Automated Harmonisation of a Melody into a Four Part Barbershop Arrangement

Steve Roberts

BSc (Hons) in Computer Information Systems

2005

Automated Harmonisation of a Melody into a Four Part Barbershop Arrangement

Submitted by Steve Roberts

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

Declaration

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

Signed:

Abstract

The domain of algorithmic composition has been combined with many music styles in the past, but not barbershop. This project aims to draw together barbershop and algorithmic harmonisation and fuse them in an investigation as to whether a fundamentally creative process can be formalised into rules. It uses knowledge of existing barbershop arrangement practices and existing rules in order to create arrangements from a melody input.

A system was produced based on testing possible chords and voicings against rules that were hardcoded. This functionality was split down into identifying the chords then voicing them. The latter function proved to be successful; however the implementation of the former was naive and did not produce arrangements of a professional standard.

The conclusion drawn from this work is that although it is possible to formalise the creative process into rules, it does not yield 'good' arrangements. Rules are not enough on their own to create a good arrangement as experience from previous arranging will influence the arranger when approaching a melody.

Acknowledgements

For guidance throughout the project:
Professor J. Fitch

For moral support througout:
Meri Williams

For showing an interest in the research and evaluating the outputs:
Rick Spencer (SPEBSQSA)
John Wiggins (BABS)
Mike Lofthouse (BABS)
The Rivertones
The Alley Barbers
Robin Jackson

For getting me started with initial pointers:
Neil Watkins (BABS)

For being someone to measure progress against and exchange knowledge of Csound and Lilypond:
Paul Morris

For proof–reading the drafts:
Anne Roberts
Anna Skeoch
Caroline Reay

Dedicated in Memory of Steve Hall

Contents

Chapter 1

Problem Identification

> *'Rock 'n' Roll had become stagnant. 'Achy Breaky Heart' was seven years away. Something had to fill the void, and that something was barbershop'*
>
> - Homer Simpson in 'Homer's Barbershop Quartet'

Algorithmic composition is the process of creating a piece of music using rules laid down by a human operator, but where the computer determines the notes and rhythms.

The SPEBSQSA's (Society for the Preservation and Encouragement of Barber Shop Quartet Singing in America) Contest and Judging Handbook defines barbershop as '...a style of unaccompanied vocal music characterized by consonant four-part chords for every melody note in a predominantly homophonic texture'.

Barbershop music is one of many styles where the harmonies are dictated by a set of rules. The origins of barbershop actually stem from the circle of fifths; a method of harmonisation that has been in use since the 16th Century. As the rules are so strictly defined for barbershop harmonies, the style lends itself to algorithmic composition, because the rules can be laid down by a human operator, whilst a computer can determine the details.

Algorithmic composition is not a new concept, and existing projects have used this tool to generate music in the style of Bach, as well as using it for personal compositions. As such, there is no similar tool for arranging barbershop, therefore, as much as this is research into whether barbershop music can be arranged just through rules, it may also prove to be a tool for amateur arrangers.

Automated arrangement of barbershop has previously been attempted, but this was 20 years ago in the days of FORTRAN and assembler. More recently, there have been many attempts at algorithmic composition and harmonisation, using a variety of methods.

This project includes an investigation into possible methods of algorithmic harmonisation, with particular focus on three areas:

- Backtracking through data structures
- Analysis of existing pieces to build a knowledge base
- Rule–based harmonisation

The main task to be completed will be to create a program that will accept a melody as an input and add 3 harmony lines before outputting a barbershop arrangement. Within this program, there will have to be procedures to deal with identifying the primary harmonies of the melody, voicing the harmonies (and possibly moving the melody to a different voice), ensuring that each harmony is a melodic line in its own right, and making sure that there is a general cohesion to the piece.

There will be outputs in the form of sheet music and an audio file. These will allow the output to be used as both a preview function (the audio), and will also in the future have the potential of producing sheet music.

Chapter 2

Background and Previous Work

'Learning music by reading about it is like making love by mail'

- Luciano Pavarotti

2.1 Literature search definition

This literature review will be used to identify the key concepts within the project, bringing them together into one coherent entity. The concepts that are relevant, which will be researched are:

- The style of barbershop
- Representing music within a computer
- Algorithmic composition

These will then be brought together and links and dependencies between the subject areas will be identified.

2.2 Arranging barbershop music

2.2.1 Introduction to the style

Barbershop originated from groups of men who would sing songs together in the 19th Century by ear. In its formal definition, barbershop is 'unaccompanied vocal music characterized by consonant four-part chords for every melody note' [18]. The four parts are:

- Bass, with a range of G♭ to B
- Baritone, with a range of B to E♭
- Lead, with a range of D below middle C to F a fourth above
- Tenor, with a range of B♭ to B♭, spanning middle C

This is the simple definition of barbershop, however there are many constraints which have evolved with barbershop singing over the years.

The main constraints are:

- 'The melody is sung by the lead with the tenor harmonizing above the melody'[18]
- 'Barbershop music features Major and minor chords and barbershop (dominant type) seventh chords, resolving primarily on the Circle of Fifths'[18]
- 'Sixth, ninth and Major seventh chords are avoided'[18]
- Just intonation is used in singing - which varies from even temperament of fixed-pitch instruments

The Circle of Fifths in itself has many constraints, but these will not be discussed here, as the rules leave little room for discussion.

Similarly, 'Just Intonation' is a huge subject to investigate. The mainstream revival of just intonation was inspired by Harry Partch [3], although the emerging of the barbershop style happened while he was just a child. Just intonation is defined by Gerhard [5] as 'a perfect ratio scale' where each note in the scale is a whole number ratio from the tonic. This is a key part of barbershop, which gives it a unique sound over other a cappella music styles.

Fundamental to the style of barbershop are the tags and introductions to arrangements. The introduction sets the scene for the piece and the mood, whereas the tag is purely for the enjoyment of the performer; a chance to show off and ring chords to produce overtones. However, these are outside the scope of this project, therefore they will not be examined.

Embellishments also form a key part of barbershop music, as they are used to join phrases, provide harmonic interest or drive the piece forward. These are essential as they fill the role of the orchestra, so there are common (and almost cliched) embellishments which are often used.

2.2.2 Arranging method one

The first method to arranging barbershop assumes that the arranger has access to the chords. This is fundamentally a problem of voicing the parts to make a coherent and sensible sound. The main rules presented in this part of the Society Arranger's Manual [19] regarding the voicing of given chords are 'basses

sing roots and fifths, while tenors and baritones are more likely to sing thirds and sevenths'.

However, it is not as simple as just assigning the notes using these rules. As barbershop is a contrapuntal style [14], it is not possible to examine chords in isolation; the voice leading (voice progressing to the next note) should be taken into consideration and made as easy as possible. Sometimes the melody line will not be a note within the chord of the bar, in which case there are rules specified in the Manual [19] for dealing with this.

This method can be formulated into a simple algorithm:

- Acquire the primary harmony
- Determine the harmonic rhythm (how often the harmony changes)
- Harmonise the notes that fit in the chords of the primary harmony
- Harmonise the remaining notes

This seems to be relatively easy, until it is pointed out that there will often be several solutions, with arrangers deciding on which solution to use by previous experience, and by examining the lyrics. Each note is determined by the previous note, therefore decisions early on in the piece can totally change the arrangement.

2.2.3 Arranging method two

The second approach focuses mainly on how to find the primary harmony and it assumes a knowledge of the melody. Two methods are suggested in the Manual [20]:

- Learn the melody, then try working out the primary harmony by ear
- Examine the melody and harmony parts of a piano accompaniment and use the Circle of Fifths

The optimum primary harmony will come out of the two of these methods working together.

The simple algorithm for this process is:

- Work out a bass part
- Add in chord names and Roman numeral analysis (used in the Circle of Fifths), identifying primary and secondary melody notes
- Voice the chords (see previous arranging method)
- Iterate over the arrangement to correct weak chording

Similarly to the first method, this problem seems trivial, but it requires an in depth knowledge of barbershop and music theory.

2.2.4 Arranging method three

The third given method has a focus on where the harmony is going and what the target harmony is. This relies on a tonal centre being identified which is where the chord is stable. The melody moves us along 'the harmonic highway' [21], where it moves in, out and around the tonal centre. This approach centres itself on five fundamental rules:

1. There is root movement by perfect fifths
2. There is root movement by chromatic steps up or down
3. Chords which are a diminished fifth or augmented fourth apart can be swapped
4. Diminished sevenths affect progression, anticipate a change of harmony or hold the chord
5. The barbershop seventh chord can progress to a chord rooted a major third above

As with the other two methods, a simple algorithm can be constructed:

- Find the harmonic targets suggested by the melody
- Fill in any chords which will keep the flow of the piece
- Apply the five rules to the rest of the piece to add in the harmony
- Voice the harmonies

2.2.5 Analysis of arranging methods

All of the approaches have their merits within certain situations. Obviously the first approach can be used singularly when a primary harmony has been provided, and in all cases, this approach will be used to voice the parts.

It is not a case of one approach being the correct approach, as they are equally valid. Some arrangers prefer method two to method three, but both methods could potentially lead to the same arrangement. John Wiggins, (a leading arranger in Britain) prefers method two [25], as does Neil Watkins (a music judge of the British Association of Barbershop Singers), who is referenced in most British arranging classes [24]

It is necessary to say that none of these methods are purely mechanical. All of them rely on some knowledge of the piece of music that is being arranged; both lyricly and rhythmically. The lyrics of the piece may lend themselves to 'squashy' chords, and similarly, a ballad may require closer harmonies. A lot of the creative input that makes Barbershop what it is, is therefore down to the arranger.

2.3 Music and computers

2.3.1 Representing music in computers

There are many different ways to represent music in the modern world; not just as notated score on a piece of manuscript paper. Therefore when a computer stores a representation of some music, it should reflect the tangible world. The types of stored music can be generalised into two categories:

- Notated music
- Performed music

Notated music will be in some form of graphical mode, whether that be a traditional score, or some form of graph, whereas performed music will be stored as audio in either an electronic or acoustic mode.

In 2000, Haus and Pollastri [7] investigated the best way to store music within a database on a computer, for querying. From this investigation, they classified music interfaces into a hierarchy, with alphanumeric representations as the easiest mode of interaction and musical notation as the most complex. From this, it is possible to jump to the conclusion that the best way to represent music in a computer would be in a textual form, however this is just an opinion.

This problem had been addressed previously in 1996 by Prather [16] who took traditional musical notation, and created an internal representation based on it to hold the data within a computer:

- Each bar is a node holding the bar number, a pointer to the next bar, and pointers to the upper and lower staves.
- The upper and lower staves are similar nodes, holding the fraction of the way through the bar (the example in the paper is in 4/4 time so fractions are measured in eighths), a pointer to the next part of the bar, and a pointer to the notes
- Each note holds the length of the note (as a fraction), the pitch of the note, then a pointer to another note which, if present, will add another note to the chord. This pointer can be null.

This representation of the musical score is a realistic representation of traditional notation. A formal grammar of SML is used to design a recursive descent syntax checker which generates this representation. There are no limitations on the number of notes in a chord on the stave, and jumping to certain bars should not be difficult. The only observable problem might be working out corresponding links between the staves.

Music representation languages are available, with the main difference between them and general purpose languages being that they provide ‘many necessary data abstractions and control flow parameters that are more directly suited

to music' [11] This comparison of languages for computer music was made in 1985, so languages have become more advanced since then, but many of the fundamental structures are still present.

The languages mentioned are Pla and Formes which are written in SAIL and LISP respectively, and Arctic is written in terms of functional programming. The conclusion drawn by Loy and Abbott [11] is that flexibility is the one key aspect of any language in computer music, as this is such a diverse subject with many pockets of expertise. Their prediction in 1985 for the future was that artistic imperatives would begin to shape work more and more, as people moved away from the technological imperatives. This doesn't seem to have been the case though, as computer musicians have drawn the two areas together.

The conclusion that can be drawn from this research as a whole is that there is no right or wrong way of representing music in a computer, as long as it is sufficient for its purpose. Of the many languages available, there may not be one to fulfil all needs for a particular program, so it may be best to write data structures specifically for a particular project.

2.3.2 Input and output of computer music

The physical representation within a computer has been discussed, but there has been no mention of how the data is inputted. This subject was investigated by Haus and Pollastri [7] when they were looking at ways of storing music for querying. Their research concentrated on the translating of audio inputs as an alternative to textual or graphical inputs.

Of particular interest is their analysis of inputting a singing voice, as this takes different techniques to inputting a musical instrument. The input of the audio is not a problem, but the analysis and subsequent storing is problematic in that the methods used at that time were only 80% correct. This is due to the singing voice giving an approximation of rhythm and pitch.

In the last four years however, more time has been dedicated to this investigation and in 2003, Lin et al [10] developed a singing rectifier in two parts:

- A recogniser, which recognises input which is 'off key'
- A synthesiser, which 'shifts' the pitch using PSOLA (Pitch Synchronous Overlap and Add)

This now opens up possibilities for inputting singing voices into a computer and recognising the notes. Such technology is not available to all yet, as it is still in the development stage and it has not been released commercially yet.

As understanding of DSP (digital signal processing) has improved over the years, computers can now output many different sound qualities. The area of interest here is output of audio from an internal representation in the style of a human voice. Howard and Tyrell [8] have investigated DSP to work on a real-time four part singing synthesiser. This breaks down the human voice into the

power source (the lungs), the sound source (the vocal folds) and sound modifiers (velum, nose and mouth) to analyse how sounds are formed. These can then be recreated within a computer.

Another approach to creating a human singing voice output would be to use Csound (DSP software developed from Music V) and to create an instrument of a human voice. This would probably be restricted to a single vowel however, with no consonants to break up the sound. The added bonus of Csound is that the output frequency can be specified and therefore just intonation (see section on barbershop) can be used for output to create the rich harmonics.

There are several possibilities for outputting a piece of music with the sound of human voices. The method chosen depends on the end user and the complexity of the program within the context. It may be necessary to compromise certain functions in favour of efficiency.

There are many pieces of software on the market for amateur composers who wish to output their compositions in a professional format. As most of them are driven by a GUI, these would be unsuitable for outputting a musical score from a bespoke abstract data structure. Two pieces of software built on TeX for this purpose are LilyPond and MusiXTeX. Both of these require a textual input, which could be generated by traversing an internal representation of the music. LilyPond however is the better tool for output, as more complex algorithms are used for layout and presentation.

2.4 Algorithmic composition

2.4.1 Rule-based composition

Until 1972, all algorithmic composition was done using random numbers, then at Stanford, Moorer investigated a new method [13]. The discussion within the his paper looks at creating a simulation of human composition using heuristics, yet it is a very simple model. It acknowledges that composers such as Mozart composed their music using formulae, which could be described in terms of a BNF grammar.

Although Moorer's paper [13] covers the majority of aspects to be taken into consideration when composing music with computers, there is very little on the technical aspects; the focus being on the subjective parts. This discussion therefore provides many openings for further work. One interesting point to note is that in the initial (very simple) composition, the operator has control over a certain number of parameters, as a computer can never be fully creative.

David Cope was a pioneer in his work with computer composition of Mozart. His program, 'Experiments in Musical Intelligence' [2] worked through pattern matching against existing Mozart pieces and deciding what Mozart would do if faced with a certain sequence of notes. He also uses augmented transition networks to take existing Mozart phrases and augment them with passing notes in order to create new phrases. Through this, Cope was able to develop a

program that generated pieces of Mozart music which are indistinguishable from original pieces. This is similar to the rule based systems, but using Markov chains. It illustrates that there is more than one way to implement rule based systems, and in this case, it is a very effective implementation.

The method of formalising music composition into grammars was the way that computer music developed and in 1974, Rader investigated creating 'rounds' on a computer [17]. The problem was redefined from computers composing music to formalising human symbolic systems. In his research, Rader built on previous work and set clear constraints by formalising all possible paths in a stochastic grammar. In a similar methodology to Moorer, the harmonic framework was built first, with a melody then generated within it. It is interesting to note the similarity between the finite state graph produced, and the Circle of Fifths [22] - suggesting that a grammar is a good way to formalise the Circle of Fifths.

Even with the rise of other methodologies, Prather [16] chose to use a rule-based system for extracting the primary harmony from a representation of a musical score. Rather than learning from many pieces of music, the rules have been hard coded into the system through mathematical analysis. The melody is then traversed and rules are checked against each note of the melody to work out the underlying harmony. This approach does not use backtracking though, which it may benefit from (although increasing the complexity is a disadvantage).

2.4.2 Genetic Algorithms in composition

As an extension to rule-based systems, the use of genetic algorithms allows the program to use rules, but also to learn new rules and apply them. This was utilised in 1991 by Gibson and Byrne [6] in the creation of Neurogen. This program produces four-part compositions using knowledge extracted from example fragments. The two main tools are:

- Neural Networks, which capture the conceptual ideas and extract common features
- Genetic Algorithms, which have states representing musical fragments, and an evaluation function giving a fitness value to each member.

The constraints are clearly specified; the only scale used is C major, only the tonic, subdominant and dominant chords are used, and it only creates four part harmony.

This seems to be a very successful application of neural networks and genetic algorithms, as this means that the program will continue to learn through many compositions. The initial rules are worked out from examples, ensuring that they are true.

2.4.3 A comparison between these two approaches

An initial look at these two methods makes them seem as if they are very different, yet there are many similarities; mainly based around the rules and the knowledge that they hold. It is the way that this knowledge is embedded within the program that dictates the difference.

The evaluation by Rader of his compositions was done by listening to them [17], yet this doesn't give an objective way of comparing methods. The human ear and what constitutes music is very subjective, therefore it is not possible to compare how successful a method is in complying to the rules.

In 1999, Phon-Amnuaisuk and Wiggins compared the two aforementioned approaches [15], using the fitness function within the genetic algorithms to evaluate both methods. The same pieces of music were tested, and to make it fair, the same knowledge was inputted into both of the systems. The results showed that the rule based system delivered a better solution. However it is impossible to draw a conclusion from this one test.

From examining the evidence, it is clear that both methods have their merits, and an optimal solution depends on the knowledge within the system. If it is possible to formalise the rules mathematically and input them into the computer, then a rule based method would be better. However if there is little initial domain knowledge, then the use of genetic algorithms would be advisable. The method to be used therefore depends on the context of the use.

A further tool was introduced in 1998 in Spain, with the use of case-based reasoning [1]. The objective is for each note to be harmonised by finding the same situation within the case base, and applying it to the current situation. This, in conjunction with the rule based approach for harmonisation proves to be a very powerful tool. One further extension (which is not mentioned in the paper) would be to combine this with genetic algorithms, and therefore enlarge the case base over time.

2.5 How these subjects are related

It is interesting to see how these three topics fit together. In the Society's Arranging Manual, there is a section on the future of barbershop music [23], which predicts that barbershop would go down the route where the arranger has more say over the interpretation of the music. This was written over 20 years ago, yet there are no moves towards this. However there have been investigations into algorithmic arrangement.

About 20 years ago, Neil Watkins (one of the leading British arrangers, and now a music judge for BABS) worked on a program to arrange barbershop. This was in the days of FORTRAN and assembler, so it was an unsuccessful project. The primary harmony was hard coded into the system, as Neil believes it is not possible to code this, then the Arranger's manual approach 1 was used [19]. Prather [16] may hold the solution to this problem of identifying the primary

harmonies, as he did an investigation into analysing chords given a piano score. He traverses the score, extracting all of the notes, then analyses the vertical chords to give the chord name, and the chord type also.

It is interesting to see that although embellishments are a key part of barbershop, Neil suggests that these should not be added in by a computer, as computers cannot be creative in the same way as humans. If this is the thinking, then surely the whole project might as well be cut, as arranging a piece of music is down to creativity.

The sort of approach to be taken in this project with the arranging of barbershop music has been done in the past, with Bach chorales. The fundamental investigation (and still the leading work) into harmonising four part chorales was done in 1988 by Ebcioglu [4]. A new language (Backtracking Specification Language) was specifically constructed to aid this task. The general process of the composition involves generating all possible solutions for the next element, then testing these for the best match against the heuristics.

This problem is very complex, and in more recent years, there have been more attempts (not so successful) to repeat the work but with different methods. One was a computer science final year project and another was by McIntyre in 1994 [12]. Both approaches had very strict constraints about which key the music should be in, and McIntyre's approach could actually be used for barbershop. He takes an input melody then generates diatonic harmonies using vocal constraints similar to barbershop. The tools used for this are genetic algorithms, which in this context have one major advantage over rule based systems; there are many more harmonisations you can get for the one melody than with rules.

The last comment in McIntyre's paper is one to strike fear: 'Finally, users familiar with other forms of music might create fitness functions that could create harmonies for Jazz or, more frighteningly, Barber Shop Quartet Harmonies'[12]

Chapter 3

Solutions Considered

'Computers are like Old Testament gods; lots of rules and no mercy'

- Joseph Campbell

3.1 Introduction

Different solutions to the problem have been considered, and in all of them it is necessary to explicitly state some things which are outside the scope. From background reading, it has been noted that some of the similar systems to the one being designed have user interaction during the process of composition/harmonisation, with the ability to change certain parameters [13] This will not be the case with this project, and any outputs to the user during the program's execution will be purely for information purposes on the state of the system.

Barbershop music is sung and therefore normally has words associated with a melody and arrangement. However, the addition of lyrics to a piece of music is an art in itself, therefore this is also outside the scope of this project. Also, as this is not for commercial release, there will not be a graphical user interface as it is assumed that users of this system will be relatively competent with computers.

It is important to realise that the emphasis of this study is not on the actual program, but on the processes behind the program that make it possible.

3.2 The tasks to be completed

The scope of the project has been defined as five distinct steps. These can be implemented in several ways, so it is the aim of this section to provide possible solutions, which can then be evaluated.

3.2.1 Inputting a melody into the computer

Within this system, there will be one input, which is a melody. From the reading of previous research, it has been acknowledged that the best method to use is a textual representation. This would mean an internal representation would hold the melody in a textual format.

There are a two options when storing a representation:

- *Existing data structures* - these may be useful for ideas of how to store the data, but to use them for the whole purpose may not be ideal. As the implementation will be using the language C, there is the option of using the Cscore library, but this does not give much flexibility or control over the structure and traversing; its main function is to help transfer the score into Csound for playing.
- *Bespoke data structures* - this is probably the preferable way of representing data, as this allows a freedom for expansion. In the research, a computer representation of a barbershop score was not found, therefore it may prove more useful to have a data structure designed for the purpose of holding a barbershop score. This way, there will be no superfluous processes and all processes will be tailored for the need.

It would have been preferable to have an audio input as often the sheet music is not available for transcribing into a computer, but the time constraints mean that this is not possible. If an audio input were to be considered, then there would have to be a recogniser and translator to an textual internal data representation. However, there could be a whole new way of harmonising the parts, as the analysis of the melody could be done in the audio domain rather than textual. This would lead to a totally different method of working and thinking though, so not much attention will be given to this concept.

3.2.2 Adding primary harmonies to the melody

The adding of primary harmonies to the melody will have to be implemented through certain rules. This means it is necessary to have some way of holding rules, as well as part of an internal representation being able to hold the details of harmonies corresponding to the melody. The main rule mechanism for this is the circle of fifths, although the melody note also needs to be taken into consideration.

As seen in the background reading, there are different ways to hold rules within the program:

- *BNF grammar* - this can be used to check valid statements and orders. All combinations of note patterns would be coded in, and also the details of the circle of fifths (as this is what will be used for the addition of primary harmonies). This basically gives us a way of defining the language that is the circle of fifths.

- *Pattern matching and Markov chains* - this would be used with the circle of fifths as a stochastic grammar; each point in the circle is a state and there are different probabilities of going to the different states. This would be a good implementation of the circle of fifths, as noted in the background reading and it naturally fits.
- *Hard-coded into the algorithm* - this would be used by having many nested 'if' statements to give different harmonies depending on the previous note. This isn't the obvious choice as it doesn't seem the natural method.
- *Genetic algorithms* - in this case, genetic algorithms would probably not be a good choice, as there is little knowledge to learn within the circle of fifths.

It may be that the chosen solution utilises more than one of these techniques, as there is no restriction to using just one method.

3.2.3 Voicing the harmonies for four voices

Similarly to the previous section, there needs to be something within the program to hold the rules for voicing the harmonies based on the previous note and voice range constraints. There will need to be information therefore about the possible voicings, the voice ranges and how to choose between different voicings.

The choices are restricted as far as storing the voicings and voice ranges is concerned. These need to be lookup tables that are hard coded. However, there are still options for choosing the different voicings:

- *Pattern matching and Markov chains* - this would be a useful way of voicing the parts, as a lot of barbershop music already exists and a database could be created with all of the patterns; given one chord with certain voicings and a primary harmony of x, in previous cases, chord y was found to be the best voicing.
- *Genetic algorithms* - this could provide a good method of choosing the voicings of the chords as a fitness function could be used. Used with neural networks, there could be learning to speed up the process by having a record of each voicing. This option has been chosen in similar situations in the past, with the optimum voicing being produced each time after many iterations.
- *Simple fitness function tests* - similar to the genetic algorithm approach, each voicing would be given a fitness function rating, but there would be no learning involved with this. The advantage of this approach over the genetic algorithm is quick results.

There seems to be no single ideal way of doing this, so in the chosen solution a combination may be used.

3.2.4 Adding typical barbershop embellishments

The addition of typical barbershop embellishments appears to be a similar process to the voicing of the harmonies. There will be an iteration over the structure, and when a certain case is matched with a possible embellishment, it will be inserted. There will therefore need to be rules as to when embellishments should be inserted and what the embellishments are.

The two possible options available are:

- *Markov chains*- most of the states in the chain would have the single action of moving on to the next state (and next chord), but occasionally where a state matches the chance to add an embellishment, there would be other actions; the choice of which embellishment to add
- *Lookup tables* - for each chord that warrants an embellishment, there will be a table with the possible embellishment next to it. The addition of embellishments suddenly becomes a mechanical process. This is a simplistic approach though, and may not yield the best results

3.2.5 Outputting the arrangement

There will need to be a way of translating the internal data representation of the score into the two output types (audio and traditional music notation), so these can be opened in the relevant programs.

There are very few options for the translation into the music notation. The method used basically involves traversing the structure and creating a file that can be opened for viewing using a program which will translate that file into a visual output.

The output of the audio does present a couple of alternative options:

- *Equally tempered output* - this would be the simplest option as MIDI could be created from the outputted text file. However, although this would give an audio preview, it is probably too simple for use in this project
- *Just intonation output* - this would require creating a second output of the arrangement from the program, with frequencies and amplitudes calculated. This would certainly give us a true barbershop sound and 'ringing' chords which would be a more accurate preview function

3.3 Initial conclusions

From the initial thinking about the design of the system, it can be said that the problem is built around knowledge and how it is stored in the program; whether it is held in data structures or in the actual algorithm. There is also an issue of whether the knowledge is specified (hard coded), or learned from

performing tests on the data. In general, the knowledge about barbershop has to be formalised into mechanical functions that can be performed by a computer.

When this plan was put to Neil Watkins, it was met with a lot of scepticism, as two of the functions appear to be non-mechanical. The addition of primary harmonies and addition of embellishments will both require original solutions, and the project definition may therefore change over the course of the project if these are clearly not viable options.

Chapter 4

Chosen Solution

'If music be the food of love, play on'

- William Shakespeare

4.1 Introduction

There are three main things to be thought about in designing this program:

1. Data Representation within the system
2. Data Structures
3. Processes taking place

Data structures and representations overlap quite considerably as they are dependent on each other. However they will be addressed as two separate issues before they are brought together, so that all issues can be identified with clarity. They will take into consideration the languages being used for the implementation (C mainly, but Lilypond and Csound for output).

To some extent, the main processes within the program will dictate the data structures and representations needed, but equally the processes will be influenced by the data structures. It has been decided that this will be addressed in a separate section.

An interesting property brought out of these dependencies is that the order of implementation is very important. For example, for testing purposes, it is necessary to have an output working from very early on. Therefore the data structures have to have been implemented as well as the representation within them.

4.2 Data Representation

The representation of data within the program will be driven by the structure of Lilypond. This has been decided on as the output language so it also makes sense to have this as the input language. From these decisions, a natural progression is to design the internal representation of the score with reference to Lilypond.

One main problem with using the direct Lilypond structure is that the pitch of the note is written as a letter with the note duration as a number. For comparison and computation within the program, it will be simpler to deal with numbers. This gives us a certain freedom as there is no longer the restriction to one key signature, as information about the 14 different types of chord with the same tonic for all 12 semitone tonics can be held, instead of hard coding one lot with letters.

An original thought would be that the letters will be converted into numbers once the program is run, with the tonic of the melody being number one. This allows us to work with the circle of fifths for the primary harmony progression. However, once the chords are being voiced, the tonic of every chord will have to be number one, as this is how the chords are worked out. Again this presents a problem though, as a comparison has to be made between the current chord and the previous chord, so they both need to be numbered from the same starting point.

Another issue which arises from this solution is that of the octave. It is noted that all of the voice parts in barbershop singing (excluding the tenor) have a range greater than one octave. It is necessary therefore to have a way of indicating which part of the range a note occurs in. One solution may be to have the numbers ranging from 1 to 12 (an octave in semitones), with another to have a variable within the encoded note, specifying the octave.

The general solution for all of these problems seems to be that the score representation will hold letters (as with Lilypond), then there will be various lookup tables for converting these letters to numbers for any operations that take place on the score:

- For the addition of primary harmonies, the letters will be changed into numbers with the key signature (tonic) as 1, with the next semitone being 2, incrementing up the octave; so numbers 1 to 12. This means that the encoding of the circle of fifths will need to be in numerical semitones also, with tonic as 1.
- For working out the chords possible for a given melody note and primary harmony, the primary harmony (which will have a letter included in the representation) will use that letter as number 1, as this is the tonic, and everything shall be computed from there.
- For the comparison of the chord with the previous chord, it does not matter which note is number 1 as long as both chords have the same number system, as this is a relative comparison; how far apart the notes are.

In conclusion, although the initial solution appeared to be the use of numbers throughout the implementation, it is now seen that the use of letters and converters may be more appropriate.

4.3 Data structures

4.3.1 The score

The score will be the main data structure within the system, as this will be what holds the input and all of the stages between this and the output. The final output will be a translation of the score into a Lilypond output and a Csound score.

The score will need to hold:

- Details of the harmonic rhythm (how often the primary harmony changes)
- Time signature
- Key signature
- Chords, which each have 4 notes and a name (e.g. F7). Each note has a duration and pitch too

The time signature is only really important for the output, but the key signature information is needed for the primary harmony to be added. It specifies the tonic of the piece, and subsequently the tonic for the circle of fifths.

As well as the score holding information about all of these things, there will need to be ways of connecting these things together and traversing the score. A chord is an entity, and within this project there is the restricted scope that all four parts will have the same rhythm. This gives us the option of creating a chord object which has four note objects in it, a name, and a pointer to the next chord and previous chord.

A possible extension to this project is the addition of embellishments, where one part has a slightly different rhythm and notes. This is not possible with the above option, therefore with future work in mind, it is necessary to think of a new structure. The proposed changed structure would still have the chord object with names and notes. However, the notes would be the ones with the pointers to the next and previous notes, as well as their pitch and duration. This gives the flexibility for a sustained chord with one part having an extra few notes; the pointers will just change.

4.3.2 Circle of Fifths

The circle of fifths will be encoded into the system as a set of 12 nodes with a number attached (the degree of the scale), then pointers to the possible pro-

gressions. It should be noted that degree of the scale is measured in semitones, which is unlike the traditional circle of fifths as seen in Figure 4.1:

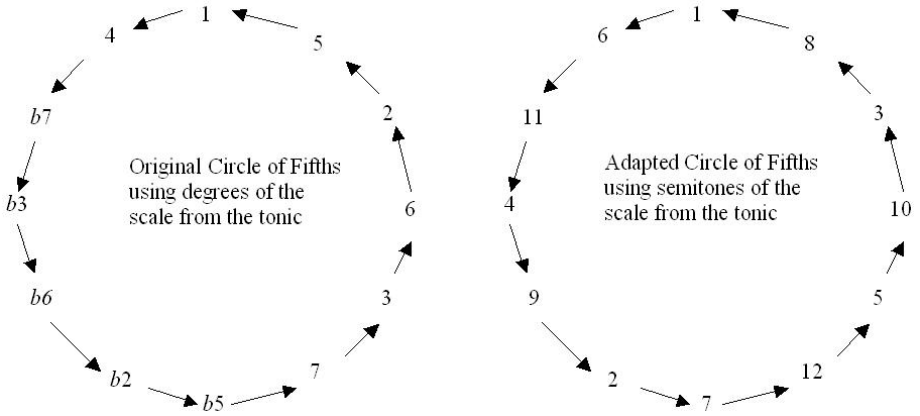

Figure 4.1: Original and altered circle of fifths

An option may be to implement a 'progression' object, which would be a pointer to a possible progression and then the percentage of the time to use this. This could be used to ensure that the main progression used was in an anticlockwise direction.

There will be a pointer variable as well, which will point to the current position in the circle of fifths, so that actual progression around it shall be simple.

4.3.3 Voicing comparison

When the primary harmony is generated and a chord to fit this has been identified, there is the task of voicing the chord. There will be 6 possibilities in voicing the chord (assuming that the lead part will be fixed on a certain note and it will be necessary to decide on which voicing to use).

There will therefore be a table which will be able to hold 6 possible voicings of the chord, with each chord being given a rating. The rating will be given depending on whether each voice part is within their range, and how far apart the note is from the previous one for each voice part. The data held within this table will be in letter form, but all of the comparisons will take place numerically by changing the letters to numbers.

4.3.4 Lookup table of every 4-part chord

There are 14 different chords possible for each tonic and these need to be stored within the system in a lookup table. Rather than creating a table with the 14 combinations for each of the 12 semitones of an octave, if the chords are coded as numbers, then these can be translated back to letters afterwards.

It will also be necessary to have some form of ordering to this table, as within the rules of barbershop, there are some which constrain the number of times a certain type of chord is used. As well as this ordering, it may be interesting to have an additional variable within the table which keeps track of the number of times a certain type of chord has been used within the arrangement.

The combinations which shall be coded into the system shall be measured in semitones, and they are:

1. Major triad (1, 5, 8,)
2. Minor triad (1, 4, 8,)
3. Augmented triad (1, 5, 9,)
4. Diminished triad (1, 4, 7,)
5. Thirteenth (1, 5, 10,)
6. Dominant 7th (1, 5, 8, 11)
7. Diminished 7th (1, 4, 7, 10)
8. 9th (1, 5, 8, 11, 3)
9. Major 6th (1, 5, 8, 10)
10. Minor 6th (1, 4, 8, 10)
11. Major 7th (1, 5, 8, 12)
12. Minor 7th (1, 4, 8, 11)
13. Half-diminished 7th (1, 4, 7, 11)
14. Augmented 7th (1, 5, 9, 11)

An interesting observation is that in the first 5 cases, the chord is only made up of three notes. In these situations, the most common option in barbershop arrangement would be to double the root (so two occurrences of 1). In a similar situation is the 9th chord, which has 5 notes. Again, the generally accepted and most common method to remedy this situation would be to remove the root.

Lookup table for translating numbers to letters

In several situations, it will be necessary to translate letters to numbers and vice versa. The structure for this lookup table will be circular, with 12 nodes. Each node will have the letter hard coded into it, a pointer to the next node and space for a number. This way, the same structure can be used for any translation, by '1' being put in at any point, then numbers being incremented as it is moved round.

4.4 Processes + Techniques

4.4.1 Translating input into the representation

The input will be a Lilypond file and a variable with the harmonic rhythm in it. The first thing to be extracted from the input file will be the key signature and time signature, which will be held in variables. The time signature is not needed during the processes that take place on the score. The key signature will then be used to set the tonic in the letter/number lookup table and populate it.

The input file will be traversed and new notes will be recognised. As this is just a prototype/research program, it is assumed that the input will be a valid Lilypond file, so there will only be simple error handling. Every time a note is found, a new chord will be created with links from the previous chord notes to the new chord notes. The lead note will then be filled in with the note length (an integer) and the pitch of the note (a `char*` as this could involve a sharp *'is'* or a flat *'es'* as well as the letter of the note)

4.4.2 Adding primary harmonies

The first chord name will be set to be the tonic, which will be taken from the tonic variable which was created during the translation into the representation. There will then be a traversal through the score from chord to chord and there will be a check to see whether the harmonic rhythm specifies a change in chord (from the harmonic rhythm variable passed as input). If there has not been a change, then the previous chord name will be copied into the current chord name. However, if there has been a change, then a lookup process will take place.

The first process will be to check the position in the circle of fifths and to work out the possible progressions around it. Once these have been identified, they can be ranked using a couple of criteria:

- Most common progression - this is important for 'singability' of a piece, as barbershop singers can normally sense where a piece is going next
- Whether the melody note is part of the chord - this is quite important for simple pieces, as normally the melody is on the tonic or fifth, however with more complex pieces, this is less of an issue, as interesting chords can be produced when the melody does not have a primary harmony note. All that will be tested will be whether the melody note is part of the major or minor triad (the simplest chords).

Once the possible candidates have been ranked, the highest ranking one shall be chosen and written into the chord name of that chord. At this stage the chord name shall just be a letter, without details such as whether it is a diminished chord, or a ninth.

A useful bit of functionality that may be added as an extra is knowledge of the end of a phrase of the music. This could be useful as often a phrase ends on the tonic and for tension to be kept, it is important the the chord does not come too soon.

This is certainly the hardest bit of functionality to implement, as there is no tried and tested method for successfully finding the primary harmony. All arrangers normally sit by a piano and try it by ear. This method will not work for pieces of music that do not follow the circle of fifths, so there may be some interesting results (or even failures) as non-conforming pieces cannot be forced to follow the rules.

4.4.3 Voicing the harmonies

To voice the harmonies, the score will be traversed again. For each chord, the letter/number lookup table will be populated with number 1 at the tonic of the chord. From here, the lookup table for all four part chords will be compared against the lead note to decide on the variation of the chord that will be used. The reason for comparing with the lead voicing is in case the lead is on a note which is not part of the primary harmony; in which case a more interesting chord is needed. The chosen chord full name will replace the letter in the chord name in the score.

Once this name is fixed, the six possible voicings of this will be generated and the following rules will be tested:

- Each part shall be in its own range
- The bass should be on the root or 5th of the chord
- For each voice part, the interval between the previous note and the proposed note should be 'easy to sing', so the notes should be as close together as possible
- There shall be no parallel octaves or parallel 5ths (where two parts an octave or a 5th apart move in parallel)
- The tenor notes should be no more than a 4th apart.

Each of these tests will have a value attached to it and after every test has been performed, the voicing will be given a total score. Once all the voicings have been tested, the one with the highest score shall be selected for use in the arrangement, and each voice part shall have the note name inserted.

If there are two voicings which give the same score at the end of the tests, then one of them will be picked by using the random function. This has an interesting effect as it could change the whole arrangement because each note is dependent upon the previous one (as seen in the test criteria above).

4.4.4 Outputting to Lilypond

To output to a Lilypond file, there shall be a traversal of the score to output the relevant pitches and note durations for every note and every part. This should be a trivial process, as the pitches and durations will be in the Lilypond format already.

Bar lines are not necessary in the translation, as Lilypond adds them in during the compilation process. The only possible difficulty may be in ensuring that the note stems are all going in the correct directions (pointing upwards for tenor and baritone and pointing downwards for lead and bass). This file can then be opened in Lilypond to produce the final arrangement in traditional music notation.

4.4.5 Outputting to Csound

The outputting to a Csound file will be much more complex and involved as all the note values and pitches will need to be translated. The only easier aspect to this output will be that the order of the traversal does not make a difference.

The lead note for each chord will be the first note to be translated, as this is the only part for which an equal temperament scale is used. Once this has been turned into an absolute frequency, each other note in the chord will be outputted as a frequency by calculating the relative frequency to the lead note (to make it just intonation). The lead part will be given a slightly louder amplitude, as this is where the melody lies.

When thought was going into the design of this output, it was very easy to get carried away and forget that this is a preview function rather than a performance function. Possible extensions to the simple output are:

- An instrument will be created for each of the four voice parts from sampling them, as the 4 voices would each have different types of resonance and timbre
- Whenever a voice part has a third or seventh of a chord, it will be made quieter, whereas the fifths and roots would be made louder. This is the way that quartets balance the chords.

This is verging on the pedantic, and it is important to remember that this is just a preview function. This would not be the first addition to make to the project if there was extra time.

Chapter 5

Implementation

'The last good thing written in C++ was the Pachelbel Canon'

- Jerry Olson

5.1 The flow through the program

The program is broken into five basic sections of functionality as can be seen in figure 5.1. The boxes are the sections of the functionality and the inputs and outputs are shown next to the arrows.

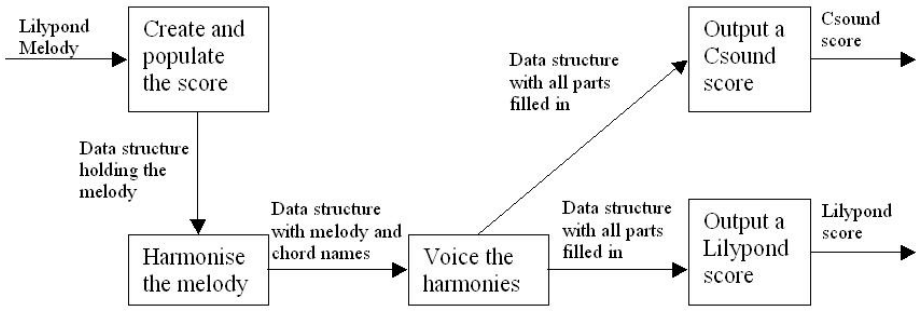

Figure 5.1: Flow of data through the program

Within these five functions, there are nine separate traversals through the score once it has been created to either add more data, or to print out the score. The next section explains the processes within each of the functions.

5.2 Details of the function algorithms

5.2.1 Creating and populating the score

The score is created as it is populated. The structure of the score allows chords to be added, so as a new melody note is recognised, a new chord is created which is linked to the previous chords.

The input is tokenised by a simple lexer, then unused tags are ignored while important melody information is extracted. This is then parsed straight into the score rather than a parse tree. The grammar can be found in section 5.3.2.

5.2.2 Harmonising the melody

The harmonising of the melody involves 3 separate iterations through the data structure. The first information to be extracted from the score is the key signature, and this is used to create a lookup table of numbers against characters, where the key signature tonic is at position 1. The first note is given the chord name from the key signature.

In the first traversal, the harmonic rhythm is checked, and at every stage that a new chord is required, the following takes place:

- The octaves are stripped off the chord to allow for calculations
- The possible chord names are calculated from the melody note and the previous position on the circle of fifths
- The chord name with the highest ranking is placed into the score

The second traversal searches for positions without a chord name entered, then copies across the chord from the previous position.

The third traversal of the score fills in the type of the chord. This is done by searching through the 'AllChords' structure to find the first chord which contains the lead note. Because the chords are ordered within the structure by commonality of use in barbershop, the chord type which is chosen will normally be the best choice.

5.2.3 Voicing the harmonies

The harmonies are voiced in one traversal through the score. For each position, the notes making up the chord are looked up in the 'AllChords' structure and all possible combinations of voicings are generated. As the octaves had been stripped off earlier, this will always be six combinations.

Each of these six chords is given a rating based on the rules of voicing that can be found in section 4.4.3. These rules cover comparisons between the proposed

voicing and the previous chord as well as rules for positions of notes within chords. The chord with the best voicing is then copied across into the score.

5.2.4 Creating a Lilypond output

The creation of the Lilypond output is the simplest function in the program. It is generated using `printf()` statements; hardcoding the same tags, then four simple traversals through the score. Each traversal extracts the note pitch and duration of one of the four voice parts and prints it out. The final tags are then included, completing the output.

5.2.5 Creating a Csound output

The creation of a Csound score file is similar to the Lilypong output method in that `printf()` statements are used, but there are necessary calculations. Firstly, the lead frequency is calculated from the name of the note. This is worked out from an equal temperament scale lookup table of all of the frequencies that are used in the barbershop singing range.

The output can be produced with just one traversal through the score structure, as Csound doesn't work linearly. Therefore within this traversal, all of the frequencies of the other parts are calculated straight after each other. The difference between the lead note and the calculated note is calculated in semitones, then the frequency ratio is looked up. This gives the ratio to work out the new note frequency. Information about octaves is then added back in to calculate the final frequency which is then outputted.

5.3 Decisions made and deviations from chosen solution

5.3.1 Alterations to the data structures

The original design for the main 'score' structure within the program was altered slightly in the implementation from the proposed structure. There was the temptation just to create a 2D array of 'note' structures as these provide natural vertical and horizontal links between the notes, but it was decided that a more bespoke solution was necessary.

The only main change was in the addition of a 'chord type' variable in each chord. Chord names are normally made up of two parts - the pitch of the chord, then whether the chord is major, minor, or any other type. The original plan was to hold all of this information in one variable 4.3.1, but this would involve unnecessary tokenisation later, so the natural solution was to split the chord name variable.

Using the same principles, perhaps within the note structure the pitch should have been split into the note name, whether it is a sharp or a flat, then which octave it is. This however was not adopted, as the pitch is a combination of all of these. It does not make sense to split up the name and details of whether there is a sharp or a flat as this would only help the programmer and would not intuitively make sense. The separation of the octave makes more sense, but it was decided to keep the pitch as one object for simplicity

A graphical representation of the score data structure can be seen in Figure 5.2. It is not totally accurate, as the chord structure holds pointers to notes rather than notes themselves.

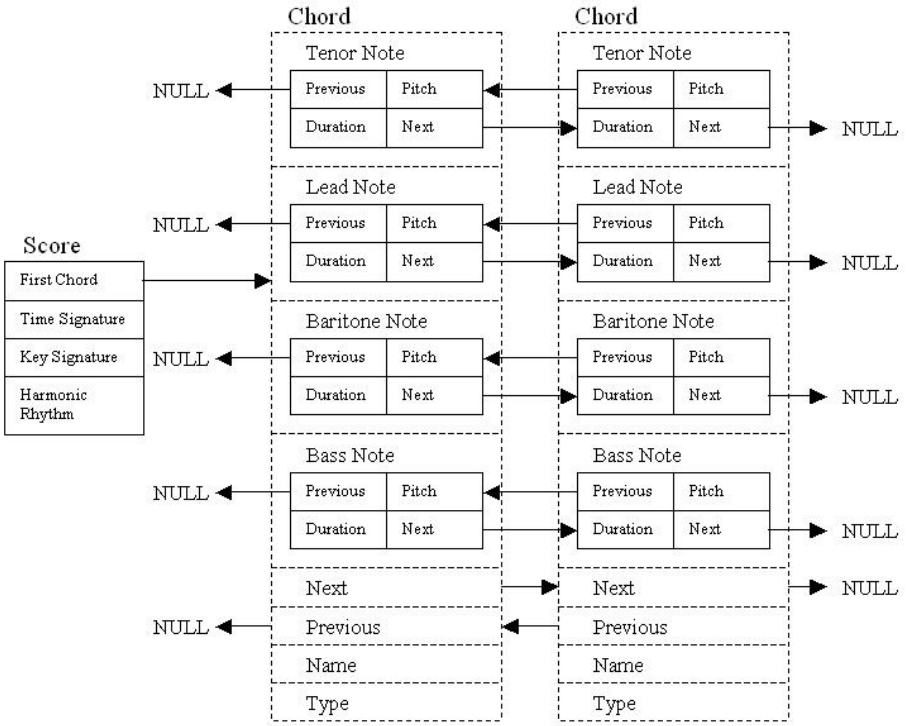

Figure 5.2: Score data structure

It was recognised in the literature survey that the circle of fifths looked similar to finite state graphs (see section 2.4.1), but it was only in the implementation phase that it was decided to implement this data structure as a finite state machine. Each node contains the degree of the scale (a numerical semitone using the tonic as number 1), then an array of 'next node' objects. These objects contain a pointer to the next node and a rating for that.

It was decided that an array was the best option because for each node there is only a maximum of 7 'legal' progressions. If the FSM was much larger, it would be more logical to have a list, but that would be 'overkill' for this situation and use. When initialising this array, it was decided that the initialisation should set all of the unused places in the array to NULL, to give an indication when the values had stopped.

For the sake of this project, the progression of going straight across the circle of fifths is not a legal progression. This is not strictly a progression anyway - it is a substitution, so it was declared out of the scope as it could lead to unnecessary complications.

5.3.2 Inputting of the melody

When choosing the possible solution, it was not recognised that to input the melody, a simple tokeniser and recogniser were needed. One way round this would have been to input just the melody line, but it was decided that it would be preferable if the input could be previewed in Lilypond, so LEX was used with the following grammar to tokenise the Lilypond input:

```
whitespace   [\t \n]+
duration     [1-9]+
timesig      {duration}\/{duration}
upoctave     [']
downoctave   [,]
shafla       es|is
pitch        [abcdefg]+({shafla})*({upoctave}*|{downoctave}*)
majmin       \\(major|minor)
keysig       {pitch}{whitespace}{majmin}
rest         [r]
```

In addition to this, there were several tokens which were to be ignored, as these are just the tags from Lilypond:

```
\\score
{
}
\\notes
\\time
\\key
\\layout
\\clef
violin
```

By calling `yylex()`, the individual tokens could be parsed into the score.

5.3.3 Process for extracting the primary harmonies

The general process for extracting the primary harmonies from the melody never had a single solution. The chosen solution in the design phase was to rank each of the possible 'next' chords by the most common progression and whether the melody note was in the chord. The type of the chord would then be added later.

It takes little investigation to realise that this is too vague an approach to adopt when implementing this functionality. Constraints were added to clarify this progress. In the chosen solution, the circle of fifths would have probabilities for each of the next nodes, but to keep this data structure aligned with the rest of the system, it was decided that these should be replaced with a numerical rating. These match with points given for whether the melody note was in the chord, which provides a compromise between the circle of fifths and having a consonant chord.

Von Helmholtz did research into consonant and dissonant intervals. Rather than just testing if the melody note is in a chord, the aim is to find the chord where the melody note is the most consonant note. Unison, octaves and perfect 5ths are all the most consonant, so they receive a higher ranking; then perfect 4ths, major and minor 3rds are also considered. Any other notes are considered to be dissonant with the chord. Unfortunately by this method, some of the 'juicier' chords such as the 9th and the 6th are lost, indicating that this is not the optimum method.

The rankings from the consonance test and the circle of fifths progressions are then combined. If a possible chord will not yield a consonant chord, then it is ignored, as there is no guarantee that the note can be found in that chord.

The second part of this process is to traverse the structure containing all the chords to find the chord type. As the chord types are ordered by most common (see section 4.3.4, it is the first matching type that is assigned. This order was derived from the barbershop principles that the dominant seventh and major triads are the most common.

5.3.4 Audio Output

It had been decided that a separate output would be generated for audio as well as the sheet music to allow for just-intonation playback. No implementation details had been decided in the design, which left all decisions to be made at implementation.

The chosen solution was to work using frequencies in Csound rather than relative pitches. This therefore called for two new data structures (lookup tables):

- *Absolute values of semitones* - this was for the lookup of the lead line, as this follows the equally tempered scale. The 29 degrees of the scale were put in a lookup table, despite only 16 being used by the lead. This is to allow for any extensions in future, as in some barbershop pieces, the melody is passed to other parts and the melody always has to be equally tempered.
- *Ratios of the degress of the scale* - this is for working out the other parts' frequencies. Simple arithmetic can be performed using the ratios and the lead frequency to get the just intonation notes.

There were a few difficulties in working out which octave the frequencies should

be in, as checks have to be made for apostrophes and commas in the representation. It was at this stage that the weakness of the data representation was particularly noted; it would be much more sensible to have a separate octave variable in the 'Note' structure, as mentioned first in section 5.3.1.

The implementation of this function yielded audio output that was correct, but due to the use of just intonation, sounded not dis-similar to car horns. A considered solution to this problem was to add a little dither to each voice, as the one instrument was being used. However a better solution was discovered. Gustaf Kalin [9] completed a thesis during the implementation of this dissertation which was an investigation into the formants used in barbershop singing. From this research, it would be possible to adjust the formant variables within the voice instruments to reflect the tenor, lead, baritone and bass formants and therefore produce a better barbershop sound.

5.3.5 Musical decisions

There were several decisions within the piece that had to be made based on musical knowledge rather than computer science knowledge.

Within barbershop, there are clear rules about how to voice chords, but they all hold different weightings. Some rules are solid and must never be broken, whereas others can be manipulated to a certain extent. These were separated into positive and negative rules:

- Positive rule - If a rule is met, then points are added onto the rating for that chord. An example would be adding points for a small distance between the previous note and current note for each part.
- Negative rule - If a rule is not met, then points are deducted from the rating for that chord. An example would be that 10 points are deducted if there are parallel 5ths in a chord progression.

It can be seen that the solid rules have been turned into negative rules where points are subtracted. Those rules which can be twisted are positive rules in that in the worst case scenario they score 0 points.

An issue which arose during the representation of chords was that in some cases there is a duality of accidentals (i.e. a G♭ is the same as an F♯). It was necessary to make a compromise between labelling them all as sharps, or all as flats.

The solution was to use sharps and flats to cover the most common key signatures. As barbershop is a cappella, the key signature is only an issue when using an instrument to learn the piece. The following decisions were made:

- F♯ will always be used instead of a G♭
- C♯ will always be used instead of a D♭
- B♭ will always be used instead of a A♯

- E♭ will always be used instead of a D♯
- A♭ will always be used instead of a G♯

This covers the 6 key signatures from D major through to E♭ major, which are the most common key signatures used in barbershop.

5.4 Problems encountered

5.4.1 Problems with octaves

The main problem with the implementation was with the use of octaves. In manual arranging, all thoughts about octaves are done implicitly throughout the process. Because of the lack of specific rules within the current domain of arranging, the octave of each note was ignored, and just added on as a final piece of functionality.

The first problem with this comes when comparing the voicings and generating each voicing. It is not possible at this stage to know whether the current note is higher or lower than the previous one, or whether it is still within the range of the voice. The solution to this was a naive calculation of distance between the notes - ignoring whether it goes up or down.

The second problem is that it is not possible to compare whether there are parallel octaves in changing notes. An octave in this representation is exactly the same as unison, and parallel unison is perfectly acceptable in barbershop.

This shows an underlying problem with the representation of the scale. The original representation was designed for ease of translation between numbers and characters, but focused on a 12 note scale, rather than the full range of notes that are available in barbershop (a total range of 29 notes ranging from F♯ one and a half octaves below middle C throught to B♭ above middle C).

The general solution was to create a function to put each note of every part into the correct octave. For the tenor part, this is not a problem, as there is only an octave of range, but for the baritone and bass parts there is more of a problem. The simple solution was to put every note into the most common part of the range where there is a choice. A better solution may be to compare each note with the previous one to find the smallest distance, and to make sure that the baritone part is always above the bass part.

To do this problem justice, an investigation into why certain octaves are chosen for parts would have to be made. This would then allow for rules to be formulated within this augmented domain of arranging.

5.4.2 Problems with the harmonic rhythm

The other problem encountered was regarding the harmonic rhythm of a melody. For most melodies, this is a constant rhythm - changing once or twice in a bar, but with some melodies, the chords change with no particular pattern. In itself, this did not prove to be much of a problem for this project, as melodies with a constant harmonic rhythm were chosen to test the system.

The problem encountered was more to do with the notes between the notes that are analysed to work out the primary harmony. These 'odd' notes are not considered when working out the primary harmony for that bar, so there is a possibility that they will not form any of the notes in the chord. If this is the case, it causes the system to crash. It may therefore be necessary to check the consonance of these notes in addition to the others. However, a result of this may be very dull arrangements without 'exciting' chords.

The other solution to this problem may be to introduce some way of assigning the chord name and type in the same pass over the score. This way, any backtracking that is necessary would not increase the complexity as much, as future chords will not have names and types assigned.

Chapter 6

Conclusions from solutions

'If true computer music were ever written, it would only be listened to by computers'

- Michael Crichton

6.1 Introduction

This project at times did not follow the original plan for the implementation, yet it managed to achieve the basic outcomes. It has succeeded to draw together algorithmic composition methods and barbershop music coherently, and although it has only scratched the surface in places, it has left opportunities for further work in this combined area.

6.2 Evaluation of the representation

Without a good underlying data structure, this project would have failed. However, the main score data structure provided flexibility and easy traversal. The original thoughts about this structure considered using existing data structures, or even a two dimensional array, but because a bespoke structure was created, there was all the necessary functionality with no superfluous parts.

At times the representation was a barrier to functionality though. The underlying problem throughout the project with octaves could have been avoided if more thought had been given to the whole range of barbershop voices. The project mainly drew upon the idea of a 12 note scale, with no use for octaves, as this is what composers and arrangers think in. However, to bridge the gap between manual arrangement and automated arrangement, the implicit procedures (such as working out octaves) need to be made explicit.

The solution to this problem would be to add an additional facet to the view of the representation. It was perceived in the implementation phase that there

was one model of the score in progress, and the use of numbers and characters to represent this was just a different way of viewing the same data. It therefore would not be too difficult to add another view/interface. This would require another lookup structure which would convert between the character representation, the 12-note scale representation, and a new representation. The new representation would cover the 29 notes in use within typical barbershop and would make the octaves function obsolete.

Another weakness in the current representation is with the duality of accidentals (called 'intentionals' in barbershop!). The duality was handled with a compromise in this implementation by setting some semitones as sharps the whole time and the rest as flats the whole time (see section 5.3.5). This is not ideal, as it means in some cases that in an arrangement with sharps in the key signature, one may have an accidental in the arrangement as a flat, which is not good musical practice. To solve this problem, it may be useful to traverse the score once more when the arrangement is complete, and correct any wrongly notated accidentals by comparing them with the key signature. By doing this traversal at the end, there is still the simplicity when calculating chords and voicings in the main part of the program.

6.3 Evaluation of the main functions

It was decided that there should be more than one method of evaluating the functions within this program to give a complete and reliable view. The first evaluation that was done was an examination of the code. This evaluation was carried out during the coding when decisions were made, using test programs which can be found in Appendix C.2.

The second type of evaluation was of the output. Two outputs are produced every time the program is run; a piece of sheet music and an audio file. This duality of output means that evaluating the output can be done on two levels. Those who are not experts can listen to the audio, but those who wish to analyse the output in more depth can examine the sheet music and analyse the chords and voice leading.

Both the audio and sheet music output were circulated amongst barbershop singers and arrangers both in the UK and America (where barbershop singing is higher profile). Six arrangements of the first eight bars of 'Down Our Way' were given in both audio and sheet music. Two of the arrangements were generated from the melody and primary harmony (see appendix B.2), and the other four arrangements were generated from just the melody (see appendix B.3). This information was not given to the evaluators as this might have biased their evaluations.

One of the first comments received from John Wiggins, (an 'expert' evaluator) was that the output file should not have been a WAV file, as the voice leading is not clear from the audio output, despite the chords and progressions being clear. This indicates that experts use audio to evaluate pieces of music as well as just analysing sheet music. Jim Clancy (the director of 'Vocal Majority') will

write an arrangement, hand it out to the chorus; they sing it and depending on how it sounds he might change several of the voicings, or even the chords. This shows how much emphasis there is on how barbershop sounds, and not just the technicalities.

6.3.1 Voicing of the harmonies

Analysis of the code

The voicing of the harmonies once they had been identified proved to be one of the most successful parts of the implementation. All of the rules were implemented using 'if-then' statements and the ratings work well in practice with a barbershop arrangement given the primary harmony. The process for deciding the rating for each of the rules was not very scientific however. Each rating was decided upon from experience and background knowledge of barbershop, as there appeared to be no universal weighting system. It is therefore expected that optimisations to this functionality may be possible by altering the numbers in the ratings.

Unfortunately two of the rules for voicing the harmonies had to be ignored for this implementation due to the self-imposed constraint of the data representation. As octaves in the implemented program are the same as unison, it was not possible to check for parallel octaves. With the addition of another representation as suggested in section 6.2, this rule could be added in quite simply. Similarly, the checking of the range would be possible at this stage instead of manipulating the arrangement to ensure that it is all within the range of each of the voice parts.

One area in which further investigation may be interesting is that of a fitness function for the whole of an arrangement. In the current case, the decision made at the first chord for the voicing will affect the whole of the arrangement due to the comparison with the previous chord in the voicing of each part. This can lead to dramatically different arrangements each time the program is run, with a huge range of quality of outputs. This proposed area for investigation would involve a higher degree of complexity in the running of the program, as each variation of an arrangement would be generated, then rated. However, the quality of outputs should remain consistently good after this investigation.

User evaluation

The users evaluated the first two arrangements which were those where the voicing function was tested. As noted by Rick Spencer (a Music Educator for the American Barbershop Harmony Society), the first output is the closest to the society arrangement of 'Down Our Way'. From this it is possible to draw a conclusion that the voicing function is a success. Nevertheless it is necessary to note that there were two outputs from the same input (see appendix B.2), and second one is clearly worse.

The feedback received about the second output was quite slating. '[The second output]... in spite of using the correct chords, is almost unsingable' said John Wiggins (one of the most respected arrangers in the British Association of Barbershop Singers). This can be attributed to the intervals that have to be sung by each part. Currently there is no function to test the 'singability' of each part by checking easy intervals to sing, but this would not be too difficult to add in. This could lead to several iterations of the voicings where each arrangement was given a rating, before another arrangement was generated.

Within the first output, there was a divided opinion about whether the bass part in bar 6 was a legal note in barbershop. Because there is a gap of over an octave between the bass and the other parts (the interval is a 10th), John Wiggins has the opinion that this divorced bass is illegal. However, Robin Jackson (Head of Student Music at the University of Bath) disagrees and comments that it is perfectly legal, and often the bass part in different styles of music will have a 10th between it and the next note. The conclusion which has therefore been drawn from this is that it is perfectly legal, and musical too, but it isn't strictly barbershop. As an aside, it is the octaves function that created this octave gap, and it was put in totally randomly, therefore in another case there may not be this large interval. The best way to solve this problem would be to change the octave function and to tie it in with the voicings, when the representation is changed.

6.3.2 Identifying the primary harmonies

Analysis of the code

As acknowledged in the initial consideration of solutions, the implementation of the identification of primary harmonies is the weak link in this program. A thorough investigation into this part of the program was not possible and this is the least documented part of the arrangement process. As discussed in the implementation section, the solution finally implemented followed vague principles and tried to combine consonance and the circle of fifths (see section 5.3.3. It produces arrangements which follow the circle of fifths, but which are musically 'weak'.

As this has never been done before, this could warrant a whole project in itself, which could then be applied to many styles of music. Prather's work [16] enables the primary harmonies for a melody to be extracted given a piano score, which could be implemented again for this project. This would create barbershop arrangements from a piano score, which is an improvement on the current situation, but does not address the problem of identifying primary harmonies from just a melody.

It is through the weakness of this implementation that only short pieces have been generated using this program. If longer pieces were tested and proved to be too innaccurate, the deviation from the actual harmony of the piece would be greatly exaggerated. It may be that the program will crash when a note found will not fit with any progressions.

User evaluation

It was expected that the user evaluation for the four arrangements that were entirely arranged by the computer would be quite critical. In the majority of the feedback, this was the case. John Wiggins made very little comment, but gave a summary of all of the arrangements, saying '[only the first two files]... show the correct chords so the other 4 can be considered failures'. This is an understandable reaction, but it is possible that he did not realise that this is just an initial investigation into algorithmic harmonisation within the style. This feedback highlights just how difficult the task in hand was.

Mike Lofthouse (Education and Music Services Director for the British Association of Barbershop Singers) was not quite so critical, but still stated that the computer–arranged pieces were not quite what he was used to. This indicates that musically the pieces may be acceptable, but not within the barbershop style. This suggestion is confirmed by another barbershop singer, John Pengelly who sings with the Taunton Rivertones Chorus. He is an amateur arranger with interests in other musical styles as well, which means his view on the outputs will be slightly less strict. His feedback was, 'some of the other interpretations are 'interesting'...but my bag is jazz comping and I tend to do a lot of chord substitutions so I guess they sound 'interesting' but then I'm trying to achieve tonal colour.' This is indeed confirmation that the outputted arrangements are consonant, and in some cases verging on musical.

The final comment that was made about the four computer–arranged pieces is one of special interest. It was commented on by both John Pengelly and Robin Jackson that it is the voicing of these four arrangements that stops them from being fully musical. Whereas an assumption had been made that the voicing function was complete and successful, these comments suggest otherwise. They also suggest that there is maybe a more fundamental link between the primary harmony identification and voicing function than first imagined. This could potentially mean that the whole of the process used to arrange these pieces is flawed!

It was noted by another member of the Rivertones chorus that this was 'admirable as first steps'. However John Wiggins is right when he says that '...it's unlikely to rival the human variety'.

6.4 Evaluation of the general methods

When arranging manually, the process is very linear; starting with identifying primary harmonies, then moving on to voicing the harmonies, finishing with adding any embellishments. This was the approach that was therefore adopted for the project. It worked well as an approach when implementing, as it allowed for the modularisation of the program, so individual components could be tested before being integrated into the final program. However, it may be interesting to investigate whether the addition of primary harmonies could be combined with the voicing, to give one pass at the arrangement.

There are other possibilities for the implementation methods but they still rely on the underlying data structure though. This is one thing that would not be changed with a repeated implementation.

6.4.1 Rules or learning

At the start of the project, there were two possible paths to take with regard to the method of creating barbershop harmonies from a melody. One was a rule–based system, and the other was a genetic algorithm and neural network route. The former was chosen at that time, as issues of copyright with existing barbershop arrangements had not been clarified. This restricted the use of existing arrangements as inputs to a learning system.

If a learning system had been used, it is still forseen that similar arrangements would have been produced. It would have been useful to use a learning system in conjunction with the current implementation in order to clarify decisions. Such a system could have helped with:

- Finding the most common chord types. An analysis would help to rank the chord types in the structure holding all chord types. It would have allowed a probability system too, which would have added interest to chords, with not just the most common being used the whole time
- Adding weightings to the voicing rules. A statistical analysis of existing arrangements could be used to work out which rules are followed the most strictly.
- Weightings on the Circle of Fifths. The inputted arrangements would have to include chord names to identify the paths most commonly taken when following this structure.

These were the areas where knowledge was not made explicit in any barbershop manuals and therefore could only be gained through experience. With all three areas, it would be necessary to enter traditional barbershop arrangements which adhere to the rules more than modern arrangements do. This would build up the knowledge within the system with less of a degree of error than the ratings and weightings on the rules in the current system.

It should be noted that two types of rule system were used in the current implementation. The first was the use of simple conditional statements, which was used for the voicing of chords. The second was the use of data structures to aid, such as the finite state machine in the circle of fifths. Implementing the circle of fifths as conditional statements was considered to keep a general coherence, but it was decided that this would not be the natural solution. It may be interesting to investigate whether the conditional statements in the rest of the program could be replaced by similar finite state machines.

6.4.2 Backtracking

Currently within the implemented system, if a melody note is found which is not part of the chord assigned to that note, then the system will crash. This could be the case for notes between those falling on the harmonic rhythm. It seems logical therefore that the solution would be to include backtracking in the system. This backtracking would traverse back through the score, examining possible other chords that could be used previously, which would change the current chord (because of the dependencies on previous chords).

The alternative which only became apparent once the implementation was completed was to choose a chord from the secondary harmony possibilities. This gives options to replace the chord with:

- The chord up a 5th
- The chord down a 5th
- The chord up a semitone
- The chord down a semitone
- The chord up a diminished 5th

In effect, this is merely shifting the chord around on the circle of fifths, and opens up many more possibilities for the chords. However, it does not necessarily guarantee to provide a chord to fit the melody note, so backtracking may still be necessary.

The other solution would be a lookahead. This would add a lot of complexity to the problem, but it would ensure that it would be possible to get to the end of a piece and match a chord to every note. A tree would be built as the possible chords for each note are worked out. From here, if there is more than one route from root to leaf that includes all notes of the arrangement, they would be evaluated and one would be chosen. Alternatively, all routes could be outputted, leaving the choice to the person executing the program.

Chapter 7

Critical appraisal and further work

'If the time comes when your computer will churn out an arrangement which somebody is going to use...as an arranger, I will of course commit suicide'

- John Wiggins

7.1 Appraisal of the process

7.1.1 Approach to the implementation

Throughout this project, the emphasis was on a research implementation to investigate the methodologies rather than moving through a software development process. This has led to a program which delivers the functionality when provided with a valid and 'arrangable' input. However, if the input melody contains melody notes which do not fit with the implemented version of the circle of fifths, the program is likely to crash.

The main problem with the implemented system is the lack of error handling in several places. Initially, the input is not checked for format errors, which could be introduced to the system. In the case of an invalid input, the program will crash and will stop responding. The other possible errors which are not addressed are handling of memory allocation errors and problems where a melody note is not within the suggested primary harmony chord.

As this project was addressed with research in mind, the code is not optimised, therefore wastes time and memory. The algorithms used have not been analysed for more optimal solutions, and the memory allocation is wasteful. There are occurrences of duplicate data structures (namely the AllChords structure) which could be created just once as a global variable, because it never changes.

In all, there are many alterations necessary before this code could be used for anything but research purposes. Although the current code does work, it is very fragile and not suitable for use by inexperienced users. For any form of user testing, it would need to be made much more robust.

7.1.2 Unimplemented functionality

Embellishments

Unfortunately, there was planned functionality that was not implemented due to time constraints. This is the addition of embellishments to a finished arrangement. The reasons for disregarding this functionality was that it seemed more important to get the primary harmony function working to an acceptable standard than to implement more functionality. This would have led to more functions, but without the quality. The embellishments were always there as additional work that was not core .

It was realised when compiling the chosen solution that time would probably not allow for the implementation of embellishments, so the solution was not fully considered. With hindsight, the full solution would not have been simple, and may have been of the same level of difficulty to implement as the primary harmonies. The simple solution however would be to traverse the score until sustained notes were found. At this stage there would be several options:

- *Add an echo* - there are several standard (and cliched) echoes which could randomly be chosen and inserted. In all of them, one, two or three of the parts have a sustained note, and the other(s) add in a little melody.
- *Add a pickup* - again there are several standard pickups, which could be used. The first main type is where the lead sings the first few notes of the next line as a solo with rests in the other three parts. The other main type is for another of the parts to perform a run, up or down into the next phrase.

Again with hindsight, it can be observed that this would have required much time to implement and it is reliant on the primary harmony and voicing functions. The right decision was made in not implementing this functionality, although it would have added much value to the program.

Tags and introductions

One area of implementation which would be necessary before outputs could be called proper barbershop arrangements is that of introductions and tags. This was well beyond the scope of the project and would involve some complex functionality:

- Analysis of the melody for themes - this is necessary as the introduction

and tag need to use melodic themes from the main body of the arrangement to give the piece cohesion.

- Knowledge of all cliched tags - this is necessary to make sure that the tag can be defined as true barbershop.
- Analysis of the melody for genre - the program must be able to decipher the genre of the piece to apply the appropriate tag, as there are different tags for different styles of arrangement - ballads, upbeat, swung etc.

This is moving into the area of algorithmic composition and not just harmonisation, as the melody line would have to be composed from knowledge of the rest of the piece.

Modulation

Only once the coding was finished was modulation considered. This is where the melody changes key in the middle of the piece. There is nothing in place within the current system that would allow for this to happen, therefore the melodies which could be inputs are restricted. The algorithms associated with modulation would be very complicated, as firstly a modulation point has to be identified within the melody, then the key that it modulates into has to be identified. From here, the key change would have to be written in, which would require the system to hold knowledge of how to change key into each of the degrees of the scale.

From these initial thoughts about modulation, it is clear to see that this is a piece of functionality which would only be added once the rest of the work was up to an acceptable standard. This is an impressive extra bit of functionality, but it is not essential and there would be a large cost in time to develop it for not much return in the final program.

7.1.3 Further comments

It has become apparent through the design and implementation process of this project that a firm basis of both knowledge and experience are necessary for a project of this nature. As a basic program, the functionality works, but there is certainly a limit to the quality of work that can be produced with a small time of investigation into the domain of barbershop arranging. Experience is just as important as knowledge, and as there are so few barbershop arrangers worldwide, much of the necessary knowledge has not been explicitly recorded. Instead it is passed on largely by word of mouth and by practical experience within the domain.

Before further work can be attempted to improve the current implementation, experience should be gained of manual practices within the domain before they can be automated efficiently. An alternative to this ‘manual’ experience would be to implement a learning system that could work through this experience by arranging many pieces which could then be rated by humans. This is necessary

because some form of personality needs to be injected into a program like this. Each arranger has certain ways to deal with situations where the rules are broken, and these can only be developed through experience. Another method of adding in this experience would be to work closely with an existing arranger to try to formalise their practices and approaches to arranging, rather than working from a manual.

7.2 Work that could build on this

7.2.1 Human/Agent intervention

In the current form of implementation, there are distinct steps in the arrangement of the melody. Because of this separation of functionality, it might be useful to make an output available at each step. This way a human could preview the arrangement and make any necessary alterations to improve the harmonies. Arranging music is fundamentally a creative activity, so to have some human input would be very appropriate.

Another possible view on this intervention at the preview stages is that it could be an agent that makes the alterations. This way, the current program could be built on easily with very little modifications to the code, and the arrangements could increasingly become better as these external modules of agent code improve. The code for generating the harmonies and voicings could remain untouched, therefore leaving less to go wrong.

The use of agents and humans working together in conjunction could lead to a learning system for amateur arrangers. This could further lend itself to a commercially viable package within the barbershop world.

7.2.2 Extensions to the work

The separation of the data structure from the rules opens up the scope for further projects which involve four part scores. This generic score structure is reusable for any projects which may require a score, and is easily extendable to add more parts to the chords if necessary. This would mean that any future work would have the foundations prepared already.

As the rules are separated, these could be also easily altered to accommodate other styles of music. These rules could be added to existing functionality to possibly create a program that could arrange one melody in many different styles. Gospel music would be one of the natural steps on from barbershop, as this too is polyphonic and follows some of the same principles. However, the chords are much simpler and there are many more embellishments to add. Given that the data structure is easy to traverse, it could also be ideal for arranging music such as fugues or even simple rounds.

7.2.3 New directions

On a slightly more superficial level, there is a possibility of introducing algorithmic composition into this process. As the rules of barbershop are held as knowledge within the system, there may be a possible extension to the program of composing full arrangements from scratch. In a change from the current system, a melody would not be the input; rather the program would generate a melody after harmonies had been generated.

This is very improbable even though it would be feasible to implement. Barbershop songs are very rarely written from a specially composed melody; instead they are arranged from existing melodies that are perceived to fit the style. This proposed extension is therefore unlikely to be of much use in furthering knowledge and speed of barbershop arranging (although it could be an interesting investigation).

Bibliography

[1] J. L Arcos, J Sabater, and R. Lopez de Mantaras. Using Rules to support Case-based Reasoning for harmonizing melodies. *Multimodal Reasoning: Papers from the 1998 AAAI Spring Symposium*, pages 147–151, 1998.

[2] David Cope. Computer Modeling of Musical Intelligence in Experiments in Musical Intelligence. *Computer Music Journal*, 16:2:69–83, 1992.

[3] Encyclopædia Britannica Online. Partch, Harry' Britannica Concise Encyclopædia. Available from: http://www.britannica.com/ebc/article?tocId=9374629 (Accessed 22 November 2004).

[4] Kemal Ebcioglu. An expert system for harmonizing four-part chorales. *Machine models of music*, pages 385–401, 1992.

[5] David Gerhard. Automatic Interval Naming Using Relative Pitch. *Bridges: Mathematical Connections in Art, Music and Science*, pages 37–48, 1998.

[6] P. M. Gibson and J. A. Byrne. Neurogen, Musical Composition Using Genetic Algorithms and Cooperating Neural Networks. In *Second International Conference on Artificial Neural Networks*, pages 309–313, 1991.

[7] Goffredo Haus and Emanuele Pollastri. A Multimodal Framework for Music Inputs. In Marina del Rey, editor, *Proceedings of the 8th ACM International Conference on Multimedia*, pages 382–384. ACM Press, 2000.

[8] David Howard and Andy Tyrrell. The Harmonius Chipsmith [DSP Chips for Four-part Singing Synthesiser]. *IEE Review*, 45(5):215–218, September 1999.

[9] Gustaf Kalin. *Formant Frequency Adjustment In Barbershop Quartet Singing*. PhD thesis, KTH, Department of Speech, Music and Hearing, 2005.

[10] Cheng-Yuan Lin, J.-S. Roger Jang, and Mao-Yuan Hsu. An Automatic Singing Voice Rectifier Design. In *Proceedings of the eleventh ACM international conference on Multimedia*, pages 267–270. ACM Press, 2003.

[11] G. Loy and C. Abbott. Programming Languages for Computer Music. *Computing surveys*, 17(2):256–263, June 1985.

[12] Ryan A. McIntyre. Bach in a Box: The Evolution of Four Part Baroque Harmony Using the Genetic Algorithm. In *First IEEE Conference on Evolutionary Computation*, pages 852–857, 1994.

[13] James Anderson Moorer. Music and Computer Composition. *Communication of the ACM*, 15(2):104–113, February 1972.

[14] J. M. H. Peters. The Mathematics of Barbershop Quartet Singing. *Bulletin. The Institute of Mathematics and Its Applications*, 17(7):137–143, July 1981.

[15] Somnuk Phon-Amnuaisuk and Geraint A. Wiggins. The Four-Part Harmonisation Problem: A comparison between Genetic Algorithms and a Rule-Based System. In *1999 - Proceedings of the AISB'99 Symposium on Musical Creativity*, 1999.

[16] Ronald E. Prather. Harmonic Analysis from the Computer Representation of a Musical Score. *Communications of the ACM*, 39(12es (electronic supplement)), 1996.

[17] Gary M. Rader. A Method for Composing Simple Traditional Music by Computer. *Communications of the ACM*, 17(11):631–638, November 1974.

[18] SPEBSQSA. *Barbershop Arranging Manual*. The Barbershop Harmony Society, Kenosha, Wisconsin, 1980.

[19] SPEBSQSA. *Barbershop Arranging Manual*, chapter Barbershop harmonization: Approach One, pages 74–141. The Barbershop Harmony Society, Kenosha, Wisconsin, 1980.

[20] SPEBSQSA. *Barbershop Arranging Manual*, chapter Barbershop harmonization: Approach Two, pages 141–206. The Barbershop Harmony Society, Kenosha, Wisconsin, 1980.

[21] SPEBSQSA. *Barbershop Arranging Manual*, chapter Barbershop harmonization: Approach Three, pages 206–245. The Barbershop Harmony Society, Kenosha, Wisconsin, 1980.

[22] SPEBSQSA. *Barbershop Arranging Manual*, chapter Definition of barbershop harmony, pages 3–27. The Barbershop Harmony Society, Kenosha, Wisconsin, 1980.

[23] SPEBSQSA. *Barbershop Arranging Manual*, chapter The evolution and future of barbershop, pages 379–406. The Barbershop Harmony Society, Kenosha, Wisconsin, 1980.

[24] J. Sutton. A Cappella Alchemy 4 - Basic Arranging. Available from: http://www.labbs.org.uk/educ/resources.html (Accessed 30 October 2004).

[25] John Wiggins. Beginner's barbershop arranging. BABS Harmony College, University of Worcester, August 27–29 2004.

Appendix A

Methodology used for this project

A.1 Introduction

It was decided that although this project was for research purposes only, some software engineering methods would be used to focus the direction of the project. Therefore, any tools that were used were just to clarify details and aid the research. As this was a research project, there was no software engineering lifecycle used as such, but elements were drawn from experiences of past projects and tools used through those.

The identification of possible solutions, the chosen solution and the implementation sections are all well documented in the main section, with no need to add more information.

A.2 Requirements elicitation and analysis

A.2.1 Requirements elicitation

The requirements elicitation and analysis process utilised many tools, as although this is a software development project, it is just for research and not for a specific end user. The following tools were used:

- Analysis of the problem
- Brainstorming
- Experience from previous software development processes
- Examining functionality of systems in previous academic research within this field

As this was to be a piece of academic software used to solve a problem, there are no 'users' as such, so interviews could not be conducted. Any decisions made were evaluated by comparing possible paths to existing work and examining the problem definition. It was also accepted that this is an iterative process and the problem definition evolved as the understanding of the domain changed.

A.2.2 Requirements Specification

When writing the requirements, most of the functional requirements were derived from working through possible arranging methods, as these provided a walk-through of the system functions. However, in a project such as this, it was not possible to use just one tool for the requirements elicitation. The further requirements have been broken down into system requirements and domain (external) requirements. This was necessary, as it is important to be aware of any effects that this software may have on the environment, and any effects the environment may have on it.

A.2.3 Functional requirements

The user shall be able to input a melody

Description: The user shall be able to input a text file of the melody to be arranged into four parts
Rationale: There has to be a way to input a melody to the system, and the most reliable method for this has been identified as in the form of a textual input.
Change History: Created on 5th December 2004

The system shall hold rules for adding a primary harmony to the melody

Description: The system shall hold a table or database of rules for adding the primary harmony to the melody for this stage in the arranging process
Rationale: It is necessary to add the primary harmony by applying rules in some form and these rules need to be stored somewhere within the system
Change History: Created on 5th December 2004

The system shall hold some representation of the melody

Description: The system shall have to hold an encoded version of the melody which can then be added to and altered as part of the arrangement
Rationale: As there will be additions to the melody, it is obvious that an internal representation of the melody and progressing arrangement should exist.
Change History: Created on 5th December 2004

The system shall hold some representation of the circle of fifths

Description: The system shall hold the rules which are represented by the circle of fifths which are used in deciding the primary harmonies

Rationale: This is necessary as the rules form one of the main tools within barbershop arranging, and therefore would be useful to utilise in the existing form without changing the paradigm for the use of them.
Change History: Created on 5th December 2004

The system shall be able to output the representation into Csound

Description: The system shall be able to output the finished arrangement into Csound to be played through audio. This will include a transformation of the representation into the Csound syntax, and a function to change the chords into just intonation.
Rationale: One of the outputs of the system will be audio, so the representation should be transformable into a Csound file for it to be output. As barbershop works on the principle of just intonation, it is important that the output is played in just intonation, hence the need for a function to change the chords.
Change History: Created on 5th December 2004

The system shall be able to output the representation to Lilypond

Description: The system shall be able to output the finished arrangement into Lilypond to be viewed as a piece of sheet music in the traditional notation.
Rationale: One of the outputs of the system will be a visual output, so the representation should be transformable into a file which can be opened in Lilypond to view the sheet music.
Change History: Created on 5th December 2004

There shall be a function to voice the primary harmonies

Description: There shall be part of the system that will examine the primary harmonies and choose the notes from the chord that each part will sing based on some fundamental laws.
Rationale: Each part will have to be assigned a note from the primary harmonies and this function will do this.
Change History: Created on 5th December 2004

The system shall hold rules about voicing the harmonies

Description: The system will hold rules about voicing the harmonies - rules that take into consideration the previous notes and the range of each of the four parts
Rationale: In the previous requirement, these rules were used to voice the harmonies, so they need to exist within the system as a form of knowledge
Change History: Created on 5th December 2004

There shall be iteration around the representation to improve the voicings

Description: Once the voicing have been set, there shall be a function to make more passes around the representation to improve the voicing where the previous and next notes are both examined

Rationale: It is accepted that the first pass through the representation of the arrangement may not produce the best voicing, so it is necessary to make multiple passes.
Change History: Created on 5th December 2004

The system shall hold rules about embellishments

Description: The system shall hold rules about adding in embellishments - what the typical embellishments are, and where they apply
Rationale: Adding embellishment will require some rules, so they have to be present within the system
Change History: Created on 5th December 2004

There shall be a function to add embellishments

Description: The system shall have a function to add embellishments to the arrangement once the voicings have been applied
Rationale: To make the arrangement slightly more personal and interesting, embellishments shall be added to the arrangement.
Change History: Created on 5th December 2004

A.2.4 System requirements

Lilypond and Csound are necessary

Description: The Lilypond program and Csound must be installed on the computer that the software will be run on, so that the outputs can be viewed in a meaningful format.
Rationale: The outputs of the program are in the form of inputs for these two pieces of software, so the software must be present
Change History: Created on 5th December 2004

Speakers are needed for the system

Description: The computer must have speakers
Rationale: As one of the outputs is in the form of audio, it is necessary to have some way of listening to it
Change History: Created on 5th December 2004

A keyboard and monitor are needed for the system

Description: The computer must have a monitor and keyboard
Rationale: The main input method will be through text commands and text files, therefore a keyboard is needed, and the main outputs will be visual (either the sheet music or the program information), so a monitor is needed.
Change History: Created on 5th December 2004

The interface shall keep the user informed of actions being taken

Description: The interface will let the user know the phase of the arranging process that is taking place and the state of the system. It will output Lilypond

files after the primary harmonies have been added, and again once the voicings have been applied.
Rationale: This is necessary as if the program crashes, the user will want to know where it crashed to fix the problem.
Change History: Created on 5th December 2004

A.2.5 Domain Requirements

Permission will be obtained for music used in testing

Description: Existing barbershop melodies will be used in the testing of the system to compare the results of the arrangements with existing arrangements, so permission must be granted for this use
Rationale: It is necessary to get permission, otherwise this will be an infringement of copyright
Change History: Created on 5th December 2004

The system shall be for testing purposes, not general public release

Description: The created software shall be for testing purposes only and not released to the general public
Rationale: It is necessary to state this, as it will not be built to industry standards and may not be robust code. The code will be there to investigate possibilities, not to endure heavy use.
Change History: Created on 5th December 2004

The system shall be designed for a small user set

Description: The system is not designed to accommodate blind people, or those who are computer illiterate
Rationale: The system shall not have usability as its main goal and therefore will not be appropriate for certain users.
Change History: Created on 5th December 2004

The system shall be driven by a command prompt

Description: The system shall not have a graphical user interface, but shall be driven by typed commands
Rationale: It is felt that a graphical user interface would be redundant for this software, as it is for testing purposes only.
Change History: Created on 5th December 2004

The necessary legislation will be observed

Description: Any information on the system shall have the permission of the authors and acknowledgements to the source of the data
Rationale: This is to keep in line with the legislation and to ensure that the software is legal
Change History: Created on 5th December 2004

Performance shall not be a priority

Description: The algorithms in the system shall not be optimised unless there is a special need
Rationale: The software shall be for research purposes, therefore the performance speed is not a priority
Change History: Created on 5th December 2004

A.3 Analytical testing

The initial plan when developing the software was that a separate test program would be written to compare the output of the program with an actual barbershop arrangement, but it was realised that this was not a good option. There are possibly many solutions for one input, and the published arrangement may be just one, and not necessarily the best option. For this reason another way of analysing the output had to be found.

Most of the analytical testing was actually of the functionality and whether it yielded results as a valid program. The details of the output and evaluation of the output was left until later. It was only in the implementation of the voicing function and the primary harmony function that the output was examined, so that rules and ratings could be tuned to fit with proper barbershop pieces.

An example of this testing was with a short melody with 8 notes to test the primary harmonies function (using CircleFifthsTest2.c which can be found in Appendix C.2.3). The actual chord progression in the manual arrangement was:

I to III to VI to VI to II to II to II to II

The first output progression was:

I to I to I to ♭V to IV to IV to III to III

This is clearly wrong, so the rankings and balance between the circle of fifths and the consonance was changed. The second output progression was:

I to I to III to ♭III to ♭VI to I to VII to III

This is a better progression, but still deviates from the manual arrangement. The calculation for the difference between two notes was re-numbered slightly and the third output progression was:

I to VII to III to ♭III to II to V to V to I

Again it is a good progression, but it still was not right. The sixth note was then omitted (as this is a ninth in the manual) to see whether this would have an impact. Also the rule that an interval of a fifth between the lead note and the primary harmony is better than an octave, was changed, which yielded the fourth output progression:

I to III to VI to II to II to NULL to V to I

There was then a final change to the weighting of staying in the same position on the circle of fifths, which yielded the best output:

I to III to VI to II to II to NULL to II to II

Here the fourth chord is wrong, but it was decided that the rules would remain unchanged, as this may be picked up by the harmonic rhythm function.

A.4 User testing

It was decided that the best way to evaluate the output would be to ask experts to comment, as they would be able to give a more in depth analysis of the arrangements. A relatively diverse range of users was attempted in order to show different viewpoints, so the user groups that were given the outputs were:

- Arrangers and staff members of the British Association of Barbershop Singers
- Rick Spencer, a staff member of the Barbershop Harmony Society in America
- All members of the Rivertones Chorus, Taunton
- All members of the Alley Barbers Chorus, Bath
- Robin Jackson, Head of Student Music at the University of Bath

This provides the range from amateur arrangers to professional arrangers, and also from barbershop singers through to a classically trained musician. The feedback was mainly via email, but there was a discussion with the Alley Barbers and Robin Jackson, where the processes used to produce the outputs were discussed. It was found that this was the part that the users were interested in more than the output, and all users requested more information on this part. There is great interest in this work within the barbershop world.

Appendix B

Input and Outputs

B.1 Input to system

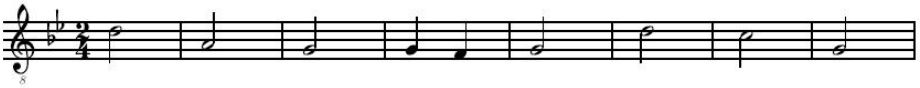

Figure B.1: The inputted melody

B.2 Outputs from the voicing tests

The following two outputs can be found on the accompanying CD as files Output1.wav and Output2.wav.

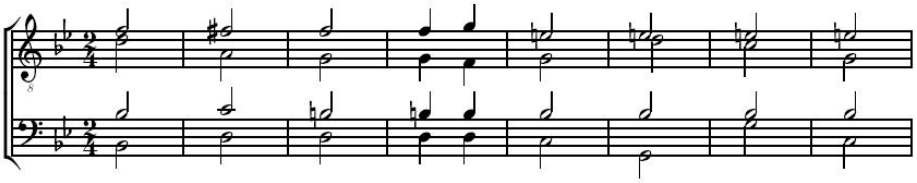

Figure B.2: Output 1 from the voicing tests

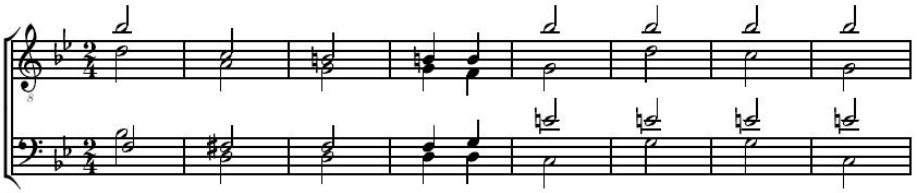

Figure B.3: Output 2 from the voicing tests

B.3 Outputs from the primary harmony tests

The following four outputs can be found on the accompanying CD as files Output3.wav, Output4.wav, Output5.wav and Output6.wav.

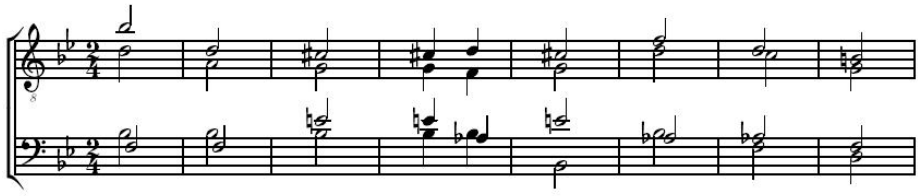

Figure B.4: Output 1 from the primary harmony tests

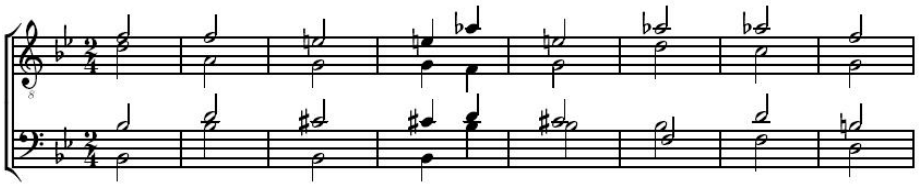

Figure B.5: Output 2 from the primary harmony tests

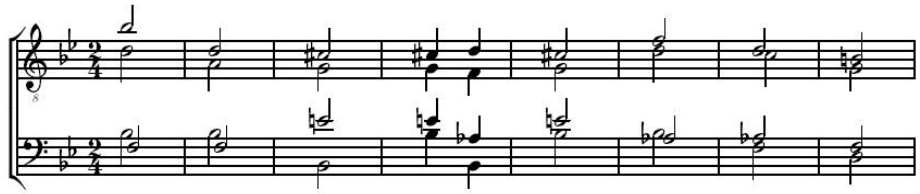

Figure B.6: Output 3 from the primary harmony tests

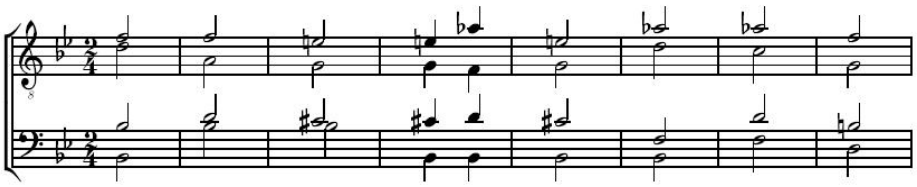

Figure B.7: Output 4 from the primary harmony tests

Appendix C

Program Code

C.1 Libraries

C.1.1 AllChords.h

```
/*******************************
 * Header file for AllChords  *
 * Created 22nd February 2005 *
 * Modified 4th May           *
 * Author: Steve Roberts      *
 *******************************/

#ifndef ALLCHORDS_H
#define ALLCHORDS_H

/*********************************
 * Define an AllChords structure *
 *********************************/

typedef struct allchords
{
  char* name; /* the chord name */
  int firstnote; /* the semitone of the first note */
  int secondnote; /* the semitone of the second note */
  int thirdnote; /* the semitone of the third note */
  int fourthnote; /* the semitone of the fourth note */
  struct allchords* next; /* a pointer to the next chord */
}
AllChords;

/*******************************
 * Declare AllChords functions *
 *******************************/
```

```
/* create a new chords structure given the provided data */
extern AllChords* AllChords_onenew(char* name, int first, int second,
                                   int third, int fourth, AllChords* next);

/* create an array of chords */
extern AllChords* AllChords_new(void);

/* search for a chord */
extern AllChords* Chord_lookup(char* name, AllChords* chord);

/* find first chord containing this note */
extern char* Chord_search(int name, AllChords* chord);

#endif
```

C.1.2 AllChords.c

```
/*****************************
 * Source file for AllChords  *
 * Created 22nd February 2005 *
 * Modified 4th May           *
 * Author: Steve Roberts      *
 *****************************/

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "AllChords.h"

/* create a new chords structure given the provided data */
AllChords* AllChords_onenew(char* name, int first, int second, int third,
                            int fourth, AllChords* next)
{
  AllChords* new;
  if((new = (AllChords *)malloc(sizeof(AllChords))))
    {
      new->name = name;
      new->firstnote = first;
      new->secondnote = second;
      new->thirdnote = third;
      new->fourthnote = fourth;
      new->next = next;

      /* printf("Created a chord\n"); */

      return new;
    }
  else
    {
      return NULL;
    }
}

/* create an array of chords */
extern AllChords* AllChords_new(void)
{
  AllChords* one = AllChords_onenew("dom7th", 1, 5, 8, 11, NULL);
  AllChords* fourteen = AllChords_onenew("13th", 1, 5, 10, 1, one);
  AllChords* thirteen = AllChords_onenew("maj7th", 1, 5, 8, 12, fourteen);
  AllChords* twelve = AllChords_onenew("halfdim7th", 1, 4, 7, 11, thirteen);
  AllChords* eleven = AllChords_onenew("aug7th", 1, 5, 9, 11, twelve);
  AllChords* ten = AllChords_onenew("min6th", 1, 4, 8, 10, eleven);
  AllChords* nine = AllChords_onenew("maj6th", 1, 5, 8, 10, ten);
  AllChords* eight = AllChords_onenew("9th", 5, 8, 11, 3, nine);
```

```
  AllChords* seven = AllChords_onenew("augtriad", 1, 5, 9, 1, eight);
  AllChords* six = AllChords_onenew("dimtriad", 1, 4, 7, 1, seven);
  AllChords* five = AllChords_onenew("min7th", 1, 4, 8, 11, six);
  AllChords* four = AllChords_onenew("dim7th", 1, 4, 7, 10, five);
  AllChords* three = AllChords_onenew("mintriad", 1, 4, 8, 1, four);
  AllChords* two = AllChords_onenew("majtriad", 1, 5, 8, 1, three);

  one->next = two;

  /* printf("Filled the all chords structure\n"); */

  return one;
}

/* search for a chord */
extern AllChords* Chord_lookup(char* name, AllChords* chord)
{
  AllChords* foundchord = chord;
  int counter = 0;

  /* printf("Looking up the chord\n"); */

  while (strcmp(foundchord->name, name) && counter<15)
    {
      /* printf("Name found is %s, name looking for is %s\n",
         foundchord->name, name); */
      foundchord = foundchord->next;
      counter = counter+1;
    }

  if (!strcmp(foundchord->name, name))
    {
      return foundchord;
    }

  else
    {
      return NULL;
    }
}

char* Chord_search(int name, AllChords* chord)
{
  AllChords* chordtable = chord;
  int found = 0;

  /* go through the chords and try to find one which contains this note */
  while (found!=1)
    {
      if (chordtable->firstnote==name)
```

```
        {
          found=1;
        }
      else if (chordtable->secondnote==name)
        {
          found=1;
        }
      else if (chordtable->thirdnote==name)
        {
          found=1;
        }
      else if (chordtable->fourthnote==name)
        {
          found=1;
        }
      else
        {
          chordtable = chordtable->next;
        }
    }

  /* when found return the name of that chord */
  return chordtable->name;
}
```

C.1.3 Chord.h

```
/*******************************
 * Header file for Chord       *
 * Created 14th February 2005 *
 * Modified 4th May            *
 * Author: Steve Roberts       *
 *******************************/

#ifndef CHORD_H
#define CHORD_H

#include "Note.h"

/****************************
 * Define a Chord structure *
 ****************************/

typedef struct chord
{
  Note* tenor; /* the tenor note */
  Note* lead; /* the lead note */
  Note* bari; /* the baritone note */
  Note* bass; /* the bass note */
  char* name; /* the name of the chord */
  char* type; /* the type of the chord */
  struct chord *previous; /* the previous chord */
  struct chord *next; /* the next chord */
}
Chord;

/*************************
 * Declare Chord functions *
 *************************/

/* create a new chord given the provided data */
extern Chord* Chord_new(char* leadpitch, int duration, char* name,
                        Chord* previous, Chord* next);

#endif
```

C.1.4 Chord.c

```
/*****************************
 * Source file for Chord      *
 * Created 15th February 2005 *
 * Modified 4th May           *
 * Author: Steve Roberts      *
 *****************************/

#include "Note.h"
#include "Chord.h"
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

/* create a new chord given the provided data */
Chord* Chord_new(char* leadpitch, int duration, char* name, Chord* previous,
                 Chord* next)
{
  Chord* new;
  if((new = (Chord *)malloc(sizeof(Chord))))
    {
      /* set pointers and values within this chord */
      new->next = next;
      new->previous = previous;
      new->name = name;
      new->type = NULL;

      if(previous)
        {
          /* set new notes to point to previous notes */
          new->tenor = Note_new("r", duration, NULL, previous->tenor);
          new->lead = Note_new(leadpitch, duration, NULL, previous->tenor);
          new->bari = Note_new("r", duration, NULL, previous->bari);
          new->bass = Note_new("r", duration, NULL, previous->bass);

          /* set previous notes to point to new notes */
          previous->tenor->next = new->tenor;
          previous->lead->next = new->lead;
          previous->bari->next = new->bari;
          previous->bass->next = new->bass;

          /* set previous chord to point to new chord */
          previous->next = new;
        }

      else
        {
          /* create new notes with null pointers */
          new->tenor = Note_new("r", duration, NULL, NULL);
```

```
          new->lead = Note_new(leadpitch, duration, NULL, NULL);
          new->bari = Note_new("r", duration, NULL, NULL);
          new->bass = Note_new("r", duration, NULL, NULL);
        }

      return new;
    }
  else
    {
      return NULL; /* malloc error */
    }
}
```

C.1.5 CircleFifths.h

```
/*********************************
 * Header file for CircleFifths *
 * Created 3rd March 2005       *
 * Modified 10th March          *
 * Author: Steve Roberts        *
 *********************************/

#ifndef CIRCLEFIFTHS_H
#define CIRCLEFIFTHS_H


/*************************************
 * Define a Circle of Fifths structure *
 *************************************/

typedef struct circlepoint
{
  struct circlefifths* state;
  int probability;
}
CirclePoint;

typedef struct circlefifths
{
  int notename;
  struct circlepoint* pointers[8];
}
CircleFifths;

/************************************
 * Declare Circle of Fifths functions *
 ************************************/

/* create a new Circle of Fifths node */
extern CircleFifths* CircleFifths_onenew(int number);

/* create a new Circle of Fifths structure */
extern CircleFifths* CircleFifths_new(void);

/* move to the next position (given the lead note) */
extern CircleFifths* CircleFifths_next(CircleFifths* current, int leadnote);

/* give a ranking to an note interval */
extern int notedifference(int i, int j);

#endif
```

C.1.6 CircleFifths.c

```
/*********************************
 * Source file for CircleFifths *
 * Created 3rd March 2005       *
 * Modified 20th April          *
 * Author: Steve Roberts        *
 *********************************/

#include "CircleFifths.h"
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* create a new Circle of Fifths node */
CircleFifths* CircleFifths_onenew(int number)
{
  CircleFifths* new;
  if((new = (CircleFifths *)malloc(sizeof(CircleFifths))))
    {
      /* set pointers and values within this chord */
      new->notename = number;
      new->pointers[0] = (CirclePoint *)malloc(sizeof(CirclePoint));
      new->pointers[1] = (CirclePoint *)malloc(sizeof(CirclePoint));
      new->pointers[2] = (CirclePoint *)malloc(sizeof(CirclePoint));
      new->pointers[3] = (CirclePoint *)malloc(sizeof(CirclePoint));
      new->pointers[4] = (CirclePoint *)malloc(sizeof(CirclePoint));
      new->pointers[5] = (CirclePoint *)malloc(sizeof(CirclePoint));
      new->pointers[6] = (CirclePoint *)malloc(sizeof(CirclePoint));
      new->pointers[7] = (CirclePoint *)malloc(sizeof(CirclePoint));
      return new;
    }
  else
    {
      return NULL; /* malloc error */
    }
}

/* create a new Circle of Fifths structure */
CircleFifths* CircleFifths_new(void)
{
  /* create the nodes */
  CircleFifths* one = CircleFifths_onenew(1);
  CircleFifths* two = CircleFifths_onenew(2);
  CircleFifths* three = CircleFifths_onenew(3);
  CircleFifths* four = CircleFifths_onenew(4);
  CircleFifths* five = CircleFifths_onenew(5);
  CircleFifths* six = CircleFifths_onenew(6);
  CircleFifths* seven = CircleFifths_onenew(7);
```

```
CircleFifths* eight = CircleFifths_onenew(8);
CircleFifths* nine = CircleFifths_onenew(9);
CircleFifths* ten = CircleFifths_onenew(10);
CircleFifths* eleven = CircleFifths_onenew(11);
CircleFifths* twelve = CircleFifths_onenew(12);

/* printf("Have created all 12 nodes in the Circle of Fifths\n"); */

/* say where each of the nodes points */
one->pointers[0]->state = six;
one->pointers[0]->probability = 1;
one->pointers[1]->state = seven;
one->pointers[1]->probability = 2;
one->pointers[2]->state = twelve;
one->pointers[2]->probability = 2;
one->pointers[3]->state = five;
one->pointers[3]->probability = 2;
one->pointers[4]->state = ten;
one->pointers[4]->probability = 2;
one->pointers[5]->state = three;
one->pointers[5]->probability = 2;
one->pointers[6]->state = eight;
one->pointers[6]->probability = 2;
one->pointers[7] = NULL;

six->pointers[0]->state = one;
six->pointers[0]->probability = 5;
six->pointers[1]->state = eleven;
six->pointers[1]->probability = 2;
six->pointers[2]->state = five;
six->pointers[2]->probability = 2;
six->pointers[3]->state = seven;
six->pointers[3]->probability = 1;
six->pointers[4] = NULL;
six->pointers[5] = NULL;
six->pointers[6] = NULL;
six->pointers[7] = NULL;

eleven->pointers[0]->state = four;
eleven->pointers[0]->probability = 1;
eleven->pointers[1]->state = ten;
eleven->pointers[1]->probability = 5;
eleven->pointers[2]->state = twelve;
eleven->pointers[2]->probability = 2;
eleven->pointers[3] = NULL;
eleven->pointers[4] = NULL;
eleven->pointers[5] = NULL;
eleven->pointers[6] = NULL;
eleven->pointers[7] = NULL;
```

```
four->pointers[0]->state = nine;
four->pointers[0]->probability = 1;
four->pointers[1]->state = five;
four->pointers[1]->probability = 2;
four->pointers[2]->state = three;
four->pointers[2]->probability = 5;
four->pointers[3] = NULL;
four->pointers[4] = NULL;
four->pointers[5] = NULL;
four->pointers[6] = NULL;
four->pointers[7] = NULL;

nine->pointers[0]->state = one;
nine->pointers[0]->probability = 5;
nine->pointers[1]->state = two;
nine->pointers[1]->probability = 1;
nine->pointers[2]->state = eight;
nine->pointers[2]->probability = 3;
nine->pointers[3]->state = ten;
nine->pointers[3]->probability = 2;
nine->pointers[4] = NULL;
nine->pointers[5] = NULL;
nine->pointers[6] = NULL;
nine->pointers[7] = NULL;

two->pointers[0]->state = one;
two->pointers[0]->probability = 5;
two->pointers[1]->state = seven;
two->pointers[1]->probability = 1;
two->pointers[2]->state = three;
two->pointers[2]->probability = 2;
two->pointers[3] = NULL;
two->pointers[4] = NULL;
two->pointers[5] = NULL;
two->pointers[6] = NULL;
two->pointers[7] = NULL;

seven->pointers[0]->state = twelve;
seven->pointers[0]->probability = 1;
seven->pointers[1]->state = six;
seven->pointers[1]->probability = 5;
seven->pointers[2]->state = eight;
seven->pointers[2]->probability = 5;
seven->pointers[3] = NULL;
seven->pointers[4] = NULL;
seven->pointers[5] = NULL;
seven->pointers[6] = NULL;
seven->pointers[7] = NULL;

twelve->pointers[0]->state = one;
```

```
twelve->pointers[0]->probability = 5;
twelve->pointers[1]->state = five;
twelve->pointers[1]->probability = 4;
twelve->pointers[2]->state = eleven;
twelve->pointers[2]->probability = 1;
twelve->pointers[3] = NULL;
twelve->pointers[4] = NULL;
twelve->pointers[5] = NULL;
twelve->pointers[6] = NULL;
twelve->pointers[7] = NULL;

five->pointers[0]->state = ten;
five->pointers[0]->probability = 5;
five->pointers[1]->state = six;
five->pointers[1]->probability = 2;
five->pointers[2]->state = four;
five->pointers[2]->probability = 1;
five->pointers[3] = NULL;
five->pointers[4] = NULL;
five->pointers[5] = NULL;
five->pointers[6] = NULL;
five->pointers[7] = NULL;

ten->pointers[0]->state = three;
ten->pointers[0]->probability = 5;
ten->pointers[1]->state = nine;
ten->pointers[1]->probability = 1;
ten->pointers[2]->state = eleven;
ten->pointers[2]->probability = 1;
ten->pointers[3] = NULL;
ten->pointers[4] = NULL;
ten->pointers[5] = NULL;
ten->pointers[6] = NULL;
ten->pointers[7] = NULL;

three->pointers[0]->state = eight;
three->pointers[0]->probability = 5;
three->pointers[1]->state = two;
three->pointers[1]->probability = 1;
three->pointers[2]->state = four;
three->pointers[2]->probability = 1;
three->pointers[3] = NULL;
three->pointers[4] = NULL;
three->pointers[5] = NULL;
three->pointers[6] = NULL;
three->pointers[7] = NULL;

eight->pointers[0]->state = one;
eight->pointers[0]->probability = 5;
eight->pointers[1]->state = seven;
```

```
  eight->pointers[1]->probability = 1;
  eight->pointers[2]->state = nine;
  eight->pointers[2]->probability = 1;
  eight->pointers[3] = NULL;
  eight->pointers[4] = NULL;
  eight->pointers[5] = NULL;
  eight->pointers[6] = NULL;
  eight->pointers[7] = NULL;

  return one;
}

/* move to the next position (given the lead note) */
CircleFifths* CircleFifths_next(CircleFifths* current, int leadnote)
{
  int bestrating = notedifference(leadnote, current->notename)+5;
  int currentrating;
  CircleFifths* best = current;
  int j = 0;

  /* printf("Best rating is %d\n", bestrating); */

  while (current->pointers[j]!=NULL)
    {
      int notediff =
        notedifference(leadnote, current->pointers[j]->state->notename);

      if (notediff==0)
        {
          currentrating = 0;
        }
      else
        {
          currentrating = notediff + current->pointers[j]->probability;
        }

      /* printf("Current rating is %d\n", currentrating); */

      if (currentrating > bestrating)
        {
          bestrating = currentrating;
          best = current->pointers[j]->state;
        }
      else if (currentrating == bestrating)
        {
          float random;

          random = (float)rand()/RAND_MAX;

          if (random < 0.5)
```

```
            {
              bestrating = currentrating;
              best = current->pointers[j]->state;
            }
        }

      j = j+1;
      /* printf("current->pointers[%d] = %p\n", j, current->pointers[j]); */
    }
  return best;
}

/* give a ranking to an note interval */
int notedifference(int i, int j)
{
  int k = i - j;

  if (k < 0)
    {
      k = -k;
    }
  if (k>7)
    {
      k = 12 - k;
    }

  /* printf("Difference is %d\n", k); */

  if (k==0) /* octave or unison */
    {
      return 5;
    }
  else if (k==7) /* perfect 5th */
    {
      return 5;
    }
  else if (k==5) /* perfect 4th */
    {
      return 3;
    }
  else if (k==4) /* major 3rd */
    {
      return 2;
    }
  else if (k==3) /* minor 3rd */
    {
      return 1;
    }
  else
    {
```

```
      return 0;
    }
}
```

C.1.7 Compare.h

```
/*****************************
 * Header file for Compare    *
 * Created 23rd February 2005 *
 * Modified 4th May           *
 * Author: Steve Roberts      *
 *****************************/

#ifndef COMPARE_H
#define COMPARE_H

#include "Chord.h"

/*****************************
 * Define a Compare structure *
 *****************************/

typedef struct compare
{
  int id;
  char* tenorpart;
  char* leadpart;
  char* baripart;
  char* basspart;
  int rating;
  struct compare* next;
  struct compare* previous;
}
Compare;

/*****************************
 * Declare Compare functions *
 *****************************/

/* create a new compare entry given certain parameters */
extern Compare* Compare_new(int id, char* tenor, char* lead, char* bari,
                            char* bass, int rating, Compare* next,
                            Compare* previous);

/* create a new comparison structure */
extern Compare* Compare_newtable(char* tonic, char* lead, char* chordname);

/* assign all of the chords a value */
extern Compare* Compare_assignratings(Chord* chord);

/* find the distance in semitones between two parts */
extern int distanceparts(int previousnote, int currentnote);

/* returns a negative score if there are parallel 5ths */
```

```
extern int parallel(int noteone, int noteoneactual, int notetwo,
                    int notetwoactual);

/* find the best voicing */
extern Compare* Compare_getbest(Compare* table);

#endif
```

C.1.8 Compare.c

```
/*****************************
 * Source file for Compare    *
 * Created 23rd February 2005 *
 * Modified 4th May           *
 * Author: Steve Roberts      *
 *****************************/

#include "Compare.h"
#include "Lookup.h"
#include "AllChords.h"
#include "Chord.h"
#include "Note.h"
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/* create a new compare entry given certain parameters */
Compare* Compare_new(int id, char* tenor, char* lead, char* bari,
                     char* bass, int rating, Compare* next, Compare* previous)
{
  Compare* new;
  if((new = (Compare *)malloc(sizeof(Compare))))
    {
      new -> id = id;
      new -> tenorpart = tenor;
      new -> leadpart = lead;
      new -> baripart = bari;
      new -> basspart = bass;
      new -> rating = rating;
      new -> next = next;

      if (previous)
        {
          new -> previous = previous;
          previous -> next = new;
        }

      return new;
    }
  else
    {
      return NULL;
    }
}

/* create a new comparison structure */
```

```
Compare* Compare_newtable(char* tonic, char* lead, char* chordname)
{
  int leadnote;
  int firstnote;
  int secondnote;
  int thirdnote;
  char* first;
  char* second;
  char* third;
  char* leadthing;
  char* leadchar;
  char lead1[5] = "     ";
  Compare* compareone;
  Compare* comparetwo;
  Compare* comparethree;
  Compare* comparefour;
  Compare* comparefive;
  Compare* comparesix;
  AllChords* chords;
  AllChords* found;
  Lookup* table;

  /* printf("Just about to create our table of all chords\n");
     printf("Tonic is %s, lead is %s, chordname is %s\n", tonic, lead,
     chordname);*/

  /* Get the four numbers from the allchords thingy */
  chords = AllChords_new(); /* global as same is used for everything? */
  found = Chord_lookup(chordname, chords);

  /* printf("Created a table of all of the chords and found
     is our chord\n"); */

  /* Create a lookup starting with tonic */
  table = Lookup_newtable();
  table = Lookup_populate(tonic, table);

  /* printf("Populated our lookup table\n"); */

  /* strip off the octave from the leadnote */
  leadthing = strcpy(lead1, lead);
  leadchar = strtok(lead1, "',");

  /* Create an integer which is the forbidden lead note */
  leadnote = Lookup_byname(leadchar, table);

  /* For each of the others, convert to letters and put in table */

  if (found->firstnote==leadnote)
    {
```

```
    firstnote = found->secondnote;
    secondnote = found->thirdnote;
    thirdnote = found->fourthnote;
  }
else if (found->secondnote==leadnote)
  {
    firstnote = found->firstnote;
    secondnote = found->thirdnote;
    thirdnote = found->fourthnote;
  }
else if (found->thirdnote==leadnote)
  {
    firstnote = found->firstnote;
    secondnote = found->secondnote;
    thirdnote = found->fourthnote;
  }
else if (found->fourthnote==leadnote)
  {
    firstnote = found->firstnote;
    secondnote = found->secondnote;
    thirdnote = found->thirdnote;
  }

first = Lookup_bynumber(firstnote, table);
second = Lookup_bynumber(secondnote, table);
third = Lookup_bynumber(thirdnote, table);

compareone = Compare_new(1, first, lead, second, third, 0,
                         NULL, NULL);
comparetwo = Compare_new(2, first, lead, third, second, 0,
                         NULL, compareone);
comparethree = Compare_new(3, second, lead, first, third, 0,
                           NULL, comparetwo);
comparefour = Compare_new(4, second, lead, third, first, 0,
                          NULL, comparethree);
comparefive = Compare_new(5, third, lead, first, second, 0,
                          NULL, comparefour);
comparesix = Compare_new(6, third, lead, second, first, 0,
                         NULL, comparefive);

compareone->next = comparetwo;
compareone->previous = comparesix;
comparetwo->next = comparethree;
comparethree->next = comparefour;
comparefour->next = comparefive;
comparefive->next = comparesix;
comparesix->next = compareone;

/* printf("Leaving the compare_newtable function\n"); */
```

```
  return compareone;
}

/* assign all of the chords a value */
Compare* Compare_assignratings(Chord* chord)
{
  /* we create the comparison table for the chord */
  Compare* comparison = Compare_newtable(chord->name, chord->lead->pitch,
                                         chord->type);

  /* create a chord object of the previous chord with octaves stripped */
  Chord* previous = chord->previous;

  Lookup* table = Lookup_newtable();
  Lookup* lookup = Lookup_populate(chord->name, table);
  int i;
  int rating;

  /* printf("In the assignratings function\n"); */

  /*
  for (i=0; i<6; i++)
    {
      printf("Tenor note is %s, ", comparison->tenorpart);
      printf("Lead note is %s, ", comparison->leadpart);
      printf("Bari note is %s, ", comparison->baripart);
      printf("Bass note is %s\n", comparison->basspart);
      comparison = comparison->next;
    }
  */

  /* for each entry in the table, we need to do the following: */
  for (i=1; i<7; i++)
    {
      /*** for this lot we just check within the chord ***/
      int tenor = Lookup_byname(comparison->tenorpart, lookup);
      int lead = Lookup_byname(comparison->leadpart, lookup);
      int bari = Lookup_byname(comparison->baripart, lookup);
      int bass = Lookup_byname(comparison->basspart, lookup);
      rating = 0;

      /* printf("Just reset rating to zero...rating is %d\n", rating); */

      /* do a large if statement to check the bass part */
      if (lead==1 && bass==8)
        {
          rating = rating + 10;
        }
      else if (lead==3 && bass==8)
        {
```

```
    rating = rating + 10;
  }
else if (lead==8 && bass==1)
  {
    rating = rating + 10;
  }
else if ((!strcmp(chord->type, "mintriad") ||
          !strcmp(chord->type, "majtriad") ||
          !strcmp(chord->type, "augtriad") ||
          !strcmp(chord->type, "dimtriad")) && bass==1)
  {
    rating = rating + 10;
  }
else if (bass==1 || bass==8)
  {
    rating = rating + 10;
  }
else
  {
    rating = rating;
  }

/* check that parts are not a semitone apart */
if (bass-bari==1 || bari-bass==1)
  {
    rating = rating - 5;
  }

if (lead-bari==1 || bari-lead==1)
  {
    rating = rating - 5;
  }

if (lead-tenor==1 || tenor-lead==1)
  {
    rating = rating - 5;
  }

/*** for this lot we need to check against the previous chord ***/
/*** we skip this section if the previous chord is null ***/
if (previous)
  {
    int tenorprevious = Lookup_byname(previous->tenor->pitch, lookup);
    int leadprevious = Lookup_byname(previous->lead->pitch, lookup);
    int bariprevious = Lookup_byname(previous->bari->pitch, lookup);
    int bassprevious = Lookup_byname(previous->bass->pitch, lookup);
    int distance;

    /* check that the tenor has no more than a 4th */
    distance = distanceparts(tenorprevious, tenor);
```

```
          if (distance<6)
            {
              rating = rating + 10;
            }

          /* give better marks for closer together with previous note
             (for all parts) */
          rating = rating + (7-distance);

          distance = distanceparts(leadprevious, lead);
          rating = rating + (7-distance);

          distance = distanceparts(bariprevious, bari);
          rating = rating + (7-distance);

          distance = distanceparts(bassprevious, bass);
          rating = rating + (7-distance);

          /* checking for parallel octaves and fifths is an expensive check */
          rating = rating + parallel(tenor, tenorprevious, lead, leadprevious);
          rating = rating + parallel(tenor, tenorprevious, bari, bariprevious);
          rating = rating + parallel(tenor, tenorprevious, bass, bassprevious);
          rating = rating + parallel(lead, leadprevious, bari, bariprevious);
          rating = rating + parallel(lead, leadprevious, bass, bassprevious);
          rating = rating + parallel(bari, bariprevious, bass, bassprevious);
        }

      /* put the value into the comparison table */
      comparison->rating = rating;
      comparison = comparison->next;

      /* printf("Rating for comparison %d is %d\n", i, rating); */
    }
  /* we then have got all parts, so we can call the best voicing function */
  comparison = Compare_getbest(comparison);

  return comparison;
}

/* returns the positive distance between two parts */
int distanceparts(int previousnote, int currentnote)
{
  int distance = previousnote-currentnote;

  if (distance<0)
    {
      distance = 0 - distance;
    }
  if (distance>7)
```

```
    {
      distance = 12 - distance;
    }

  return distance;
}

/* returns a negative score if there are parallel 5ths */
int parallel(int noteone, int noteoneactual, int notetwo, int notetwoactual)
{
  int distanceone = distanceparts(noteone, noteoneactual);
  int distancetwo = distanceparts(notetwo, notetwoactual);

  if (distanceone==8 && distancetwo==8)
    {
      return -10;
    }
  else
    {
      return 0;
    }
}

/* find the best voicing...if there is more than one, then choose
   which one randomly */
Compare* Compare_getbest(Compare* table)
{
  Compare* best = table;
  Compare* bestid;
  int bestvalue = 0;
  int i;

  srand((unsigned)time(NULL));

  for (i=1; i<7; i++)
    {
      if (best->rating > bestvalue)
        {
          bestvalue = best->rating;
          bestid = best;
          best = best->next;
        }
      else if (best->rating == bestvalue)
        {
          float random;

          random = (float)rand()/RAND_MAX;

          /* printf("Random is %f\n", random); */
```

```
          if (random < 0.5)
            {
              bestvalue = best->rating;
              bestid = best;
              best = best->next;
            }
          else
            {
              best = best->next;
            }
        }
      else
        {
          best = best->next;
        }
    }

  /* printf("Best id is %d\n", bestid->id); */

  return bestid;
}
```

C.1.9 Lookup.h

```
/*****************************
 * Header file for Lookup     *
 * Created 22nd February 2005 *
 * Modified 23rd February     *
 * Author: Steve Roberts      *
 *****************************/

#ifndef LOOKUP_H
#define LOOKUP_H

/*****************************
 * Define a Lookup structure *
 *****************************/

typedef struct lookup
{
  int note;
  char* name;
  struct lookup* next;
}
Lookup;

/*****************************
 * Declare Lookup functions *
 *****************************/

/* create a new lookup entry given the chord name and number */
extern Lookup* Lookup_new(char* name, int number, Lookup* next);

/* create a new lookup structure */
extern Lookup* Lookup_newtable(void);

/* search for a note name given the number */
extern char* Lookup_bynumber(int number, Lookup* table);

/* search for a note number given the name */
extern int Lookup_byname(char* name, Lookup* table);

/* re-populate the lookup structure */
extern Lookup* Lookup_populate(char* name, Lookup* table);

#endif
```

C.1.10 Lookup.c

```
/*****************************
 * Source file for Lookup     *
 * Created 22nd February 2005 *
 * Modified 4th May           *
 * Author: Steve Roberts      *
 *****************************/

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Lookup.h"

/* create a new lookup entry given the chord name and number */
Lookup* Lookup_onenew(char* name, int number, Lookup* next)
{
  Lookup* new;
  if((new = (Lookup *)malloc(sizeof(Lookup))))
    {
      new -> note = number;
      new -> name = name;
      new -> next = next;

      return new;
    }
  else
    {
      return NULL;
    }
}

/* create a new lookup structure */
Lookup* Lookup_newtable(void)
{
  Lookup* one = Lookup_onenew("c", 0, NULL);
  Lookup* twelve = Lookup_onenew("b", 0, one);
  Lookup* eleven = Lookup_onenew("bes", 0, twelve);
  Lookup* ten = Lookup_onenew("a", 0, eleven);
  Lookup* nine = Lookup_onenew("aes", 0, ten);
  Lookup* eight = Lookup_onenew("g", 0, nine);
  Lookup* seven = Lookup_onenew("fis", 0, eight);
  Lookup* six = Lookup_onenew("f", 0, seven);
  Lookup* five = Lookup_onenew("e", 0, six);
  Lookup* four = Lookup_onenew("ees", 0, five);
  Lookup* three = Lookup_onenew("d", 0, four);
  Lookup* two = Lookup_onenew("cis", 0, three);
  one -> next = two;
```

```
  /* printf("LOOKUP: Lookup table has been populated\n"); */

  return one;
}

/* search for a note name given the number */
char* Lookup_bynumber(int number, Lookup* table)
{
  Lookup* found = table;
  int counter = 0;

  while (found->note != number && counter<13)
    {
      /* printf("Found->note is %d and one we're looking for is %d\n",
         found->note, number); */
      found = found->next;
      counter = counter+1;
    }

  if (found->note==number)
    {
      return found->name;
    }
  else
    {
      return NULL;
    }
}

/* search for a note number given the name */
int Lookup_byname(char* name, Lookup* table)
{
  Lookup* found = table;
  int counter = 0;

  while (strcmp(found->name, name) && counter<13)
    {
      /* printf("Found->name is %s and one we're looking for is %s\n",
         found->name, name); */
      found = found->next;
      counter = counter+1;
    }

  if (!strcmp(found->name, name))
    {
      return found->note;
    }
  else
    {
      return 0;
```

```
    }
}

/* populate the lookup structure */
Lookup* Lookup_populate(char* name, Lookup* table)
{
  Lookup* populated = table;
  int i;
  /* printf("LOOKUP: Name is %s and name looking for is %s\n",
     populated->name, name); */

  while(strcmp(populated->name, name))
    {
      /* printf("Moving to position to populate table\n");
         printf("Name is %s and name looking for is %s\n",
         populated->name, name); */
      populated = populated->next;
    }

  for (i=1; i<13; i++)
    {
      /* printf("Populating %s with %d\n", populated->name, i); */
      populated->note = i;
      populated = populated->next;
    }

  return populated;
}
```

C.1.11 Note.h

```
/*******************************
 * Header file for Note        *
 * Created 14th February 2005 *
 * Modified 20th April         *
 * Author: Steve Roberts       *
 *******************************/

#ifndef NOTE_H
#define NOTE_H

/***************************
 * Define a Note structure *
 ***************************/

typedef struct note
{
  char* pitch; /* the pitch of the note will be a string (or char*) */
  int duration; /* the duration of the note will be an integer */
  struct note *next; /* each note will have a pointer to the next note */
  struct note *previous; /* each note has a pointer to the previous note */
}
Note;

/***************************
 * Declare Note functions *
 ***************************/

/* Create a new note using provided data */
extern Note* Note_new(char* pitch, int duration, Note* next, Note* previous);

#endif
```

C.1.12 Note.c

```
/*****************************
 * Source file for Note      *
 * Created 14th February 2005 *
 * Modified 20th April       *
 * Author: Steve Roberts     *
 *****************************/

#include "Note.h"
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

/* Create a new note using provided data */
Note* Note_new(char* pitch, int duration, Note* next, Note* previous)
{
  Note* new;
  if((new = (Note *)malloc(sizeof(Note))))
    {
      new->pitch = pitch;
      new->duration = duration;
      new->next = next;

      /* set next and previous Notes to point to this Note */
      if (next)
        {
          next->previous = new;
        }

      new->previous = previous;
      if (previous)
        {
          previous->next = new;
        }

      return new;
    }
  else
    {
      return NULL; /* malloc error */
    }
}
```

C.1.13 Pitches.h

```
/*******************************
 * Header file for Pitches     *
 * Created 20th April 2005     *
 * Modified 20th April         *
 * Author: Steve Roberts       *
 *******************************/

#ifndef PITCHES_H
#define PITCHES_H

/*******************************
 * Define a Pitches structure *
 *******************************/

typedef struct pitches
{
  float hz;
  char* name;
  struct pitches* next;
}
Pitches;

/*****************************
 * Declare Pitches functions *
 *****************************/

/* create a new Pitches entry given the chord name and number */
extern Pitches* Pitches_new(char* name, float hz, Pitches* next);

/* create a new Pitches structure */
extern Pitches* Pitches_newtable(void);

/* search for a note name given the number */
extern float Pitches_lookup(char* name, Pitches* table);

#endif
```

C.1.14 Pitches.c

```
/*****************************
 * Score file for Pitches     *
 * Created 20th April 2005    *
 * Modified 20th April        *
 * Author: Steve Roberts      *
 *****************************/

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Pitches.h"

/*****************************
 * Declare Pitches functions *
 *****************************/

/* create a new Pitches entry given the chord name and number */
Pitches* Pitches_new(char* name, float hz, Pitches* next)
{
  Pitches* new;
  if((new = (Pitches *)malloc(sizeof(Pitches))))
    {
      new -> hz = hz;
      new -> name = name;
      new -> next = next;

      return new;
    }
  else
    {
      return NULL;
    }
}

/* create a new Pitches structure */
Pitches* Pitches_newtable(void)
{
  Pitches* one = Pitches_new("fis,", 92.50, NULL);
  Pitches* twentynine = Pitches_new("bes'", 466.16, one);
  Pitches* twentyeight = Pitches_new("a'", 440.00, twentynine);
  Pitches* twentyseven = Pitches_new("aes'", 415.30, twentyeight);
  Pitches* twentysix = Pitches_new("g'", 391.99, twentyseven);
  Pitches* twentyfive = Pitches_new("fis'", 369.99, twentysix);
  Pitches* twentyfour = Pitches_new("f'", 349.23, twentyfive);
  Pitches* twentythree = Pitches_new("e'", 329.63, twentyfour);
  Pitches* twentytwo = Pitches_new("ees'", 311.13, twentythree);
  Pitches* twentyone = Pitches_new("d'", 293.66, twentytwo);
```

```
  Pitches* twenty = Pitches_new("cis'", 277.18, twentyone);
  Pitches* nineteen = Pitches_new("c'", 261.63, twenty);
  Pitches* eighteen = Pitches_new("b", 246.94, nineteen);
  Pitches* seventeen = Pitches_new("bes", 233.08, eighteen);
  Pitches* sixteen = Pitches_new("a", 220.00, seventeen);
  Pitches* fifteen = Pitches_new("aes", 207.65, sixteen);
  Pitches* fourteen = Pitches_new("g", 196.00, fifteen);
  Pitches* thirteen = Pitches_new("fis", 184.99, fourteen);
  Pitches* twelve = Pitches_new("f", 174.61, thirteen);
  Pitches* eleven = Pitches_new("e", 164.81, twelve);
  Pitches* ten = Pitches_new("ees", 155.56, eleven);
  Pitches* nine = Pitches_new("d", 146.83, ten);
  Pitches* eight = Pitches_new("cis", 138.59, nine);
  Pitches* seven = Pitches_new("c", 130.81, eight);
  Pitches* six = Pitches_new("b,", 123.47, seven);
  Pitches* five = Pitches_new("bes,", 116.54, six);
  Pitches* four = Pitches_new("a,", 110, five);
  Pitches* three = Pitches_new("aes,", 103.83, four);
  Pitches* two = Pitches_new("g,", 98.00, three);

  one -> next = two;

  return one;
}

/* search for a note name given the number */
float Pitches_lookup(char* name, Pitches* table)
{
  Pitches* found = table;

  /* printf("Searching for %s\n", name); */

  while(strcmp(found->name, name))
    {
      found=found->next;
    }
  return found->hz;
}
```

C.1.15 Ratios.h

```
/*******************************
 * Header file for Ratios      *
 * Created 20th April 2005     *
 * Modified 4th May            *
 * Author: Steve Roberts       *
 *******************************/

#ifndef RATIOS_H
#define RATIOS_H

/*******************************
 * Define a Ratios structure   *
 *******************************/

typedef struct ratios
{
  float ratio;
  int position;
  struct ratios* next;
}
Ratios;

/*******************************
 * Declare Ratios functions    *
 *******************************/

/* create a new Ratios entry given the ratio and number */
extern Ratios* Ratios_new(int position, float ratio, Ratios* next);

/* create a new Ratios structure */
extern Ratios* Ratios_newtable(void);

/* search for a ratio given the number */
extern float Ratios_lookup(int number, Ratios* table);

/* calculate the frequency of a part */
extern float Ratios_calculate(float leadfreq, int difference, char* leadorig,
                              char* noteorig);

#endif
```

C.1.16 Ratios.c

```
/*****************************
 * Source file for Ratios     *
 * Created 20th April 2005    *
 * Modified 4th May           *
 * Author: Steve Roberts      *
 *****************************/

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Ratios.h"

/* create a new Ratios entry given the ratio and number */
Ratios* Ratios_new(int position, float ratio, Ratios* next)
{
  Ratios* new;
  if((new = (Ratios *)malloc(sizeof(Ratios))))
    {
      new -> position = position;
      new -> ratio = ratio;
      new -> next = next;

      return new;
    }
  else
    {
      return NULL;
    }
}

/* create a new Ratios structure */
Ratios* Ratios_newtable(void)
{
  float temp1 = (float)1/1;
  Ratios* one = Ratios_new(0, temp1, NULL);
  float temp2 = (float)15/8;
  Ratios* twelve = Ratios_new(11, temp2, one);
  float temp3 = (float)9/5;
  Ratios* eleven = Ratios_new(10, temp3, twelve);
  float temp4 = (float)27/16;
  Ratios* ten = Ratios_new(9, temp4, eleven);
  float temp5 = (float)8/5;
  Ratios* nine = Ratios_new(8, temp5, ten);
  float temp6 = (float)3/2;
  Ratios* eight = Ratios_new(7, temp6, nine);
  float temp7 = (float)45/32;
  Ratios* seven = Ratios_new(6, temp7, eight);
```

```
  float temp8 = (float)4/3;
  Ratios* six = Ratios_new(5, temp8, seven);
  float temp9 = (float)5/4;
  Ratios* five = Ratios_new(4, temp9, six);
  float temp10 = (float)6/5;
  Ratios* four = Ratios_new(3, temp10, five);
  float temp11 = (float)9/8;
  Ratios* three = Ratios_new(2, temp11, four);
  float temp12 = (float)135/128;
  Ratios* two = Ratios_new(1, temp12, three);

  one->next = two;

  return one;
}

/* search for a ratio given the number */
float Ratios_lookup(int number, Ratios* table)
{
  Ratios* found = table;

  while(number!=found->position)
    {
      found=found->next;
    }

  return found->ratio;
}

/* calculate the frequency of a part */
float Ratios_calculate(float leadfreq, int difference, char* leadorig,
                       char* noteorig)
{
  Ratios* table = Ratios_newtable();
  float frequency;
  float ratio;
  char comma = ',';
  char apostrophe = '\'';

  if (difference < 0)
    {
      difference = 0 - difference;
      ratio = Ratios_lookup(difference, table);
      frequency = leadfreq * ratio;
    }
  else if (difference > 0)
    {
      difference = 12 - difference;
      ratio = Ratios_lookup(difference, table);
      frequency = leadfreq * ratio;
```

```
      frequency = frequency/2;
    }
  else
    {
      frequency = leadfreq;
    }

  if (strchr(leadorig, apostrophe)!=NULL)
    {
      if (strchr(noteorig, apostrophe)!=NULL)
        {
          frequency = frequency;
        }
      else if (strchr(noteorig, comma)!=NULL)
        {
          frequency = frequency/4;
        }
      else
        {
          frequency = frequency/2;
        }
    }
  else
    {
      if (strchr(noteorig, apostrophe)!=NULL)
        {
          frequency = frequency * 2;
        }
      else if (strchr(noteorig, comma)!=NULL)
        {
          frequency = frequency/2;
        }
      else
        {
          frequency = frequency;
        }
    }

  return frequency;
}
```

C.1.17 Score.h

```
/*******************************
 * Header file for Score       *
 * Created 16th February 2005 *
 * Modified 4th May            *
 * Author: Steve Roberts       *
 *******************************/

#ifndef SCORE_H
#define SCORE_H

#include "Chord.h"

/*****************************
 * Define a Score structure *
 *****************************/

typedef struct score
{
  Chord* firstchord; /* pointer to the first chord */
  char* timesig; /* time signature will be in the form "4/4" or
                    alternative values */
  char* keysig; /* key signature will be in the form "c \minor" or
                   alternative values */
  int harmonrhythm; /* this will be how often the chord changes */
}
Score;

/*************************
 * Declare Score functions *
 *************************/

/* Create a new Score structure given the parameters */
extern Score* Score_new(Chord* firstchord, char* timesig, char* keysig,
                        int harmonrhythm);

/* Print out the score */
extern void Score_print(Score* score);

/* Print out the csound file */
extern void Score_csound(Score* score);

/* Fill in the primary harmony chords */
extern Score* Score_harmonise(Score* score);

/* Fill in the voicings of the chords chords */
extern Score* Score_voice(Score* score);

/* Set the correct octaves */
```

```
extern Score* Score_octaves(Score* score);

#endif
```

C.1.18 Score.c

```
/*****************************
 * Source file for Score      *
 * Created 16th February 2005 *
 * Modified 4th May           *
 * Author: Steve Roberts      *
 *****************************/

#include "Chord.h"
#include "Score.h"
#include "CircleFifths.h"
#include "Lookup.h"
#include "AllChords.h"
#include "Compare.h"
#include "Pitches.h"
#include "Ratios.h"
#include "csound.h"
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Create a new Score structure given the parameters */
Score* Score_new(Chord* firstchord, char* timesig, char* keysig,
                 int harmonrhythm)
{
  Score* new;
  if((new = (Score *)malloc(sizeof(Score))))
    {
      /* set pointers and values within this chord */
      new->firstchord = firstchord;
      new->keysig = keysig;
      new->timesig = timesig;
      new->harmonrhythm = harmonrhythm;

      return new;
    }
  else
    {
      return NULL; /* malloc error */
    }
}

/* Print out the score */
void Score_print(Score* score)
{
  Chord* chord = score->firstchord;

  printf("\\version \"2.4.2\"\n");
```

```
printf("global = {\n");
printf("\\key %s\n", score->keysig);
printf("\\time %s\n", score->timesig);
printf("}\n");

printf("tenorMusic = {\n");

do
  {
    printf("%s%d ", chord->tenor->pitch, chord->tenor->duration);
    chord = chord->next;
  }
while (chord!=NULL);
printf("}\n");

chord = score->firstchord;
printf("leadMusic = {\n");
do
  {
    printf("%s%d ", chord->lead->pitch, chord->lead->duration);
    chord = chord->next;
  }
while (chord!=NULL);
printf("}\n");

chord = score->firstchord;
printf("bariMusic = {\n");
do
  {
    printf("%s%d ", chord->bari->pitch, chord->bari->duration);
    chord = chord->next;
  }
while (chord!=NULL);
printf("}\n");

chord = score->firstchord;
printf("bassMusic = {\n");
do
  {
    printf("%s%d ", chord->bass->pitch, chord->bass->duration);
    chord = chord->next;
  }
while (chord!=NULL);
printf("}\n");

printf("\\score {\n");
printf("\\context ChoirStaff <<\n");

printf("\\context Staff = tops <<\n");
printf("\\clef \"G_8\"\n");
```

```
  printf("\\context Voice = tenors { \\voiceOne << \\global
          \\tenorMusic >> }\n");
  printf("\\context Voice = leads { \\voiceTwo << \\global
          \\leadMusic >> }\n");
  printf(">>\n");

  printf("\\context Staff = bottoms <<\n");
  printf("\\clef bass\n");
  printf("\\context Voice = baritones { \\voiceOne << \\global
          \\bariMusic >> }\n");
  printf("\\context Voice = basses { \\voiceTwo << \\global
          \\bassMusic >> }\n");
  printf(">>\n");
  printf(">>\n");
  printf("\\layout {\n");
  printf("}\n");
  printf("}\n");
}

Score* Score_harmonise(Score* score)
{
  float harmonrhythm;
  float actualrhythm;
  float firstleadnote;
  CircleFifths* circle = CircleFifths_new();
  Chord* current;

  /* set a char* to be the parts of the key sig before the space */
  static char key[5] = "     ";
  char* keything = strcpy(key, score->keysig);
  char* keysig = strtok(key," ");

  /* create a lookup table for the whole score to find the primary harmony */
  Lookup* table = Lookup_newtable();

  /* printf("SCORE: Just copied string across and created lookup table\n"); */

  table = Lookup_populate(keysig, table);

  /* work out if we need a chord here */
  harmonrhythm = 4;

  /* printf("SCORE: Harmon rhythm is %f\n", harmonrhythm); */

  /* give first note a chord */
  score->firstchord->name = keysig;

  /* printf("SCORE: firstchord name is %s\n", score->firstchord->name); */

  firstleadnote = score->firstchord->lead->duration;
```

```
actualrhythm = 1/firstleadnote;

/* printf("SCORE: Actual rhythm is %f\n", actualrhythm); */

current = score->firstchord->next;

while (current)
  {
    float currentlead = current->lead->duration;

    actualrhythm = actualrhythm + 1/currentlead;

    /* printf("SCORE: Actual rhythm is %f\n", actualrhythm); */

    /* if actualrhythm==harmonrhythm, we put a chord in */
    if (actualrhythm==harmonrhythm || actualrhythm == 2*harmonrhythm)
      {
        if (strcmp(current->lead->pitch, "r")!=0)
          {
            /* strip off octaves for calculations */
            static char lead[5] = "     ";
            char* leadthing = strcpy(lead, current->lead->pitch);
            char* leadchar = strtok(leadthing, "',");
            int leadno = Lookup_byname(leadchar, table);

            circle = CircleFifths_next(circle, leadno);
            current->name = Lookup_bynumber(circle->notename, table);

            /* printf("SCORE: Filled in chord name as %s\n",
               current->name); */

            actualrhythm = 0;
            current = current->next;
          }
      }
    else
      {
        current = current->next;
      }
  }

/* fill in the rest of the chords */
current = score->firstchord;

while (current)
  {
    if (current->name==NULL)
      {
        /* printf("In the if statement\n"); */
        current->name=current->previous->name;
```

```
        /* printf("SCORE: Filled in chord name as %s\n", current->name); */
      }

    /* printf("SCORE: Current->name is %s\n", current->name); */

    current = current->next;
  }

/* traverse the score and fill in what type of chord it is */
current = score->firstchord;

current->type = "majtriad";
current = current->next;

while (current)
  {
    /* for each chord, get the chord name */
    char* name = current->name;
    char* chordtype;
    AllChords* chords = AllChords_new();

    /* create a lookup table */
    Lookup* lookup = Lookup_newtable();
    lookup = Lookup_populate(name, lookup);

    /* printf("SCORE: Created a lookup table\n"); */

    /* get the lead note */
    if (strcmp(current->lead->pitch,"r"))
      {
        static char lead[5] = "     ";
        char* leadthing = strcpy(lead, current->lead->pitch);
        char* leadchar = strtok(lead, "',");

        int leadno = Lookup_byname(leadchar, lookup);

        /*  printf("SCORE: Current lead pitch is %s\n",
            current->lead->pitch);
            printf("SCORE: Got out the lead note, which is %d\n", leadno);
            printf("SCORE: Our current tonic is %s\n", current->name); */

        /* call the AllChords search to return the string of chord type */
        chordtype = Chord_search(leadno, chords);
        /* printf("SCORE: Our chord type is %s\n", chordtype); */
      }

    /* put that in the score */
    current->type = chordtype;

    /* printf("SCORE: Set the chord type to be %s\n", current->type); */
```

```
      current = current->next;
    }

  current = score->firstchord;

  while (current)
    {
      /* printf("SCORE: Chord name is %s %s\n", current->name,
         current->type); */
      current = current->next;
    }

  return score;
}

Score* Score_voice(Score* score)
{
  Chord* current = score->firstchord;
  Compare* compare;

  /* printf("SCORE: Created our comparison table\n");
     printf("SCORE: First chord name is %s\n", score->firstchord->name); */


  while (current && strcmp(current->lead->pitch, "r"))
    {
      compare = Compare_assignratings(current);
      current->tenor->pitch = compare->tenorpart;
      current->bari->pitch = compare->baripart;
      current->bass->pitch = compare->basspart;

      current = current->next;
    }

  return score;
}

Score* Score_octaves(Score* score)
{
  Chord* current = score->firstchord;

  /* printf("SCORE: In the octaves function\n"); */

  while (current)
    {
      /* set the tenor octaves */

      if (strcmp(current->tenor->pitch, "b")!=0  &&
          strcmp(current->tenor->pitch, "r")!=0)
```

```
  {
    static char* tenortemp[50];
    static int position = 0;
    tenortemp[position] = strdup(current->tenor->pitch);
    strcat(tenortemp[position], "'");
    current->tenor->pitch = tenortemp[position];
    position = position+1;

    /* printf("SCORE: Current tenor pitch is %s\n",
       current->tenor->pitch); */
  }

/* set the bari octave to keep as high as possible */
if (strcmp(current->bari->pitch, "c")==0
    || strcmp(current->bari->pitch, "cis")==0
    || strcmp(current->bari->pitch, "d")==0
    || strcmp(current->bari->pitch, "ees")==0
    || strcmp(current->bari->pitch, "e")==0)
  {
    static char* baritemp[50];
    static int position2 = 0;
    baritemp[position2] = strdup(current->bari->pitch);
    strcat(baritemp[position2], "'");
    current->bari->pitch = baritemp[position2];
    position2 = position2+1;

    /* printf("SCORE: Current bari pitch is %s\n",
       current->bari->pitch); */
  }

/* set the bass octave */
if (strcmp(current->bass->pitch, "fis")==0
    || strcmp(current->bass->pitch, "g")==0
    || strcmp(current->bass->pitch, "aes")==0
    || strcmp(current->bass->pitch, "a")==0
    || strcmp(current->bass->pitch, "bes")==0
    || strcmp(current->bass->pitch, "b")==0)
  {
    float random;

    random = (float)rand()/RAND_MAX;

    if (random<0.5)
      {
        static char* basstemp[50];
        static int position3 = 0;
        basstemp[position3] = strdup(current->bass->pitch);
        strcat(basstemp[position3], ",");
        current->bass->pitch = basstemp[position3];
        position3 = position3+1;
```

```
            /* printf("SCORE: Current bass pitch is %s\n",
               current->bass->pitch); */
          }
        }

      current = current->next;
    }



  return score;
}

void Score_csound(Score* score)
{
  Chord* current = score->firstchord;
  float distance = 0;
  float duration;
  float leadfreq;
  float tenorfreq;
  float barifreq;
  float bassfreq;
  int tenordif;
  int baridif;
  int bassdif;

  Pitches* pitches = Pitches_newtable();
  Ratios* ratios = Ratios_newtable();

  Lookup* lookup = Lookup_newtable();
  lookup = Lookup_populate("c", lookup);

  printf("%s", ftables);

  while (current)
    {
      if (strcmp(current->lead->pitch, "r"))
        {
          /* strip off the octaves */
          static char lead[5] = "     ";
          char* leadthing = strcpy(lead, current->lead->pitch);
          char* leadchar = strtok(leadthing, "',");
          int leadno = Lookup_byname(leadchar, lookup);

          static char tenor[5] = "     ";
          char* tenorthing = strcpy(tenor, current->tenor->pitch);
          char* tenorchar = strtok(tenorthing, "',");
          int tenorno = Lookup_byname(tenorchar, lookup);
```

```
static char bari[5] = "     ";
char* barithing = strcpy(bari, current->bari->pitch);
char* barichar = strtok(barithing, "',");
int barino = Lookup_byname(barichar, lookup);

static char bass[5] = "     ";
char* bassthing = strcpy(bass, current->bass->pitch);
char* basschar = strtok(bassthing, "',");
int bassno = Lookup_byname(basschar, lookup);

leadfreq = Pitches_lookup(current->lead->pitch, pitches);

tenordif = leadno - tenorno;
baridif = leadno - barino;
bassdif = leadno - bassno;

tenorfreq = Ratios_calculate(leadfreq, tenordif,
                             current->lead->pitch,
                             current->tenor->pitch);
barifreq = Ratios_calculate(leadfreq, baridif,
                            current->lead->pitch,
                            current->bari->pitch);
bassfreq = Ratios_calculate(leadfreq, bassdif,
                            current->lead->pitch,
                            current->bass->pitch);

/* calculate the duration */
if (current->lead->duration==1)
  {
    duration = 4.0;
  }
else if (current->lead->duration==2)
  {
    duration = 2.0;
  }
else if (current->lead->duration==4)
  {
    duration = 1.0;
  }
else if (current->lead->duration==8)
  {
    duration = 0.5;
  }
else if (current->lead->duration==16)
  {
    duration = 0.25;
  }

/* print out the notes */
```

```
        printf("i1   %f   %f   3000  %f\n", distance, duration, tenorfreq);
        printf("i1   %f   %f   3000  %f\n", distance, duration, leadfreq);
        printf("i1   %f   %f   3000  %f\n", distance, duration, barifreq);
        printf("i1   %f   %f   3000  %f\n", distance, duration, bassfreq);

        distance = distance+duration;
      }
    else
      {
        if (current->lead->duration==1)
          {
            duration = 4.0;
          }
        else if (current->lead->duration==2)
          {
            duration = 2.0;
          }
        else if (current->lead->duration==4)
          {
            duration = 1.0;
          }
        else if (current->lead->duration==8)
          {
            duration = 0.5;
          }
        else if (current->lead->duration==16)
          {
            duration = 0.25;
          }

        distance = distance+duration;
      }

    current = current->next;
  }

printf("e\n");
}
```

C.2 Test code

All of the code in this appendix was used to test the libraries. The voicing outputs were produced using `CompareTest2.c` and the full arrangement outputs were produced using `ScoreTest2.c`.

C.2.1 AllChordsTest.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "AllChords.h"

int main()
{
  AllChords* table = AllChords_new();
  AllChords* chord = Chord_lookup("min7th", table);
  int first = chord->firstnote;
  int second = chord->secondnote;
  int third = chord->thirdnote;
  int fourth = chord->fourthnote;

  printf("An minor 7th is made up of %d, %d, %d, %d\n", first, second,
         third, fourth);

  return 0;
}
```

C.2.2 CircleFifthsTest.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "CircleFifths.h"

int main()
{
  CircleFifths* circle = CircleFifths_new();

  int i;

  for (i = 0; i<12; i++)
    {
      int best;
      int j=0;
      CircleFifths* next = circle->pointers[j]->state;
      best = circle->pointers[j]->probability;

      printf("In node %d\n", circle->notename);

      while (circle->pointers[j])
        {
          if (circle->pointers[j]->probability > best)
            {
              best = circle->pointers[j]->probability;
              next = circle->pointers[j]->state;
            }
          j = j+1;
        }

      circle = next;
    }

  return 0;
}
```

C.2.3 CircleFifthsTest2.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "CircleFifths.h"

int main()
{
  CircleFifths* circle = CircleFifths_new();

  printf("First chord is tonic, first note is 5\n");

  /* feed it lead notes and go round circle of fifths */
  circle = CircleFifths_next(circle, 12);
  printf("Circle->notename is %d, lead note is 12\n", circle->notename);

  circle = CircleFifths_next(circle, 10);
  printf("Circle->notename is %d, lead note is 10\n", circle->notename);

  circle = CircleFifths_next(circle, 8);
  printf("Circle->notename is %d, lead note is 8\n", circle->notename);

  circle = CircleFifths_next(circle, 10);
  printf("Circle->notename is %d, lead note is 10\n", circle->notename);

  /*
  circle = CircleFifths_next(circle, 5);
  printf("Circle->notename is %d, lead note is 5\n", circle->notename);
  */

  circle = CircleFifths_next(circle, 3);
  printf("Circle->notename is %d, lead note is 3\n", circle->notename);

  circle = CircleFifths_next(circle, 10);
  printf("Circle->notename is %d, lead note is 10\n", circle->notename);
  return 0;
}
```

C.2.4 CircleFifthsTest3.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "CircleFifths.h"

int main()
{
  CircleFifths* circle = CircleFifths_new();

  printf("First chord is tonic, first note is 1\n");

  /* feed it lead notes and go round circle of fifths */
  circle = CircleFifths_next(circle, 12);
  printf("Circle->notename is %d, lead note is 12\n", circle->notename);

  circle = CircleFifths_next(circle, 1);
  printf("Circle->notename is %d, lead note is 1\n", circle->notename);

  circle = CircleFifths_next(circle, 10);
  printf("Circle->notename is %d, lead note is 10\n", circle->notename);

  circle = CircleFifths_next(circle, 8);
  printf("Circle->notename is %d, lead note is 8\n", circle->notename);

  circle = CircleFifths_next(circle, 8);
  printf("Circle->notename is %d, lead note is 8\n", circle->notename);

  return 0;
}
```

C.2.5 CompareTest.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "Compare.h"

int main()
{
  Compare* compare = Compare_newtable("f", "ees", "dom7th");
  int i;

  for (i=0; i<6; i++)
    {
      printf("Tenor note is %s, ", compare->tenorpart);
      printf("Lead note is %s, ", compare->leadpart);
      printf("Bari note is %s, ", compare->baripart);
      printf("Bass note is %s\n", compare->basspart);
      compare = compare->next;
    }

  return 0;
}
```

C.2.6 CompareTest2.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "Compare.h"
#include "Score.h"
#include "Chord.h"

int main()
{
  Chord* chord = Chord_new("d", 2, "bes", NULL, NULL);
  Score* score = Score_new(chord, "4/4", "a \\major", 2);
  Compare* compare;

  chord->type = "majtriad";
  chord->next = Chord_new("a", 2, "d", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("g", 4, "g", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("f", 2, "g", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("g", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("d", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "9th";
  chord->next = Chord_new("c", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("g", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";

  chord = score->firstchord;

  while (chord)
    {
      compare = Compare_assignratings(chord);
      chord->tenor->pitch = compare->tenorpart;
      printf("Chord tenor is %s, compare tenor is %s\n",
             chord->tenor->pitch, compare->tenorpart);
      chord->bari->pitch = compare->baripart;
      printf("Chord bari is %s, compare bari is %s\n",
             chord->bari->pitch, compare->baripart);
      chord->bass->pitch = compare->basspart;
```

```
      printf("Chord bass is %s, compare bass is %s\n",
             chord->bass->pitch, compare->basspart);

      chord = chord->next;
    }

  printf("Score->firstchord tenor is %s\n", score->firstchord->tenor->pitch);
  printf("Score->firstchord lead is %s\n", score->firstchord->lead->pitch);
  printf("Score->firstchord bari is %s\n", score->firstchord->bari->pitch);
  printf("Score->firstchord bass is %s\n", score->firstchord->bass->pitch);

  score = Score_octaves(score);
  Score_print(score);
  printf("************************************************************\n");
  Score_csound(score);
  return 0;
}
```

C.2.7 CompareTest3.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "Compare.h"
#include "Score.h"
#include "Chord.h"

int main()
{
  Chord* chord = Chord_new("c", 2, "c", NULL, NULL);
  Score* score = Score_new(chord, "3/4", "c \\major", 2);
  Compare* compare;

  chord->type = "majtriad";
  chord->next = Chord_new("b", 2, "g", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("c", 4, "c", chord, NULL);
  chord = chord->next;
  chord->type = "majtriad";
  chord->next = Chord_new("a", 2, "f", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("c", 2, "f", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("g", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "majtriad";
  chord->next = Chord_new("e", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "majtriad";
  chord->next = Chord_new("g", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";

  chord = score->firstchord;

  while (chord)
    {
      compare = Compare_assignratings(chord);
      chord->tenor->pitch = compare->tenorpart;
      printf("Chord tenor is %s, compare tenor is %s\n",
             chord->tenor->pitch, compare->tenorpart);
      chord->bari->pitch = compare->baripart;
      printf("Chord bari is %s, compare bari is %s\n",
             chord->bari->pitch, compare->baripart);
      chord->bass->pitch = compare->basspart;
```

```
      printf("Chord bass is %s, compare bass is %s\n",
             chord->bass->pitch, compare->basspart);

      chord = chord->next;
    }

  printf("Score->firstchord tenor is %s\n", score->firstchord->tenor->pitch);
  printf("Score->firstchord lead is %s\n", score->firstchord->lead->pitch);
  printf("Score->firstchord bari is %s\n", score->firstchord->bari->pitch);
  printf("Score->firstchord bass is %s\n", score->firstchord->bass->pitch);

  score = Score_octaves(score);
  Score_print(score);
  return 0;
}
```

C.2.8 CompareTest4.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "Compare.h"
#include "Score.h"
#include "Chord.h"

int main()
{
  Chord* chord = Chord_new("e", 2, "c", NULL, NULL);
  Score* score = Score_new(chord, "2/4", "c \\major", 2);
  Compare* compare;

  chord->type = "majtriad";
  chord->next = Chord_new("g", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "majtriad";
  chord->next = Chord_new("c", 4, "c", chord, NULL);
  chord = chord->next;
  chord->type = "majtriad";
  chord->next = Chord_new("g", 2, "g", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("c", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "majtriad";
  chord->next = Chord_new("c", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "majtriad";
  chord->next = Chord_new("e", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("d", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "9th";
  chord->next = Chord_new("c", 2, "f", chord, NULL);
  chord = chord->next;
  chord->type = "majtriad";
  chord->next = Chord_new("a", 2, "a", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("a", 2, "d", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("a", 2, "d", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("d", 2, "d", chord, NULL);
```

```
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("a", 2, "d", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("d", 2, "g", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("d", 2, "g", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("b", 2, "g", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("g", 2, "g", chord, NULL);
  chord = chord->next;
  chord->type = "dom7th";
  chord->next = Chord_new("e", 2, "c", chord, NULL);
  chord = chord->next;
  chord->type = "majtriad";


  chord = score->firstchord;

  while (chord)
    {
      compare = Compare_assignratings(chord);
      chord->tenor->pitch = compare->tenorpart;
      printf("Chord tenor is %s, compare tenor is %s\n",
             chord->tenor->pitch, compare->tenorpart);
      chord->bari->pitch = compare->baripart;
      printf("Chord bari is %s, compare bari is %s\n",
             chord->bari->pitch, compare->baripart);
      chord->bass->pitch = compare->basspart;
      printf("Chord bass is %s, compare bass is %s\n",
             chord->bass->pitch, compare->basspart);

      chord = chord->next;
    }

  printf("Score->firstchord tenor is %s\n", score->firstchord->tenor->pitch);
  printf("Score->firstchord lead is %s\n", score->firstchord->lead->pitch);
  printf("Score->firstchord bari is %s\n", score->firstchord->bari->pitch);
  printf("Score->firstchord bass is %s\n", score->firstchord->bass->pitch);

  score = Score_octaves(score);
  Score_print(score);
  return 0;
}
```

C.2.9 csound.h

```
char* ftables = "

f1 0 4096 10 1 ; sine
f  2  0  1024  19  .5  1  270  1  ; granule tex shape
f3 0 1025 05 0.01 256 .3 256 .8 512 1 ; envlpx rise shape

";
```

C.2.10 input.ly

```
\score {

  \time 4/4
  \clef violin
  \key bes \major
  {d'2 a2 g4 f2 g2 d'2 c'2 g2}

  \layout { }
}
```

C.2.11 LookupTest.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "Lookup.h"

int main()
{
  char* letter;
  int number;
  Lookup* table = Lookup_newtable();
  printf("just about the call lookup_populate\n");
  table = Lookup_populate("f", table);
  letter = Lookup_bynumber(4, table);
  number = Lookup_byname("ees", table);

  printf("Found letter %s at position 4\n", letter);
  printf("Found number %d at position e flat\n", number);
  return 0;
}
```

C.2.12 lpondlexer.h

```
#ifndef LPONDLEXER_H
#define LPONDLEXER_H

#define SCORE 257
#define BRA 258
#define KET 259
#define NOTES 260
#define TIME 261
#define KEY 262
#define LAYOUT 263
#define TIMESIG 264
#define UPOCTAVE 265
#define DOWNOCTAVE 266
#define SHAFLA 267
#define PITCH 268
#define DURATION 269
#define MAJMIN 270
#define KEYSIG 271
#define REST 272
#define CLEF 273
#define VIOLIN 274

typedef struct token
{
  char *lexeme;
  struct token *next;
  int type;
  int value;
}TOKEN;

extern TOKEN *mktoken(int type);
extern TOKEN *mkval(int type);
TOKEN * yylval;


#endif
```

C.2.13 lpondparser.c

```
#include <stdio.h>
#include "lex.yy.c"

int cursym;
TOKEN* curtoken;

void nextsym(void)
{
  cursym = yylex();
  curtoken = yylval;
}

void checkfor(int type)
{
  printf("Cursym is type %d\n", type);
  if (cursym!=type)
    {
      printf("Invalid input...I think\n");
      exit(1);
    }
  nextsym();
}

int main()
{
  nextsym();
  checkfor(SCORE);
  checkfor(BRA);
  checkfor(TIME);
  if (cursym==TIMESIG)
    {
      printf("Found a token with lexeme %s\n", curtoken->lexeme);
      nextsym();
    }
  checkfor(CLEF);
  checkfor(VIOLIN);
  checkfor(KEY);
  if (cursym==KEYSIG)
    {
      printf("Found a token with lexeme %s\n", curtoken->lexeme);
      nextsym();
    }
  checkfor(BRA);
  while (cursym!=KET)
    {
      printf("Found a token with lexeme %s\n", curtoken->lexeme);
      nextsym();
    }
```

```
  checkfor(KET);
  checkfor(LAYOUT);
  checkfor(BRA);
  checkfor(KET);
  checkfor(KET);
  return 0;
}
```

C.2.14 PitchesTest.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "Pitches.h"

int main()
{
  float frequency1;
  float frequency2;
  float frequency3;
  float frequency4;

  Pitches* table = Pitches_newtable();

  printf("Created table\n");

  frequency1 = Pitches_lookup("a,", table);
  frequency2 = Pitches_lookup("f", table);
  frequency3 = Pitches_lookup("ees'", table);
  frequency4 = Pitches_lookup("bes'", table);

  printf("Found frequency %f corresponding to a,\n", frequency1);
  printf("Found frequency %f corresponding to f\n", frequency2);
  printf("Found frequency %f corresponding to ees'\n", frequency3);
  printf("Found frequency %f corresponding to bes'\n", frequency4);

  return 0;
}
```

C.2.15 RatiosTest.c

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "Ratios.h"

int main()
{
  float ratio1;
  float ratio2;
  float ratio3;
  float ratio4;

  Ratios* table = Ratios_newtable();

  printf("Created the table\n");

  ratio1 = Ratios_lookup(3, table);
  ratio2 = Ratios_lookup(6, table);
  ratio3 = Ratios_lookup(7, table);
  ratio4 = Ratios_lookup(1, table);

  printf("Found ratio %f corresponding to 3\n", ratio1);
  printf("Found ratio %f corresponding to 6\n", ratio2);
  printf("Found ratio %f corresponding to 7\n", ratio3);
  printf("Found ratio %f corresponding to 1\n", ratio4);

  return 0;
}
```

C.2.16 ScoreTest.c

```
#include <malloc.h>
#include <stdio.h>
#include "Score.h"
#include "Chord.h"
#include "lex.yy.c"


int cursym;
TOKEN* curtoken;

void nextsym(void)
{
  cursym = yylex();
  curtoken = yylval;
}

void checkfor(int type)
{
  printf("%% Cursym is type %d\n", type);
  if (cursym!=type)
    {
      printf("%% Invalid input...I think\n");
      exit(1);
    }
  nextsym();
}

/* Populate the score */
Score* Score_populate(void)
{
  Score* score = Score_new(NULL, NULL, NULL, 1);
  Chord* chord;

  nextsym();
  checkfor(SCORE);
  checkfor(BRA);
  checkfor(TIME);
  if (cursym==TIMESIG)
    {
      printf("%% Found a token with lexeme %s\n", curtoken->lexeme);
      score->timesig = curtoken->lexeme;
      nextsym();
    }
  checkfor(CLEF);
  checkfor(VIOLIN);
  checkfor(KEY);
  if (cursym==KEYSIG)
    {
```

```
      printf("%% Found a token with lexeme %s\n", curtoken->lexeme);
      score->keysig = curtoken->lexeme;
      nextsym();
    }
  checkfor(BRA);
  if (cursym==PITCH||cursym==REST)
    {
      printf("%% Found a token with lexeme %s\n", curtoken->lexeme);
      score->firstchord = Chord_new(curtoken->lexeme, 0, NULL, NULL, NULL);
      chord = score->firstchord;
      nextsym();
    }

  while (cursym!=KET)
    {
      /* Only make a chord once we have pitch and duration */
      if (cursym==DURATION)
        {
          printf("%% Found a token with lexeme %s\n", curtoken->lexeme);
          chord->tenor->duration = curtoken->value;
          chord->lead->duration = curtoken->value;
          chord->bari->duration = curtoken->value;
          chord->bass->duration = curtoken->value;
          nextsym();
        }
      else if (cursym==PITCH||cursym==REST)
        {
          printf("%% Found a token with lexeme %s\n", curtoken->lexeme);
          chord->next = Chord_new(curtoken->lexeme, 0, NULL, chord, NULL);
          chord = chord->next;
          nextsym();
        }

    }
  checkfor(KET);
  checkfor(LAYOUT);
  checkfor(BRA);
  checkfor(KET);
  checkfor(KET);

  return score;
}

int main()
{
  Score* score = Score_populate();
  score = Score_harmonise(score);
  printf("First chord is %s %s\n", score->firstchord->name,
         score->firstchord->type);
  score = Score_voice(score);
```

```
  Score_print(score);
  return 0;
}
```

C.2.17 ScoreTest2.c

```
#include <malloc.h>
#include <stdio.h>
#include "Score.h"
#include "Chord.h"
#include "lex.yy.c"


int cursym;
TOKEN* curtoken;

void nextsym(void)
{
  cursym = yylex();
  curtoken = yylval;
}

void checkfor(int type)
{
  /* printf("%% Cursym is type %d\n", type); */
  if (cursym!=type)
    {
      /* printf("%% Invalid input...I think\n"); */
      exit(1);
    }
  nextsym();
}

/* Populate the score */
Score* Score_populate(void)
{
  Score* score = Score_new(NULL, NULL, NULL, 1);
  Chord* chord;

  nextsym();
  checkfor(SCORE);
  checkfor(BRA);
  checkfor(TIME);
  if (cursym==TIMESIG)
    {
      /* printf("%% Found a token with lexeme %s\n", curtoken->lexeme); */
      score->timesig = curtoken->lexeme;
      nextsym();
    }
  checkfor(CLEF);
  checkfor(VIOLIN);
  checkfor(KEY);
  if (cursym==KEYSIG)
    {
```

```
      /* printf("%% Found a token with lexeme %s\n", curtoken->lexeme); */
      score->keysig = curtoken->lexeme;
      nextsym();
    }
  checkfor(BRA);
  if (cursym==PITCH||cursym==REST)
    {
      /* printf("%% Found a token with lexeme %s\n", curtoken->lexeme); */
      score->firstchord = Chord_new(curtoken->lexeme, 0, NULL, NULL, NULL);
      chord = score->firstchord;
      nextsym();
    }

  while (cursym!=KET)
    {
      /* Only make a chord once we have pitch and duration */
      if (cursym==DURATION)
        {
          /* printf("%% Found a token with lexeme %s\n", curtoken->lexeme); */
          chord->tenor->duration = curtoken->value;
          chord->lead->duration = curtoken->value;
          chord->bari->duration = curtoken->value;
          chord->bass->duration = curtoken->value;
          nextsym();
        }
      else if (cursym==PITCH||cursym==REST)
        {
          /* printf("%% Found a token with lexeme %s\n", curtoken->lexeme); */
          chord->next = Chord_new(curtoken->lexeme, 0, NULL, chord, NULL);
          chord = chord->next;
          nextsym();
        }

    }
  checkfor(KET);
  checkfor(LAYOUT);
  checkfor(BRA);
  checkfor(KET);
  checkfor(KET);

  return score;
}

int main()
{
  Score* score = Score_populate();
  score = Score_harmonise(score);
  /* printf("First chord is %s %s\n", score->firstchord->name,
         score->firstchord->type); */
  score = Score_voice(score);
```

```
  score = Score_octaves(score);
  Score_print(score);
  printf("************************************************************");
  Score_csound(score);
  return 0;
}
```