20 minutes to complete. Please take the time to read the **Background** and **Instructions** before starting.

Background

Financial Music is software which generates music from financial accounts. The aim is that you can tell whether a company is making money just from the music; there's no need to look at the numbers in the accounts.

Different techniques have been used to generate the music you're about to listen to. Remember that the music is generated by a computer, so its important that you don't have any preconceptions about what you're about to hear.

Instructions

1. First, fill out the preliminary information on the next page.
2. You'll answer questions by ticking boxes like this:
 - a. To **tick a box**, double click it and select "**checked**" under "**default value**".
 - b. To **un-tick a box**, double click it and select "**not checked**" under "**default value**".
3. To listen to a tune, **double-click** the blue 'play' icon at the top of each box.
4. Listen to each tune as many times as you like. If you want to take a break, then you can simply save this document and come back to it later.
5. Answer the questions for each tune you listen to.
6. When you're done, be sure to **save this file** and **exit Microsoft Word**.
7. Finally, please e-mail this file back to me at danieldomby@googlemail.com.

- Note that all information you provide is anonymous -
- You can opt out of this questionnaire any time you like -

Figure 8.11: Page 1 of the tester's questionnaire.

- Preliminary Information -
Please rate your musical background from 1 (experienced) to 5 (novice) 1 > <input type="checkbox"/> 2 > <input type="checkbox"/> 3 > <input type="checkbox"/> 4 > <input type="checkbox"/> 5 > <input type="checkbox"/>
Would you like to be informed of the results of the testing? Yes > <input type="checkbox"/> No > <input type="checkbox"/>

You're almost ready to begin the questionnaire. There are 24 "tunes" to listen to.

Please make sure that the volume on your computer is turned up


- Tune 1 -
 Wave Sound Double Click the icon above to play the tune
How many times did you listen to the tune? 1 > <input type="checkbox"/> 2 > <input type="checkbox"/> 3 > <input type="checkbox"/> More than three > <input type="checkbox"/>
Please choose one or more words that you think represent the account: Soar > <input type="checkbox"/> Slump > <input type="checkbox"/> Boom > <input type="checkbox"/> Plunge > <input type="checkbox"/> Grow > <input type="checkbox"/> Slide > <input type="checkbox"/> Consistent > <input type="checkbox"/>
Do you think this company is doing well enough to invest money in? Yes > <input type="checkbox"/> No > <input type="checkbox"/> Unsure > <input type="checkbox"/>
How musical did the tune sound? 1 > <input type="checkbox"/> 2 > <input type="checkbox"/> 3 > <input type="checkbox"/> 4 > <input type="checkbox"/> 5 > <input type="checkbox"/>

Figure 8.12: Page 2 of the tester's questionnaire. Future pages continue in the same fashion.

Appendix E: User Guide

Welcome to the user guide. Here you will learn how to operate Financial Music.

System Requirements

Before beginning, you will need to ensure you have Java 1.6 and Jython 2.2.1 installed. Java can be obtained from www.sun.com and Jython can be obtained from www.jython.org

File Formats

Financial Music expects the account information to be in CSV (Comma Separated Value) files. These can be opened or generated with a spreadsheet package such as Microsoft Excel.

Compiling Financial Music

To prepare Financial Music for operation, you will need to open a command line console. Once this is done, navigate to the directory containing Financial music and type: `java *.java`. This will perform the compilation.

Music Generation

Before generating the music, you will need to make sure that you have two subsequent years' balance sheets in two separate CSV files.

To generate music for the Signal Mapping implementation, type the following at the command line: `jython Mapping.py year1 year2`. Likewise, to generate music for the L-System Music Generation implementation, type: `jython LSS.py year1 year2`. In both instances, replace `year1` and `year2` with the appropriate file names.

Playing the Music

Once the music is generated, you can hear it by typing the following at the command line: `java MusicReader`.

Changing Program Settings

To alter the settings used to generate the music, open your favourite text editor and choose 'open file'. Navigate to the Financial Music directory, and open `settings.py`. Within, you will find instructions on how to change these settings.

Appendix F: Financial Genome Example

Full Genome

16464 : ['Boom', ['Descending', 'NULL', 'Chord']]
4200 : ['Plunge', ['Descending', 'Minor', 'Scale']]
49392 : ['Boom', ['Ascending', 'NULL', 'Chord']]
3430 : ['Slump', ['NULL', 'Major', 'Chord']]
360 : ['Plunge', ['Ascending', 'Major', 'NULL']]
82320 : ['Boom', ['Descending', 'Major', 'Chord']]
8400 : ['Boom', ['Descending', 'Minor', 'Scale']]
350 : ['Slump', ['NULL', 'Minor', 'Scale']]
6860 : ['Soar', ['NULL', 'Major', 'Chord']]
12600 : ['Plunge', ['Ascending', 'Minor', 'Scale']]
720 : ['Boom', ['Ascending', 'Major', 'NULL']]
336 : ['Boom', ['Descending', 'NULL', 'Scale']]
10290 : ['Slump', ['Descending', 'Major', 'Chord']]
1470 : ['Slump', ['Descending', 'Major', 'Crescendo']]
700 : ['Soar', ['NULL', 'Minor', 'Scale']]
13720 : ['Plunge', ['NULL', 'Major', 'Chord']]
246960 : ['Boom', ['Ascending', 'Major', 'Chord']]
25200 : ['Boom', ['Ascending', 'Minor', 'Scale']]
4900 : ['Soar', ['NULL', 'Minor', 'Crescendo']]
686 : ['Slump', ['NULL', 'NULL', 'Chord']]
300 : ['Soar', ['Descending', 'Minor', 'NULL']]
17150 : ['Slump', ['NULL', 'Minor', 'Chord']]
294 : ['Slump', ['Descending', 'NULL', 'Crescendo']]
29400 : ['Plunge', ['Descending', 'Minor', 'Crescendo']]
4116 : ['Soar', ['Descending', 'NULL', 'Chord']]
1050 : ['Slump', ['Descending', 'Minor', 'Scale']]
20580 : ['Soar', ['Descending', 'Major', 'Chord']]
280 : ['Plunge', ['NULL', 'Major', 'Scale']]
11760 : ['Boom', ['Descending', 'Major', 'Crescendo']]
1800 : ['Plunge', ['Ascending', 'Minor', 'NULL']]
2940 : ['Soar', ['Descending', 'Major', 'Crescendo']]
411600 : ['Boom', ['Descending', 'Minor', 'Chord']]
102900 : ['Soar', ['Descending', 'Minor', 'Chord']]
252 : ['Soar', ['Ascending', 'NULL', 'Scale']]
1400 : ['Plunge', ['NULL', 'Minor', 'Scale']]
630 : ['Slump', ['Ascending', 'Major', 'Scale']]
27440 : ['Boom', ['NULL', 'Major', 'Chord']]
1008 : ['Boom', ['Ascending', 'NULL', 'Scale']]
240 : ['Boom', ['Descending', 'Major', 'NULL']]
1764 : ['Soar', ['Ascending', 'NULL', 'Crescendo']]
30870 : ['Slump', ['Ascending', 'Major', 'Chord']]
9800 : ['Plunge', ['NULL', 'Minor', 'Crescendo']]
1372 : ['Soar', ['NULL', 'NULL', 'Chord']]
2520 : ['Plunge', ['Ascending', 'Major', 'Scale']]
176400 : ['Boom', ['Ascending', 'Minor', 'Crescendo']]
22050 : ['Slump', ['Ascending', 'Minor', 'Crescendo']]

600 : ['Plunge', ['Descending', 'Minor', 'NULL']]
980 : ['Soar', ['NULL', 'Major', 'Crescendo']]
34300 : ['Soar', ['NULL', 'Minor', 'Chord']]
210 : ['Slump', ['Descending', 'Major', 'Scale']]
588 : ['Soar', ['Descending', 'NULL', 'Crescendo']]
58800 : ['Boom', ['Descending', 'Minor', 'Crescendo']]
200 : ['Plunge', ['NULL', 'Minor', 'NULL']]
4410 : ['Slump', ['Ascending', 'Major', 'Crescendo']]
196 : ['Soar', ['NULL', 'NULL', 'Crescendo']]
8232 : ['Plunge', ['Descending', 'NULL', 'Chord']]
2100 : ['Soar', ['Descending', 'Minor', 'Scale']]
24696 : ['Plunge', ['Ascending', 'NULL', 'Chord']]
180 : ['Soar', ['Ascending', 'Major', 'NULL']]
41160 : ['Plunge', ['Descending', 'Major', 'Chord']]
560 : ['Boom', ['NULL', 'Major', 'Scale']]
6300 : ['Soar', ['Ascending', 'Minor', 'Scale']]
168 : ['Plunge', ['Descending', 'NULL', 'Scale']]
7056 : ['Boom', ['Ascending', 'NULL', 'Crescendo']]
123480 : ['Plunge', ['Ascending', 'Major', 'Chord']]
3600 : ['Boom', ['Ascending', 'Minor', 'NULL']]
2450 : ['Slump', ['NULL', 'Minor', 'Crescendo']]
150 : ['Slump', ['Descending', 'Minor', 'NULL']]
1680 : ['Boom', ['Descending', 'Major', 'Scale']]
14700 : ['Soar', ['Descending', 'Minor', 'Crescendo']]
144 : ['Boom', ['Ascending', 'NULL', 'NULL']]
2058 : ['Slump', ['Descending', 'NULL', 'Chord']]
140 : ['Soar', ['NULL', 'Major', 'Scale']]
5880 : ['Plunge', ['Descending', 'Major', 'Crescendo']]
900 : ['Soar', ['Ascending', 'Minor', 'NULL']]
205800 : ['Plunge', ['Descending', 'Minor', 'Chord']]
51450 : ['Slump', ['Descending', 'Minor', 'Chord']]
126 : ['Slump', ['Ascending', 'NULL', 'Scale']]
5488 : ['Boom', ['NULL', 'NULL', 'Chord']]
504 : ['Plunge', ['Ascending', 'NULL', 'Scale']]
120 : ['Plunge', ['Descending', 'Major', 'NULL']]
2800 : ['Boom', ['NULL', 'Minor', 'Scale']]
882 : ['Slump', ['Ascending', 'NULL', 'Crescendo']]
112 : ['Boom', ['NULL', 'NULL', 'Scale']]
1260 : ['Soar', ['Ascending', 'Major', 'Scale']]
88200 : ['Plunge', ['Ascending', 'Minor', 'Crescendo']]
490 : ['Slump', ['NULL', 'Major', 'Crescendo']]
100 : ['Soar', ['NULL', 'Minor', 'NULL']]
98 : ['Slump', ['NULL', 'NULL', 'Crescendo']]
12348 : ['Soar', ['Ascending', 'NULL', 'Chord']]
90 : ['Slump', ['Ascending', 'Major', 'NULL']]
3920 : ['Boom', ['NULL', 'Major', 'Crescendo']]
137200 : ['Boom', ['NULL', 'Minor', 'Chord']]
3150 : ['Slump', ['Ascending', 'Minor', 'Scale']]
84 : ['Soar', ['Descending', 'NULL', 'Scale']]

3528 : ['Plunge', ['Ascending', 'NULL', 'Crescendo']]
80 : ['Boom', ['NULL', 'Major', 'NULL']]
61740 : ['Soar', ['Ascending', 'Major', 'Chord']]
840 : ['Plunge', ['Descending', 'Major', 'Scale']]
7350 : ['Slump', ['Descending', 'Minor', 'Crescendo']]
72 : ['Plunge', ['Ascending', 'NULL', 'NULL']]
70 : ['Slump', ['NULL', 'Major', 'Scale']]
450 : ['Slump', ['Ascending', 'Minor', 'NULL']]
19600 : ['Boom', ['NULL', 'Minor', 'Crescendo']]
2744 : ['Plunge', ['NULL', 'NULL', 'Chord']]
5040 : ['Boom', ['Ascending', 'Major', 'Scale']]
60 : ['Soar', ['Descending', 'Major', 'NULL']]
56 : ['Plunge', ['NULL', 'NULL', 'Scale']]
44100 : ['Soar', ['Ascending', 'Minor', 'Crescendo']]
2352 : ['Boom', ['Descending', 'NULL', 'Crescendo']]
1200 : ['Boom', ['Descending', 'Minor', 'NULL']]
50 : ['Slump', ['NULL', 'Minor', 'NULL']]
48 : ['Boom', ['Descending', 'NULL', 'NULL']]
6174 : ['Slump', ['Ascending', 'NULL', 'Chord']]
1960 : ['Plunge', ['NULL', 'Major', 'Crescendo']]
35280 : ['Boom', ['Ascending', 'Major', 'Crescendo']]
68600 : ['Plunge', ['NULL', 'Minor', 'Chord']]
42 : ['Slump', ['Descending', 'NULL', 'Scale']]
40 : ['Plunge', ['NULL', 'Major', 'NULL']]
420 : ['Soar', ['Descending', 'Major', 'Scale']]
36 : ['Soar', ['Ascending', 'NULL', 'NULL']]
30 : ['Slump', ['Descending', 'Major', 'NULL']]
28 : ['Soar', ['NULL', 'NULL', 'Scale']]
1176 : ['Plunge', ['Descending', 'NULL', 'Crescendo']]
24 : ['Plunge', ['Descending', 'NULL', 'NULL']]
17640 : ['Plunge', ['Ascending', 'Major', 'Crescendo']]
20 : ['Soar', ['NULL', 'Major', 'NULL']]
18 : ['Slump', ['Ascending', 'NULL', 'NULL']]
784 : ['Boom', ['NULL', 'NULL', 'Crescendo']]
400 : ['Boom', ['NULL', 'Minor', 'NULL']]
16 : ['Boom', ['NULL', 'NULL', 'NULL']]
14 : ['Slump', ['NULL', 'NULL', 'Scale']]
12 : ['Soar', ['Descending', 'NULL', 'NULL']]
8820 : ['Soar', ['Ascending', 'Major', 'Crescendo']]
10 : ['Slump', ['NULL', 'Major', 'NULL']]
392 : ['Plunge', ['NULL', 'NULL', 'Crescendo']]
1234800 : ['Boom', ['Ascending', 'Minor', 'Chord']]
8 : ['Plunge', ['NULL', 'NULL', 'NULL']]
6 : ['Slump', ['Descending', 'NULL', 'NULL']]
4 : ['Soar', ['NULL', 'NULL', 'NULL']]
617400 : ['Plunge', ['Ascending', 'Minor', 'Chord']]
2 : ['Slump', ['NULL', 'NULL', 'NULL']]
308700 : ['Soar', ['Ascending', 'Minor', 'Chord']]
154350 : ['Slump', ['Ascending', 'Minor', 'Chord']]

Vailid Genes

58800 : ['Boom', ['Descending', 'Minor', 'Crescendo']]
2940 : ['Soar', ['Descending', 'Major', 'Crescendo']]
5880 : ['Plunge', ['Descending', 'Major', 'Crescendo']]
8820 : ['Soar', ['Ascending', 'Major', 'Crescendo']]
176400 : ['Boom', ['Ascending', 'Minor', 'Crescendo']]
27440 : ['Boom', ['NULL', 'Major', 'Chord']]
840 : ['Plunge', ['Descending', 'Major', 'Scale']]
11760 : ['Boom', ['Descending', 'Major', 'Crescendo']]
14700 : ['Soar', ['Descending', 'Minor', 'Crescendo']]
17640 : ['Plunge', ['Ascending', 'Major', 'Crescendo']]
1680 : ['Boom', ['Descending', 'Major', 'Scale']]
12600 : ['Plunge', ['Ascending', 'Minor', 'Scale']]
630 : ['Slump', ['Ascending', 'Major', 'Scale']]
2520 : ['Plunge', ['Ascending', 'Major', 'Scale']]
17150 : ['Slump', ['NULL', 'Minor', 'Chord']]
1470 : ['Slump', ['Descending', 'Major', 'Crescendo']]
29400 : ['Plunge', ['Descending', 'Minor', 'Crescendo']]
420 : ['Soar', ['Descending', 'Major', 'Scale']]
88200 : ['Plunge', ['Ascending', 'Minor', 'Crescendo']]
8400 : ['Boom', ['Descending', 'Minor', 'Scale']]
4410 : ['Slump', ['Ascending', 'Major', 'Crescendo']]
13720 : ['Plunge', ['NULL', 'Major', 'Chord']]
35280 : ['Boom', ['Ascending', 'Major', 'Crescendo']]
7350 : ['Slump', ['Descending', 'Minor', 'Crescendo']]
6300 : ['Soar', ['Ascending', 'Minor', 'Scale']]
25200 : ['Boom', ['Ascending', 'Minor', 'Scale']]
1260 : ['Soar', ['Ascending', 'Major', 'Scale']]
137200 : ['Boom', ['NULL', 'Minor', 'Chord']]
44100 : ['Soar', ['Ascending', 'Minor', 'Crescendo']]
4200 : ['Plunge', ['Descending', 'Minor', 'Scale']]
210 : ['Slump', ['Descending', 'Major', 'Scale']]
6860 : ['Soar', ['NULL', 'Major', 'Chord']]
3150 : ['Slump', ['Ascending', 'Minor', 'Scale']]
68600 : ['Plunge', ['NULL', 'Minor', 'Chord']]
22050 : ['Slump', ['Ascending', 'Minor', 'Crescendo']]
2100 : ['Soar', ['Descending', 'Minor', 'Scale']]
3430 : ['Slump', ['NULL', 'Major', 'Chord']]
5040 : ['Boom', ['Ascending', 'Major', 'Scale']]
1050 : ['Slump', ['Descending', 'Minor', 'Scale']]
34300 : ['Soar', ['NULL', 'Minor', 'Chord']]
40 out of 144 genes are valid.

Appendix G: Progress Log

This log details the progress of my dissertation through its final few weeks. The purpose of the log is to provide evidence of the work that has applied to the task, just in case the worse should happen (such as all the departmental servers melting at the same time as my laptop explodes, setting fire to all printed documents in the vicinity).

I think its fair to say that as Deliverables 1 & 2 are now a matter of record. These in themselves provide adequate evidence of work achieved over the earlier sections of the project.

(Each day's entries were written as I was working, so the tenses may fluctuate between past, present and future!)