Unix Style Computer Aided Composition

Ray Garner

20156967

psyrg4@nottingham.ac.uk

Computer Science with Year in Industry Bsc

ABSTRACT

Computer aided composition is when a musician employs software tools to create music. The scope of computer aided composition varies from the production of small ideas such as melodies or chords to entire pieces. In computer science terms, computer aided composition software reduces the search space a musician explores to find successful ideas. This paper attempts to evaluate the effectiveness of applying Unix philosophy to computer aided composition by implementing a system using orthogonal programs, text streams and pipes and comparing the results to existing popular compositions. Eight different programs were produced which could be combined in various ways to provide different functionality including mode generation, mode filtering, chord generation, melody generation and melody harmonisation. Output was was quantitativly analysed, with the results being compared against Bach compositions and folksongs to show that they align with the fundamental ideas of what makes music pleasant. The system was designed so that in the future new programs could be written and integrated with the existing ones to provide extra operations.

Keywords: unix, pipelines, music, computer aided composition, melody, harmony, tonality

I would like to thank God, my family, friends and supervisors Nazia Hameed and Adam Walker for their support throughout the project.

Table of Contents

List of abbreviations

CAC	Computer aided composition
DAW	Digital audio workstation
IRCAM	Institut de recherche et coordination acoustique/musique
LSEPI	Laws, social, ethical and professional issues
MIT	Massachusetts Institute of Technology

Music Glossary

Pitch	A value used to represent how high or low a note sounds
Interval	Difference between 2 pitches
Scale	A circular sequence of intervals and optionally a starting pitch
Degree	Ordinal representing where a pitch lies in a given scale. Also used to describe intervals
Mode	A scale with a specific interval treated as the first
Melody	A line either played alone or higher in pitch than all other concurrent parts
Chord	Two or more notes played concurrently
Harmony	The combination of notes in a sequence of 1 or more chords
Tonality	The mode or scale which the melody and harmony is built from primarily
Key	Representation of tonality for practical interpretation
Line	A sequence of single notes
Natural	Unaltered pitch
Sharp	Pitch raised by one semitone
Flat	Pitch lowered by one semitone

List of Tables

List of Figures

List of Equations

1. Introduction

1.1. Background

Computer aided composition is when a musician employs software tools to create music. The scope of computer aided composition varies from the production of small ideas such as melodies or chords to entire pieces. In computer science terms, computer aided composition software reduces the search space a musician explores to find successful ideas. Software can generate music from a range of input types: 'Bach in a Box' (McIntyre, 1994) shows harmony being generated from a specifically defined melody and 'COMPoZE' (Zimmermann, 1996) shows music being generated from variable descriptors such as 'ambition' and 'distribution'.

A scorewriter is a tool for writing and formatting sheet music. A digital audio workstation is a tool for manipulating audio data. The requirements for a tool to be a scorewriter, digital audio workstation and composition aid are different but a tool may be any combination of the three. This project focuses on computer aided composition.

The IRCAM (Delerue, 1999) computer aided composition philosophy is that a user is best served by a 'visual programming language' because it provides the flexibility required for accurate expression. IRCAM's 'PatchWork' (Duthen, 1989) proved the effectiveness of this approach. 'OpenMusic' (Bresson, 2011) , an IRCAM PatchWork successor, is used by institutes for research and education as well as by individuals for composition. IRCAM solutions are an abstraction of Lisp, with 'boxes' corresponding to Lisp functions. The IRCAM style solutions allow for ideas to be built by combining multiple individual ideas, each operating in one different element of music. For example, you could combine melody data with a tonality data to produce music.

'Unix Music Tools at Bellcore' (Langston, 1990) demonstrates music software written for Unix and explains the motivations for the design choices made. Langston says that consumer music programs lack the ability to communicate with each other, an issue caused by limitations of consumer PC operating systems. The music software written at Bellcore was written with the Unix design philosophy in mind: an approach combining small orthogonal programs to solve larger problems (Kernighan, 1984). Using a shell script, the Bellcore music tools can be combined to generate music and were even combined to form the backend of 'IMG/1'(Langston, 1991), a tool for generating backing music for presentations. More detail on the languages used to transmit musical data between programs can be seen in 'Little Languages for Music'(Langston, 1990).

1.2. Motivation

The motivation for this project follows from the flaws in the Bellcore music tools and IRCAM tools. These tools share many similarities and what one fails at, the other tends to succeed at. As shown in table 1, this project attempts to combine the successes of both of these systems.

Table 1: How this project combines elements of existing works

	OpenMusic workflow	IMG/1 workflow
OpenMusic implementation		
IMG/1 implementation	This project	

Parallels between the IRCAM style solutions and the Bellcore music tools can be drawn: both make the user interact with the system by sequentially applying functions to a flow of data. Functions in the IRCAM solutions are abstractions of Lisp functions, shown as 'functional boxes' but in the Bellcore solutions, they are standalone programs written in C which read from `STDIN` and write to `STDOUT`. Data-flow handling for the Bellcore solutions is handled by the Unix operating system with pipes and streams but in the IRCAM solutions it is done with Lisp data structures during the runtime of the main program. A further parallel can be drawn between this contrast and the contrast between 'MIT' and 'New Jersey' approaches described in 'The Rise of Worse is Better' (Gabriel, 1991), with OpenMusic falling into the 'MIT' category (Lisp, correctness) and the Bellcore music tools falling into the 'New Jersey' category (see the literature review section for more on this).

Viewing the IRCAM methodology through the lens of Unix philosophy raises the question- why implement functionality already implemented by the operating system? That is, why should the IRCAM solutions build another data flow framework when one already exists built into Unix-style operating systems?

Comprising of over 90 separate programs, becoming acquainted with the Bellcore music tools would be a daunting challenge for a non-technical composer and the more user friendly 'IMG/1', built on top of said tools, fails to provide an interface facilitating sequential function application on a data stream like the 'visual programming language' of OpenMusic does.

This project attempts to create a modern Unix style counterpart to OpenMusic, preserving the generality and expressiveness of its interface but implementing its functionality using traditional Unix methods.

2. Aims and Objectives

2.1. Musical

Primarily, this software will need to produce musical output in the form of tonal, melodic and harmonic ideas for a human composer to interpret. These ideas must adhere appropriately to established music theory principles and relate accordingly to the input used to generate them. For this project we will define 'melody' as single voice sequential lines, 'harmony' as a chord or sequence of chords (where a chord is 2 or more notes played concurrently) and 'tonality' as scales (where a scale is a set of intervals combined with a starting note).

2.2. Architectural

For this project it is important that the musical goals are achieved using the right means. The function of the system should be broken down into small orthogonal programs which are combined by the user using Unix pipes to produce the various outputs. This is advantageous to a user because it shows them clearly how the aspects of the system can be rearranged to produce a different desired output. Monolithic systems such as IRCAM style solutions have a huge amount of internal functions implemented to facilitate proper output production but they are not exposed to the user for them to utilise, even though they may be of use. Building the system up in a modular fashion allows the user to just use the specific functions which they need rather than having to load the whole program just to use a small portion of it.

The output format must be easy for a human to read, but also simple for a computer program to interpret. This will make the user interaction more intuitive by removing the need for intermediary translation programs in the pipeline.

2.3. Omissions from the project scope

This omitted from the scope of this project are:

MIDI output

Audio output

Entire piece composition

Rhythmic and textural manipulation

This is not to say that these things may not be built on top of this system in the future. It is important that this solution is extendable but this dissertation is not concerned with implementing these features.

MIDI describes more than just tonality, harmony and melody so it is beyond the initial focus of this project. Audio output would require implementing support for a whole new interface: speakers. This project is focused on human readable text output which could be interpreted by a composer.

This software is not trying to be a composer, it is trying to be a tool which a composer can use to generate prompts which they can implement. The composition of an entire piece is a different problem to what is being solved by this project.

Tonality, harmony and melody can all be handled in the same terms: sets of pitches. Rhythm and texture require special notation beyond this for accurate representation so are outside the scope of this solution.

The execution of each of the 3 programs will begin with the reading of data and end with the writing of data. Between these two points in time, no further data will be inputted to the program. This contrasts IRCAM solutions which are running constantly while a user works on them but is in line with the Bellcore approach.

3. Literature Review

The crux of this project is combining elements IRCAM and Bellcore approaches to computer aided composition. The 'IRCAM' approach refers to PatchWork (Duthen, 1989) and OpenMusic (Bresson, 2011), systems which provide a real-time, monolithic system developed using Lisp based languages. The 'Bellcore' approach refers to the tool-set developed at Bellcore which provides a wide array of music functionality. Since there are so many tools listed we will focus on one example demonstrated in 'Unix Music Tools at Bellcore' (Langston, 1990): generating chord progressions and generating melodies.

The contrast of approaches here is a strong reflection of the contrast of approaches described in 'The Rise of Worse is Better' (Gabriel, 1991). Gabriel compares what he called the 'MIT approach' and the 'New Jersey' approach. The IRCAM approach is in line with the MIT approach because of its Lisp style and the Bellcore approach is inline with the New Jersey approach because of its Unix style. Initially Gabriel frames the MIT approach to be superior thanks to its unwillingness to compromise correctness, consistency and completeness for the sake of simplicity. By contrast, the New Jersey approach assigns greater value to simplicity, going as far as to say that it is 'slightly better to be simple than correct'. Following this, it may be surprising to read further and discover Gabriel praising the New Jersey approach for its 'better survival characteristics', saying that software written in that style is more portable, allowing it to spread faster and gain more use. Currently there is no 'New Jersey' style counterpart to the 'MIT Style' software like OpenMusic, so with this project I intend to explore the application of 'New Jersey' style software development in the field of computer aided composition, building on ideas demonstrated by the Bellcore music tools.

3.1. IRCAM

IRCAM say the purpose of computer aided composition research was to 'provide composers with the means to develop musical ideas and models using the computer.' Contrast with the Bellcore philosophy can be seen here because Bellcore tools attempt automate composition but IRCAM leave the composition up to the composer and just provide a means for them to work expressively with the computer. My goal with this project is inline with the IRCAM philosophy, however I want to implement a solution using a methodology inline with the Bellcore philosophy (Unix philosophy).

3.2. Bellcore

Figure 1 in 'Unix Music Tools at Bellcore' shows a script generating a 'march' style piece of music. This task is decomposed into generating a chord chart, generating an accompaniment, generating a melody and then merging the melody and accompaniment. For each of these tasks, there is an individual program to perform it and each of these programs communicate by writing and reading to files. First a 32 bar chord chart in the key of F with a 'march' structure is generated. This is then used to generate an accompaniment, and then used again to generate a melody. Finally the melody and accompaniment are merged to produce the finished piece.

This example shows an almost textbook application of the unix philosophy: the system is broken down into orthogonal programs which each solve a general problem and they are tied together using a shell script. This makes things simpler for a developer because each individual program can be debugged on its own and it is more expressive for a user because a system structured this way allows for the components to be combined in various ways, producing interesting results. One shortcoming apparent here is that the tonality aspect of the system is limited: the user appears to be limited to only a major and minor key for each note in a western harmony system. 7 different modes can be derived from just a standard western 7 note major scale, these being used in different styles of music (more on this later). What this example shows is also closer to computer composition than computer aided composition. For this project I am more interested in a computer aided composition system producing prompts for a composer to arrange and implement. In this context, the flexibility and expressiveness of the user interaction is more important than the output being a finished piece.

A system built on top of the Bellcore tools is IMG/1 (Langston, 1991). IMG/1 is used to generate musical accompaniment for powerpoint style presentations. This system falls more into the category of algorithmic composition than computer aided composition because it is aimed at users unskilled in music composition. This contrasts OpenMusic and similar IRCAM projects because they try to provide as much

flexibility and freedom to allow skilled composers to express themselves as accurately as possible.

3.3. Justification for this work

This project attempts to combine the implementation philosophy of IMG/1 (Bellcore, Unix, New Jersey) with the composition and UI philosophy of OpenMusic (general, flexible, and expressive). The justification for this project follows from there being no 'New Jersey' or Unix-style counterpart to the 'MIT' style IRCAM computer aided composition software such as OpenMusic. The closest thing there has been to this was definitely the Bellcore music tools, however they were only available internally and not to real world composers. Not only this, but the the Bellcore tools aren't focused on enabling computer aided composition and would be daunting and confusing for a composer to use rather than a Unix expert. IMG/1, built on top of the Bellcore music tools and aimed at unskilled users, doesn't offer the generality, flexibility or expressiveness which IRCAM style tools such as OpenMusic do. This project attempts to fill this gap in the field and evaluate whether this style of development can lead to effective computer aided composition software being produced.

4. Methodology

4.1. Music Theory

To understand the algorithms used in this project, it is essential to have a basic grasp of western music theory. In this section I will describe the core aspects of music theory which this project primarily deals in: melody, chords and modes. If at any point you are unsure the meaning of a musical term, please refer to the music glossary at the beginning of this document. Some useful further reading if you desire it is "The AB Guide to Music Theory" (Taylor, 1991).

Traditionally there are 12 tones used to represent pitch: C, C#, D, D#, E, F, F#, G, G#, A, A# and B (using only sharps and no flats to represent them). The interval between two adjacent tones in this sequence is known as a semitone and the interval between every other tone in this sequence known as a whole-tone. For example, C and D are a whole tone apart and C and C# are a semitone apart.

The most fundamental scale in western music is the major scale, which is a sequence of seven intervals: tone, tone, semitone, tone, tone, tone, semitone. Scales such as this can be given a root note (note to start on) to produce a set of notes which can be used to create music. For example, a major scale with a root of C (aka C major scale) contains the following notes: C, D, E, F, G, A and B because C is the first, a tone above C is D, a tone above D is E, a semitone above E is F, and so on (recall the sequence of intervals defining the major scale if this is not clear to you). Each pitch in a scale can be given an ordinal to represent its function within the scale and this is known as the degree of the scale which it is. For example, C is the first degree of the C major scale and D is the second, E is the third and so on. Each interval between each adjacent note in a scale is known as a 'step' and the other various intervals between notes in the scale can be described in a similar way to how we use degrees. For example, E is a third above C and A is a third above F.

Bare in mind that not every step in the scale represents the same difference in pitch because some intervals of the scale are a tone and some are a semitone. This means that the difference in pitch between C and E is different to that of D and F. All 'thirds' in the scale are either the same as the difference between the first degree and the third degree in the major scale intervals or of that in the minor scale intervals. Hence, thirds are always 'major' or 'minor'. Whether something is major or minor is an example of tonality, and when tonality is not explicitly stated it is assumed major is being refered to.

Once we have established what scale we are using, we can begin to build chords. The most fundamental chord structure is known as the triad which contains 3 notes: the first, third and fifth. For each note of the major scale there is a triad chord where it is the root. For example, chord 1 in the major scale is C, E and G. Chord 4 in the major scale is F, A and C. Table 2 shows the tonality for each chord of the major scale

Table 2: Chord tonality in the major scale

Chord	Tonality
I	Major
II	Minor
III	Minor
IV	Major
V	Major
VI	Minor
VII	Diminished

Other than chord VII, the fifth in each of these chords is known as 'perfect'. It is called a perfect fifth because the difference in pitch between the first and fifth degrees is the same in the major and minor scales. Chord VII is special because it is the only one where the interval between its first and fifth is not that of the major scale, it is one semitone smaller. The third of chord VII is minor.

The major scale has 7 modes because a mode is defined by treating a specific degree as the first. Table 3 shows the modes of the major scale, where "relativity" is the degree of the major scale treated as the first to define that mode.

Table 3: Modes of the major scale shown with their intervals and what degree their first degree is in the major scale (relativity)

Name	Relativity	Intervals
Ionian	1	TTSTTTS
Dorian	2	TSTTTST
Phrygian	3	STTTSTT
Lydian	4	TTTSTTS
Mixolydian	5	TTSTTST
Aeolian	6	TSTTSTT
Locrian	7	STTSTTT

Ionian is the modal name for the major scale and Aeolian is the modal name for the minor scale. These 2 modes are by far the most common in western pop music but others are used more frequently in different genres and styles. To help understand the relativity of the modes, consider why C Ionian contains all of the same notes as A Aeolian.

4.2. Algorithms

4.2.1. Melody Generation

‘‘Melody Generator: A Device for Algorithmic Music Construction’’ (Povel, 2010) discusses some ideas for algorithmic melody generation. One of these it calls the ‘‘Chord-Based Model’’ and it fundamentally works by using a chord as a template for melody construction. Notes within the chord are placed on ‘strong’ beats in the rhythm to create a ‘skeleton melody’ and notes of the scale are used to connect them to create the final melody. This simple technique serves as an ideal basis for a program within our system which could be passed a chord as input and produce as melody as output. Such input and output types are ideal for a program designed to sit in a pipeline of others such as ours.

4.2.2. Harmony Generation

Harmonising Bach chorales is a common exercise for music students, whereby they are given a melody line and tasked with adding 3 additional lines beneath it in pitch to produce a satisfying piece of music. ‘‘Bach in a box’’ (McIntyre, 1994) attempts to automate this process by producing a large amount of potential solutions and evaluating them on criteria established by musicology. This criteria is standard for Bach chorales and includes smoothness, range and motion. ChoraleGUIDE (Pankhurst, 2009) is a popular resource for undergraduate and A Level music students looking to improve their chorale harmonisation skills. The algorithm I implemented for the harmony generating aspect of the system is based on the method outlined by Pankhurst:

add simple bassline beneath melody

modify bassline to boost its evaluation against certain criteria

fill in middle part

This same idea of first writing a simple bass line, carefully improving it according to criteria and then filling in the middle part accordingly is also described in another popular textbook ‘‘Harmonising Bach Chorales’’ (Gill, 2018).

To implement this we do a depth-first search in the bassline improvement step to find the solution which best satisfies the criteria proposed in ChoraleGUIDE including:

balance of steps and leaps

no consecutive leaps in same direction

no repeated notes

not too similar to melody line

no ‘sirening’ (up-down-up-down stepwise repetition)

5. Design

5.1. Overview

As there are many ways to cut a cake, there are many ways to divide the overall functionality of our system into orthogonal programs. Figure 1 outlines how the system is broken down into building blocks which can be combined in various ways to produce output.

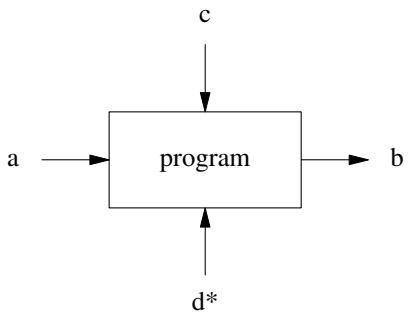


Figure 1: graphic representation of example program functionality

Figure 1 shows an example program in the format we use to represent the components of our system. The program takes input of type `a` via `STDIN` as well as input of type `c` as a command line argument and optionally input of type `d` as another command line argument. After reading these inputs, the program writes output of type `b` to `STDOUT`.

5.2. Mode Generator

The ‘‘mode generator’’ is a program which takes as command line arguments a set of notes and outputs the set of modes which each contain all of the notes in the input. If STDIN input is supplied then only modes also in the input set will be in the output set, otherwise all modes are considered. This functionality can be formalised using the following notation: This functionality is formally defined in equation 1

$$INP = \text{set of modes inputted via STDIN}$$
$$M = \text{set of all modes}$$
$$N = \text{set of notes passed as arguments}$$
$$P(m) = \forall n \in N : n \in m$$
$$output = \{m \in M \mid (m \in INP \vee INP = \varnothing) \wedge P(m)\}$$

Equation 1: mode generator output defined using set notation

Figure 2 shows the functionality of the mode generator in terms of its input and output types.

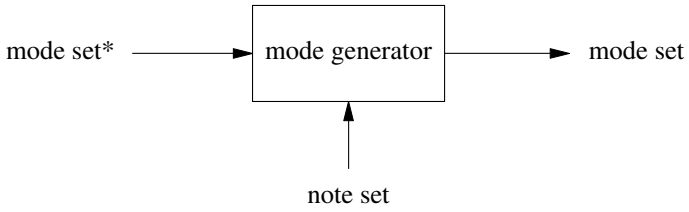


Figure 2: graphic representation of mode generator functionality

5.3. Interval filter

The interval filter is a program which takes as input a set of modes and also a set of intervals. It outputs all of the modes from the input which have the intervals specified. If no modes are given as input it outputs the modes from the set of all modes which have those intervals. Equation 2 formally defines this functionality.

$$INP = \text{set of modes inputted via STDIN}$$
$$M = \text{set of all modes}$$
$$I = \text{set of intervals passed as arguments}$$
$$P(m) = \forall i \in I : i \in m$$
$$output = \{m \in M \mid (m \in INP \vee INP = \varnothing) \wedge P(m)\}$$

Equation 3: interval filter output defined using set notation

Figure 3 shows the functionality of the interval filter in terms of its input and output types.

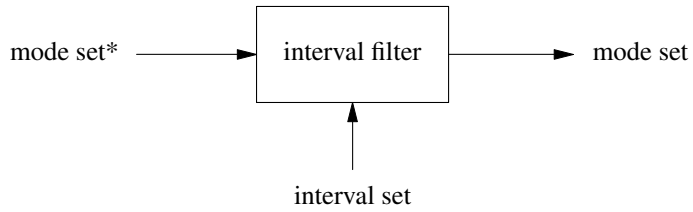


Figure 3: graphic representation of interval filter functionality

5.4. Chord Builder

The chord builder is a program which takes as input a set of modes and also a degree of the scale to build a chord from with it as the root. Optionally, it may also take the degrees of any extensions to be added to the chord, relative to the chord root. For each mode in the input there is a corresponding chord in the output set. Chords are written to the output paired with the mode from the input used to build them. As with the previously mentioned programs which take as input a mode set via `STDIN`, if no modes are supplied the set of all modes is used.

Figure 4 shows the functionality of the chord builder in terms of its input and output types.

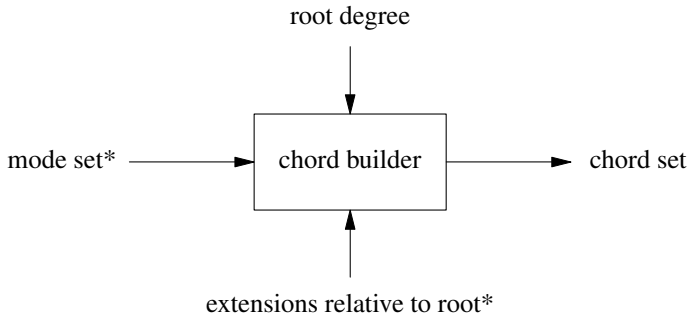


Figure 4: graphic representation of chord builder functionality

5.5. Melody Generator

The melody generator is a program which takes as input as set of chords and produces a melody for each one which would work played concurrently with that chord. In addition to a set of chords as input, it takes

the length of the melody to be generated and a seed value for randomness.

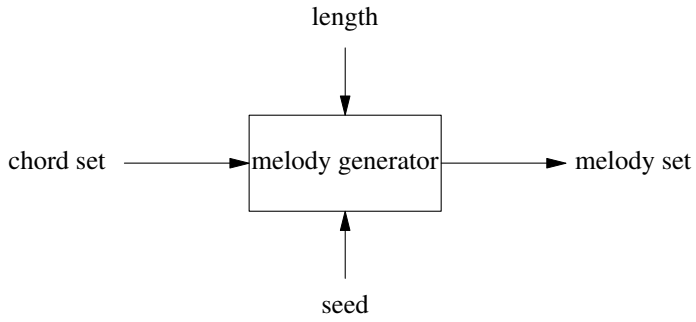


Figure 5: graphic representation of melody generator functionality

5.6. Melody Harmoniser

The melody harmoniser takes a set of melodies as input and for each one writes 3 part harmony for it to the output set of harmonised melodies. Each 3-part harmony has the melody in the highest pitch line, with 2 accompanying lines beneath it in pitch.

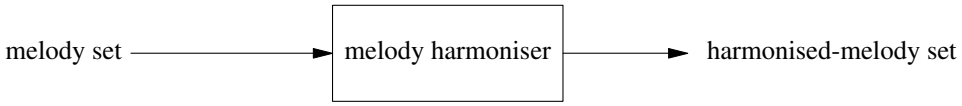


Figure 6: graphic representation of melody harmoniser functionality

5.7. MusicXML Formatter

The musicxml formatter reads a set of harmonised melodies and outputs sheet music containing each one after the other, represented using MusicXML (Good, 2001).

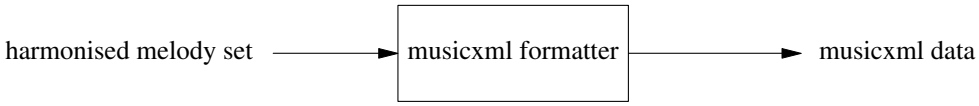


Figure 7: graphic representation of musicxml formatter functionality

5.8. Mode Displays

To demonstrate the extensibility of the system, I developed 2 alternate end-points for mode sets to be pipes into. Instead of generating musical ideas which could be piped into other programs, these draw alternate representations of modes in the terminal for the user to read. There is one to display modes on a guitar fret-board and one to display them on a stave using traditional key signature notation.

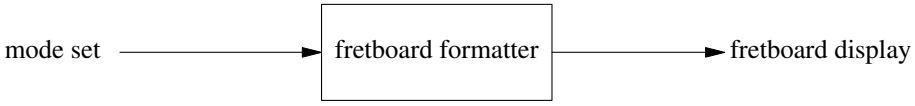


Figure 8: graphic representation of fretboard formatter functionality

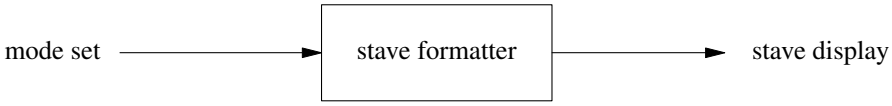


Figure 9: graphic representation of stave formatter functionality

5.9. Component Combinations

Although each component program provides useful functionality alone, it is the compatibility between them which is the main asset of this design. Various permutations of the programs can be used to produce different results.

Figure 10 shows a particularly long pipeline beginning with generating a set of modes and ending with a set of harmonised melodies represented in MusicXML format. The number of harmonies in the output will be the same as the number of chords outputted from the chord builder because for each of the programs between them there is one member in the output set generated from each member of the input set.

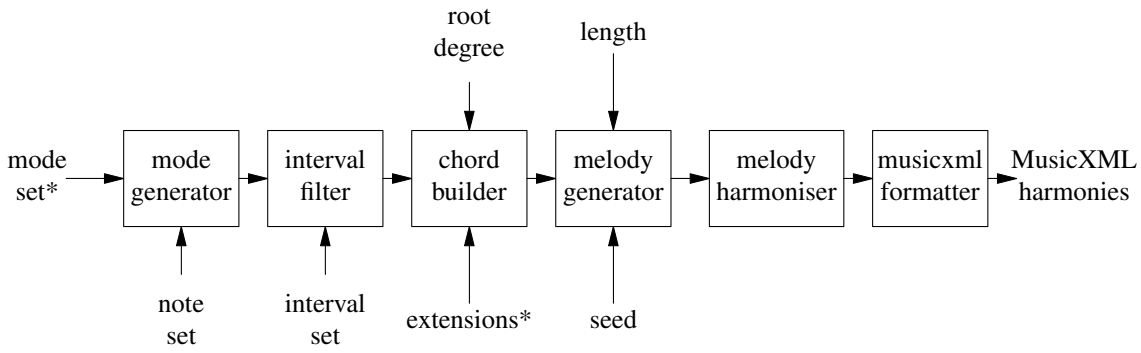


Figure 10: graphic represenetation of pipeline from modes to harmonies

A user may choose to build up such a pipeline incrementally by first only only using the mode generator, analysing the output and then deciding to append the interval filter to the pipeline, thereby removing modes which do not meet their criteria. Once satisfied, they could examine the chord of a given nature for each of these modes using the chord builder and then if they wish append the melody generator to produce a melody to work over each of the chords built. From here it would be simple to append the harmoniser and MusicXML formatter to the pipeline and redirect the output to a file using the standard > Unix operator. MusicXML files can be opened in a variety of scorewriters, such as Musescore (Todea, 2015), which offer playback functionality, allowing the harmonies to be listened to and edited in other programs.

It is important to note that the pipeline could end with any of the programs and the output could be redirected to a file by the user. This may be satisfactory and the end of the users interaction with the system, or they may wish to use the file later to input into another pipeline build using the components of the

system. Not only this, but the user may wish to manually write musical data to files and then input them into a pipeline. The transmission language used by the system has deliberately been kept as simple as possible to make this easier for users.

Table 4 lists the STDIN input and STDOUT output types for each of the component programs. If a program Y has input type of A and and a program X has output type of A then X can be piped into Y.

Table 4: Component programs shown with their input and output types

Program	Input Type	Output Type
Mode generator	Mode set	Mode set
Interval filter	Mode set	Mode set
Chord builder	Mode set	Chord set
Melody generator	Chord set	Melody set
Melody harmoniser	Melody set	Harmonised melody set
MusicXML formatter	Harmonised melody set	MusicXML
Stave formatter	Mode set	Stave display
Fretboard formatter	Mode set	Fretboard display

6. Implementation

6.1. Technologies

Since this project is about exploring the effectiveness of applying Unix philosophy to computer aided composition, the software targets Unix based platforms. These include operating systems based on Linux, Hurd and BSD. The basic requirement for the platform is that is provides Unix pipes for the programs to communicate with.

The language with the most portability across Unix-like platforms is C. Like Unix, C is strongly associated with the 'New Jersey' philosophy (Gabriel, 1991). According to Gabriel it was 'designed using the New Jersey approach' and 'designed for writing Unix'. He attributes its popularity to its simplicity because it makes C compilers easier to develop. As mentioned in the 'Program Design in the UNIX Environment' (Kernighan, 1984), C was originally the language for the Unix kernel and applications and 'essentially everything was written in C', which made the software easy to modify and customise. Continuing with the theme of Unix style and 'New Jersey' style, I wrote the software in C. This also helps make the software as portable as possible between the various Unix-like operating systems. To further maximise the compatability and portability, the standard of C I wrote in was ANSI 99 (ISO, 1999). The majority of the development was carried out using GCC as a compiler but the code has proven to be perfectly compatable with the minimalist C compiler TCC (https://bellard.org/tcc/). All development and testing was done targeting an x86 Linux system.

6.2. Common Data Types

There are a number of data types used by the programs which make up the overall system. The goal of the encoding method is to represent musical data in a way which is simple for a computer to interpret and manipulate but also simple to encode and decode when reading input or writing output.

"Bach in a box" (McIntyre, 1994) chooses to represent pitch using integers but in such a way that each integer maps to a pitch in the C major scale. For example, 0 represents C, 1 represents D, 2 represents E and so on. The advantage of this method is that programming the internal logic for building chords, melodies and harmonies becomes simpler because to go up a step within the mode you can just add 1 to a number, to go up a fifth in the step you can just add 4 to the value and so on. However, the problem with this encodiing method is that if you wish to work in any mode other than C major (C Ionian) then transposition is required when encoding/decoding takes place. Additionally, it is impossible to produce music using more pitches that what can be found in a single mode (chromaticism). This method is unsuitable for our usage because the pipeline design of our system places high importance on the simplicity and efficiency of reading and writing data. Encoding data this way would also mean that future extensions implementing chromaticism would be made more difficult. Encoding data this way would also reject the extensibility objective of the project by making the implementation of chromaticism more difficult.

Our pitch encoding method is simlilar to that of MIDI, stemming from USI (Smith, 1981), whereby all 12 tones are accounted for and the interval between any 2 adjacent integers is 1 semitone. This means that the fundamental encoding and decoding of each note is simple and can be done without knowledge of a tonal context, which is particularly useful for the mode generator. Additionally this representation means that it is no harder to program functionality using chromaticism or modes other than C major than it is to program functions working entirely within the confines of C major. To implement scales on top of this we represent a whole tone as the integer 2, a semitone as the integer 1 and the major scale as a circular list comprising of these types. It follow then that modes of the major scale can then be intuitively derived by treating different elements in the list as the first. This encoding opens up an interesting avenue for future exploration: using a different set of intervals as the base scale to derive modes from. Although not used in popular music, alternate modes such as "The modes of limited transposition" (Street, 1976) are of interest in the field of musicology and could be easily implemented into our system.

Table 5 shows the types used to encode musical data in this project.

Table 5: Musical data types and how they are encoded in our system. ○[x] represents a circular list of elements of type x

Data	Encoding
Pitch	Int
Interval	Int
Degree	Int
Scale	○[Interval]
Mode	(Scale, Degree)
Root	Pitch
Key	(Root, Mode)
Chord	[Pitch]
Line	[Pitch]
Harmony	[Line]
Alteration	Int

6.3. Common Functions

6.3.1. Input/Output

todo: explain pitch spelling, refer to papers

Input and output functions play an imporant role in the working of the system because they are what allow the component programs to be combined to provide different operations.

The function to encode a note is surjective but not injective which means that every internal integer used to represent pitch has at least one string such as 'C' which maps to it but some have two as, for exampe, C# and Db will map to the same internal representation because they have the same pitch. The nature of the decoding function depends on the mode of the data being decoded as, for example, the pitch represented internally by the integer 1 may be outputted as C# or Db. In traditional music representation, each key signature is written using either entirely sharps or entirely flats and if it a key signature contains C# then it cannot contain a C natural. However, if the key signature contained a Db it would mean that there is no D natural in the key signature (but there could be a C natural). This means that some key signatures must be written using sharps and others must be written using flats else double sharps or double flats would be necessary to accurately describe it. In computer science this is known as the problem of 'pitch spelling' and powerful algorithms such as 'ps13' (Meredith, 2006) have been developed to solve it accurately even when the mode context is not known. The design of our system keeps track of the mode context so such complicated algorithms are not necessary but some work is still required to produce valid output. Our pitch spelling algorithm basically checks whether the current mode should be written using sharps or flats and then uses whichever accidental is correct for that to write non-natural notes with. Figure 11 shows a high level representation of the function used to check if a key can be represented with a given accidental where accidental is either sharp or flat:

```
is_correct_accidental(key, accidental)
    if key is one of the few which can be written with either
        return true
    if accidental is flat then dir <- 1 else dir <- -1
    for each pitch in key
        if pitch is non-natural and
        pitch + dir is in key and
        pitch + dir*2 is accidental
            return false
    return true
```

Figure 11: pseudocode for determining whether a given accidental is the correct one for writing a given key signature with

The idea here is two iterate through pitches in the key and check that if there is, for example, a Db in the key there is not also a D natural. We must check an extra pitch in the same direction direction though to avoid false negatives in cases such as when there is a Bb and Cb because Cb has the same pitch as B

natural. We must also check that the next pitch in the same direction is accidental before saying its invalid else we would produce false negatives in such cases as when there is a Bb and a Cb in the key (because Cb and B natural have the same pitch so the integer encoding could mean that there is a Bb and B natural).

6.3.2. Internal

Amongst other things, the shared internal functions faciliate using modes as frameworks within our semitonal pitch encoding scheme. The methods to implement this functionality make use of modular arithmetic to make sure that scales and pitches 'wrap around' (one semitone above G# is A).

Figure 12 pseudocode shows how you can work with steps in our pitch system. Additions to degree and pitch remain within their respective fields (using modular arithmetic in the proper implementation).

```
apply_steps(degree, key, steps)
    pitch <- key(degree)
    if steps < 0 dir <- -1 else dir <- 1
    if steps < 0 degree <- degree - 1
    for s <- 0 to steps
        interval <- MAJOR_SCALE[degree+key.mode]
        pitch <- pitch + dir*interval
        degree <- degree + dir
    return pitch
```

Figure 12: pseudocode for appling a given positive or negative number of steps to a given degree of a given key and return the final pitch

As you can see, the idea here is to repeatedly add the intervals of the major scale to a starting pitch. The mode of the major scale which is being used can be thought of as an index offset. This function is used extensively when work must be done using a mode as a framework, for example building chords, melodies, harmonies and filtering by intervals.

6.4. Mode Generating

The mode generator works by taking a list of notes as input and returning a list of all the keys which contain all of the nodes where each key is a root note paired with a mode. Figure 13 shows how this core functionality is achieved:

```
process_notes(notes)
    for root_note in notes
        for mode in MAJOR_SCALE.modes
            key <- new_key(root_note, mode)
            for degree in MAJOR_SCALE.degrees
                key_freq[key(d)][d+key.mode]++
    return all keys (r, m) where key_freq[r][m] is notes.len
```

Figure 13: pseudocode showing how a list of all keys where each contains all of the notes given as input is returned

The idea here is to iterate through all the inputted notes and for each one, treat it as the root of each of the modes of the major scale and for each mode where that is so find all of the relative modes and mark them as containing that note. Recall that relative modes are ones which contain all of the same notes. In the proper implementation this is broken down into 2 functions where each one calls common functions for working with steps and key matrices. As mentioned in the design section, the mode generator can optionally take additional input in the form of a list of modes whereby modes not in this list will not be outputted even if they contain all of the input notes. This functionality is achieved by applying a mask to the key matrix before outputting the keys in it which match the criteria. The common functions `read_key_list`, `init_key_field` and `print_matching_keys` enable this functionality.

6.5. Interval Filtering

Interval filtering works by reading a set of keys and only outputting the ones from that set which contain all the intervals given as additional input. In this context, the intervals refer to the difference in pitch between the root (first pitch) of the scale and a given degree relative to the difference in pitch between the root and that degree of the major scale. For example, the minor scale (Aeolian) has a flat 3rd, flat 6th, flat 7th because each of those intervals is 1 semitone smaller than it would be in the major scale (Ionian). It has a natural 2nd, natural 4th and natural 5th because those intervals are the same size as they would be in the major scale.

```
correct_alteration(degree, mode, alteration)
    interval <- 0
    major_interval <- 0
    for step is 0 to degree-1
        interval <- interval + MAJOR_SCALE[step+mode]
    for step is 0 to degree-1
        major_interval <- major_interval + MAJOR_SCALE[step]
    diff <- interval - major_interval
    if alter is natural
        return true if diff is alter else return false
    elif alter is flat
        return true if diff < 0 else return false
    elif alter is sharp
        return true if diff > 0 else return false
```

Figure 14: Pseudocode showing how to check whether a given degree is altered by a given alteration in a given mode relative to the major scale.

The idea in the code shown in figure 14 is to calculat the difference in semitones between the first of the mode and the given degree of that mode, then do the same for the major scale and then compare the results. If it is claimed that the given degree of the given mode is natural then the claim is correct if and only if the calculated intervals are equal. If the claim is that the degree is flat then it is true if and only if the major interval is greater than that of the given degree in the given mode. If the claim is that the degree is sharp then it is true if and only if the major interval is less than that of the given degree in the given mode.

6.6. Chord Building

Chord building is a relatively simple exercise given we have access to the `apply_steps` function defined earlier and the mode to build the chord from. For example, outputting the notes of a triad is a case of outputting the root note, the note 2 steps above that (the third) and finally the note 4 steps above the root (the fifth). Remember that the interval in semitones a pitch refers to depends upon the mode we are working in.

```
build_chord(key, degree, extensions)
    chord <- list()
    root <- key(d)
    chord.append(root)
    chord.append(apply_steps(degree, key, 2))
    chord.append(apply_steps(degree, key, 4))
    for each extdegree in extensions
        chord.append(apply_steps(degree, key, extdegree)2
    return chord
```

Figure 15: Pseudocode for building the triad of a given degree of a given key with any given extensions

6.7. Melody Generation

As mentioned in the methodology section earlier, the ‘‘chord-based model’’ from ‘‘Melody Generator: A Device for Algorithmic Music Construction’’ (Povel, 2010) is used in our melody generator because it takes chords as input and generates a melody for each one. The concept is to create a ‘skeleton melody’ on all the weak beats (every other beat) using only pitches from the chord used as input and then fill in the gaps using any pitch from the modal context.

```
generate_line(len, tones, k)
    melody <- list(len)
    for each strong beat b in melody
        b <- rand_element(tones)
    for each weak beat b in melody
        steps <- difference in steps between b.prev and b.next
        if steps is 0
            passing_step <- 1 if even(rand()) else -1
        else
            passing_step <- random_element([0..steps])
        b <- b.prev.apply_step(passing_step)
    return melody
```

Figure 16: Pseudocode showing how melody generation in our system is achieved using a stochastic implementation of the chord-based model.

The idea shown in figure 16 is an implementation of the chord-based model using randomness as a decider where necessary. This is chiefly how the melody generator component in our system works. Smooth stepwise motion is encouraged by the algorithm because this is traditionally what makes melodies sound ‘musical’ to our ears but randomness is injected as a decider when there are equally ‘smooth’ choices available to break up any accumulating monotony.

As with elsewhere in the system, modular arithmetic is employed to ensure that results from operations on pitches and degrees remain within their respective fields.

6.8. Melody Harmonisation

The method used by our melody harmoniser is inspired by ChoraleGUIDE (Pankhurst, 2009), where the fundamental idea is to first generate a bassline complimenting the melody such that each chord they form together is a primary chord (1, 4 or 5) with its root note in the bass part. Next, the bassline should be improved by either creating a different primary chord with the combination of pitches, putting the 3rd of the chord in the bass instead of the root, applying both those techniques or applying none of those techniques (4 total options). During this phase of improving the bass line it is important that the violation of certain musical rules must be minimised to produce pleasent harmonies. Finally, the middle part should be added such that it is as smooth as possible and combines with the melody and bass parts to form a sequence of chords.

This algorithm can be effectively carried out by a computer by breaking down the problem into searching and evaluating. The most delicate and important part of the process is that of improving the bassline, so to do it we build a search tree by treating one end of the line as the root. Each node can have up to 4 children corresponding to the 4 improvement options for that note in the bassline. Moving to the next layer in the tree represents choosing that improvement option and examing the next note in the bass line sequence. The leaves are reached once the end of the bass line is reached, which is the same length as the melody it is being written to compliement. Once a leaf has been reached the bassline resulting from that path is evaluated by counting its faults. These evaluations then back-propagate up the tree so that the optimal path in the tree can be determined by picking the child with the least faults at each stage.

$$n = \text{melody length}$$

$$\text{tree height} = n - 1$$

$$\text{maximum possible viable basslines (leaves)} = 4^n$$

$$\text{maximum total nodes in tree} = \frac{4^{n+1} - 1}{3}$$

Equation 4: Properties of the search tree explored during the bassline improvement step of harmonisation

```
improve_bassline(bassline, melody, current_beat, key)
    if current_beat is bassline.len
        bassline.faults <- count_faults(bassline, melody, key)
        return bassline
    bassline_b <- bassline.copy()
    bassline_c <- bassline.copy()
    bassline_d <- bassline.copy()
    bassline_b[current_beat].alternate_chord()
    bassline_c[current_beat].invert_chord()
    bassline_d[current_beat].alternate_chord()
    bassline_d[current_beat].invert_chord()
    a = improve_bassline(bassline, melody, current_beat+1, key)
    b = improve_bassline(bassline_b, melody, current_beat+1, key)
    c = improve_bassline(bassline_c, melody, current_beat+1, key)
    d = improve_bassline(bassline_d, melody, current_beat+1, key)
    return least_faults([a, b, c, d])
```

Figure 17: Pseudocode demonstrating the bassline improvement search, where `improve_bassline` returns an improved bassline

Figure 17 shows how `improve_bassline` is defined recursively to perform a depth first and return the optimal result according to evaluations carried out by `count_faults`. Once the optimal bassline has been returned by `improve_bassline` the middle part is generated in a linear fashion such that, the final chord the most complete one possible and each preceding note is the nearest pitch within the current chord to the middle pitch in the neighbouring chord. By a chord being 'complete' it is meant that it contains all degrees of the triad: the 1st, 3rd and 5th. For the other chords in the harmony, some omissions are allowed if it means that faults can be avoided. Table 6 shows the possible degrees of the chord being considered which can be put in the bass and melody parts to create pleasant harmony. The middle part may be degree 1, 3 or 5. These rules are based off of the theory described in ''Harmonising Bach Chorales'' (Gill, 2018).

Table 6: Allowed combinations of bass and melody degrees of the chord they are being treated as

Bass degree	Melody degree
1	1
1	3
1	5
3	1

Features `count_faults` considers faults are the following:

- consecutive bass notes
- bass and melody notes being exactly an octave (12 tones) apart for consecutive beats
- leap of a tritone (flattened fifth interval)

- consecutive intervals of greater than a step in the same direction
- consecutive repetition of a 2 note pattern (eg up 1 step, down 1 step, up 1 step, down 1 step)

These faults are outlined also in ChoraleGUIDE (Pankhurst, 2009).

6.9. Conversion to MusicXML

To avoid manually playing or transcribing output from the harmony generator, its output can be piped into the musicxml formatter which takes the input and writes the musicxml representation of it to `STDOUT` where it can be easily redirected to a file using the standard unix > operator. Since overall the pipeline processes several ideas in parallel (melody harmoniser can take any ammount of melodies and will output harmonisations for all of them) we need a system of presenting multiple ideas in a legible way. The way we do this in our MusicXML representation is to put each idea in sequence with a pause seperating each one. This means when the MusicXML data is interpreted by a scorewriter like MuseScore the ideas can be played back on after the other or the user can easily skip to a specific one.

Another challenge involved in this process is how to convert our encoding of pitch to theirs; our system simply uses the 12 tones of western music but in MusicXML the octave of each pitch must be manually specified. It follows intuitively that the default octave for the melody must be the highest, with the middle part beneath that and the bass part beneath that but each part may cross over into adjacent octaves at some point. An octave in MusicXML is from C to B so without proper handling a step up of a semitone in our system from B to C would be interpreted as a step down of 11 semitones.

Additionally, the MusicXML representation of pitch does regard semitones as primitives. That is, each pitch is represent as a pair of a natural note and and alteration applied to it. So C# which is encoded as the integer 1 in our system must be converted to (C, #) to be valid in MusicXML. For simplicity, we represent all non-naturals as sharps for our MusicXML output. Proper key signatures and pitch spelling could be implemented as a future improvement to readability, but accurate playback is not compromised at all by this decision.

```
write_part_line(line, octave)
    for each pitch in line
        if pitch is non-natural
            write_mxml_note(pitch-1, SHARP, octave)
        else
            write_mxml_note(pitch, natural)
        interval <- shortest_interval(pitch.prev, pitch)
        if interval < 0 and pitch.prev < pitch
            octave <- octave - 1
        elif  interval > 0 and pitch.prev > pitch
            octave <- octave + 1
```

Figure 18: Pseudocode showing the logic for writing our internal line representation as MusicXML

In figure 18 `shortest_interval` calculates the shortest interpretation of any interval (there are 2 possible interpretations of any interval due to the circular nature of pitch) and assumes this is what is meant by the representation. For example, the shortest interval betweeen pitch 11 and pitch 0 is 1, not 11. If it turns out that the interval is negative (a step down has happened) yet the value of the first pitch is less than the second we must have 'wrapped-around' the circle of pitches and we have entered the octave below. The same principle applies in reverse if a step up has happened yet the value of the second note is less than the value of the first so the octave above has been entered.

6.10. Stave Key Signature Display

As a component program our system provides functionality to display modes as a key signature on a stave in the terminal. A core part of the algorithm we use to do this is where a list of flags is produced where each element represents whether that line on the stave should have an accidental (sharp or flat) on it. This

list of flags must be calculated from a number which represents the total number of accidentals and the type of accidental being used because each key is mapped to a position on the 'circle of fifths' where its position on this circle corresponds to those numbers.

```
note_status(alteration, quantity)
    line <- B_LINE if alteration is FLAT else F_LINE
    interval <- PFOURTH if alteration is FLAT else PFIFTH
    for a in accidentals
        stave[line] <- true
        line <- line + interval
    return stave
```

Figure 19: Pseudocode showing how to produce a list representing which lines of the stave should be altered to represent a given key signature

Traditionally alterations must be added to the lines in a specific order and, as in figure 19, this is done by starting on a specific pitch (line) and moving from it by repeatedly adding a specific interval. The starting pitch and interval depend on whether the key is represented using sharps or flats; if flats are used then the starting line is B and the interval is a perfect fourth otherwise the starting note line is F and the interval is a perfect fifth.

Once the list from `note_status` has been obtained, it is a matter of printing the lines to screen. Do to the nature of printing to the terminal being such that lines must be printed sequentially from top to bottom, the correct indentation spacing must be calculated. Figure 20 shows example output from this program and demonstrates correct indentation of the symbol for each line. The number of spaces in the indentation corresponds to the order in which they are traditionally writtten: from a starting note with a repeating interval, a similiar idea to in the `note_status` function.

```
Key: F# major (6#)
------------------

   #
=#================
      #
====#=============
  #
==================
     #
==================

==================
```

Figure 20: Output from the terminal key signature stave display for F# major, which has 6 sharps.

As with any other component in the system, input maybe given as a list in which case multiple staves would be printed sequentially corresponding with each element of the input list.

6.11. Fretboard Mode Display

A common way for guitar students to learn modes is to view them on a fretboard diagram. These look as if you had put the guitar facing upwards on your lap and were viewing the fretboard from directly above with the lowest string closest to your body and the highest the furthest away. It is assumed that the guitar is a right handed one, which means the frets lowest in pitch are to the left. Without going into too much detail, the circular nature of modes mean that they are repeated all over the fretboard. Figure 21 shows the fretboard diagram outputted by our program for C Ionian.

```
Cn Ionian:
| 4|   | 5|   | 6|   | 7| 1|   | 2|   | 3| 4|   | 5|   | 6|   | 7| 1|   | 2|   | 3|
| 1|   | 2|   | 3| 4|   | 5|   | 6|   | 7| 1|   | 2|   | 3| 4|   | 5|   | 6|   | 7|
|   | 6|   | 7| 1|   | 2|   | 3| 4|   | 5|   | 6|   | 7| 1|   | 2|   | 3| 4|   | 5|
|   | 3| 4|   | 5|   | 6|   | 7| 1|   | 2|   | 3| 4|   | 5|   | 6|   | 7| 1|   | 2|
|   | 7| 1|   | 2|   | 3| 4|   | 5|   | 6|   | 7| 1|   | 2|   | 3| 4|   | 5|   | 6|
| 4|   | 5|   | 6|   | 7| 1|   | 2|   | 3| 4|   | 5|   | 6|   | 7| 1|   | 2|   | 3|

  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

Figure 21: Fretboard diagram for C natural Ionian outputted by our program. Numbers along the bottom represent the fret numbers and the numbers on the fretboard represent the degree of the mode which resides on that fret for that string.

This problem rather lends its-self to our pitch representation scheme since each integer value we use to represent a different pitch corresponds to the fretboard as adjacent frets are 1 all semitone apart too. Additionally, this means that if we can create a method to write one string then this can be applied to the rest of the strings with an offset to produce a correct result.

```
write_string(fret, mode)
    string <- list()
    for degree is 0 to MAJOR_SCALE.degrees
        string[f] <- d
        fret <- fret + MAJOR_SCALE[mode + degree]
    return string
```

Figure 22: Pseudocode showing how to write the degrees of a given mode with its root note at a given fret to a string represented as a list of integers

Table 7: Each string of the guitar and the corresponding value passed to the ‘fret’ parameter in `write_string` where f is the starting fret of the mode on the low E string

String	Starting fret
E	f+5*7+1
B	f+4*7+1
G	f+3*7
D	f+2*7
A	f+7
E	f

The offset added to f in table 7 is that way simply because that is the definition of standard guitar tuning in semitones. For alternate tunings the offset would need to be adjusted accordingly.

7. Evaluation

7.1. Methodology

For effective testing and evaluation of the system a quantitative method of output analysis must be established. Empirical analysis on Bach chorales has been done by segmenting the the music into 'pitch-class sets' (Rohrmeier, 2008). This abstracts away intricacies of individual voice lines and represents the music as a sequence of chords. With a simpler representation of the music, frequency of pitch-class sets and pitch-class set transitions can be examined. Rohrmeier discusses the significance of symmetry in pitch-class set transitions. He finds that transitions show a high degree of symmetry. That is, for all pitch-class sets X and Y, the frequency of X-Y transitions is roughly equal to the frequency of Y-X transitions. This corresponds with music theory ideas of 'tension' and 'resolution'.

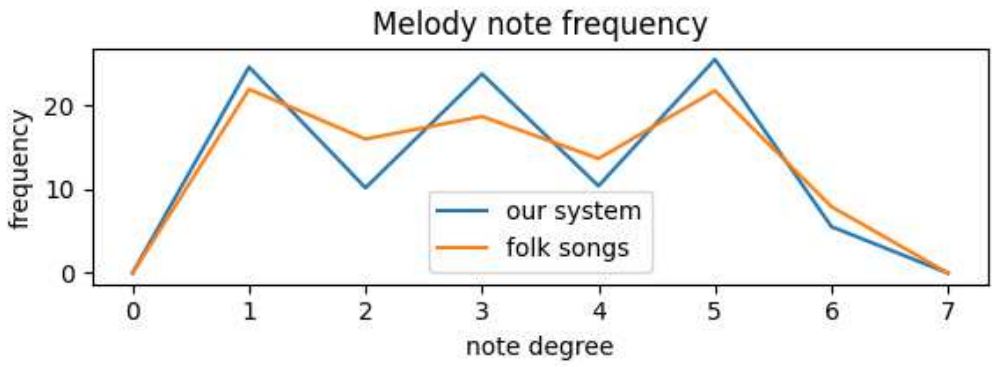
A 'tonal hierarchy' represents the importance of each diatonic note in a given tonality. An empirical investigation into this concept has been done in 'A Theory of Tonal Hierarchies in Music' (Krumhansl, 2010) where listeners were played an incomplete scale followed by the tonic of the scale and then rated the completeness of what they heard. This experiment is known as the 'probe tone method' and figure 3.1 in that paper shows the results. These results also reflect ideas established in traditional western music theory because notes belonging to the tonic triad scored the highest. The results from this experiment provide a good benchmark for the frequency of notes in music. That is, the frequency of notes in melodies which listeners find satisfying will roughly match the results of the 'probe tone' experiment.

With these ideas in mind, we can start to apply similar methods to existing compositions and build a picture to which we can compare analysis of the output of this project against. 'Music21' (Cuthbert, 2010) is a Python module which provides a framework for musicology. As well as providing a rich toolkit for the analysis of music, it also has a built in corpus of roughly 3000 pieces comprised of popular folk songs and works by over 20 iconic classical composers from varying eras. To create a benchmark to compare the output of my system to, I will apply the aforementioned music analysis techniques to the Music21 corpus using the functions it provides.

7.2. Melody

To evaluate the musicality of melodic ideas generated using our system, we compared output from the melody generator to data in the ''Essen Folksong Collection'' (Schaffrath, 1995), which is included in the Music21 corpus. Folksongs are quintessentially melodic; they propagated through societies and gained cultural significance not because populations learned the chord progressions but simply because the melodies where 'catchy' to the ear. Explained more concretely, there structure agrees with the findings of the aforementioned 'probe tone' experiment which found that listeners consider melodic phrases to sound more 'complete' when they end on specific notes of the scale (notes in the tonic triad). It follows then that if we use as input to the melody generator the tonic triad of a scale, the melodic output will have similar characteristics to folksongs. Conveniently, the Essen Folksong Collection is encoded in ABC format (Walshaw, 1997), which focuses primary on representing the main melody line of the music while additionally providing the tonality of the piece (the musical mode it is built from).

To test this theory, I produced 100 6-note melodies (each with a different seed value) using the melody generator and used Music21 to carry out frequency analysis on degrees of the scale in the melody. For example, if a C note is found in the key of C then the frequency of 1st degree notes has increased by 1. If a D note is found in the key of C then the frequency of 2nd degree notes has increased by 1, and so on. Evaluating the frequency of degrees of the scale rather than of notes is much more useful because the degree of the scale it is built from represents the function of that note in its melodic context. This idea of scalic context being what gives a note its character to listeners is proven by the probe-tone experiment because they were only able to assign a 'completeness' value to each note of the scale when it was preceded by stepwise ascension of the scale. Corresponding analysis was produced a subset of the Essen Folksong Collection. The pieces in the Essen Folksong Collection use a variety of tonalities so the tonality information in the ABC encoding was used to convert notes to degrees of their respective scales. Figure 23 shows a comparison of the frequency analysis of my system compared with that of the folksongs.



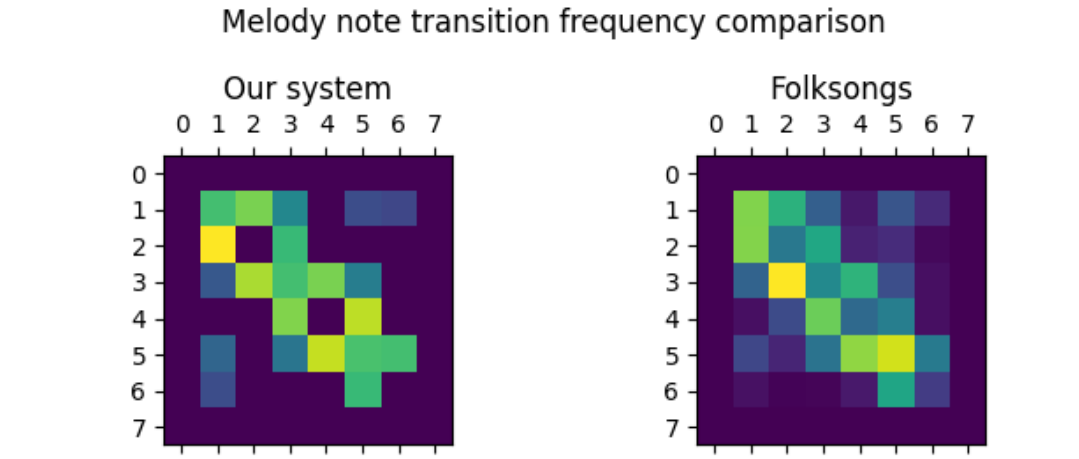
Difference between folk songs and our system

	Note I	Note II	Note III	Note IV	Note V	Note VI	Note VII	Total	Average
0	3	6	5	3	4	2	0	23	2

Figure 23: The distribution of note degrees as a percentage for each degree in output produced
by our system and in folksongs in the Essen Folksong Collection. The table shows, for each degree, the difference in percent between ours and the folksongs as an absolute value followed by the sum of these and the mean average of them.

As you can see, the notes of the tonic triad (1, 3 and 5) are the most common in both, which correlates with the results of the probe-tone experiment. However, the extent to which these notes are emphasised is higher in the output of our system. This is probably because the melodies were generated using a tonic triad as input. The emphasis of these notes could be made less prominent by passing extensions to the chord generator as arguments, but the level of emphasis played on the tonic triad notes in the example is acceptable. Alternatively, the amount of randomness used in the generation of melodies could be increased to even out the distribution.

The next graph shows the same method analysis applied note transitions rather than notes. It is essential to analyse the frequency of transitions as well as the frequency of individual notes to prove the musicality of the output of our system because it shows that the distribution of the note instances in the melody is satisfactory Aspell as just the overall frequency. The frequencies are shown on matrices whereby the value at row N column M represents the frequency of N-M transitions. Lighter colours represent higher frequencies.



% Difference between ours and folksongs

	I	II	III	IV	V	VI	VII
0	0	0	0	0	0	0	0
I	1	1	1	1	0	1	0
II	1	3	1	1	1	0	0
III	0	1	2	1	1	0	0
IV	0	2	0	3	4	0	0
V	1	1	0	1	2	2	0
VI	1	0	0	1	1	1	0
VII	0	0	0	0	0	0	0

Total and average % difference

Total	Average
37	1

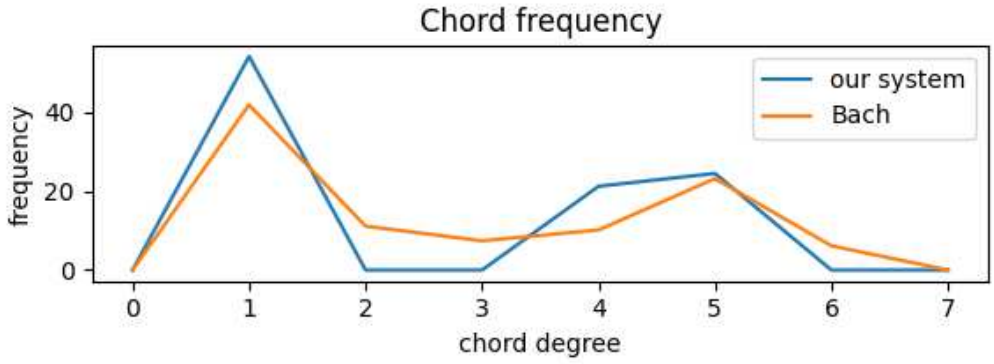
Figure 24: The distribution of note transitions as a percentage for each transition in output produced by our system and in folksongs in the Essen Folksong Collection. Lighter colours represent higher percentages. The tables show the difference in percentage for each transition between our system and the folksongs as an absolute value along with the sum of these and the mean average of them.

This analysis shows that both sets of melodies have a tendency to move stepwise, that is most transitions are to an adjacent note in mode used to build the melody. The analysis shows that our systems tends to produce a pattern-like and uniform melodies, which could be deemed as un-musical according to taste. A simple way to rectify this would be to increase the amount of randomness used in the algorithm. Both sets also show to have high levels of transition symmetry. That is, the amount of N-M transitions is similar to the amount of M-N transitions.

7.3. Harmony

Harmonic analysis was performed by comparing output from the melody harmoniser to a subset of the Music21 Bach corpus. As in ‘‘Statistical Properties of Tonal Harmony in Bach’s Chorales’’, (Rohrmeier, 2008) the music of Bach has been selected due to its consistent style and overwhelming popularity. Additionally, the principle of taking a single line melody and adding additional voices beneath it in pitch to create harmony was popularised by his working applying this technique to hymn tunes and it is this principle which the melody harmoniser applies. The techniques I used to analyse harmony are similar to those used to analyse melody, except instead of converting notes to scale degrees, chords are converted to scale degrees. The scale degree a chord maps to is the scale degree of its root note. The Essen Folksong Corpus would have been a poor choice for comparison because the pieces lack the harmonic depth which Bach provides and also the ABC format cannot accurately represent multiline harmonies such as what Bach and the output of the melody harmoniser deal in. As with the melodic analysis, I used our system to generate 100 6-note melodies but then piped them into the melody harmoniser and then into the musicxml formatter.

The Music21 Bach corpus is also encoded using MusicXML which means we can use the Music21 MusicXML 'chordify' function on our data as well as the Bach corpus to convert them to a list of chords. Bach's music changes key often (the mode used to generate the music can change to a different one during a piece) so this must be considered when converting the chords to scale degrees. Thankfully, Music21 provides functionality to re-analyse the key frequently at any point during the MusicXML data, which allows us to accurately perform frequency analysis on the scale degrees of the chords used. Figure 25 compares the frequency of chords in Bach's compositions and the output of the melody harmoniser.



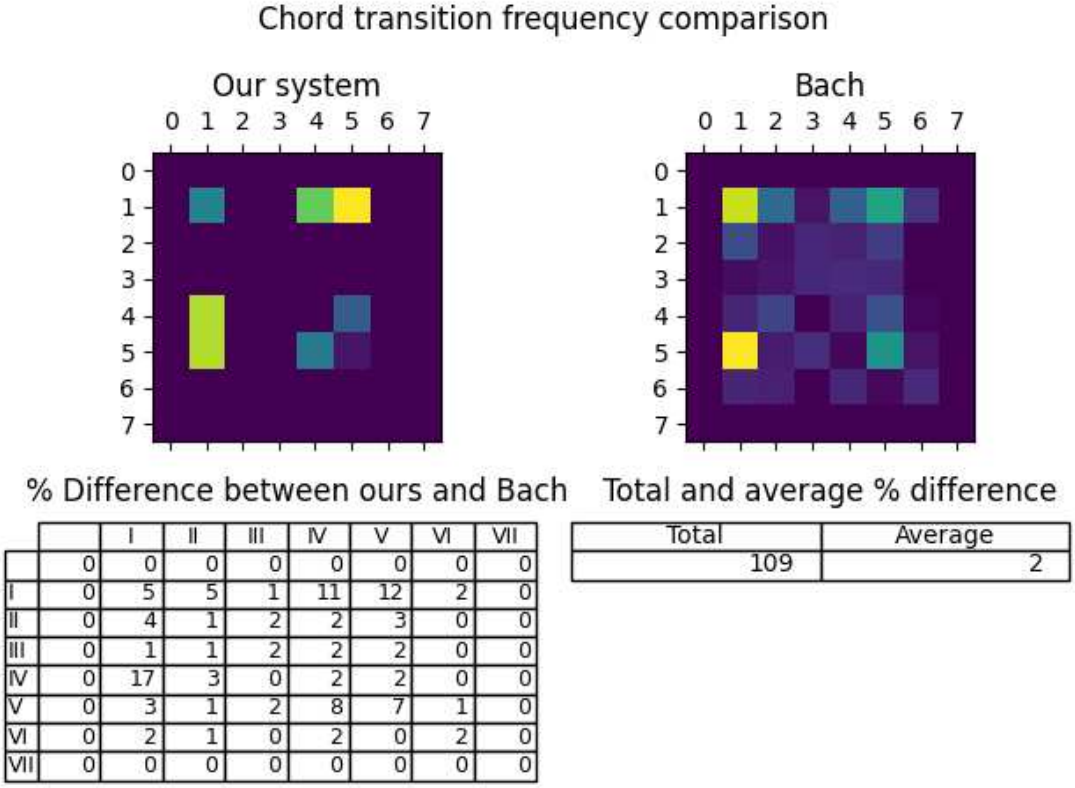
Difference between Bach and our system

	Chord I	Chord II	Chord III	Chord IV	Chord V	Chord VI	Chord VII	Total	Average
0	12	11	7	11	1	6	0	48	7

Figure 25: The distribution of chord degrees as a percentage for each degree in output produced by our system and in Bach compositions in the Music21 corpus. The table shows, for each degree, the difference in percent between ours and the Bach corpus as an absolute value followed by the sum of these and the mean average of them.

In music theory, chords 1, 4 and 5 are considered the primary chords and have particular importance. (Taylor, 1991), so it follows that they should be the most frequent in music adhering to a fundamental style. The analysis shows that this is the case for the output of our melody harmoniser and that it is the case for the Bach, apart from he seems to emphasise chord 2. The emphasis on chord 2 could be due to Bach's heavy usage of the 2-5-1 progression in his music, which is a staple of Baroque era music (Andrews, 1999).

As with the melody analysis, it is essential to examine the frequency of transitions as well of individual chords to ensure the distribution is also accurate. The next graph presents chord transition data the same way as was done for melody note transitions.



		I	II	III	IV	V	VI	VII
	0	0	0	0	0	0	0	0
I	0	5	5	1	11	12	2	0
II	0	4	1	2	2	3	0	0
III	0	1	1	2	2	2	0	0
IV	0	17	3	0	2	2	0	0
V	0	3	1	2	8	7	1	0
VI	0	2	1	0	2	0	2	0
VII	0	0	0	0	0	0	0	0

Total	Average
109	2

Figure 26: The distribution of chord transitions as a percentage for each transition in output produced by our system and in Bach compositions in the Music21 corpus. Lighter colours represent higher percentages.
The tables show the difference in percentage for each transition between our system and the Bach corpus as an absolute value along with the sum of these and the mean average of them.

In music a 'cadence' refers to a chord transition at the end of a phrase and the most fundamental cadences are the 'perfect cadence' (5-1), 'plagal cadence' (4-1) and 'imperfect cadence' (1-5) (Taylor, 1991). It is satisfying then that these transitions are shown to be the most common in the output of the melody harmoniser. The Bach analysis shows some emphasis towards the primary cadences, however not as much as I expected, with a distinct lack of 4-1 transitions. This could be due to the algorithm which Music21 uses to 'chordify' the MusicXML representations misinterpreting some instances of chord 4 as chord 1 (they both have the first degree of the scale as a note). The MusicXML for the Bach works is far more complex than the MusicXML for my harmonised melodies so its possible this is causing some complications for the chordify algorithm. Nonetheless, the correlation between the harmonised melodies produced by my system, Bach works and fundamental music theory is strong enough to be acceptable in my opinion as subjective factors such as 'taste' and 'style' mean this kind of analysis is not an exact science.

8. Summary

8.1. Project Management

8.2. Contributions and Reflections

The appropriate role of this software within the sphere of computer aided composition became more specific when I categorised existing solutions into 'MIT' and 'New Jersey' styles. This contrast of approaches provides a map of development philosophies, simplifying the identification of gaps in existing research.

The conception of the idea for this project was based on producing a computer aided composition tool which adheres to Unix philosophy (Kernighan, 1984). This idea has remained central to the project, but the discovery of the Music21 (Cuthbert, 2010) python musicology library was something I did not anticipate. This tool has proved to be valuable for its powerful analysis functions and rich built in corpus. Despite not being used for the primary software produced for this project, Music21 has played an important role in analysing existing compositions and providing a quantitative benchmark which the software developed for this project aims to match with its output.

A potential intellectual property issue arose when considering how to evaluate the software developed for this project. After deciding to use a method whereby existing compositions and the output of this project would be quantitatively analysed and compared, it became apparent that a source of existing compositions was required which permits their usage in this research. As mentioned in the evaluation section, the existing music used as benchmarks was Bach compositions and the Essen Folksong database, both of which are included in the Music21 corpus and are free to use for research. Thankfully, no compromises in quality had to be made due to copyright restrictions because these 2 corpuses made ideal benchmarks for our use case.

The work for this project did not involve any human participants or data subjects. Initially, qualitative evaluation of the software by human participants was considered but decided against on account of composition being highly subject to personal preference. The work for this project also did not use any personal data.

A broader consideration for this project is how its role will evolve as music styles change over time. Another benefit of the Unix style approach is that the modularity it provides makes modification of the software simpler than if it was a monolithic system. Within the context of music, this is useful because the demands of users will certainly change over time as conventions and tastes in music develop. This principle is also partly why I believe a system which outputs musical prompts for a human composer to arrange and implement is more useful than a system which attempts to entirely automate the music composition process, outputting complete pieces. A system such as that will stay relevant for less time because as time goes on what it produces will be further from what people desire. Ideas which are meant for a human composer to arrange and implement will stay relevant for longer because the human composer will be able to arrange them in a way which adheres to whatever conventions they choose.

Throughout the development of this project, its use as a teaching or self-study tool has become more and more apparant. The flexibility and interactivity of the system allow for core concepts to be demonstrated and experiemented upon in various ways, which would be of great use to someone just becoming acquantined with such music theory concepts.

8.3. Future Extensions

The possibilities for extending this project are almost endless, but I will briefly mention a few which seem natural additions to the existing functionality here.

First, the number of voices (lines) in the output of the melody harmoniser could be parameterised, allowing the user to request 2 part or 4 part harmony instead of 3 part, for example. Additionally, something like a length parameter could also be added to allow the user to have the chords change, for example, every other beat rather than every other beat for an alternate musical effect.

A new component program to take a set of notes and simply return the name of the chord they form would be useful, especially for anyone using the system for self-study of music theory. There are a nearly

endless amount of chords so categorising and naming them can be challenging, but they have a systematic nature so logic could be programmed into a computer to process them.

An additional parameter could be given to the mode generator such that when it is set, it expects the note input via `STDIN` rather than as a command line argument. This would allow it to read the output from the chord builder and melody generator and return all the modes which those ideas could also be treated in without incurring dissonance.

The ability to read abc notation input (Walshaw, 1997) would enable corpuses such as the Essen Folksong Collection to be used as input, allowing for functionality such as re-harmonisation of existing melodies possible without transcribing them to our input syntax. The simple and un-columnised nature of abc notation makes it a good candidate for integrating into a unix style system too, as has been explored already in "ABC with a UNIX Flavor" (Azevedo, 2013).

References

Andrews, 1999.
Andrews, William G and Sclater, Molly, *Materials of western music,* Alfred Music (1999).

Azevedo, 2013.
Azevedo, Bruno M and Almeida, Jose Joao, *ABC with a UNIX Flavor* (2013).

Bresson, 2011.
Bresson, Jean and Agon, Carlos and Assayag, G´erard, *OpenMusic: visual programming environment for music composition, analysis and research,* pp. 743-746 (2011).

Cuthbert, 2010.
Cuthbert, Michael Scott and Ariza, Christopher, *music21: A toolkit for computer-aided musicology and symbolic music data,* International Society for Music Information Retrieval (2010).

Delerue, 1999.
G´erard Assayag and Camilo Rueda and Mikael Laurson and Carlos Agon and Olivier Delerue, "Computer-Assisted Composition at IRCAM: From PatchWork to OpenMusic," *Computer Music Journal* **23**(3), pp. 59-72, MIT Press - Journals (1999).

Duthen, 1989.
Mikael Laurson and Jacques Duthen, *Patchwork: a Graphic Language in preFORM* (1989).

Gabriel, 1991.
Gabriel, Richard, "The rise of worse is better," *Lisp: Good News, Bad News, How to Win Big* **2**(5) (1991).

Gill, 2018.
Gill, C., *Harmonising Bach Chorales: The Definitive Guide for Students and Teachers,* CreateSpace Independent Publishing Platform (2018).

Good, 2001.
Good, Michael, "MusicXML for notation and analysis," *The virtual score: representation, retrieval, restoration* **12**(113-124), p. 160, MIT Press, Cambridge, MA (2001).

ISO, 1999.
ISO, *ISO/IEC 9899: 1999 Programming Languages-C,* American National Standards Institute, New York (1999).

Kernighan, 1984.
R. Pike and B. W. Kernighan, "Program Design in the UNIX Environment," *AT&T Bell Laboratories Technical Journal* **63**(8), pp. 1595-1605, Institute of Electrical and Electronics Engineers (IEEE) (1984).

Krumhansl, 2010.
Krumhansl, Carol L and Cuddy, Lola L, *A theory of tonal hierarchies in music,* pp. 51-87, Springer (2010).

Langston, 1990.
Langston, Peter S, "Unix music tools at Bellcore," *Software: Practice and Experience* **20**(S1), pp. S47-S61, Wiley Online Library (1990).

Langston, 1990.
Langston, Peter S, "Little languages for music," *Computing Systems* **3**(2), pp. 193-288 (1990).

Langston, 1991.
Peter S. Langston, "IMG/1: An Incidental Music Generator," *Computer Music Journal* **15**(1), pp. 28-39, The MIT Press (1991).

McIntyre, 1994.
R.A. McIntyre, *Bach in a box: the evolution of four part Baroque harmony using the genetic algorithm,* IEEE (1994).

Meredith, 2006.
Meredith, David, "The ps13 pitch spelling algorithm," *Journal of New Music Research* **35**(2),

pp. 121-159, Taylor Francis (2006).

Pankhurst, 2009.
Pankhurst, Tom, *ChoraleGUIDE: harmonising Bach chorales,* King Edward VI College (2009).

Povel, 2010.
Povel, Dirk-Jan and others, "Melody generator: A device for algorithmic music construction," *Journal of Software Engineering and Applications* **3**(07), p. 683, Scientific Research Publishing (2010).

Rohrmeier, 2008.
Rohrmeier, Martin and Cross, Ian, *Statistical properties of tonal harmony in Bach's chorales* **6**(4), pp. 123-1319 (2008).

Schaffrath, 1995.
Schaffrath, Helmut, *Essen Folksong Collection* (1995).

Smith, 1981.
Smith, Dave and Wood, Chet, *The 'USI', or Universal Synthesizer Interface,* Audio Engineering Society (1981).

Street, 1976.
Street, Donald, "The modes of limited transposition," *The Musical Times* **117**(1604), pp. 819-823, JSTOR (1976,).

Taylor, 1991.
Taylor, Eric and Smith, Stuart James and Richards, Mark and Smith, Chris and Cook, Frank D, *The AB Guide to Music Theory Vol 1,* London, ABRSM (1991).

Todea, 2015.
Todea, Diana, "The Use of the MuseScore Software in Musical E-Learning," *Virtual Learn,* p. 88 (2015).

Walshaw, 1997.
Walshaw, Chris, *ABC Notation* (1997).

Zimmermann, 1996.
M. Henz and S. Lauer and D. Zimmermann, *COMPOzE-intention-based music composition through constraint programming,* IEEE Comput. Soc. Press (1996).

Unix Style Computer Aided Composition

Appendix A: Function Listings

Ray Garner

20156967

psyrg4@nottingham.ac.uk

Computer Science with Year in Industry Bsc

1. Common Data

Data	Encoding
Pitch	Int
Interval	Int
Degree	Int
Scale	○[Interval]
Mode	(Scale, Degree)
Root	Pitch
Key	(Root, Mode)
Chord	[Pitch]
Line	[Pitch]
Harmony	[Line]
Alteration	Int

2. Common Functions

2.1. Input/Output

Function	Description
read_accidental(a)	Return encoded alteration a
read_note(p)	Return encoded natural pitch p
read_tone(p, a)	Return encoded pitch p with alteration a
read_mode(m)	Return encoded mode m
init_key_field(k, i)	Initialise all cells of M×N matrix k with value i where M is number of pitches and N is number of major scale modes
read_key_list(k, x)	For each key(root, mode) read from STDIN set k[root][mode] to x
print_matching_keys(k, x)	For each k[root][mode] equal to x print key(root, mode)
is_accidental(p)	Returns true if the decoding of p requires a sharp or flat else returns false
is_correct_accidental(k, a)	Returns true if the decoding of key k can be represented using accidental a
get_correct_accidental(k)	Returns an accidental which the decoding of key k can be written using
print_note(a, p)	Print decoding of pitch p using accidental a

2.2. Internal

Function	Description

clock_mod(x, m)	Returns a member of {0..m} congruent to x where x may be positive or negative
step(d, k)	Returns the pitch one step up from degree in key
calc_degree(p, k)	Returns the degree of pitch p in the context of key k
is_diatonic(p, k)	Returns true if pitch p is in key k, false otherwise
apply_steps(d, k, s)	Returns the pitch s steps from degree d in key k where s may be positive or negative
min_tone_diff(p, q)	Returns the minimum pitch difference between pitches p and q in semitones

3. Mode Generating

Function	Description
check_relative_modes(r, k)	For all k[root][mode] in matrix k which are relative to key r, increment the cell value
process_notes(n, k)	For each pitch in list n call check_relative_modes(key(note,m), k) for each mode m

4. Interval Filtering

Function	Description
degree_val(d, m)	Return the interval between the first and degree d in mode m of the major scale in semitones
correct_alteration(d, m, a)	Returns true if the interval between the first and degree d in mode m is different to the corresponding interval in the major scale

5. Melody Generation

Function	Description
count_scale_steps(k, start, end)	Return the steps it takes to reach pitch end from pitch start in key k
generate_line(len, tones, k)	Returns a melody line of length len using pitches from list tones as a skeleton and filled out with pitches from key k

6. Melody Harmonisation

Function	Description
is_primary_degree(p, k)	Return true if pitch p is degree 1, 4 or 5 in key k else returns false
add_middle_note(b, m, k)	Return the pitch x such that the chord made up of pitch b in the bass, pitch x in the middle and pitch m in the melody forms the most complete chord possible in key k
generate_middle_line(b, m, k)	Return a line between the bass line b and melody line m that would such that they would be harmonious together in the key k
pick_primary_chord(d)	Return a primary chord degree which melody degree d is a part of
faulty_note(b, m, k)	Return the number of faults incurred by having bass pitch b with melody pitch m in key k
count_faults(b, m, k)	Return the number of faults incurred by having the bass line b with melody line m in key k
alt_chord_choice(c, d)	Return another primary chord degree other than c which degree d is a part of if possible, otherwise return degree c
improve_bass_line(b, m, k)	Returns an improved version of a simple bass line b using melody line m and key k as context
generate_bass_line(m, k)	Returns a simple bass line to work with melody line m in key k

7. Conversion to MusicXML

Function	Description
write_headers()	Print MusicXML headers
write_part_def(i, n)	Print the definition for a part with name n and ID i
write_part_line(i, l, o, c)	Print the MusicXML representation of line l with ID i in octave o using clef c

7.1. Stave Key Signature Display

Function	Description
spacing(a, l)	Returns the indent as a number of spaces required for correct placement of accidental a on stave line l
print_key_sig(a, l)	Prints the key signature on a stave to the terminal where p is a list of flags defining which lines should be altered and a is the alteration which should be applied if so
note_status(a, n)	Returns a list of flags representing which lines should be altered using accidental a to represent the key signature with n instances of accidental a
is_flat_key(k)	Returns true if key k must be represented using flats rather than sharps, otherwise returns false
calc_accidentals(a, k)	Returns the number of accidentals of type a which must be used to represent key k
relative_ionian(k)	Return the root pitch of the relative Ionian for key k
note_to_cf(p)	Returns the number of sequential perfect fifth steps pitch p is from the pitch C

7.2. Fretboard Mode Display

Function	Description
write_string(k)	Return a single guitar string representation of key k
note_to_fret(p)	Return the guitar fret which pitch p lies on a guitar E string