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.

5 March 2023

I would like to thank God for everything.

Table of Contents

List of Figures

List of Tables

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

1. Introduction

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).

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.

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

Table 1

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.

3. Aims and Objectives

The aims and objectives of this project can be categorised into those which are primary and those which are architectural. Primary objectives are goals which will mean the problem is solved if they are met. Architectural objectives define how the implementation of the system will be done. For clarity, I have also listed some points which are outside of the scope of the project.

3.1. Primary objectives

Generate tonality, harmony, and melody prompts for a composer to implement

Generate tonality from melody or harmony, harmony from melody or tonality and melody from harmony or tonality

To aid a composer this system will provide tonal, harmonic and melodic prompts based on given musical input. These prompts must adhere to established rules of western music theory, corresponding accurately to the input data used to generate them. Additionally, the output must correlate with modern empirical analysis of existing compositions. Elements of style and taste are minimised since these are just prompts rather than complete compositions.

For this project we will define tonality as which mode of the major scale the music is based on, harmony as chords and melody as single note lines.

3.2. Architectural Objectives

Split into 3 programs: a mode generator (tonality), chord generator (harmony) and melody generator

Each program must be able to read output from either of the other programs

Each program must be able to read input from a user

The output from each of the programs must be human readable

Splitting the functionality of the system into 3 different programs allows for them to be combined in various ways, producing different results. It follows that the system is divided this way since the focus of this tool is these 3 areas of music. Output to input compatibility and vice versa is important so that the programs can be combined using pipes. The output must also be human readable so that it can be interpreted without additional translation software. Figure 1 shows a simple usage example:

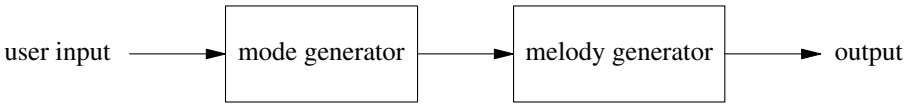


Figure 1

Here the user inputs either a melody or chord into the mode generator, and the mode outputted by that program is fed straight into the melody generator which uses it to produce a melody. The melodic output goes to `STDOUT`, allowing the user to redirect it to a file or other program using standard Unix operators.

3.3. Not objectives

MIDI output

Audio output

Entire piece composition

Rhythmic and textural manipulation

Real time interaction

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.

4. Dissertation Structure

The question this project attempts to answer is as follows: ‘‘Can Unix-style computer aided composition software be effective?’’ To answer this I will develop a computer aided composition system by combining elements of OpenMusic and the Bellcore music tools and then quantitatively compare the output from it with existing compositions (see the methodology section for more on this).

After explaining the background and motivation for the project and reviewing the primary relevant existing literature on the topic, this report explores methodologies for development and evaluation and goes on to explain preliminary design choices and prototypes. The final chapter of this report reflects on LSEPI issues considered during the work on this project.

5. 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.

5.1. 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).

5.2. 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.

5.3. 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.

6. Methodology

I have categorised the methodologies for this project into technologies and evaluation methods. 'Technologies' refers to target platforms and programming language whereas evaluation methods are what will be

applied upon existing compositions and the output of the software developed for this project to provide data which can be meaningfully compared.

6.1. Technologies

Since this project is about exploring the effectiveness of applying Unix philosophy to computer aided composition, the software will target 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 will write the software in C. This is also to make the software as portable as possible between the various Unix-like operating systems.

6.2. Evaluation

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 rich toolkit for 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.

The python style pseudocode function `analyse_chord_freq` returns the frequencies of chords for a given score split into two categories based on the tonal context of each chord. It gets the key signature of each bar of the score and uses that to determine the role of each chord in the bar. The function also accounts for tonal certainty, which means that results derived from bars with a more ambiguous key signature will contribute less to the frequency evaluation.

Figure 3 shows results from `analyse_chord_freq` applied when it is fully implemented using Music21 and applied to over 400 Bach works from the Music21 corpus. It shows that for both major and minor keys, over 40% of chords are chord 1 and over 20% are chord 5.

```
def analyse_chord_freq(score):
    for m in score.makeMeasures()
        k = score.measure(m).analyze('key')
        for c in m.chordify()
            if k.mode == "major":
                ma[degree(c, k)] += k.tonalCertainty()
            elif k.mode == "minor":
                mi[degree(c, k)] += k.tonalCertainty()
    return (ma,mi)
```

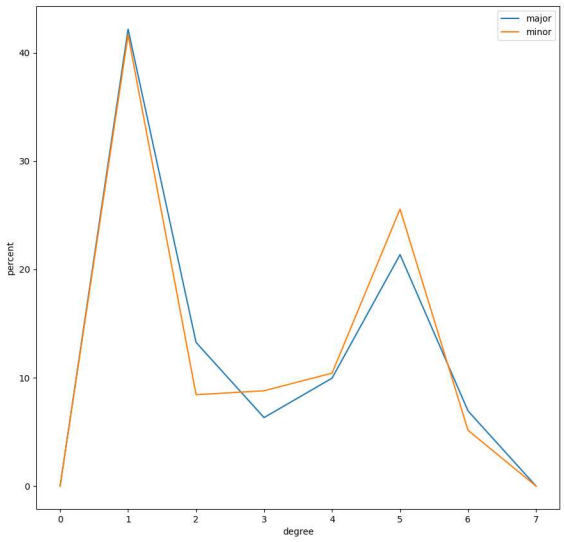


Figure 3

The function `analyse_chord_trans_freq` is a variation on `analyse_chord_freq` and returns the frequency of chord transitions. For major and minor, it uses a matrix rather than a list to store frequency data. The same principles of examining the key signature for each bar and accounting for tonal certainty apply. Results from a full implementation of `analyse_chord_trans_freq` are shown in figure 4. It shows that the most common asymmetric chord transitions are 1 to 5 and 5 to 1. The results from both of these functions correspond with traditional music theory ideas, particularity for the Baroque style.

```
def analyse_chord_trans_freq(score):
    for m in score.makeMeasures()
        k = score.measure(m).analyze('key')
        mchords = m.chordify()
        for c in len(mchords):
            ca = degree(mchords[c],k)
            cb = degree(mchords[c+1],k)
            if k.mode == "major":
                ma[ca][cb] += k.tonalCertainty()
            elif k.mode == "minor":
                mi[ca][cb] += k.tonalCertainty()
    return (ma,mi)
```

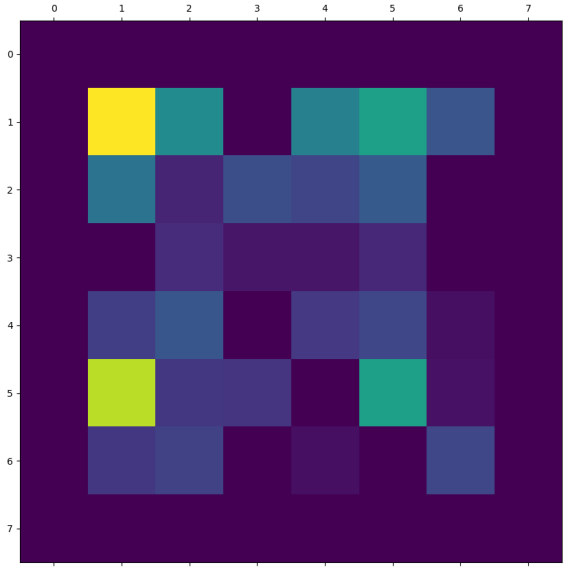


Figure 4

These two functions are relevant because I intend on running them on output from the software written for this project and comparing the results to those from works by traditional composers such as Bach. In addition to chord frequency and chord transition frequency, I intend on examining note frequency in the melody (highest line) and inversion frequency for each chord.

Similarly, figure 5 shows the frequency of notes in the melodies of over 500 folk songs from the Music21 corpus. That is, the x axis represents the degree of the scale (1 means the first note in key scale) and the y axis shows the number of occurrences in the data analysed. The frequency of note transitions is shown in Figure 6. Contrasting the frequency of chord transitions, this analysis shows that melodies tend to move in a step-wise fashion with the majority of notes being followed by another that is 1 degree up or down in the key scale.

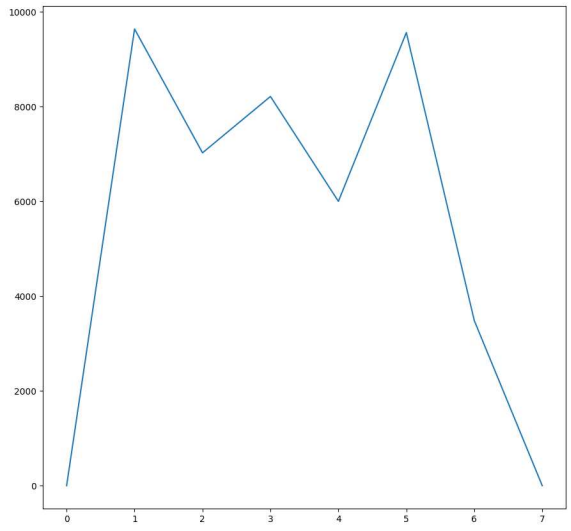


Figure 5

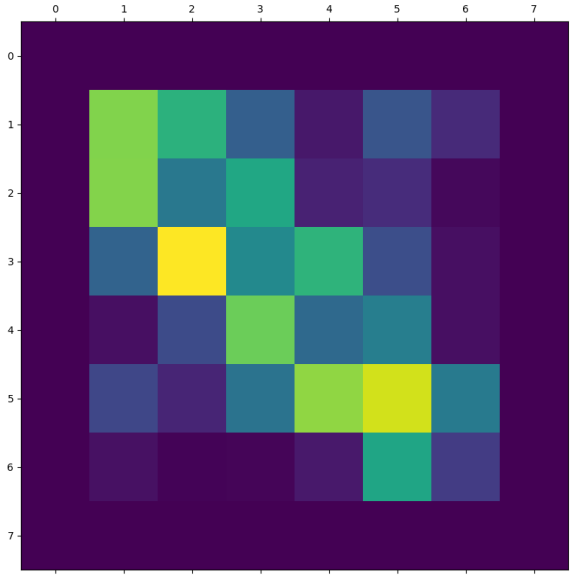


Figure 6

Going further, variations on these analysis methods can be tested. Pitch range, feature distribution and inversion (order of notes in a chord) frequency can all be examined to help provide a quantitative goal for computer aided composition software output.

7. Design

The design for this project will need to tackle issues such as representing musical data in code and how the component programs of the system should be able to be arranged to maximise versatility.

7.1. Internal Encoding of Musical Data

A musical mode can be though of as a permutation of intervals. For example, a standard western major scale (Ionian) is W-W-H-W-W-W-H (Temperley, 2012) where 'W' represents a whole-tone and 'H' represents a half-tone (semitone). This sequence of intervals is circular and therefore repeats forever. The concept of 'modes of the major scale' refers to treating the first/tonic/root of the scale as a different note in this specific sequence of intervals. For example, the intervals of the Dorian scale (second mode of the major scale) are W-H-W-W-W-H-W. (Temperley, 2012).

An intuitive implementation concept following from this principle is to represent a mode as a circular list, a half-tone as the integer 1 and a whole-tone as the integer 2. Alternatively, modular arithmetic could be used to index a standard list of intervals in a circular manor. In correspondence with this, the entire standard western note vocabulary could be represented by integers, allowing scales (modes) to be built by repeatedly adding intervals to a note defined as the tonic.

```
#define DEGREES 7
#define NOTES   12
#define EMPTY   -1

enum Interval {
    H = 1,
    W = 2
};

const int IONIAN[DEGREES] = {W, W, H, W, W, W, H};

void
fill_notes(int notes[NOTES], int start, int m)
{
    int n = (start - 1) % NOTES, d = 0;

    while (notes[n] == EMPTY) {
        notes[n] = d;
        n = (n + IONIAN[(d + m) % DEGREES]) % NOTES;
        d = (d + 1) % DEGREES;
    }
}
```

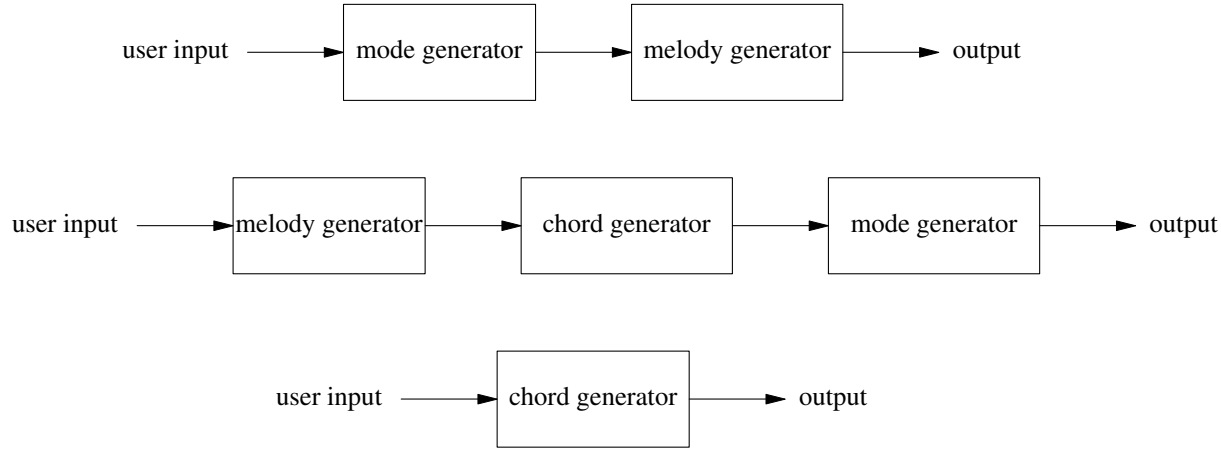
The code above is an excerpt from a program I developed to explore this method of music representation. The function `fill_notes` produces a list where the index corresponds to a note and the value of the element is the degree of the key scale that note is (the value is left as `EMPTY` if it is not part of the key). The variable `m` is the mode of the major scale being mapped (which degree is being treated as the first from the circular definition of the major scale) and `start` is the note the tonic is defined as (a combination of tonic note and mode is adequate to determine a key signature). `NOTES` is defined as 12 because there are 12 semitones in an octave and the scale being mapped is a one octave scale.

An alternative approach is applied in ‘‘Bach in a Box’’ (McIntyre, 1994). To simplify the problem, their approach only operates in the key of C major and does not consider any notes at all which aren’t in that key signature. Similar to the method discussed above, Bach in a Box uses integers to represent notes, however their system means that the pitch difference between each adjacent pair of notes isn’t uniform. For example, the different between 0 and 1 is a whole-tone but the difference between 2 and 3 is a half-tone. Of course, this is due to the W-W-H-W-W-W-W-H configuration of the major scale. By contrast, the arrangement of the previous method means that the pitch difference between any pair of adjacent integers representing notes is always a half-tone. In addition to limiting the key signature to only C major, the exclusion of non-diatonic notes (notes not in the key signature) means that some techniques used to add extra ‘colour’ to melodies and chords are omitted.

With a means of establishing tonality (key signature/mode) in place, the framework for building chords and melodies is established. The overwhelming majority of notes, whether in a melody or in a chord, in western music are diatonic (part of the key signature). The presence of a key signature reduces the search space for satisfactory notes in a chord or melody because notes which are likely to sound dissonant are excluded by the key.

7.2. Architecture

As stated, the system will be split into 3 separate programs to facilitate the generation of tonality (modes), harmony (chords) and melodies. To communicate, the programs will use Unix style text streams and pipes. Some examples of combinations of the programs are shown below:



On a unix-style command line this will look something like this:

```
$ echo $INPUT | modegenerator | melodygenerator
abccdc
$ echo $INPUT | melodygenerator | chordgenerator | modegenerator
cwwhwwwh
$ echo $INPUT | chordgenerator
cadgb
```

In addition to general usage shown above, each individual program will be able to take extra arguments to apply logical functions to the musical data. This is inspired by the functionality of OpenMusic (Bresson, 2011) which encourages the composer (user) to modify musical elements using mathematical and logical functions. This allows for more flexibility and variation of output than a system such as COMPoZE (Zimmermann, 1996) or IMG/1 (Langston, 1991) which interact with the user via natural language musical descriptors. The superiority of this approach is evidenced by the wide usage of OpenMusic by composers and lack of adoption of COMPoZE and IMG/1 by composers.

7.3. Data Transmission Protocol

To facilitate the transmission of information via pipes between the individual programs making up the system, a protocol must be established whereby data is serialised in a way which is simple for both user and computer to work with. Choices can be made in regards to the design of the protocol whereby they incur a tradeoff. This is generally between how simple it is for a computer to interpret versus how simple it is for a human to interpret. A good example of this is outputting modes; we could choose to represent Ionian as 'wwwhwwwh' or 'Ionian'. With the software representation of music we have currently established in mind, it is intuitive to parse and output modes represented using the former notation because it corresponds to our hardcoded Ionian representation:

```
#define DEGREES 7

const int MAJOR_SCALE[DEGREES] = {W, W, H, W, W, W, H};

void
print_mode(int m)
{
    int d, i = MAJOR_SCALE[m];
    char c;

    for (d = 0; d < DEGREES; d++) {
        c = i == W ? 'w' : 'h';
        putchar(c);
        m = (m + 1) % DEGREES;
        i = MAJOR_SCALE[m];
    }
}
```

Alternatively, printing of modes using the natural language representation requires hardcoding of strings:

```
#define DEGREES 7

const char *MODES[DEGREES] = { "Ionian", "Dorian", "Phrygian", "Lydian",
                               "Mixolydian", "Aeolian", "Locrian" };

void
print_mode(int m)
{
    printf("%s", MODES[m]);
}
```

The first option is more extendable since there is no hardcoding and it computes the sequence of intervals based on a given scale (major scale). This means that modification to make it use a different set of intervals as a base scale (such as harmonic minor) would be easy to implement. In contrast, the method based on natural language would require all the new names of the modes to be encoded. However, I believe that to have the modes printed in the form of natural language is important enough to warrant sacrificing extensibility in this case because it is a great deal more intuitive for a user to interpret than simply reading a sequence of intervals.

In addition to representing modes, we will need a method for representing pitches. Traditionally in music, pitches are represented by notes where a note can be considered as 'sharp', 'flat' or 'natural'. These terms are relative; a note being sharp means that it is one semitone higher than it would otherwise be in a given context and similarly a note being flat means that it is one semitone lower. A note being natural means it is neither higher or lower pitched. A simple approach to representing these using a computer would be to have two ascii characters per note. For example `an` for A natural, `a-` for A flat and `a+` for a sharp. Traditionally in music, if there is no symbol it is assumed that the note is natural but explicitly stating it, as we have with the 'n' character makes parsing this data simpler because no inferences need to be made and all notes will be of equal length (two characters). All characters here are standard ASCII characters so there will not be any requirements for extended character sets to be installed. The use of - instead of `b` to represent a flat is done because `b` could be interpreted as a B note by a user at a glance and the usage of + follows from this choice as it is universally recognised as the inverse operation of -.

Implementing wildcards to represent all/any notes would increase the flexibility of the system. For example 'X Ionian' could be passed to a program to pass C Ionian, C# Ionian, D Ionian and so on. To keep all pitch representations as two characters long, the notation `xx` could be used.

7.4. Streams and Command Line Arguments

Programs in the system will generally require two sets of data inputted to produce useful output. For example, the mode generator could take a list of notes and a list of modes and return all modes from the list which contain all notes from the list. Note that the output here is only one set of data and that this will be the case for other programs too. This means that simply piping the output from one program to another with no additional data inputted is not sufficient. Normally, Unix style command line programs take command line arguments, for example `ls -l` and `rm -r`. We can use a similar idea here for our mode generator:

```
$ echo "$MODES" | modegenerator "$NOTES"
```

If we wanted our mode generator to operate on the set of all modes then we could implement another command line argument to specify this:

```
$ modegenerator "$NOTES" -
```

To use this approach effectively, we must consider which data is appropriate to input via a stream and which data is appropriate to input via command line arguments. The main rule here is that if input data is more likely to be outputted by another program then it should be read from a stream so that the programs can be connected with pipes. However if the data is more likely to be directly inputted by a user then it should be passed as a command line argument.

8. Implementation

8.1. Common Functions

In this section I will explain some of the common functions implemented for this project. That is, functions that are put in the 'library' and called by at least 2 different programs in the system.

Ring Modulus

A function which is fundamental to the system is a simple one I have defined as `clock_mod`. All it does is extend the functionality of C's modulo operator so that works as for a ring of numbers (which is how we are thinking of scales).

```
int
clock_mod(int x, int mod)
{
    return x < 1 ? mod - (abs(x) % mod) : x % mod;
}
```

Although `mod` is a parameter, it is always passed the value `TONES` which is defined as the number of notes in a scale. Hardcoding or parameterising this value is largely inconsequential in my opinion but I parameterised it to make it more general. If the value of `x` is positive then we just apply the regular C modulo operator as normal but if it is negative then find the value in the ring congruent to that of the absolute value of `x` and then subtract it from the greatest number in the ring. That is, we go round the ring in reverse.

Stepping through a scale

For the various operations of this system is it frequently required to step through a scale. That is, apply the intervals of a mode to a given pitch context. Recall that a mode of the major scale uses the same intervals but starts on a different degree and you will understand that we can use the mode as an offset to represent this. Combine this with the offset representing the current degree of the scale we are on (a major scale has degrees 1-7) and you can use this to index the intervals of the major scale to find the correct one. Once you know whether the interval is a tone (2) or semitone (1) then you can just add it to the value of the current note to step to the next note in the scale. Of course this must operate within the bounds the rings we have defined, that is a 7 note scale and 12 tone pitch system.

```
int
step(int degree, int note, int mode)
{
    return (note + MAJOR_SCALE[(degree + mode) % DEGREES]) % TONES;
}
```

Determine if a note is diatonic

A note being diatonic in the context of a given key means that the key contains that note. For example, the key of C major contains CDEFGAB so G is diatonic but G# is not. In that last particular example the note is G#, the root is C and the mode is Ionian (major). The function simply steps through all notes in the scale and returns true if it finds one the same as the `note` argument. If it steps through the whole scale and has not found the note yet then it returns false.

```
int
is_diatonic(int note, int root, int mode)
{
    int d, cn = root;

    for (d = 0; d < DEGREES; d++) {
        if (cn == note)
            return TRUE;
        cn = step(d, cn, mode);
    }
    return FALSE;
}
```

Determine if a key is defined using the correct accidental

Even though two notes such as A# and Bb may have the same pitch to the ear, these two definitions mean different things. A# implies that A natural is not to be used in this context and Bb implies that B natural is not to be used in this context. This means that certain key signatures must be defined using one and not the other if you want to avoid using double accidentals. For example, Ab major is defined using 4 flats but G# major is defined using 6 sharps and 1 double sharp. The behaviour of the function `is_correct_accidental` would be such that if it takes G#/Bb as the root (both represented by the same number), Ionian (major) as the mode and flat as the accidental it would return true. However if it received the same arguments except with sharp as the accidental it would return false. The function essentially returns whether the key can be defined that way without using double accidentals. This is important to consider when outputting data for a user to interpret because as we have shown, even through G# major and Ab major are enharmonic (sound the same), Ab major is significantly easier for a human musician to work with.

```
int
is_accidental(int note)
{
    return !is_diatonic(note, C, IONIAN);
}

int
is_correct_accidental(int root, int mode, int accidental)
{
    int d, cn = root;

    if (clock_mod(root+mode,TONES) == C+1 ||
    clock_mod(root+mode,TONES) == F+1 || clock_mod(root+mode,TONES) == B)
        return TRUE;
    for (d = 0; d < DEGREES; d++) {
        if (is_accidental(cn) &&
        is_diatonic(clock_mod(cn+accidental*-1, TONES), root, mode) &&
        is_accidental(clock_mod(cn+accidental*-2, TONES))) {
            return FALSE;
        }
        cn = step(d, cn, mode);
    }
    return TRUE;
}
```

I found that the simplest way to achieve this functionality was to hardcode keys which can be written using either sharps or flats (C+1, F+1, B) to return true and otherwise iterate through the scale evaluating a condiction. This condition is defined such that if the current note is between two natural notes and if the note sharped or flattened to get the pitch of the current note is also in the scale and if the note adjacant to that in the same direction is between two natural notes then writing the key this way requires double accidentals. If after iterating through every degree of the scale this condition has not been met at least once then we can say that the key can be written without using double accidentals. C Ionian contains all natural notes and no accidentals so we can say that if a note is diatonic to C Ionian it is natural and if it is not then it is between 2 natural notes (requires an accidental to represent).

8.2. Mode Generator

8.3. Interval Filter

8.4. Chord Builder

8.5. Fretboard Display

9. Evaluation

9.1. Compile Time

9.2. Run Time

9.3. Distribution Size

9.4. Compatability

9.5. Comparison to Existing Music

10. Summary and Reflections

10.1. Project Management

How to quantitatively evaluate the output of the software for this project has been the problem I've focused on most so far. Solving this has involved getting to grips with Music21 and analysing existing music such as Bach chorales and folk-songs in a way which allows the results to be effectively compared against the analysis of the output of the software for this project. Of course, researching existing literature has been a large part of the work so far as well. The main topics of the existing literature researched were computer aided composition, music analysis and development methodologies. Figure 7 shows an updated gantt chart based on work currently carried out and work required to complete the project.

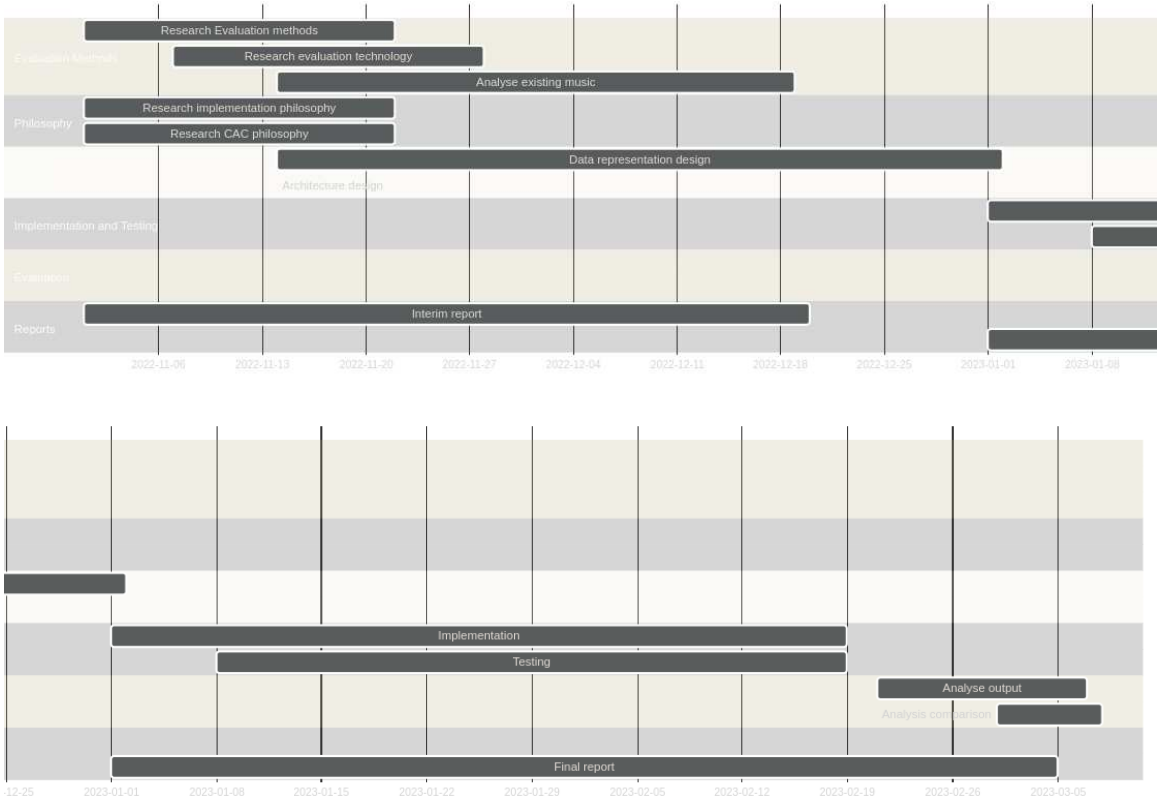


Figure 7

10.2. 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. The existing works the analysis will focus on are Bach chorales because there are established quantitative analysis methods which have been applied to them in previous research (Rohrmeier, 2008) and folk-songs because they offer a contrast in style and are unlikely to be restricted by copyright. Additionally, the Music21 (Cuthbert, 2010) built in corpus provides many Bach chorales and folk-songs for analysis.

The work for this project does 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 so highly subject to personal preference. The work for this project also does 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.

References

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).

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).

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

Temperley, 2012.
Temperley, David and Tan, Daphne, "Emotional connotations of diatonic modes," *Music Perception: An Interdisciplinary Journal* **30**(3), pp. 237-257, University of California Press (2012).

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