DEPARTMENT OF COMPUTER SCIENCE

Creating a Tool to Analyse Contrapuntal Music

Lisa Bowles

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering

April 2005 | CSBSC-05

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Lisa Bowles, April 2005

Abstract

The area of musical analysis, extracting and disassembling the structure of a piece, is widely practised and could be enhanced by a simple, user-friendly analysis tool. Such a tool is proposed and implemented here. The tool aids in the 2 main areas of contrapuntal (more specifically fugal) music analysis - voice separation and pattern extraction. Voice separation involves separating the voices (or parts) of the piece and pattern extraction involves extracting the musically significant patterns which make up the piece's basic structure. The tool is designed to be very user friendly and interactive, allowing the user to alter the parameters used in these 2 areas of analysis and to both see and hear the resulting outputs. Implementation of this tool has proven successful, both in ease of use and accuracy of outputs.

Contents

Chapter 1

Introduction and Statement of Problem

1.1 Introduction

The area of musical analysis is extensively researched and practised, not only by music students but also by recreational enthusiasts in the area.

In particular, analysis of works by the great composers throughout the history of classical music can provide an insight into the complexity of such compositions and attempt to work out how they were constructed. Analysing music in this way can be fascinating, breaking such a work of art into its constituent parts, and can also aid modern day composers to create works as masterful, or to mimic the styles of past composers.

The basis of musical analysis is to extract and disassemble the structure of a piece - breaking it down into the structural parts which are central to the piece. The most common, and easily comprehensible of these is the theme, or subject of the piece. Figure 1.1 shows the themes extracted from pieces of music from 2 different genres.

The area of classical music most commonly studied is that of counterpoint. A contrapuntal piece of music incorporates two or more independent melodic parts/voices sounding in parallel (see section 2.1 for more information). Counterpoint is probably so often studied due to the precise, mathematical nature of the way it can be constructed and deconstructed. Analysis in this area can be very useful and can be logically followed from piece to piece.

1.2 Analysis Areas

Musical analysis can be thought of as involving 2 main areas. The first of these is the extraction of the voices making up a piece. Any piece of contrapuntal music can be broken down into its constituent voices, allowing the analyser to see how different melodic parts have been constructed and slotted together in order to produce the final output. The second area is the extraction of the basic melodic patterns and parts which make up the structure of these voices (the subject, answer, countersubject, exposition, redundant entries etc.). It is of use to the analyser to see what these are and how, when and where they have been worked into the piece.

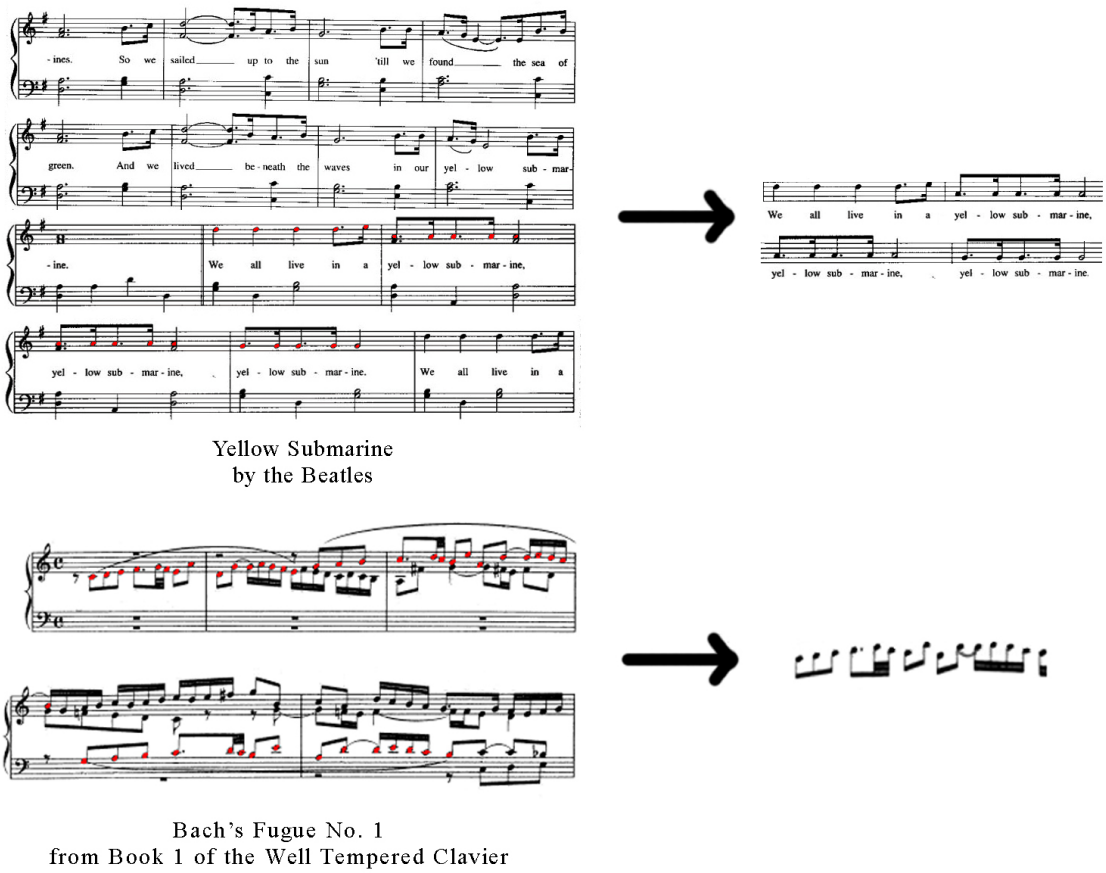


Figure 1.1: Extracting themes

(Yellow Submarine sheet music taken from http://mrpiano.bestmusicpages.com/beatles.html,
Bach sheet music taken from http://www.sheetmusicarchive.net)

1.3 Statement of Problem

The problem in this project is, therefore, to create a tool with which anyone can deconstruct and analyse a piece of contrapuntal music, inputted in MIDI format. The tool must carry out analysis in the 2 areas discussed and must be as interactive as possible to be of the fullest use to the user. As music is such an aural and visual medium (listening to the music and viewing its graphical form in sheet music), the tool must make full use of audio and visual presentation techniques to communicate with the user.

Creating such a tool will amalgamate a number of currently available tools/programs, however creating a more specialised, lightweight, user friendly application with more specific, useful outputs. Currently available tools lie in the 2 main analysis areas. In the voice separation area, current tools include the non-graphical, large scale program midi2gmn [5] which converts MIDI files to a visual form of musical notation, carrying out voice separation during the process, and VoSA (the Voice Separation Analyzer) [2] which is a large scale, graphical application used to separate voices of any MIDI file input. There are fewer tools in the pattern extraction area as literature in the area mostly concentrates on algorithms which can be implemented in much larger scale applications (see section 2.3.2). One program currently available is SIA(M) Express [11], which returns frequently occurring patterns from an input piece.

The main use for the tool created in this project is to aid music students in carrying out specific musical analysis, but it should be simple enough to use in order to also be of use to recreational enthusiasts.

1.4 Other Applications and Utilisations

The sequence analysis section of the tool could be slightly altered in order to carry out a different function. Cataloguing of musical databases is very desirable, especially with the large catalogues of MIDI or MP3 files owned by most computer users in the world of today. This would be a possible further function of the tool, which could use its pattern extraction techniques to extract the theme of all pieces in a catalogue, and allow the user to access the pieces therein by referencing them from the catalogue of themes. This would be a very good way of referencing pieces as the theme of a piece is almost always the factor which enables listeners to identify the piece.

The remainder of this paper is structured as follows:

Chapter 2 explains the musical concepts used throughout the project, explains the MIDI file format and provides a review of relevant literature in the areas covered.

Chapter 3 explains the algorithms which will be implemented and this implementation is detailed in chapter 4.

Chapter 5 discusses the results gained from testing the tool and chapter 6 forms overall conclusions from the project and makes suggestions for further work.

Chapter 2

Background Information

This chapter describes and explains the musical concepts required for full understanding of this project. It also gives an overview of the MIDI file format and the specialist Java MIDI libraries available. Finally, it reviews some of the literature in the areas covered by the project - voice separation, sequence analysis and conversion from MIDI data to traditional musical notation, making conclusions about which pieces of relevant literature will be used in implementation of the tool.

2.1 Musical Background

In this section, the relevant basics of music theory will be explained along with overviews given of the basic contrapuntal forms, including a detailed explanation of the most important form - fugue. A musical glossary, providing more information and definitions of the musical terms used, can be found in Appendix II.

2.1.1 Counterpoint

Counterpoint is a type of musical writing which creates pieces of music comprising “a combination of two or more parts, independent in melody and/or rhythm” [6]. These independent parts may be standalone (musically satisfactory to hear on their own) but the combination of these, the way they are woven together, creates the overall effect which is heard by the listener.

Contrapuntal music differs from normal harmonic music (for example, a string quartet with 4 different string instruments, each playing a different part) in the way that the interest lies in the individual parts rather than the harmonic/chordal result of harmonic parts sounding at the same time. Harmonic music concentrates more on a main part carrying the main melody and the other parts supporting the melody, harmonising with it to create a satisfactory chordal combination of notes. For example, the main melody could take a G value (as shown in figure 2.1) and 2 supporting parts could play the 3rd and 5th notes above this root note, creating the simplest chord - a triad.

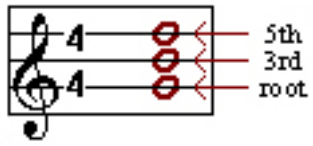


Figure 2.1: Notes forming a triad chord

2.1.2 Contrapuntal Forms

There are a number of forms of counterpoint - ranging from the simplest forms, such as canon, to the most complicated, namely fugue.

Canons involve parts which exactly imitate each other, although sounding at different times. The different types of canon are distinguished by the number of parts in the piece, and the number of melodies which are used - for example, 2 parts can be involved in a canon, both taking the same melody, or 4 parts can be involved, taking 2 different melodies (2 parts imitate 1 melody and the remaining 2 imitate another - just like 2 of the aforementioned canons sounding in parallel). There are a number of more complicated forms of canon. Some examples are explained below (for more examples, see [3], page 108).

- "Canon by diminution" - incorporates 2 voices where the second voice's notes are a fraction (e.g. half) of the length of the notes in the first voice.
- "Canon cancrizans" (crab cannon) - incorporates 2 voices where the second voice is the same as the first voice played backwards.

Examples are shown in figure 2.2 of a simple 4 part, 1 melody canon and a more complex 4 part, 2 melody crab canon.

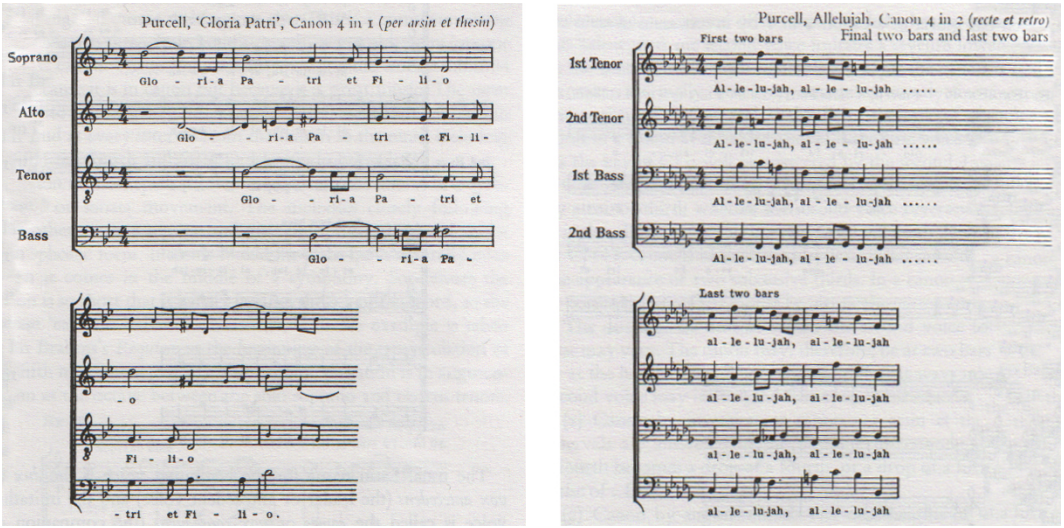


Figure 2.2: Example canons (taken from [3], Ex. 86, page 109 and Ex. 87, page 110.)

A perhaps more commonly known contrapuntal form is that of **round**. A round is a vocal type of canon where each sung voice enters at a different point and returns to the beginning after reaching the end of the melody. A round will continue indefinitely unless the number of repeats is decided prior to commencement.

Canzona is another form in which a group of instruments go through a number of sections where a theme (or subject) goes through stages of increasing elaboration. The *subject* of a piece of music (of any genre) is "the material on which part or all of a composition is based" [6]. This is a short piece of a melody (a pattern of notes) which defines the piece, and is repeated multiple times throughout the piece. See figure 1.1 for examples of subjects/themes being extracted from extracts of music. These subjects can be thought of as the most memorable part of a piece.

A theme is also predominant in the **ricercar** contrapuntal form. This form preceded fugue, and greatly influenced the fugal form. A ricercar involves all voices entering with a given subject, then re-entering with transpositions and inversions of this subject, as well as alterations to it similar to those used in the more complicated types of canon.

Transposition means altering a melody (or part of a melody) to a different pitch from the original, while retaining the same note lengths and intervals between consecutive notes. Figure 2.3 shows a simple transposition of a short original melody. In this case, the melody has been transposed up a 3rd.

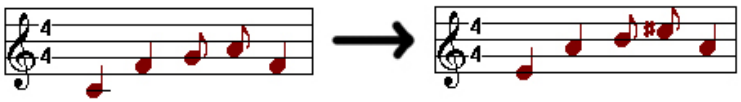


Figure 2.3: Transposition

Inversion means turning a chord or sequence of notes 'upside down' - replacing pairs of notes at a certain interval so that the formerly lower note becomes at the same interval above the formerly upper note. Example of this are shown in figure 2.4.

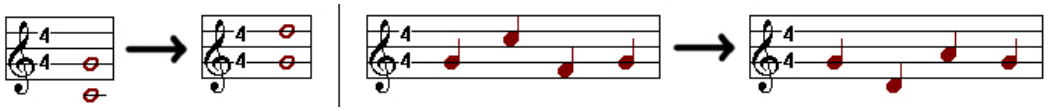


Figure 2.4: Chord and sequence inversion

Ricercars also incorporate a secondary subject which slots in with the main subject, called a *countersubject*. This is another section/pattern of notes which help to define the piece of music and appear throughout the piece, but are secondary to the main subject on both counts.

The contrapuntal forms mentioned here are the most commonly used forms and are those on which any further forms take their base, with only slight variations.

2.1.3 Fugue

The fugal form requires its own section as it is the most complex, highly developed and most regularly used of all the contrapuntal forms. Fugue extends all the forms previously mentioned, most closely matching that of ricercar.

Fugue is, again, based around a subject - the pattern of notes defining the piece. Another structural part is defined in fugue - the *answer* - a pattern which follows the subject. It is sometimes an exact transposition of the subject and sometimes varies slightly. The answer may slot in with the subject in a number of ways - either it starts immediately after the subject ends, before it ends or sometime after. Figure 2.5 shows examples of all 3 different placements of the answer, taken from 3 of Bach's fugues from Book 1 of the Well Tempered Clavier.

All voices play the subject in a section at the start of a fugue, which is called the *exposition*. The order of appearance of the subject and answer (during this exposition) is not fixed, i.e. the order is usually subject, answer, subject, answer etc. but this may be altered, for example so that the ordering subject, answer, answer, subject occurs.

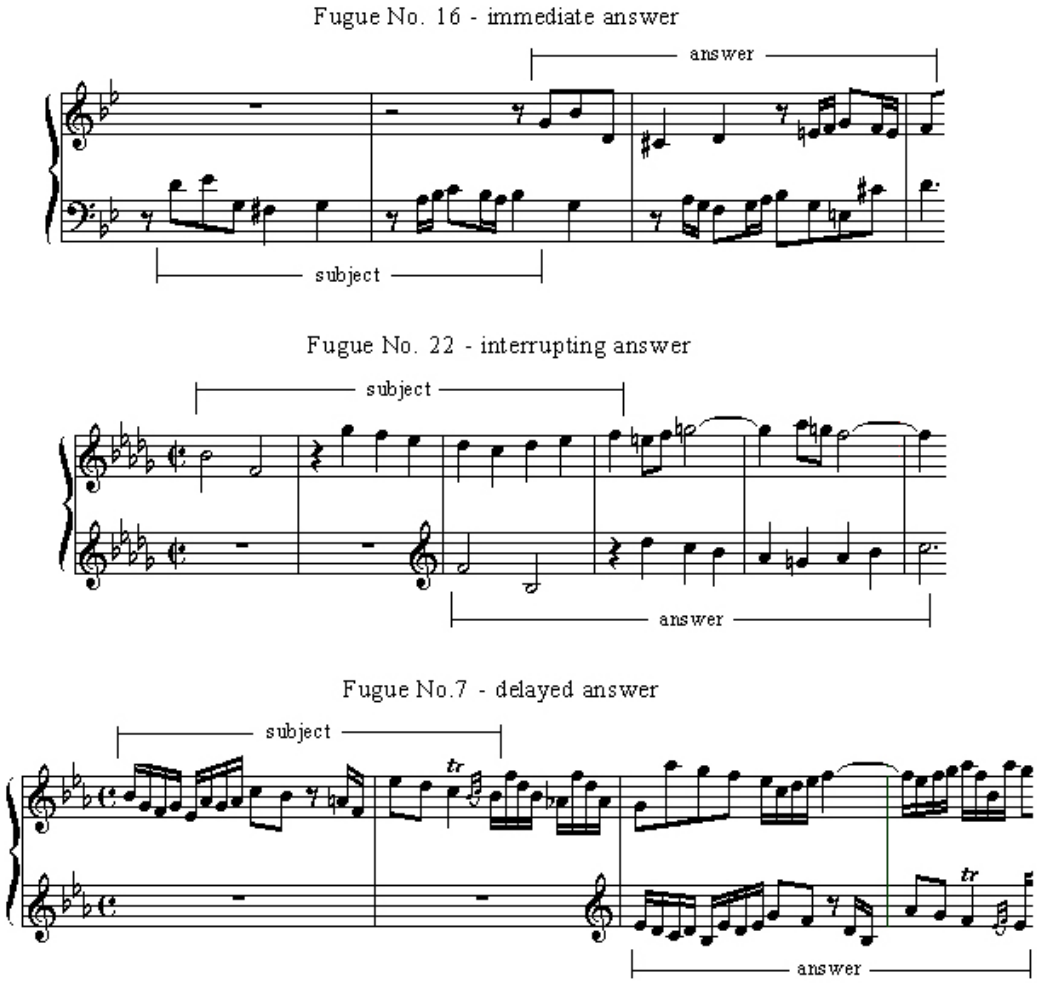


Figure 2.5: Different placements of the answer (sheet music taken from [14])

The countersubject appears again in fugue. Fugues may or may not include a secondary subject like this, and sometimes two such countersubjects are present. A countersubject can be told apart from a free part (a non-structurally important part of the piece) by its frequency - if it appears a number of times in parallel with the subject it is a countersubject, and also it is a countersubject if transpositions of it are also present in the piece. An example countersubject, slotting in with the subject, is shown in figure 2.6.

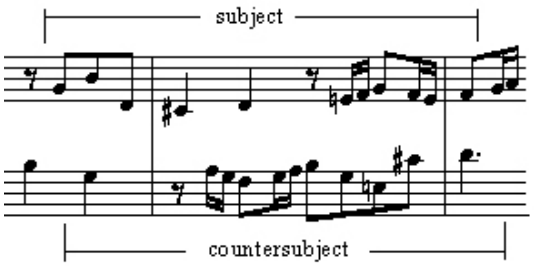


Figure 2.6: A countersubject (sheet music taken from [14])

Further to these basic structural patterns, there are appearances of these which go by different names. A *redundant entry* is an occurrence of the subject or answer which may happen in order for the countersubject to be heard in a different position. For example, if the voices all start in such an order that the entry points get lower at every new voice entry, the countersubject will only ever

appear above the subject (e.g. the first part will play the countersubject while the lower second part plays the subject), a redundant entry will allow the countersubject to be played below the subject.

There can also be multiple redundant entries. When this occurs to the extent that a new exposition section has been created (where each voice plays the subject), this section is then called a *counter-exposition*.

Fugue can use all the techniques which are also used by canon, but can sometimes also use another technique called *stretto*. This 'fugal device' is like having a small canon within the fugue. It involves one voice playing the subject and another starting to play another statement of the subject before the first has finished.

Any section within a fugue where the subject does not appear is called an *episode*.

Overall, the structure of a fugue can be split into 3 sections - the *exposition*, any number of *middle entries* and the *final section*. This final section occurs at the end of a fugue, where the subject reappears at its original pitch. Middle entries are simply the sections which occur between the exposition and final section.

2.1.4 Inventions

An invention is another kind of contrapuntal music - a simple, 2 or 3 part piece, similar in style to fugue but much less complex. Inventions are not written for performance but as exercises with which a learner can practise. The exposition section is as in fugue, but the middle entries are a lot simpler (not incorporating the techniques described for canon and fugue). The concepts of subject, answer and countersubject are still vaguely maintained.

2.2 MIDI Format and Java MIDI Libraries

The MIDI protocol (Musical Instrument Digital Interface) is a platform independent method of storing musical information in order for that information to be read by pieces of hardware and software, especially sequencers. The standard MIDI file (SMF) format allows MIDI data to be stored in binary files. MIDI data comprises a timestamp and an associated MIDI message, which gives all the required information as to how the relevant MIDI information should be interpreted or "played".

There are 3 types of basic MIDI file formats - most common are those that put all information into one track (type 0) and those that have a track for each part - each separate instrument or melodic part (type 1). Type 2 files are similar to type 1 files but differ in the way that they include tempo and time signature information for each separate track, where type 1 files store this information in a single track at the start of the file.

The binary MIDI files are able to be read by the javax.sound.midi package, which uses a MidiFileReader to extract the relevant information. MidiFileFormat stores all the information about the whole file - type (0, 1 or 2), length etc. A 'Sequence' can then be extracted from this file - a data structure which holds all the musical information about the file.

This representation works in a number of levels of abstraction - the Sequence contains Tracks which each contain a number of MidiEvents which, in turn, contain a MidiMessage. These messages are in byte form[1] - expressing note on and note off (including the note number and velocity) as well as meta-events which contain information about time/key signatures, tempo values etc.

[1] http://www.harmony-central.com/MIDI/Doc/table1.html provides tables of byte and integer values for all midi events and for notes (e.g. middle C is 60).

2.3 Literature Review

This project covers a number of different research areas, the most fundamental of these being the separating of voices in a piece of polyphonic music (in MIDI file format), and the analysis of these voices' melodies, by sequence analysis techniques, to find patterns significant to the structure of the music. Secondary to these, the area of converting MIDI representation of notes and structure to traditional musical notation must be covered in order to present the findings of the previous algorithms to the user.

2.3.1 Voice Separation

Many different methods for voice separation exist, reporting differing levels of success.

Split Point Separation

The simplest method, Split Point Separation (discussed by Kilian and Hoos [5]), assigns notes to voices depending only on which pitch range they fall into. This method is very basic and therefore not as successful as other proposed methods.

Kilian and Hoos

Kilian and Hoos [5] lay out a local optimisation approach, involving splitting the piece of music into slices and assigning notes within each slice to a different voice. The assignments are decided by minimising a cost function and taking into consideration the voice separations of previous slices. This then gives the best assignment given the set of user defined parameters (maximum number of voices, cost function parameters etc). This approach differs from others in the way that it allows notes to be assigned into chords within a voice. Also, by allowing the user to alter the given parameters, a range of different types of separation can be achieved.

Chew and Wu

In the most recent addition to this field, Chew and Wu [2] have created a new algorithm which performs in O(n^2) time with a slightly higher accuracy rate than the previous algorithms. This algorithm is based on the idea of splitting a piece of music into sections and rejoining these sections using a shortest distance strategy to identify the lines of the different voices. A voice count is taken from chords in the piece where it is assumed that all voices are sounding at once, and the algorithm then works outwards from these parts to join the different lines in the piece.

Interestingly, these algorithms focus on separating the voices of a piece not to match the original score but the perception of human listeners. This seems to be common throughout the literature. It could be thought that it is more important for a voice separation algorithm to return the voices as shown in the score but this would lead to a much more complicated algorithm due to voice crossing. This is dismissed in the current methods due to the findings of research into human perception, but would be important to bear in mind if trying to recreate the original voices as voice crossing is stated by Mann [7] as being an important tool in the writing of contrapuntal music, and will therefore be commonly found in such music.

2.3.2 Sequence Analysis

There is a lot of information available on the subjects of data mining and sequence analysis, much of which focuses specifically on automatic musical analysis. The most common techniques appear to be string based.

Rolland

Rolland [13] concentrates on the musical style of jazz improvisation, finding patterns between slightly differing (e.g. ornamented) phrases. This could be useful in this project as structural parts of contrapuntal music are often repeated with slight variation. Rolland's algorithm moves through the input corpus building a graph representing passages (vertices) and the value of their similarity to other passages (of variable lengths - edges). This value is calculated using a specialist model which compares all pairs of notes within the aligned passages, giving each pair a positive or negative similarity value, contributing to a final figure. The algorithm then extracts the actual patterns using the similarity values (over a certain threshold) of passages' edges in the graph.

Various inputs must also be given to the algorithm in order for it to run reasonably efficiently - e.g. the maximum and minimum allowed length of a passage, maximum difference in length between passages being compared etc.

Meredith et al.

Meredith et al. [11] introduce a new algorithm which discovers maximal repeated patterns within a dataset (which can, unlike Rolland's algorithm, be polyphonic music). The first part of the algorithm finds all the maximal subsets of the input dataset and passes these onto the next part. This then computes all the occurrences of the given subsets/patterns and puts them into sets of equivalent patterns (which can be translations of each other). Unlimited gaps between elements of a pattern are allowed, therefore allowing for variations between repetitions of the same pattern (as Rolland, above).

This information can be passed into the final stage - a separate algorithm which uses a set of heuristics to decide the "musical significance" of the discovered patterns. The simplest of these heuristics being that the most important patterns are likely to be those which are repeated the most (the largest sets).

Hsu et al.

Hsu et al. [4] propose a simpler method, suitable only for monophonic data sources. Their method involves a correlative matrix approach by which notes in a sequence are entered into a symmetric matrix. As each new note in the sequence is encountered, numbers are added into the relevant elements of the matrix to show the results of the process of matching the new note with the notes currently in the matrix. A candidate set of patterns can then be created from the figures in this matrix. By this method, it is possible to extract maximal repeating patterns from the input dataset.

Mannila et al.

A more general purpose algorithm is given by Mannila et al. [8]. This calculates frequent episodes in an event sequence using a sliding window technique. The frequency of any one episode is given as the number of windows in which it occurs, being reported as 'frequent' according to a user defined threshold. The algorithm requires input as a set of episodes from the input sequence, which can be given simply as every event in the sequence (representing episodes with only one event - one note). From this, it will then build a set of "candidate episodes" and check their frequencies. This procedure then repeats until maximal length episodes have been found.

2.3.3 MIDI to Traditional Notation Conversion

There are a number of proposed methods (such as Cambouropoulos [1] and Meredith [9]) which apply a number of algorithms on the data, eventually representing MIDI files on the traditional stave.

However, there is also an open source program called Lilypond[2] which transcribes MIDI data into sheet music form. This first converts from midi format into its own .ly format by way of the conversion program midi2ly. .ly files (text files) are then converted into sheet music form by Lilypond's main functions.

2.3.4 Conclusion

Voice Separation

The different methods for voice separation vary quite considerably in approach but seemingly not so much in success of results. The method which will be used in this project is that of Kilian and Hoos [5] as this method is more flexible in the output it creates, allowing the user to (to some extent) control the type of resulting separation. This kind of user interaction is important in a graphical tool. They also list future work as involving creating a user interface through which to carry out voice separations on pieces of music.

Sequence Analysis

For sequence analysis, both Meredith et al. [11] and [10] and Hsu et al. [4] propose interesting and useful methods in their algorithms. [11] and [10] are very complicated, perhaps too much so for the project in question. The main areas in which they improve on the earlier method of [4] are allowing polyphonic input and translational equivalence (recognising transposed patterns).

The first of these is not an issue in this case as the output of the voice separation algorithm will be passed for sequence analysis, and each voice will consist only of a single, monophonic line. The second area is also easily overcome as instead of making up a matrix of the specific notes in a sequence, the intervals between each note could be used (therefore equating identical patterns which start on different notes). The method which will be used in this project is therefore that of Hsu et al. [4].

Taking this approach as the simple outline for the algorithm, extra features could be added into it to further improve its performance and flexibility. For example, the heuristics part of the Meredith algorithm can be used to allow for factors other than frequency of repetition to be taken into consideration when finding the most significant patterns. Also, it might be required to find some way to adapt the algorithm so that it recognises similar patterns (e.g. with slightly different rhythms) as well as identical ones.

Conversion

Converting MIDI files to traditional musical notation will be done using Lilypond.

This chapter has shown the background information required to follow the remainder of this paper, both in the areas of music and the MIDI format. Relevant literature has been reviewed and the conclusions made will now be carried into the design stage, using the methods of [5] and [4] to carry out the main analysis functions.

[2] www.lilypond.org/web

Chapter 3

Design

In this chapter, the algorithms to be used in implementation will be explained - Kilian and Hoos's local optimisation approach to voice separation [5], Hsu et al.'s method of efficient repeating pattern finding [4], and the k-nearest neighbour machine learning method used for voice separation accuracy prediction. Firstly, the scope of the project will be discussed.

3.1 Project Scope

The project was originally laid out to analyse 'contrapuntal music'. Cole [3] describes the structures of various types of contrapuntal forms, fugue being specified as the most advanced, most detailed and most widely used of these forms.

A more useful tool than that first described could therefore be a less general one built to specifically analyse fugal music. If necessary, and time allowing, this could be extended to also take input as pieces of music following the earlier contrapuntal forms - canon, ricercar etc. As these forms are less advanced than fugue, containing only a small number of the features of a fugue, analysis could be simpler. However, these forms also contain additional features not present in fugue.

The tool could therefore concentrate only on the most common contrapuntal form, fugue, be extended to include additional forms (with the input form specified by the user), or take input as any form (unspecified) and, as the first stage of analysis, automatically determine the type/form.

For this project, the scope will be limited to fugues with the extensions mentioned as possible further work. Due to the complex yet consistent nature of fugue, analysis of this form is a lot more interesting and informative, much more often carried out and more complicated - therefore requiring a tool to aid in analysis.

Examples of fugue are J.S. Bach's Book 1 of the Well Tempered Clavier, MIDI versions of which will be used as test pieces throughout the project[1], as these pieces are commonly used to test such applications. Bach's 2 and 3 part inventions[2] will also be used, as less complex test pieces.

Figure 3.1 shows the new objectives of analysis in this project, using an example piece - Bach's Fugue No. 7 from Book 1 of the Well Tempered Clavier. It shows an extract from the piece and the resulting voices and patterns which will be extracted by the analysis tool.

[1] These can be obtained from www.multimedialibrary.com/Music/dreamworld.html

[2] Also obtainable from the same source

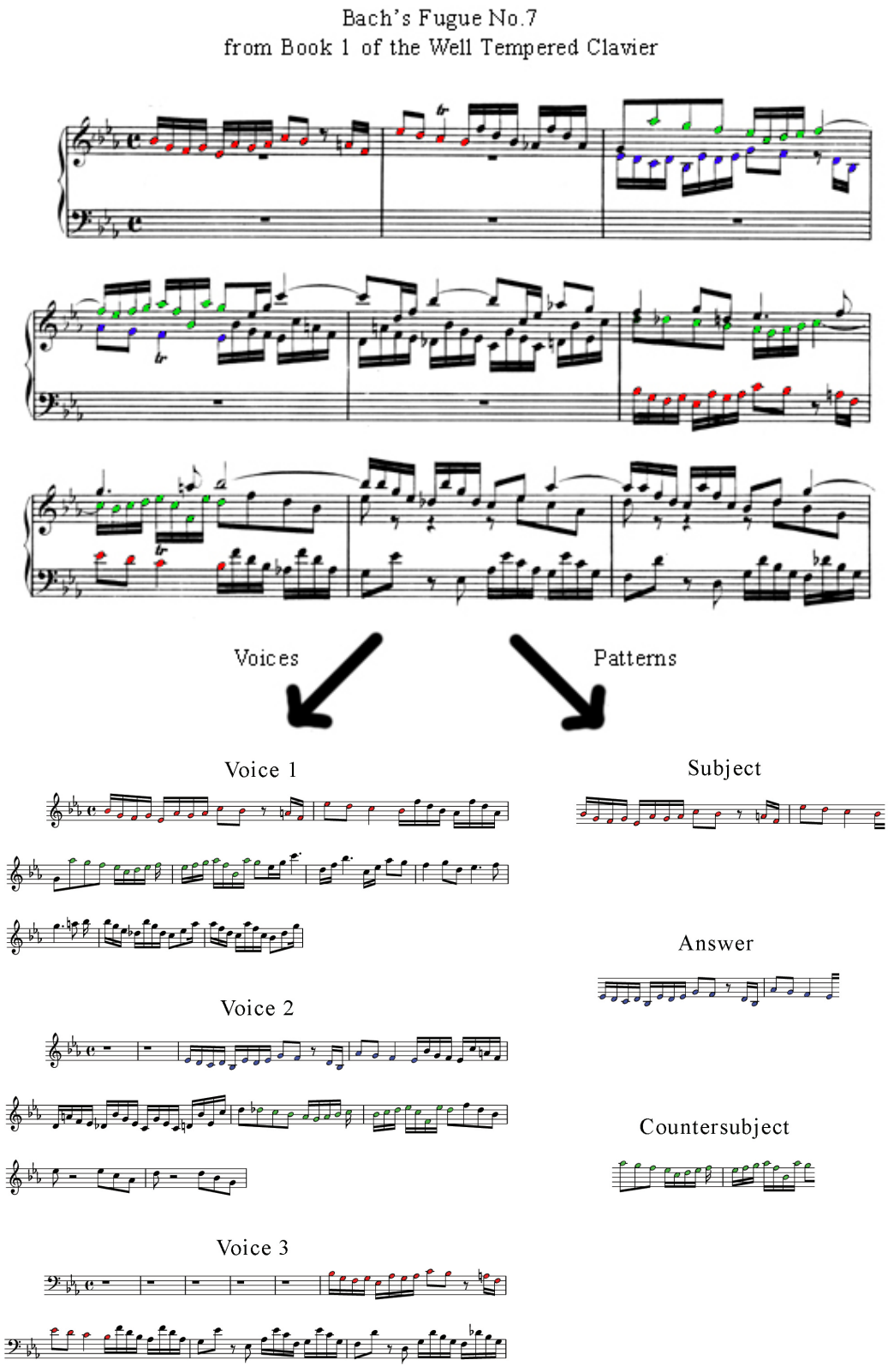


Figure 3.1: Analysis objectives

(Fugue sheet music taken from http://www.dlib.indiana.edu/variations/scores/abt8726/index.html, voices and patterns taken from pdf outputs created by the tool)

3.2 Voice Separation Algorithm

The method of Kilian and Hoos [5] first segments the input piece into slices then finds the optimum separation for each slice by calculating different separations and choosing that which minimises a cost function. These 2 processes are discussed in detail below.

3.2.1 Slice Segmentation

The segmentation process works as follows: the 1st note is automatically a member of the 1st slice. From this point, the next note is a member of the current slice if and only if it overlaps with all the members currently assigned to the slice, otherwise it becomes the first member of the next slice. Figure 3.2 shows a simple example of splitting a sequence of notes into slices.

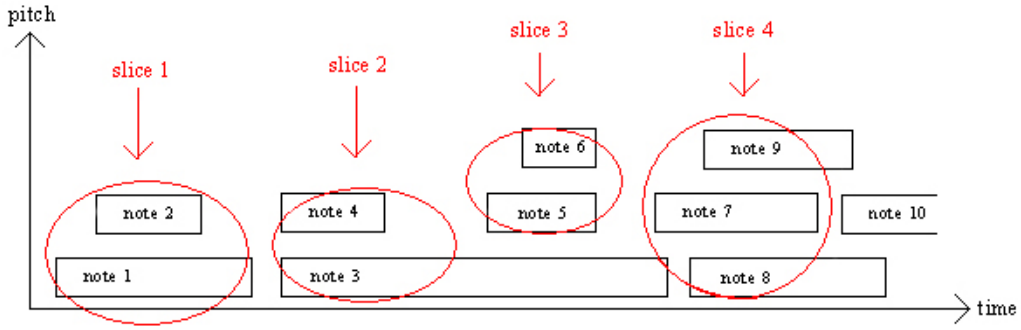


Figure 3.2: A simple example of slice segmentation

The example in figure 3.2 requires some further explanation. Notes 1 and 2 are assigned to slice 1 as they overlap with each other, even though they don't start or finish at the same time. Notes 3 and 4 are assigned to slice 2 as the first note (note 3) would be added to the new slice because it doesn't overlap with any of the notes in the previous slice, then note 4 would be assigned to the same slice as it overlaps with note 3. Note 5 will not be included in this slice as even though it overlaps with note 3, it does not overlap with note 4. This means that note 5 will start a new slice, slice 3, into which note 6 will also be allocated as it overlaps with note 5. Note 7 overlaps with note 3 only, but starts a new 4th slice as note 3 has been allocated to a previous slice in which other notes are present which don't overlap with the new note 7. Notes 8 and 9 are also added to this 4th slice as they all overlap with each other, but note 10 will start a new slice as it only overlaps with 2 out of the 3 notes in slice 4 (notes 8 and 9).

3.2.2 Cost Function

The cost function is split into 4 different parts, each of which penalises a different unwanted element within a voice:

Pitch cost is used to penalise large pitch intervals between successive notes in a voice (as shown in figure 3.3), and is calculated using 3 separate methods. The first counts the number of voices used in the given separation then calls the second method once for each voice used, calculating the overall pitch distance of the slice as:

for all voices in current slice do:

$$pD = pD + (1 - pD) \times \text{ pvD for current voice}$$

Each voice's pitch distance (pvD) is calculated by the second method - finding the note in the previous slice, or at an earlier point in the current slice, which was allocated to the same voice and calculating the distance between this note and the new note (or notes) allocated to that voice

in the current slice (making sure, if the previous note is from the same slice, that it is not part of the same chord). This involves calling the third method for all notes found to be allocated to the given voice. The third method returns the absolute distance in pitch between the previous note in the given voice and the pitch of the current note being assessed (so this method could be called multiple times with the same previous note and different current notes which form a chord). The following equation is used to find the pitch distance for a given voice:

for all notes in current voice do:

$$pvD = pvD + (1 - pvD) \times \frac{\text{pitchDifference(previous note, pitch of current note)}}{128}$$

If the previous note is part of a chord, the distance is returned between the inputted pitch and the note in the chord which is closest in pitch to this value. If the previous note is not part of a chord, the distance in pitch between this note and the inputted pitch is returned. The second method divides the returned pitch difference by 128 as there are 128 MIDI pitch values, so the maximum pitch difference in 128. Pitch difference is measured in semitones.

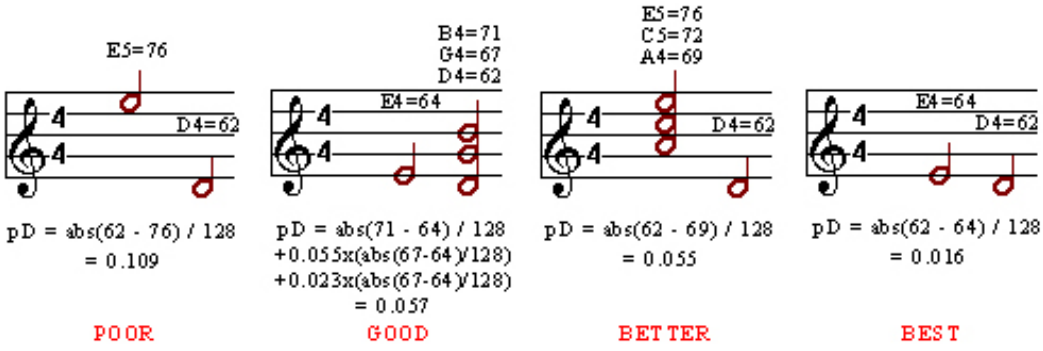


Figure 3.3: Pitch costs

The remaining 3 costs are less complex than pitch cost. **Gap cost** is used to penalise large gaps (rests) between successive notes in a voice, and comprises 2 separate methods. The first of these counts the number of voices used in the separation and then, for each of these voices, finds the earliest note in the slice which is allocated to that voice and, using the second method, finds the largest gap that would be created by adding the note to any of the voices. It then calls the second method to get the gap distance for the given note and the given voice, which is then divided by the maximal gap previously calculated . The final returned value is the average of these figures and, due to the largest gap figure used, is always between 0 and 1.

The second method searches back through previous separations to find the last occurrence of a note in the given voice. When the required note is found, the gap between the 2 notes is calculated as the difference between the on time of the later note and the off time of the earlier note.

Chord cost is used to penalise chords in which large pitch intervals are present between elements of the chord. The chord cost method first loops through the current slice, finding all the chord notes and putting them into an array of arrays (an array of chords). It then iterates through the resulting chords which comprise more than 1 note and uses the formulae below to calculate the cost of the given chord:

$$p = (1 - \frac{shortest}{longest} + (1 - (1 - \frac{shortest}{longest})) \times range$$

$$p = p + (1 - p) \times \frac{(latest - earliest)}{longest}$$

Where shortest and longest refer to the duration of the shortest/longest note, latest and earliest

refer to the onset time of the latest/earliest note and

$$range = \frac{highest - lowest}{24}$$

Where highest and lowest refer to the pitch values of the highest/lowest note.

These chord costs are amalgamated using the following formula:

for all chords in the current slice do:

$$cD = cD + (1 - cD) \times \text{ the value of p for the current chord}$$

Finally, **overlap cost** is used to penalise overlaps between successive notes in a voice, and comprises 2 methods. The first simply calculates the number of voices used in the current separation and uses the following formula to calculate the overall overlap cost:

for all voices in current slice do:

$$oD = oD + (1 - oD) \times \text{ ovD for current voice}$$

The value of ovD is calculated by the second method by finding the note in the separation with the latest onset time then moving back through the separation, comparing notes to find pairs of notes which overlap. When a pair of overlapping notes is found, the following formula is applied:

$$ovD = ovD + (1 - ovD) \times oDist$$

$$\text{Where } oDist = 1 - \frac{\text{onset time of note} - \text{onset time of previous note}}{\text{duration of previous note}}$$

The cost function weightings are discussed in more detail, with examples, in the user guide - Appendix III.

3.3 Voice Separation Accuracy Prediction Algorithms

At a later stage in the project, it was decided that a machine learning algorithm should be built into the program to predict the accuracy of the outcoming voice separations. By adding this feature, it will be possible to assess the quality of the separations, based on a set of instances for which the true accuracy is known.

Out of all the possible machine learning algorithms to use, k-nearest neighbour was chosen. This is because it easily allows for continuous data (as all the attributes are continuous integers or double values), and is a lazy algorithm which doesn't carry out any training until required (therefore not unecessarily adding to the execution time of the program). k-nearest neighbour is a simple algorithm which assumes that instances close to each other (in n-dimensional space - where n is the number of attributes used) will have similar classifications (in this application - similar accuracy rates). It has a static training set of instances and their classifications and, only when presented with a new test instance, finds the k instances in the training set which are at the closest distance from this new instance. The prediction returned will be the average of classifications of these k instances. Distances are calculated as standard Euclidean distances - this will be discussed further below.

The algorithm works by training the model when a new instance is encountered - finding the k training instances (from the training set) which are closest to the new instance, returning the average of the accuracies of these k training instances. The following formula is used to calculate the distance between 2 instances (adapted from the equations given in [12]):

$$distance(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

Where $a_r(x_i)$ is the r^{th} attribute of instance x_i

The predicted value of the new instance is therefore calculated using the following formula:

$$accuracy(x_{new}) = \frac{\sum_{i=1}^{k} accuracy(x_i)}{k}$$

It was also decided that another extension be made in order to let the user know how accurate the predictions are likely to be, as a prediction is otherwise not very useful as it could mean nothing. The accuracy rate of the model can be calculated using the 'leave-one-out' cross validation technique, following the steps shown below:

- e = 0
- for each training instance z
- train on {dataset - z}
- test on z (and obtain predicted accuracy)
- e = e + (true accuracy of instance z - predicted accuracy)
- final error estimate = $\frac{e}{|data|}$

  Where $|data|$ is the number of training instances

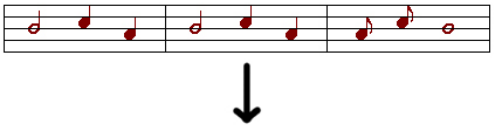
3.4 Sequence Analysis Algorithms

The sequence analysis algorithm of Hsu et al. [4] involves creating a correlative matrix from the notes of a sequence and calculating a candidate set of possible repeating patterns from this matrix. These processes are described below:

3.4.1 Creating the Matrix

Each note in an inputted sequence is moved through in step, comparing the note (from each row of the matrix) with all following notes in the sequence (from each column). This builds up the triangular matrix row by row.

For each comparison, if the notes are equal in value, the corresponding element of the matrix ($T_{i,j}$ where i = row and j = column) is set to 1. If, however, the value of the element at $T_{i-1,j-1}$ is not 0 (the default value), the current element then takes the value of this previous figure plus 1, to show that a pattern of a larger size than 1 has been found. This process is carried out until all required comparisons have been made. Figure 3.4 shows a very short sequence of repeating notes and the resulting matrix which would be created using this method.



	71	72	69	71	72	69	69	72	71
71	-			1					1
72		-			2			1	
69			-			3	1		
71				-					1
72					-			1	
69						-	1		
69							-		
72								-	
71									-

Figure 3.4: An example matrix created from the given repeating sequence

3.4.2 Calculating the Candidate Set

In order to create the candidate set, each element of the matrix is tested for 4 different cases:

1. $T_{i,j} = 1, T_{i+1,j+1} = 0$ - a pattern of size 1 has been found which is not a subsection.
2. $T_{i,j} = 1, T_{i+1,j+1} \neq 0$ - a pattern of size 1 has been found which is a subsection.
3. $T_{i,j} > 1, T_{i+1,j+1} \neq 0$ - a pattern of larger than size 1 has been found which is a subsection.
4. $T_{i,j} > 1, T_{i+1,j+1} = 0$ - a pattern of larger than size 1 has been found which is not a subsection.

Depending on which case has been matched for each element, different processes are carried out to build the candidate set. Each entry in the candidate set has a pattern, a rep count (showing the frequency of repetition of the pattern), and a sub count (showing the number of times the pattern occurs as a subsection of another, larger pattern). The processes to be carried out for these different cases are shown below:

1. If the 1 note pattern is already in the candidate set, increment its rep count, otherwise create a new entry with rep count = 1 and sub count = 0.
2. If the 1 note pattern is already in the candidate set, increment its rep count, otherwise create a new entry with rep count = 1 and sub count = 1.
3. This must be dealt with in 2 sections:
   (a) First deal with the pattern of just 1 note - if already in the candidate set, increment its rep and sub counts, otherwise create a new entry with rep count = $T_{i+1,j+1} - 1$(as this figure shows how many patterns the note has appeared in) and sub count = 1.
   (b) Second, for each of the $T_{i+1,j+1} - 1$ subpatterns of the total pattern, if the pattern is already in the candidate set, increment its rep and sub counts, otherwise create a new entry with rep count = 1 and sub count = 1.
4. This must be dealt with in 2 sections:
   (a) First deal with the pattern of just 1 note - if already in the candidate set, increment its rep and sub counts, otherwise create a new entry with rep count = $T_{i+1,j+1} - 1$ and sub count = 1 (as the single note is a subsection of the smaller patterns which make up the total pattern, even though the total pattern itself is not a subsection).
   (b) Second, for the first $T_{i+1,j+1} - 1$ subpatterns of the total pattern, if the pattern is already in the candidate set, increment its rep and sub counts, otherwise create a new entry with rep count = 1 and sub count = 1. For the final pattern (which is not a subpattern), if it is already in the candidate set, increment its rep count, otherwise create a new entry with rep count = 1 and sub count = 0.

Figure 3.5 shows the candidate set which would be created for the previously shown example.

Pattern	Rep Count	Sub Count
71	1	1
72	3	1
71-72	1	1
69	4	1
71-72-69	1	0
72-69	1	1

Figure 3.5: The candidate set produced from the matrix in figure 3.4

This chapter has given in-depth explanations of the 3 main algorithms to be used in this project - those for voice separation, accuracy prediction and sequence analysis. Implementation of these algorithms will be detailed in the following section.

Chapter 4

Implementation

In this chapter, the implementation process of the tool (using the algorithms in chapter 3) will be described. This implementation lies in 5 main areas - creating an intermediate representation (as Java MIDI libraries do not allow for direct MIDI data manipulation), voice separation, accuracy prediction, sequence analysis and GUI creation.

Full javadoc documentation for all the classes and methods in the project can be found at www.cs.bris.ac.uk/home/lb2148.

4.1 Intermediate Representation

In order to manipulate inputted MIDI data, it was required to write methods to convert to and from an intermediate representation. This is because the Java MIDI libraries do not allow for direct manipulation of MIDI information.

In order to get all notes appearing in time order (rather than split up into time order per track) the input MIDI file must first be converted into type 0 (where all the musical data is stored in one track). There are programs available (as java source code and applications) which will carry out this conversion[1].

Methods were written which take a MIDI sequence and extract the relevant data. convertFromMidi looks at all events in the sequence and creates an ArrayList containing items which take their information from events which have 3 bytes (event type, note and note velocity). These items are of the type Note (extending Object) - a class which contains methods to create, get and set fields within a Note type (name, octave, on time and off time).

convertFromMidi creates a new Note object (with time on and time off initialised to -10) for each note found (using the method getNoteName), producing a list of events. sortNotes then iterates through all events, pairing up and setting the on and off events for each note. The final output is therefore a list of notes ordered by onset time - the format required as input to Kilian and Hoos's algorithm for voice separation.

Once all alterations have been carried out on a Sequence, it can be converted back from the intermediate representation to a MIDI file by convertToMidi (using a reverse method getNoteValue to get the integer representation of a Note object).

All these methods were written into the class MidiFunctions, which acts as a library of relevant MIDI functions.

[1] currently used is the source code from http://www.jsresources.org

4.2 Voice Separation Implementation

The Note class was extended to include methods for getting and setting values representing the voice and chord of the Note. A module was also added to read in MIDI files and create separate MIDI sequences/files for each track within the original. Run on type 1 files, this produces tracks that represent part of a rough separation of the voices in the piece. This is an interesting output which can be used for comparison with the output from the voice separation part of the project.

The voice separation algorithm of Kilian and Hoos [5] first segments a list of notes into slices, then allocates the notes in each of these slices, minimising a given cost function, extracting an output file for each separate voice.

The pseudo code given in [5] is very rough and the explanations are by no means extensive, therefore implementation of the given algorithm was a lot more complicated and involved than first imagined.

4.2.1 Separating Slices

Having segmented the input piece into slices (as explained in section 3.2.1), the controlling method, voiceSeparation, calls separateSlice for each slice created, appending the returned List to the main voice separation list. Each slice is held in memory as a List of Lists of Note objects. Lists are used as items within can be easily accessed, altered, deleted etc.

separateSlice carries out randomised iterative improvements to find the cost-optimised separation for the given slice. It obtains a first separation for the first slice - assigning all notes to one voice and setting the chord values as equal if the notes onset times are equal, then loops until no improvement has been found within a fixed number of steps. 80% of these loops call getNeighbour twice (once to change the chord and once to change the voice) for each number of allowed voices for each note in the slice. This enables the full range of different neighbour slices to be computed, and separateSlice takes the one of these which returns the minimum cost. The remaining 2 out of 10 times, a randomly generated call to getNeighbour is made. During each loop, the function keeps track of the optimum separation found so far, which is returned when the loop is broken out of.

When all slices have been successfully allocated, the method createVoices searches through the list of notes resulting from the allocations, picking out those belonging to each voice. These are collected in a list, one by one, and sent to convertToMidi to be converted into separate output midi files (“voice0”, “voice1” etc.).

4.2.2 Finding Alternative Separations

getNeighbour simply copies the inputted slice and alters the voice or chord (a flag for which is passed in as a parameter) of the given note (passed as an index). The cost function takes input as the current separation and a list of previous separations and calculates the cost based on all the functions related to the 4 cost types, and the weights by which to multiply the costs.

One error in the getNeighbour function, which was necessary to correct, was making sure that a given note doesn’t overlap with the previous note in a voice before assigning it to that voice, as voices cannot contain overlapping notes which do not form a chord (notes are allocated to chords, where appropriate, when the original slice separation is computed). This check is suggested as being written into voiceSeparation by [5] to overcome these overlapping errors after each new slice separation has been added, but the getNeighbour function was decided to be the preferred location for the alteration.

4.2.3 Cost Function

The 4 parts making up the cost function - pitch cost, gap cost, chord cost and overlap cost - have all been implemented as in [5] (for a detailed explanation, see section 3.2.2). These all search through separations of previous slices and calculate the relevant cost of the given new separation. The main method controlling the calculation of the cost of a separation creates this cost as a weighted average of the 4 different costs - the weightings of which are set as global variables within the voice separation class and can therefore be altered by the user.

4.2.4 Assessing Accuracy

In order to judge how accurately the above functions separate voices in an input piece, further classes and methods had to be written to compare the voice separation output to various control outputs. 2 such controls were used - outputs gained from dividing the input piece into its constituent tracks, and the output gained from running the split point separation method on the input file (see section 2.3.1).

Firstly, an alteration was required to be made to the method which created the output files holding the separated voices as these were coming out in a different order to the outputs from the track divided control. This method makes an integer array holding the correct order of the voices in the final separated sequence (ordered by onset time of their first note - as this is how it is done in the track divided files). This is done by finding the first note listed for each voice and storing its on time and allocated voice number in a treemap. These on times are then iterated through in order, the voices associated with each on time being put into an array, which is passed back to the main function which can then use this order to create the output files.

In order to create split point separation outputs, a SplitPointSeparation class was written. This class works very simply by finding the minimum and maximum notes in the piece and creating (noVoices+1) split points (the first and last being the lowest and highest notes found). These points are found by dividing the range by the number of voices. Notes are then assigned to these ranges depending only on their values. This class also outputs voices in a different order to those outputted from the track divided separation, therefore outputs must be run through the sorting method described above.

The comparison class takes input as 2 sets of files which hold the voices to be compared from 2 different separations. Pairs of files with corresponding names are passed to compareVoices, which allocates a score depending on how many notes in the output voice being tested occur at the same time as the same notes in the control voice. Each note in the program's output voice is iterated through and the score is incremented if a matching note is found. When all tracks have been dealt with in this way, the overall percentage score is passed back to the calling function ((totalScore / totalNotes)*100), where totalScore is the summation of all comparisons and totalNotes is the total number of notes which have been used in all these comparisons (the value of which is stored during the running of the class, in order to calculate a final average).

4.3 Voice Separation Accuracy Prediction Implementation

It was at this stage, after voice separation implementation, that the improvement was decided to be made to the user's experience when using the program's functions. By implemeting the machine learning function descibed in section 3.3, the tool's outputs could be of more use to the user in the way that they would know how accurate the separation is likely to be before starting to use it in analysis.

4.3.1 Choice of Attributes

The attributes which were chosen to represent instances are number of voices in the input piece, number of events and length of the piece. Training instances are stored (in a specialist TrainingInstance class) with values for all these attributes and a double value for the accuracy gained from running the voice separation algorithm on the given input piece. These attributes were chosen as factors which largely affect the accuracy of separation. A high correlation could be seen between the number of voices in the input piece and the accuracy gained for that piece, therefore this is a very important attribute to include. It is not immediately obvious that there is any particular correlation between the number of events, length and accuracy, but these attributes represent the complexity of the piece and therefore must have a bearing on the accuracy gained from separating that piece.

4.3.2 Training and Testing

The training set is stored as a text file containing all the required attributes for all training instances. There is a trade off between the amount of training data used and the execution speed of the algorithm. The training instances chosen were a random selection of 22 of the 54 test pieces used - 7 instances each of 2, 3 and 4 voices, and 1 of the 5 voice pieces. The training set is extracted from the text file by way of a string tokeniser. This is done as the program is first run.

The training and testing algorithms were implemented as described in section 3.3. The value of k is set to 3 in the code, but was made a global variable alterable by the user. Due to the nature of the algorithm and the data, and the importance of the number of voices attribute, the values for this attribute had to be accentuated in the data (as the values run between 2 and 5 and are therefore totally outweighed by the high values of the other attributes). This was done by assigning a weighting to each attribute in the main accuracy prediction method (adding a weighting into the distance equation shown in section 3.3), depending on the attribute's influence on classification of instances. A weighting of 1000000 was added to the noVoices attribute in order for the number of voices to be taken more highly into consideration when calculating a prediction.

4.3.3 Error Rate Calculation

The error rate calculation was also implemented as in section 3.3. The calculated error figure can be returned to the user to show by what percentage any predicted accuracy rates are likely to vary from the true figures, calculated using the percentage differences between predicted and true accuracies for the elements of the training set. For example, if the error rate were 0.02 and the predicted accuracy rate were 75% - the true rate could vary by up to 1.5% (i.e. the true rate could range between 73.5% and 76.5%). This allows the user to judge whether the accuracies gained from the voice separation are likely to be high enough for the resulting separations to be of use (as the true accuracies will not be available to the user - see section 5.1).

4.4 Sequence Analysis Implementation

After carrying out the matrix creation and candidate set calculation methods (as described in section 3.4), the number of entries in this set then has to be reduced to output only the most frequently occurring patterns.

Due to the possible inaccuracy of the voice separation functions, sequence analysis has been carried out (in order to be properly tested) on the output of the track separation method (see section 4.2). These voices are combined in series and passed to the sequence analysis methods.

4.4.1 Calculating the Candidate Set

In order to create the candidate set, createCandidateSet moves through the matrix created by createMatrix, creating candidate set entries of the new type CSEntry - containing a pattern (List of Pattern types - including numbers holding the integer representations of the MIDI note and its on and off times), a rep count and a sub count.

The 4 tests shown in section 3.4.2 are carried out (as *if* statements) on each element of the matrix, each case requiring a different function to be carried out - creating a new CSEntry with the required rep and sub counts, or altering the counts of a current record, depending on whether or not the pattern in question is already stored in the candidate set. Extra functions also had to be written to search for patterns in the candidate set (getEntry) and to check for equality between patterns.

4.4.2 Altering the Candidate Set

The process above creates a large amount of unnecessary entries in the candidate set - for example, a large number of patterns with only 1 note. These can be removed simply by specifying a minimum acceptable pattern length. Two further improvements can be carried out, and these improvements have been implemented in the method improveCS.

Firstly, all entries are removed for which the rep and sub counts are equal. This is because such patterns only appear as subsections of larger patterns, and are therefore not repeating patterns in their own right. Secondly, the real repeating frequencies are calculated for each entry. The following formula is used:

$$\frac{1+\sqrt{1+8\times repCount}}{2}$$

This is done as the numbers previously held in the candidate set were not the real repeating frequencies but the number of non-zero matrix elements associated with the given pattern. This number is higher than the real frequency and is therefore reduced to its correct size by the above formula.

Further to these improvements, stated in [4], a number of further heuristics were built into the sequence analysis class, under the method reduceCS. These heuristics arose from researching analysis resources and specialist sequence analysis papers (such as [3] and [11]), by studying the full candidate set output from the program, to decide which patterns were significant and to create sensible heuristics which matched these findings. The first of these heuristics is, as stated above, to remove patterns which were deemed too short to be musically significant (after extensive trials, the number of notes signifying a long enough pattern was 6). Similarly, patterns are discarded if their rep counts are not above 1, as these patterns would only have been detected once in the piece, and stand little chance of being musically significant to that piece. A further function, comparePatterns, was written in order to return a similarity rate between 2 patterns, therefore allowing reduceCS to remove any patterns from the candidate set which are more similar than a given threshold to any other pattern already in the set. The similarity rate is calculated as the percentage of notes in the candidate pattern which match notes in the given pattern already in the set.

Matching notes, in this case, are counted as the same when they have the same value (note and octave) and the same duration. This process is carried out symmetrically, so that patterns already in the set can be removed and replaced if candidate patterns are found to which the current pattern is more similar than the candidate pattern is similar to the current pattern.

Carrying out these alterations on the example sequence shown in figure 3.4, with the minimum length of a significant pattern reduced to 2 (due to the very short length of the example), the candidate set shown in figure 4.1 is produced. This shows that the irrelevant patterns have been removed from the set - firstly the 3 for which the rep and sub counts were equal, and then the remaining 2 which comprised fewer than 2 notes. The process is simple in this case but becomes a lot more complicated as the size of the input piece (and therefore the size of the matrix and candidate set) increase.

Pattern	Rep Count	Sub Count
71-72-69	1	0

Figure 4.1: The result of altering the candidate set shown in figure 3.5

Having been reduced by these processes, the patterns remaining in the candidate set must then be ordered so that the later getSignificantPatterns method can assume the 1st pattern to be the most significant, i.e. the subject of the piece. This ordering is done by difference between each pattern's rep and sub counts. The patterns with the highest difference are those which appear independently most often, and the most significant of these mostly turns out to be the subject of the piece.

The method createPatterns is then called to create separate MIDI files for each of the remaining candidate set entries, by taking the patterns from each candidate set entry and converting them to lists of notes which can then be passed into the convertToMidi method of MidiFunctions. The created pattern files are stored in the Outputs directory of the Analyser main project folder.

4.4.3 Extracting Significant Patterns

At this point in implementation, a change was made to the original plans as to what structural parts should be sought and returned to the user. It was originally said that a number of these parts should be returned, however studying analysis resources (such as [3] and [14]) it was decided that only the significant patterns of the subject, answer and countersubject should be returned. These patterns are generally seen as the staple structural parts of fugue and are therefore by far the most significant in musical analysis. They appear consistently in almost all fugues studied today (although sometimes 'double' fugues can include 2 subjects - but the second of these can just as easily be counted as the countersubject). The subject appears in all fugues and the answer and countersubject may or may not be present, depending on the structure of the particular piece. This difference in structure between pieces has been incorporated into the significant pattern extraction routines, as discussed below. For more detailed explanations of these sections and their roles within a fugue, please see section 2.1.

The getSignificantPatterns method takes the outputted patterns from the Outputs directory and renames the first pattern (pattern0.mid) to be Subject.mid. Figure 4.2 shows an example subject - this has been taken from the same fugue as used in figure 3.1.

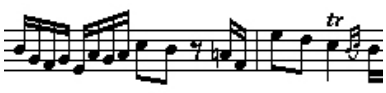


Figure 4.2: An example subject pattern (taken from [14])

The method then searches through the rest of the patterns to find an answer and/or countersubject (or neither, if no such significant patterns are found). In order to find any answer patterns, each pattern is compared to the subject pattern and the most similar pattern is renamed to Answer.mid, assuming the pattern found is more similar to the subject pattern than a given threshold (again, after extensive testing, this was set to 0.7). This comparison requires a different type of similarity to be tested for than that previously mentioned, therefore a new method was written for this purpose. compareNotePatterns works in a similar way to comparePatterns but differs in its definition of notes being the same - to allow for patterns to be counted as similar if they are transpositions of each other (i.e. the same melody but starting on different notes), notes are counted as matching when they have the same duration and the same interval from the preceding note (rather than the same distinct value). For patterns to be similar in this instance, it is also required that the sequences of matching intervals between notes occur in the same order in each pattern (rather than the notes above which may match notes in the original pattern in any order). This is done by finding the correct starting place within the comparison pattern that matches the start of the candidate pattern (i.e. 2 consecutive notes separated by the same interval as the first 2 notes of the candidate pattern). Figure 4.3 shows an example answer.

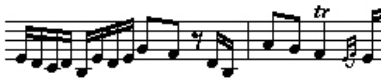


Figure 4.3: An example answer pattern (taken from [14])

Finally, to find the countersubject (if one exists), getSignificantPatterns again goes through the remaining patterns in the Outputs folder, renaming to CounterSubject.mid the longest pattern which least matches the subject and answer (using the compareNotePatterns method above). Another similarity threshold was required here, to signify how *different* a pattern must be to be considered as a possible countersubject (this was set to 0.5, as patterns must be less dissimilar to the subject to be the countersubject than they must similar to the subject to be the answer). Figure 4.4 shows an example countersubject.

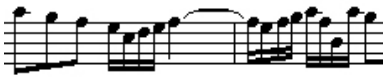


Figure 4.4: An example countersubject pattern (taken from [14])

In order to create a more interactive analysis process between the program and the user, a number of the set thresholds mentioned above were altered to be global parameters alterable by the user. The percentage similarity rates for extracting duplicate patterns and countersubjects, and the number of notes making a musically significant pattern were added alongside the cost function parameters and value of k (for the prediction function) already set as user entered values. The default values for these parameters are set as the values quoted here. It was thought that to make any more of the set parameters alterable by the user would decrease the user friendly nature of the tool and make it more complicated to use. Therefore further parameters (such as the number of repetitions of a pattern making it considerable as a musically significant pattern) remained hard coded into the program code.

4.4.4 A Different Approach

The method by which the matrix is constructed and patterns extracted from this was altered to attempt a different approach to sequence analysis. This involved using the intervals between notes in the input sequence, rather than the current method of using the absolute note values. Changes were required throughout the AnalyseSequence class - creating a matrix of intervals and creating output patterns from the longest sequences of matching intervals in the matrix (assigning note values starting from the middle C value of 60). By altering these methods, patterns could be

extracted which were transpositions of each other - therefore fewer patterns were extracted and the rep counts of these patterns were much higher. However, this method was not preferable to the original method as it would not be possible to extract the required patterns, as the answer is a transposition of the subject, and these patterns could not be worked back into the original, as every transposition of the subject or answer would also be highlighted.

4.5 GUI Implementation

Due to lack of previous experience and knowledge of creating graphical user interfaces, this part of the project was originally developed using the Netbeans IDE [2], in order to master item positioning and event listeners. Once the basics in these areas was learnt, the remaining GUI development could be done by directly manipulating the code.

4.5.1 GUI Classes

A main GUI JFrame, called Analyser, was created through which the user could access all the required functions of the program. Figure 4.5 shows a view of this main interface. Further views can be found in Appendix III.

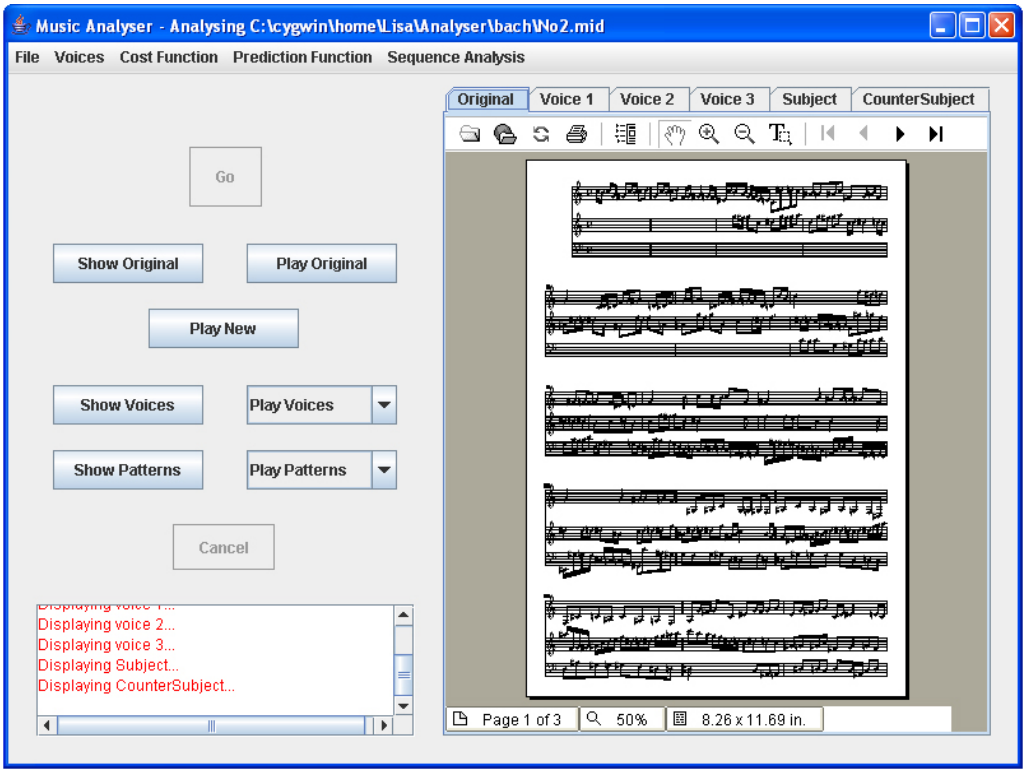


Figure 4.5: The main Analyser interface

This frame was connected to the MusicAnalyser class, which had previously been created to join together, in the correct sequence, the voice separation and sequence analysis functions. This class was connected simply by inserting a Go button which the user could press to start analysis.

The menu bar at the top of the screen was filled with various options, allowing the user to choose an input piece (by way of a JFileChooser within a JDialog) and to change all the required parameters of the program. Separate JDialogs were created to allow the user to change these parameters - these small JDialogs appear in the centre of the screen, tell the user what the current value of the relevant parameter is and prompt them to enter a new value. Due to the nature of GUI programming,

[2] www.netbeans.org/

the passing of such inputs and parameters between classes is not easy to accomplish. Therefore a Globals class was created in which all the required global variables could be stored as static variables in order to be used throughout the remainder of the program. All the classes incorporated in the program were packaged into a package called lb2148.sound, therefore the Globals variables became accessible only by classes within this package.

4.5.2 Creating Outputs

Audio

The audio outputs available to the user are the original file, the voices and patterns extracted and a new file highlighting the patterns within the original. When a button is clicked, or the name of a voice or pattern selected from the available drop-down lists, the PlayMidiFile class is called, passing the name of the required MIDI file. This class creates a Sequencer object (from the javax.sound.midi package) which plays the given MIDI file via the running machine's set MIDI sequencer. This class was set to extend Thread so any number of such threads could be run at once, and allowing the user to cancel playback of a file when desired. A Cancel button was added to the Analyser class which, when clicked, stops all currently running threads. All threads are stored in a local array on creation and are therefore available to the method which is run on clicking the Cancel button.

A method was written within the AnalyseSequence class to create the new file mentioned above. This searches through the Outputs directory to find which patterns have been created for the input file in question (i.e. any combination out of Subject.mid, Answer.mid and CounterSubject.mid), then iterating through all notes in the input file to find any occurrences of sequences of notes matching the patterns found. The matching algorithm tests all notes in the piece for equality with the first note of the pattern(s) found. If a matching note is found, a local start variable is set (holding the position in the input piece of the matching note found) and the following n notes are iterated through (where n is the number of notes in the matched pattern) to test whether or not a sequence has been found which fully matches the given pattern. If the sequence is found to be matching, the algorithm alters the velocity of the matching n notes (starting at the index specified by the start variable) so the pattern appears highlighted in the new file created.

Visual

The next step was to produce the required visual outputs. This was done using the previously mentioned program, Lilypond. Installation of Lilypond requires root access, so it was required that this be done on a Windows machine where this access was granted, therefore all the visual output classes and functions are specific to the Windows environment. The conversion program midi2ly must first be called in order to convert the given MIDI file into Lilypond's .ly format, then the output from this conversion can be sent to the main Lilypond program for final conversion into PDF format. As Lilypond is written for Linux, the Linux environment program for Windows, Cygwin[3], was downloaded through which all calls to the Lilypond programs have to be made.

The class ConvertToPDF was written to call the required Lilypond programs via Cygwin. This was done by creating new Processes for the current Java Runtime, running the cygwin.bat batch file via a Windows "cmd" call. Once this cygwin bash shell has been opened, a BufferedWriter object can be created to write a string to the input of this process (process.getInputStream()) to convert the given input file - "midi2ly -o {location of Outputs folder/output .ly filename} {input file name}". In order to wait for completion of this process, the ConvertToPDF code creates a new File object with a name equal to that of the new filename arising from the conversion process. It then waits until this file exists in the specified folder before continuing with the next process call.

[3] www.cygwin.com

Once the .ly output file exists in the Outputs folder, another call can be made to the cygwin shell to convert this file into PDF format, using the Lilypond program - “lilypond {.ly filename}”. There are no options in the Lilypond program to specify the location of the output PDF file (and intermediate files created during conversion - text, DVI, PS etc.) therefore, on completion of this command (checked by using a similar method to that above, for midi2ly) another call must be made to remove the intermediate files and move the final outputted PDF file from Lilypond’s default location to the Outputs directory. These actions are carried out using the commands “rm *.log *.tex *.dvi *.ps *.txt” and “mv {output .ly filename}.pdf {location of Outputs folder}”. In order to check when these processes have completed, ConvertToPDF first counts the number of files in the directory into which Lilypond stores its output, then waits until this number of files + 1 are to be found in this directory (after the intermediate files have been removed) then waits until the original number of files are present, signifying that the PDF file has been successfully moved.

Due to the slow running speed of the Lilypond programs (using Cygwin) a special class CreatePDFs has been created which calls ConvertToPDF for each output PDF file required. This class can be run separately from the main Analyser class, to allow the user to continue using the tool while the outputs are created. See further work (section 6.2) for further discussion on this subject.

Having been created, the next step was to find a way to display these output PDF files in the Analyser Frame (as shown in the right hand side of the screenshot shown above). A JAR file was downloaded from Adobe[4] which provides methods to convert PDF files into GUI Components, called Viewers. By including the JAR file in the compilation classpath, these methods can be used to create a Viewer for each output PDF file which can then be added to the JTabbedPane area, with an appropriate tab label, at the right of the Analyser Frame. Multiple tabs can be added to this pane at any one time and a scroll bar appears at the top right corner of the pane when more tabs have been added than are visible at a time (as shown in Figure 4.6).

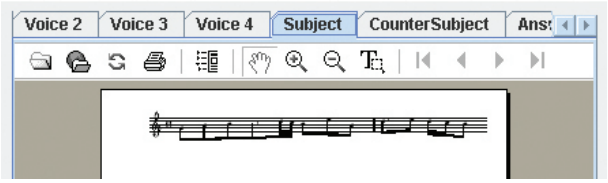


Figure 4.6: Viewing outputs on the tabbed pane

This chapter has gone through the implementation process in detail, covering the full process - from the intermediate representation created, through voice separation, accuracy prediction and sequence analysis, to GUI creation. The next chapter reports the results gained from testing the implemented tool.

[4] www.adobe.com/products/acrviewer/acrvdnld.html

Chapter 5

Results

This chapter gives detailed information about the testing which was carried out on the tool, and reports and discusses the figures which were gained from this testing.

Testing in this project lay in the main 4 areas explained throughout - voice separation, accuracy prediction, sequence analysis and the user interface. Results from these sets of tests are shown and explained in detail below. Tests were carried out on 3 different sets of input pieces - Bach's 15 2-part and 15 3-part inventions and the 24 fugues from Book 1 of the Well Tempered Clavier (limited to only the fugues for sequence analysis). Due to the differences in these types of piece, the results varied.

5.1 Voice Separation Testing

Each of the input pieces were run through the tool multiple times, with different values of the cost function weightings set, and the voice separation accuracy results noted. It must be noted here that the voice separation accuracy which is returned by the tool is only for testing and debugging purposes. This figure is gained (as explained in section 4.2.4) by comparing the outputted voice file sequences with the sequences gained from splitting the input piece into its constituent tracks (as the files used for testing are special type 1 files which allocate a track to each voice in the piece). This would not be possible to do for most input files therefore, in future, the tool would not be able to return these accuracy rates. The figures gained for split point separation were also done in this way, by comparing outputted voices to those gained from track divided separations.

All the results found here are approximate as they have been taken from a single run of the program. This approximation is due to the random nature of the voice separation algorithm (see section 4.2.1), meaning that the algorithm can work with differing levels of accuracy between runs on the same input piece. The results shown for split point separation, however, do not differ as the split point algorithm contains no random element. Testing of the extent to which accuracies differ between runs is carried out below.

Overall, the tool managed an accuracy of 73.42%, taking into account all of the input pieces. This is a very respectable figure when you take into consideration the complexity of some of the pieces used and the accuracies gained from the simplest form of voice separation - split point separation (see section 2.3.1), for which the average accuracy was only 44.8%. Testing was started using the simple, 2-part inventions then built to a higher level of complexity via the 3-part inventions to the fugues (comprising 2 to 5 parts). This was done to test the tool on a variety of pieces at different levels of complexity. As explained in section 2.1, inventions are a simple form of counterpoint, and fugues are the most complex form. This is due, in part, to the number of voices in the piece, but is also due to the way these voices interact to create the overall piece. The inventions involve few voice crossovers (pitches of voices overlapping) and, as they are designed as practice studies, they concentrate more on the voices/parts as separate entities than as interacting in a whole.

This difference between types of input can clearly be seen in the results. Using split point separation (which works best on simple, non-crossing pieces), the simple 2-part inventions produced an average accuracy of 50.27%, whereas the more complex fugues produced a lower average accuracy of 44.75%. The tool's average accuracy rates showed a higher degree of difference between results for the different types of input - starting at 90.09% for the 2-part inventions, going down to 67.06% for the 3-part inventions and finally ending on 66.97% for the fugues. It would be expected to see a smaller difference between the 2-part and 3-part inventions, and a larger difference between these and the fugues, however the 2-part inventions appear by far the most successful, leaving the 3-part inventions and the fugues roughly the same. This shows, perhaps, that the number of voices in the piece is a lot more important than first thought (as half of the fugues also comprise 3 voices) and the innate complexity has less of a bearing on voice separation accuracy than imagined.

The next stage of analysis of the results was therefore to see if big differences could be seen in the accuracies depending on the number of voices present in the input piece. Table 5.1 shows average accuracies for 2, 3, 4 and 5 voice pieces, and compares these to the results gained from the simplest form of voice separation.

No. Voices	Accuracy	Split Point Accuracy
2	90.62%	52.98%
3	71.82%	43.81%
4	56.09%	34.61%
5	43.25%	43.08%
Average	65.45%	43.62%

Table 5.1: Accuracy rates by number of voices in input piece (compared to split point separation)

Firstly, this shows that the implemented method of voice separation greatly improves on the simplest form, returning an average 21.83% higher accuracy rate. The amounts by which the figures differ decrease as the number of voices increases, showing that the algorithm used by the tool becomes less effective as the number of voices increases - the difference at the 2 voice level being 37.64%, dropping right down to 0.17% at the 5 voice level. Further proof of this is shown as the accuracy rates gained by the implemented algorithm steadily decrease as the number of voices increases, showing the number of voices to be the deciding factor in voice separation accuracy. This can also be seen in results gained from per voice analysis - returning accuracy rates for each voice within a piece. Table 5.2 shows a small sample of per voice analysis results carried out at an earlier point during implementation.

Input Piece	Overall	Voice 1	Voice 2	Voice 3	Voice 4	Voice 5
2-part invention 1	90.04%	92.86%	87.22%			
2-part invention 11	97.86%	98.75%	96.96%			
3-part invention 12	78.32%	88.97%	83.33%	62.67%		
fugue no. 15	76.68%	97.51%	85.74%	46.78%		
fugue no. 1	57.84%	86.00%	69.30%	43.18%	32.87%	
fugue no. 18	61.74%	94.17%	70.74%	52.22%	29.82%	
fugue no. 22	48.92%	90.28%	64.43%	19.84%	41.67%	28.39%

Table 5.2: Accuracy rates per voice

In all cases tested at this point, the first voice was extracted with a very high accuracy rate (at an average of 92.65%) and subsequent voices were extracted with significantly lower accuracies - from 79.67% for the second voice down to 34.79% for the fourth. An explanation for these figures could be the fact that errors appearing in the first voices are propagated through, leaving very poor

accuracy rates for the later voices. For example, if a note is mistakenly included in the first voice, the lack of this note for inclusion in its correct voice (say, the second voice) could completely throw the program and cause it to go off on the wrong track. To try and overcome this problem, the program was changed to restart the algorithm at regular intervals during the separation process. Unfortunately, this did not overcome the problem and in fact made it worse as, having been restarted when no error had been made, the algorithm would get confused as to which note in the new slice should be allocated to which voice (as no previous information was available to judge the best allocation). This issue is noted in section 6.2 - further work.

In order to test the amount by which accuracies differ between multiple runs of the program on the same input piece, a subset of the test set of pieces was taken. This set comprised randomly chosen pieces in the following categories: 2 2-part inventions, 2 3-part inventions and 6 fugues (1 with 2 voices, 1 with 3 voices, 3 with 4 voices and 1 with 5 voices). Voice separation was carried out 5 times for each of these pieces, producing the results shown in table 5.3. The standard deviations were calculated using Microsoft Excel's STDEVA function which estimates standard deviation based on a sample, using the following formula:

$$STDEVA = \sqrt{\frac{n\sum x^2 - (\sum x)^2}{n(n-1)}}$$

Input Piece	**No.**	**Accuracies**					**Average**	**Standard**
	Voices	**1**	**2**	**3**	**4**	**5**	**Accuracy**	**Deviation**
2-part invention 2	2	92.89%	93.03%	92.45%	92.31%	92.89%	92.71%	0.31%
2-part invention 6	2	78.88%	78.88%	78.88%	78.88%	78.88%	78.88%	0.00%
fugue no. 10	2	98.64%	98.64%	98.64%	98.64%	98.64%	98.64%	0.00%
3-part invention 4	3	49.10%	50.54%	50.36%	46.93%	49.46%	49.28%	1.44%
3-part invention 10	3	50.64%	43.91%	42.63%	43.27%	43.27%	44.74%	3.33%
fugue no. 9	3	80.90%	81.31%	79.81%	80.35%	79.67%	80.41%	0.70%
fugue no. 1	4	60.49%	60.22%	59.67%	61.17%	62.13%	60.74%	0.95%
fugue no. 16	4	56.93%	53.33%	56.67%	54.13%	53.07%	54.83%	1.85%
fugue no. 24	4	50.61%	51.77%	51.33%	52.43%	53.82%	51.99%	1.22%
fugue no. 4	5	42.51%	41.00%	39.41%	39.33%	39.56%	40.36%	1.38%

Table 5.3: Standard deviations in voice separation accuracy

This shows an average overall standard deviation of 1.12%, which shows that accuracy rates are unlikely to differ significantly between runs of the program. The average accuracy produced by the voice separation algorithm is 73.42%, therefore the previous test shows that the true average figure might deviate by 1.12% from this average (i.e. from 72.3% to 74.54%). These figures also show that the algorithm is likely to produce larger ranges of accuracies for pieces with larger numbers of voices.

Experimentation was carried out using the cost function weightings, which are available to be changed by the user. These 4 weightings, variable between 0 and 1, provide a vast test space - therefore only a very limited amount of this space was able to be covered in testing. This means that it may well be possible to find a better combination of weightings which would produce overall better results. The combination suggested in [5] is all weightings set at 0.5. Using this combination, the accuracies by type of input were gained as follows - 90.09% for 2-part inventions, 67.06% for 3-part inventions and 66.97% for fugues. After extensive experimentation, this was found, overall, to be the best combination for the inventions. The best overall combination for the fugues, however, was found empirically to be 0.9 for the pitch and chord weightings, 0.2 for the gap weighting and 0.5 for the overlap weighting. Faults were found in the original voice outputs which were, to some extent, remedied by these altered figures - large leaps in pitch within a voice (and within a chord) were reasonably common and there were very few gaps/rests in voices,

making it seem like the algorithm was searching for a note to allocate to a voice when really none was necessary, therefore making an allocation error. By greatly increasing the pitch and chord weightings and decreasing the gap weighting, this alters the level to which the algorithm tries to allocate notes to avoid these features appearing within voices (pitch and chord ranges become very important to avoid and large gaps become much less important). Re-running the tests using these weightings showed a 2.12% rise in accuracy - to 69.09%, showing that improvements are possible when a problem in the outputs has been successfully identified, therefore further improvements would no doubt also be possible.

One final issue that was noticeable from the outputs, and aids in explaining why the accuracy rates for the 3-part inventions were lower than expected, was that some of the 2-part and all of the 3-part inventions start with multiple voices sounding together. The voice separation algorithm could misallocate the notes in the first slice, and then find it hard to readjust onto the right track for the voices further into the piece. The way the majority of the test pieces start, with just one voice, means that the algorithm can use this sequence of notes (even if only very short) as a base for one voice from which it can extend to the remaining voices throughout the remainder of the piece.

5.2 Voice Separation Accuracy Prediction Testing

The prediction function was tested as voice separation was tested - noting the predictions which were offered by the tool against the actual separation accuracies which were returned. Due to the random nature of the voice separation algorithm, mentioned above, it was difficult to calculate the true accuracy of the prediction function, therefore the results shown below are only approximate, average figures.

While running these tests it was noticed that the length attribute was adversley affecting the accuracy of the predictions. This problem may have been due to the very large figures which are associated with length, however when these figures were reduced, so that the sizes of the figures were in better proportion to the sizes of the voices and events figures, the error rates were still higher. It was therefore decided that this attribute should be removed from the accuracy prediction function. It was also decided that the training instances that had been taken from the 3-part inventions should be removed, due to the voice separation issues discussed above that made the separation accuracies disproportional to those gained using the true fugue inputs. These pieces were also removed from the tests described below.

The only alterable parameter in this function is that of k - the number of neighbour points to use when calculating a prediction. Using the error rate feature, it was a lot easier to reduce the test space, depending on the average expected accuracies of prediction using the given values of k. Table 5.4 shows the error rates calculated for many different values of k.

K	Error
1	0.1171 (11.7%)
2	0.1011 (10.1%)
3	0.0827 (8.3%)
4	0.0826 (8.3%)
5	0.0894 (8.9%)

Table 5.4: Average error rates for different values of k

Any values of k above 5 were ignored as the accuracy rates became unacceptably low, therefore the choice appeared to be between 3 and 4, which returned very similar error rates. The functions will return an average of the 3/4 most closely matching training instances in each case. Both values were tested, using the 19 test pieces not included in the training set (therefore not creating biased results), creating the overall results shown in table 5.5.

Input Type	K	Error
2-part inventions	3	8.21%
2-part inventions	4	8.28%
fugues	3	9.95%
fugues	4	11.38%

Table 5.5: Average prediction error rates for different values of k

These results show an average error rate of 8.94% for k=3 and 9.58% for k=4, which show the prediction algorithm worked slightly worse in practice than was shown by the error rate prediction, and that k=3 was slightly better than the expected slightly worse. The errors are shown as a percentage of the prediction, therefore the error rate 8.94% on a prediction of, say, 70% means the true accuracy could range between 63.74% and 76.26% - a small enough range for the prediction to be useful to the user.

Although the predictions returned by this algorithm are good, the errors which do occur can probably be attributed to the tests used and the nature of the data being used. The test data set became small once the training set instances were removed (leaving only 19 cases), meaning that the average figures obtained may not be truly indicative of the accuracy of the prediction function. Also, the kind of music used can be very variable and unpredictable. That is, a fugue could contain only 2 voices, but these could be highly complex, for example crossing over in pitch a large number of times. Normally, a 2 voice fugue is less complex than fugues with more voices, therefore the training data would reflect this and the input piece would be an anomaly, and could possibly cause an inaccurate voice separation accuracy prediction. Also, the accuracy rates themselves are changeable (as discussed above, section 5.1) therefore the training data is not as precise as would be desired, and even if a test piece is also included in the training set, its accuracy from a previous test would be returned by 1-nearest neighbour and the resulting accuracy would probably be different (although by no means dramatically). Improvements to this prediction function are discussed in section 6.2.2.

5.3 Sequence Analysis Testing

The sequence analysis functions were tested only with the fugues, not with any of the inventions. This was because inventions are not structured the same as fugues (see section 2.1), are not studied in the same way or with the same frequency, and the tool was designed to specifically analyse fugal music. Also, testing in this particular area is very time consuming, so narrowing down the test pieces used was necessary. However, a small amount of experimentation was carried out using the inventions to see, out of interest, how far the fugal analysis techniques of the tool will stretch into other forms of counterpoint. The results of this testing will be detailed after reports on the in-depth testing of the 24 fugues.

Two sets of tests were required here as the results could rely quite heavily on the accuracy of the separation of the voices inputted (as input is taken as the extracted voices all joined together end to end in one sequence). Therefore, the tool was tested both using the output from the voice separation functions and using the sequences created from separating the voices using their individual tracks. Using these different methods it was possible to assess the tool as a whole and to assess the sequence analysis functions as a standalone feature.

For each method, the same test processes were carried out - first assessing the tool's analysis of the input piece using the default parameters (minimum pattern length and similarity rates for duplicate patterns and countersubjects), then altering these parameters to try and obtain a more accurate analysis.

5.3.1 Track Separated Input

Table 5.6 shows the percentages of subject, answer and countersubject patterns returned by the tool before and after parameter alterations. The figures shown for countersubjects are as a proportion of the number of pieces in which a countersubject appears. Assessment of the patterns was done with help from the Bach fugal analyses of [14].

	Subjects	**Answers**	**Countersubjects**	**Overall**
Before Parameter Changes	66.67%	33.33%	56.25%	52.08%
After Parameter Changes	83.33%	50.00%	56.25%	63.19%

Table 5.6: Patterns found using track separated input

In this test, patterns were counted as matching when they deviated from the real pattern by some kind of short or medium length sequence. Before parameter changes, all but 4 of the 20 matching subjects were exact, missing or including an extra 1 to 4 notes. Of the remaining 4, 3 were named incorrectly (1 as the answer and 2 as the countersubject) and 1 was missing a longer sequence at its beginning. Of the 8 answers found, 3 were named as the subject (in cases where no subject was found or the subject was wrongly named as the answer) and 1 was named as the countersubject. 4 were matched exactly and the remaining 4 were missing short sequences, either from the beginning or end. The countersubjects were slightly better matched, with only 2 out of the 9 found (out of 16 total) incorrectly named as the subject. 4 were not complete matches but 5 were exact matches. In quite a few cases, countersubjects were returned where none existed.

8 out of the 24 input pieces required (or responded to) parameter changes, quite significantly improving the success rate of the sequence analysis functions. These alterations are shown in table 5.7 - subject extracted, for example, means that no subject was extracted before the parameter changes were made and one was successfully extracted using the new parameters given. The extra countersubjects mentioned above could all be removed by lowering the countersubject similarity rate.

Fugue No.	**Pattern Length**	**Duplicate Similarity**	**CS Similarity**	**Result**
4	4	0.7	0.5	3 extra notes removed from end of subject
8	8	0.5	0.5	subject and answer extracted (incorrectly named)
9	6	0.65	0.5	answer extracted
10	20	0.7	0.5	subject extracted (incorrectly named)
12	8	0.7	0.5	subject extracted (2 missing notes at start)
13	6	0.4	0.5	answer extracted (1 missing note at start)
15	10	0.7	0.5	subject extracted (only 1st half)
19	6	0.4	0.5	answer extracted (1 missing note at start+2 at end)

Table 5.7: Alterations made to parameters for track separated input

By raising the minimum pattern lengths for fugue numbers 8, 10, 12 and 15, the subjects of these pieces were correctly returned where they hadn't been returned with the default length of 6. This is because these particular fugues contained subjects which were significantly longer than average, therefore setting the minimum pattern length too short caused shorter, less significant patterns to be incorrectly returned as the subject. Similarly, lowering the pattern length for fugue 4 enabled the function to pick out the unusually short subject where it had previously been incorrectly extended to create a pattern over 6 notes long (as it consists only of 5 notes). The answer patterns were successfully extracted for fugue numbers 8, 9, 13 and 19 by lowering the duplicate similarity rate - allowing patterns less than the default 70% similarity to the subject to be returned as the answer. Unusually, in fugue number 9, the countersubject was included at the end of both the subject

and answer. This will be the case as the countersubject runs in parallel to the subject playing in a different voice (see section 2.1) and therefore is likely to follow directly on from the subject in a voice. It is therefore surprising that this was the only case of such inclusion of the countersubject within the subject pattern. It must have been due to the unusually high frequency of occurrence of the countersubject in this particular piece, and its repeated close proximity to the subject pattern.

No success was gained in increasing the proportion of countersubjects correctly returned.

For some of the cases above where the correct patterns were not found (in any combination), the contents of the Outputs folder showed that the error was not that of the pattern finding but of the significant pattern extraction. For example, in sequence analysis of fugue number 12 (after parameter changes), the full subject pattern can be found in the extracted file “pattern5” (the returned subject pattern was missing 2 notes at the start), and the correct countersubject pattern was also to be found in the list of extracted patterns before candidate set reduction was carried out. Also, in fugue number 21, subject and answer patterns were returned but the countersubject was not - the correct countersubject pattern could, however, be found in the extracted file “pattern12’. It would be impossible to find a set of heuristics which would correctly extract all the required significant patterns for all input pieces, therefore the current heuristics work extremely well at extracting the subject patterns. However, these heuristics could probably be further altered to increase the success rate of extracting answer and countersubject patterns.

It is also possible to see, from certain examples, how missing note errors might occur. For example, in analysis of fugue number 13 (after parameter changes), the correct answer pattern was returned but was missing the first note. Figure 5.1 shows a relevant section of the input piece from which this pattern would have been extracted. It can be seen that the first and second voices share a note, this forming the beginning of the answer (in the second voice). This note can only be allocated to one of the voices, and is allocated to the first voice, therefore returning an answer pattern with the first note missing, after sequence analysis is run on the outputted voices.

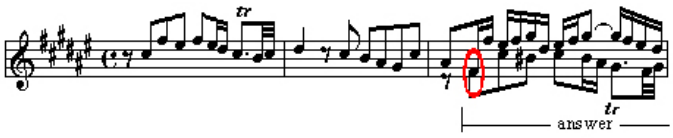


Figure 5.1: Example reason for missing notes in extracted patterns (sheet music taken from [14]

5.3.2 Voice Separated Input

The results from these tests didn’t differ by as large an amount as was expected. Again, table 5.8 shows the percentages of subject, answer and countersubject patterns returned by the tool before and after parameter alterations.

	Subjects	**Answers**	**Countersubjects**	**Overall**
Before Parameter Changes	54.17%	25.00%	31.25%	36.81%
After Parameter Changes	75.00%	33.33%	31.25%	46.53%

Table 5.8: Patterns found using voice separated input

Only 3 fewer subject patterns were returned in this test than that above. However a larger proportion of these patterns differed from the true patterns by more than a couple of notes, due to the errors in voice separation causing melodic lines to be fragmented so parts of voices containing significant patterns were less likely to be found intact. 8 of the patterns returned differed from the true patterns by more than 4 notes, leaving only 5 which were more exact. Only 1 out of the 6 answers returned differed from the true pattern by 1 to 4 notes (fugue number 18), and only 1

of the 5 countersubject fell into this category, although this 1 was exact. One advantage that was seen within this set of tests was the naming of the returned patterns - only 1 out of the 24 patterns returned was incorrectly named, compared to 8 out of the 33 incorrectly named using track separated input.

In this case, 7 of the 24 input pieces were improved by altering the parameters. These alterations are shown in table 5.9.

Fugue No.	Pattern Length	Duplicate Similarity	CS Similarity	Result
1	6	0.5	0.5	subject extracted (incorrectly named+ 3 missing notes at start)
4	4	0.7	0.5	subject extracted
9	6	0.5	0.5	answer extracted (2 missing notes at start)
10	15	0.7	0.5	subject extracted (incorrectly named+ some notes missing)
11	7	0.2	0.5	full answer extracted (incorrectly named)
17	4	0.5	0.5	subject extracted (1 missing note at start)
19	6	0.2	0.5	answer extracted (1 missing note at start+2 at end)

Table 5.9: Alterations made to parameters for voice separated input

A number of these alterations were the same or similar to those used above, and therefore require no further explanation. Altering the minimum pattern lengths was again successful in extracting the subject and answer patterns - with 2 pieces requiring lowering and 2 pieces requiring increasing the minimum. The answer pattern was successfully extracted in 3 cases by lowering the duplicate similarity rate, although these patterns tended to be returned with notes missing. Again, no improvements were able to be made to the success in returning correct countersubjects.

5.3.3 Inventions

A very quick analysis of the inventions, without altering parameters, using the track divided voice separations and only judging rough matches of the subject pattern, showed the 2-part inventions to return 77.78% of subjects correct (or reasonable) and the 3-part inventions to return a rate of 71.43%. These results are encouraging and show the tool's ability to aid with sequence analysis of other forms of counterpoint as well as fugue.

5.4 GUI Testing

Testing of the user interface was done concurrently with implementation. This way, the requirements of the user were made as simple as possible. A final round of testing was also done after completion of the tool, this being carried out by independent 3rd party users of varying computing and musical experience levels. The results of this testing were very positive, with the tool receiving an overall Excellent to Good rating throughout all categories. The questionnaires completed by the testers, and the user guide shown to them, can be found in Appendix IV and III.

This chapter has reported on all results gained from testing the tool and discussed these results, giving or suggesting reasons for the patterns and trends found within the data. The overall figures gained were as follows: 73.42% accuracy for voice separation (compared to 44.8% for simple split point separation), 8.94% accuracy for voice separation accuracy prediction, and 63.19% of significant patterns successfully extracted (after parameter changes). The conclusions arising from these results can be found in the following section.

Chapter 6

Conclusions and Further Work

This chapter draws the final conclusions from the project, details some of the main problems which were encountered and suggests further work that could be carried out in the areas covered throughout this paper.

6.1 Conclusions

The tool created carries out the required functions and is very easy to use, as specified in the statement of problem. It provides a graphical interface which is very easy to navigate, and parameters for the different functions are easy to locate and alter, quickly and easily seeing what effect these changed parameters have on the outputs of the tool. The format of its outputs enable the user to both hear and see the original file, its constituent voices and significant patterns, and to hear what the original piece sounds like with the significant patterns highlighted.

The tool provides a very good level of accuracy in voice separation of input pieces incorporating 2 voices, and good levels in input pieces of 3 voices, showing a large improvement from the simplest form of separation. The rate by which these levels decrease for further voices, however, makes the tool not acceptably accurate in pieces with more than 3 voices. The sequence analysis results are very good, regardless of the number of voices in the input piece. The success of this sequence analysis process lies mainly in the extraction of the subject pattern of a piece, with the answer and countersubject patterns less accurately extracted. The difference in accuracy between using input to sequence analysis from the voice separation or track divided separation was not very large, showing the sequence analysis works very well as a standalone function and only slightly less well when used as part of the tool as a whole.

In conclusion, the tool fully carries out its required aesthetic and usability functions and fully carries out its functions for voice separation and sequence analysis on pieces with 3 or fewer voices. When pieces with more voices are used, voice separation results become inaccurate but sequence analysis results remain very good (and only slightly less good if the input is taken from the output of voice separation).

6.2 Further Work

Although the tool fulfils all the required functions, there are still a few limitations and a few areas in which the tool could be improved, given more time. One such improvement is suggested in the design chapter (section 3.1).

6.2.1 Voice Separation Improvements

Firstly, improvements are required in the voice separation function as there are some limitations in this function, as stated above. Improvements could be to create the separations per voice, i.e. iterating through all notes in the input piece once for each voice. This way, notes in the piece could conceivably be included in multiple different voices. This could be an important improvement as, currently, the accuracy of the first voice of separation is very high, yet small inaccuracies are propagated throughout the voices, ending up with quite low accuracies for the later voices.

It would probably also be possible to find a better overall combination of weightings for the cost function - allowing the tool to create voice separations at a better accuracy rate than it does currently. [5] suggests that these parameters could be worth changing even *within* a piece, therefore showing the level of sensitivity of the results to these parameters. A large amount of experimentation and testing would, however, be required to find a better overall combination. It may also be possible to automatically find the best combination for the input piece in question, by way of some form of more complex machine learning process (see below).

6.2.2 Voice Separation Accuracy Prediction Improvements

The prediction function could be extended to provide more accurate predictions. This would involve increasing the number of attributes used in training and testing, to allow the function's methods to more accurately model the environment in which the new test instance is being analysed. Such attributes would be the user set parameters - the cost function parameters, which should be considered when predicting the voice separation accuracy, as these parameters largely alter this accuracy. This alteration would not be difficult to implement, but would be very time consuming in data collection. This prediction function could then be further altered to find the optimal combination of alterable parameters (i.e. the cost function weightings) for the test piece in question, using extensive training data.

6.2.3 Lilypond Issues

A number of problems were introduced into the project by the use of the Lilypond program, and its midi2ly conversion program. Limitations of the midi2ly program caused problems in different areas. Firstly, it doesn't cope well with files incorporating many voices, especially when the file is a type 0 file i.e. when producing the output showing the original input file. Therefore, the given implementation copies the type 1 (or type 2) input file into the Outputs directory so it can be more correctly converted, although sometimes voices are still not included in the outputted pdf file. Secondly, it doesn't allow for note colouring, which would be required to provide a visual representation of the midi file which is created and given to the user, highlighting the significant patterns found in the original piece.

The speed of running the midi2ly and Lilypond programs is very slow, and therefore the multiple calls to these 2 programs required to produce the output of the tool greatly slows down its overall running time. Ideally, the CreatePDFs class (described in section 4.5.2) would not exist as the Analyser class would be able to call the midi2ly and Lilypond programs, for the required file, whenever a button was pressed by the user asking to be shown visual output.

To attempt to overcome some of these problems with Lilypond, a possible extension to this project could be to create a conversion program (or class) similar to midi2ly, which would be able to compile MIDI (or preferably Note object) information into .ly format. This would allow the shortcomings of midi2ly to be overcome as well as speeding up the whole operation (as constant calls to the Cygwin command line would no longer be required).

The tool is almost totally platform independent, depending only on the ability to install Lilypond. At the programs start-up, it detects the location of the Analyser directory and sets this as its home path. Therefore, regardless of its position within a file system, the Outputs directory will always hold the required files, and will therefore be accessible to the Analyser class in order to load input and display output. The issue with Lilypond is that it must be installed with root access, and it stores its outputs in different places depending on the type of operating system being used. If root access were available, Lilypond could be installed on Linux and the tool could run fully, with changes made to the relocation of Lilypond output pdf files and with the Lilypond processes being called by direct command line accesses (i.e. process.exec('midi2ly inputfile') rather than via Cygwin, as required for Windows.

6.2.4 Reusable Functions

Finally, the library of functions written in MidiFunctions.java could be a useful source for further studies into this area. These functions would significantly reduce time taken at the start of a project, familiarising oneself with the MIDI file format, reading and writing MIDI files and converting the un-modifiable Java representation into a easily modifiable intermediate representation.

This chapter has reported overall very positive conclusions from the project. It has highlighted areas in which the tool under-performs and has made suggestions as to how these problems can be alleviated by further work.

APPENDIX

Bibliography

Bibliography

[1] E. Cambouropoulos. From midi to traditional musical notation. In *Proceedings of the AAAI Workshop on Artificial Intelligence and Music: Towards Formal Models for Composition, Performance and Analysis.*, Austin (TX), USA, 2000.

[2] E. Chew and X. Wu. Separating voices in polyphonic music: A contig mapping approach. Technical report, University of Southern California, 2004.

[3] W. Cole. *The Form of Music*, chapter 12 and 13. The Associated Board of the Royal Schools of Music, London, 1997.

[4] J. L. Hsu, C. C. Liu, and A. Chen. Efficient repeating pattern finding in music databases. In *Proceedings of the 1998 ACM 7th International Conference on Information and Knowledge Management.*, 1998.

[5] J. Kilian and H. Hoos. Voice separation - a local optimization approach. In *Proceedings of the 3rd International Conference on Music Information Retrieval.*, 2002.

[6] W. Lovelock. *A Student's Dictionary of Music*. William Elkin Music Services, Norfolk, 1986.

[7] Alfred Mann. Translation & edition of original work by J. J. Fux. *Gradus Ad Parnassum*, page 36 and 100. W. W. Norton, New York, 1965.

[8] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. Technical report, University of Helsinki, Finland, 1997.

[9] D. Meredith. Pitch spelling algorithms. In *Proceedings of the fifth triennial ESCOM conference*, Hanover University of Music and Drama, Germany, 2003.

[10] D. Meredith, G. Wiggins, and K. Lemstrom. Efficient algorithms for translation-invariant pattern discovery in multidimensional datasets. Technical report, City University, London, 2001.

[11] D. Meredith, G. Wiggins, and K. Lemstrom. Pattern induction and matching in polyphonic music and other multidimensional datasets. In *Proceedings of the 5th World Multi-Conference on Systemics, Cybernetics and Informatics*, Orlando (FL), USA, 2001.

[12] Tom M. Mitchell. *Machine Learning*, chapter 8. McGraw Hill, 1997.

[13] P. Y. Rolland. Discovering patterns in musical sequences. *Journal of New Music Research*, 1999.

[14] Timothy Smith. The canons and fugues of j.s. bach - http://jan.ucc.nau.edu/∼tas3/bachindex.html (fugal analyses at /∼tas/wtc.html). University of Northern Arizona.

Musical Glossary

The following definitions have been taken from Lovelock's "A Student's Dictionary of Music" ([6]):

Answer - In *fugue*: . . . is in essence the *subject transposed* up a 5th (or down a 4th), sometimes slightly modified as a 'tonal answer' in order to preserve the basic tonality. . .

Canon - The literal (Greek) meaning is law or rule. A canon is a composition in which a *melody* stated in one part is imitated exactly, after a certain time interval, by one or more other parts.

Chord - A combination of three or more sounds simultaneously.

Chromatic - Literally 'coloured'. Foreign to the *scale* (*major* or *minor*) of the *key*.

Chromatic Chord - One which contains, among the notes which form it, one or more notes foreign to the *scale* of the *key*.

Clavier - 1. A keyboard instrument. 2. The keyboard of an instrument.

Clef - A sign placed at the beginning of the *stave*, which definitely fixes the *pitch* of the line on which it stands and thus fixes that of all the other lines and spaces. E.g. 𝄞 and 𝄢.

Counterpoint - A combination of two or more *parts*, independent in *melody* and/or rhythm.

Countersubject - In *fugue*: . . . a countermelody. . .

Dimunition - The statement of a theme on notes of shorter, usually half, value.

Dominant - The fifth degree of a major or minor scale, so called because it is the next most important note after the *tonic*.

Episode - 1. A portion of a movement lying between two statements of the same material, with which it is contrasted in *key* and/or style. 2. In a *fugue*, a connecting passage, generally with *modulation*, and most frequently based on some part of the subject-matter, but without an actual statement of the *subject* itself.

Exposition - The first part of a movement (sonata or *fugue*) in which he subject-matter is 'exposed' or set forth. In *fugue*: . . . begins with the statement of the *subject* as a single-line *melody* by one *voice*, in the *tonic key*. A second *voice* then enters with the '*answer*' . . . while the first *voice* continues with a. . . *countersubject*. A third *voice* next enters with the *subject* in the *tonic*, followed by a fourth *voice* (if there is to be one) with the *answer* in the *dominant*.

Final Section - In *fugue*: The final section begins with the return of the *subject* in the *tonic key*. It may contain just the one entry, but more often there are two or three, the *answer* in the *dominant* also appearing.

Flat - 1. The sign ♭ which lowers a note a *chromatic semitone*. 2. Out of tune by being below the proper *pitch*.

Fugue - The most highly-developed form of *contrapuntal* writing, the name being derived from the Latin fuga - flight.

Harmonise - To add parts to a given *melody* (whether top, bass or inner part) so as to produce satisfactory *harmony*.

Harmony - The simultaneous combination of sounds, producing *chords*; their progression (i.e. movement from chord to chord) and relationship.

Interval - The distance, or difference in *pitch*, between two sounds.

Invention - Title of a series of fifteen two-part *clavier* pieces by Bach. (The fifteen 'Three-part Inventions' were entitled by Bach Sinfoniae).

Inversion - Literally turning upside-down. An *interval* is inverted by placing what was originally the lower note above what was originally the upper one.

Key - Key is a set of notes in relation to a principal note called the key-note or *tonic*. . . a piece which in the construction of its *melody*, *chords*, etc. employs the notes of the *scale* of C *major* is said to be in the *key* of C *major*.

Key Signature - *Sharps* or *flats* grouped at the beginning of each *stave*, immediately after the *clef*, which are essential to the *key* of the piece.

Major - Literally 'greater'. 1. Major interval. One between the *tonic* and the 2nd, 3rd, 6th or 7th degrees of a major *scale*. 2. Major *scale*. One in which the *semitones* lie between the 3rd and 4th, and the 7th and 8th degrees, the other degrees being separated by a whole tone.

Melody - In essence a series of sounds following each other, as opposed to *harmony*. . . involves rise and fall of *pitch* together with variety of sound-duration, i.e. rhythm.

Middle Entry - In *fugue*: . . . *subject* and *answer*, with or without *countersubject*, in *keys* other than the *tonic* and *dominant*. . .

Minor - Literally less, smaller. 1. A minor *interval* is a *chromatic semitone* smaller than a *major* one. 2. A minor *scale* is one in which the 3rd degree is only a minor 3rd above the *tonic*.

Modulation - Change of *key*; passing from one *key* to another.

Octave - The *interval* of an 8th, from any note to the next note of the same letter-name, either above or below.

Ornaments - The decoration or embellishment of the basic notes of a *melody*.

Part - 1. The music for a single *voice* or instrument. 2. A single *melodic* line in a composition of *contrapuntal* character.

Pitch - The height or depth of a musical sound. In sound: . . . pitch depends on the frequency (speed) of vibration; the higher the frequency the higher the pitch.

Redundant Entry - In a *fugue*, an extra entry of the *subject* or *answer* in *tonic* or *dominant key*, after the *exposition* is completed.

Root - The note on which a *chord* is founded or built up, by superimposing thirds above it.

Round - An infinite *canon* at the unison. Each singer returns from the end of the *melody* to its beginning and repeats it an indefinite number of times. The most elementary example is 'Three Blind Mice'.

Scale - A series of notes, ascending or descending, in alphabetical order. . . A *chromatic* scale consists entirely of *semitones*.

Semibreve - whole note: a musical note having the longest time value (equal to four beats in common time).

Semitone - A half-tone or *minor* 2nd; the smallest *interval* used in European music.

Sharp - 1. The sign ♯ which indicates the raising of a note by a *chromatic semitone*. 2. Out of tune by being above the proper *pitch*.

Stave - The five parallel lines, with the spaces between them, on which the notes are written.

Stretto - 1. In a *fugue*, the overlapping of entries of the *subject*, the second entry beginning before the first is finished. 2. In other compositions a concluding section at increased speed.

Subject - The material on which part or all of a composition is based.

Tempo - Speed; time.

Time Signature - A sign consisting of two figures, one above the other, placed at the beginning of a piece of music to indicate (upper figure) the number of beats or pulses in each bar and (lower figure) the value of each beat or pulse as a subdivision of a *semibreve*.

Tonic - The first degree of the *scale* of a *key*, to which it gives its name.

Transposition - The writing or performance of a composition at a different *pitch* from its original.

Triad - A *chord* of three notes, consisting of *root* with 3rd and 5th above it.

Voice - A single *melodic* line in a composition of *contrapuntal* character.

User Guide

USER GUIDE

CONTENTS

TOOL DESCRIPTION

The tool has been created to aid in analysis of contrapuntal music – most specifically fugue. It has been designed with ease of use in mind, offering a number of different functions to aid users of any computing ability to analyse music. The main functions it comprises are voice separation and pattern extraction – extracting the most musically significant patterns in the piece being analysed.

The following guide has been designed to help in the understanding and successful use of this tool and assumes an intermediate, or higher, knowledge of music theory.

GETTING STARTED

System requirements

The tool has the following minimum hardware and software requirements:

HARDWARE	SOFTWARE
Intel Pentium 4 Processor	
2.5GHz Processor Speed	Java JRE 5.0
800MB Free Hard Drive Space	Eclipse SDK 3.1
512MB RAM	Lilypond 2.4
Monitor, Keyboard & Mouse	

Installation

As this is the first release of this tool, it can currently only be run through a Java development environment. Therefore, the installation process is quite involved. Future releases should have all the required external functions in-built so such installation will be unnecessary. The tool is inherently platform-independent, depending only on the platform dependence of the musical typesetting program Lilypond, so in further releases, platform dependent features such as those stated below will no longer be relevant.

In order to install and run the tool, you must first install the software listed above. These processes will not be described here – please see the locations specified below for download and installation procedures for the required pieces of software:

- Java JRE - http://java.sun.com/j2se/1.5.0/download.jsp
- Eclipse SDK - http://www.eclipse.org/downloads/index.php
- Lilypond - http://www.lilypond.org/web/install/ (this includes installation instructions for Cygwin – a Linux environment for Windows users)

Once you have installed all of this software, you can unzip the analyser.zip file into your desired home directory. Due to Lilypond functions, Windows users must unzip the analyser folder into the home/"your username" folder

of the cygwin folder created during Lilypond installation (its default location is C:/cygwin/). Linux users may unzip the folder to any desired location, but note that root access is required on Linux systems in order to download Lilypond.

The zip file consists of a number of different directories – the main ones being the lb2148/sound folder which holds the package containing all the code for the tool, and the Outputs folder into which all outputs from the program will be stored. Other folders, such as the bach folder, hold example MIDI files which can be used for analysis.

The next step is to create a new project in Eclipse (please follow the instructions given in Eclipse, if you require help in doing this). All the relevant information is stored in the Analyser folder, so simply provide Eclipse with the location of the required information.

Running the tool

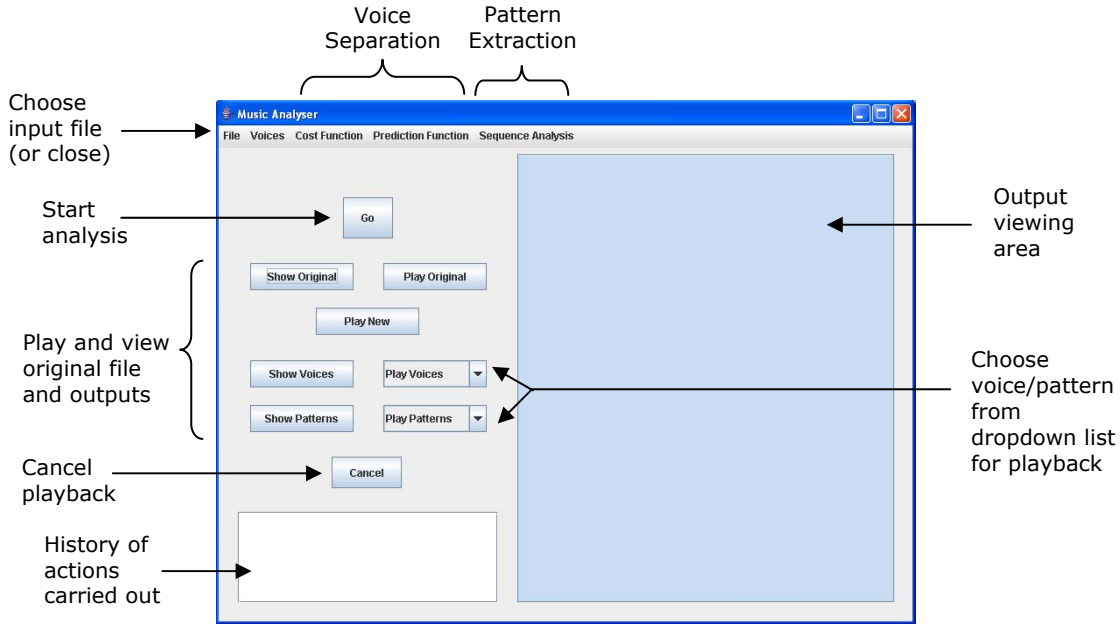
From within the newly created Eclipse project, choose the Run Configuration "GUI" and the tool will load.

TOOL FUNCTIONS

General Usage

The Interface

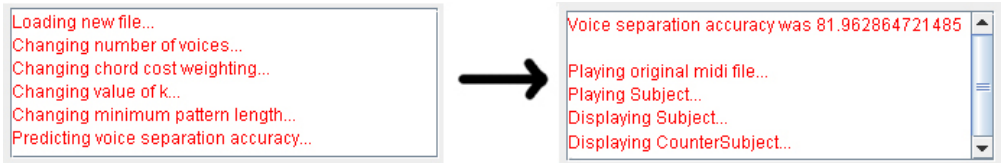
All the functions of the tool can be easily run from the main graphical interface. This provides a very intuitive menu system to change the program's parameters and buttons to hear and view outputs. The screenshot below shows the tool's interface:



The buttons shown in the previous screenshot are all enabled, meaning they can be clicked. When any buttons are disabled, they will go grey and will not be able to be selected:

Buttons are enabled and disabled according to what actions you can carry out at any one time.

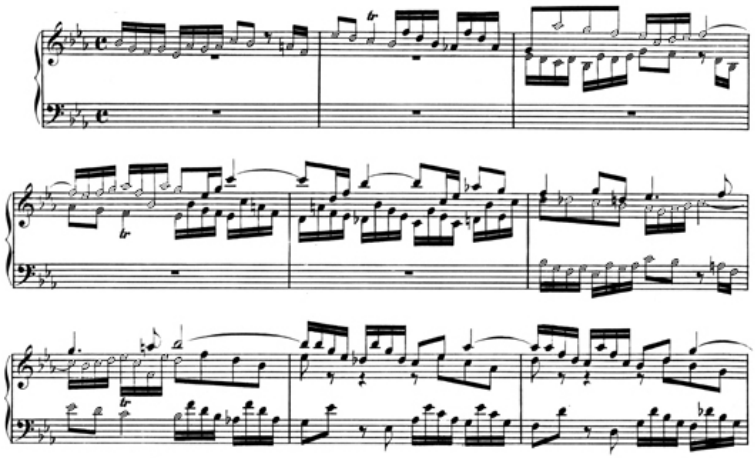
The history box will be constantly updated to provide you with a history of actions which you have carried out since the last analysis:



Analysis Steps

The following series of simple steps must be taken in order to analyse a piece of music:

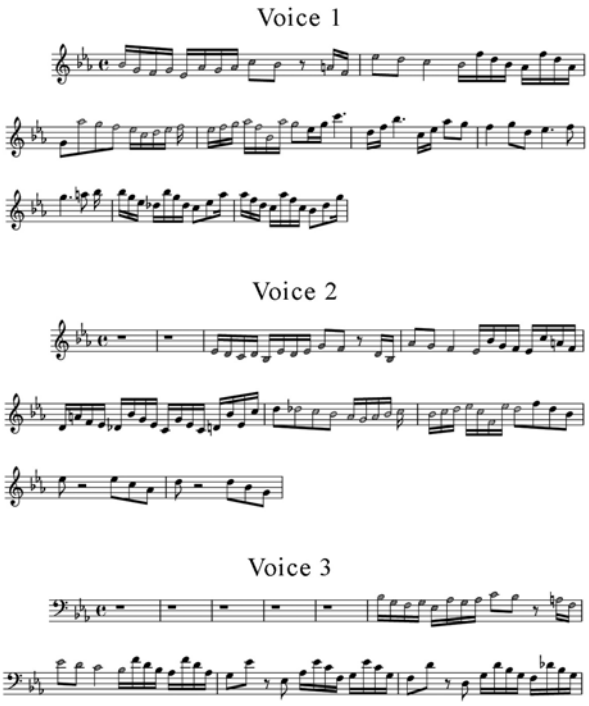
- Go to File, Load Input to load an input file to analyse. The file must be of the type MIDI (.mid). An extract from an example MIDI file is shown below. This is Bach's fugue number 7 from Book 1 of the Well Tempered Clavier:



- Go to Voices, Change Number to enter the number of voices present in the input piece (this will typically range from 2 to 5).
- Change voice separation parameters as required (from the Cost Function and Prediction Function menus).
- Change pattern extraction parameters as required (from the Sequence Analysis menu).
- Click 'Go'.
- Once analysis has been successfully carried out, the output buttons will be enabled, allowing you to select to hear and/or view outputs as required.

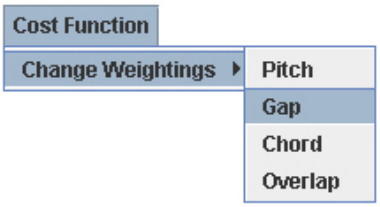
Voice Separation

The first function carried out by the tool is voice separation – this splits the inputted piece of music into its constituent parts/voices. The voices shown below were extracted from the example piece shown above:



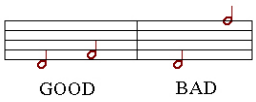
Parameters

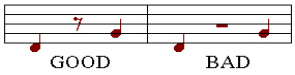
There are 4 parameters which you can change to alter the way the voice separation process is carried out. These can all be found under the Cost Function menu:

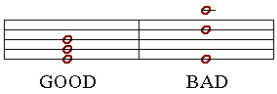


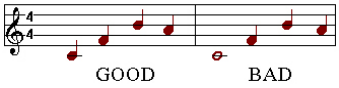
When you select one of these options, an input box will appear in which you can enter a new value for the given weighting (explained below). The box will also display the current value of the weighting:

The 4 options control the 'cost function' which is used to create the separate voices, and the weightings are used to show how important each of the 4 different unwanted features are to avoid in order to create the correct output. These features are as follows:

1. Pitch – large ranges of pitch within a voice, i.e. 

2. Gap – large rests within a voice, i.e. 

3. Chord – large ranges of pitch within chords in a voice, i.e. 

4. Overlap – overlapping notes within a voice, i.e. 

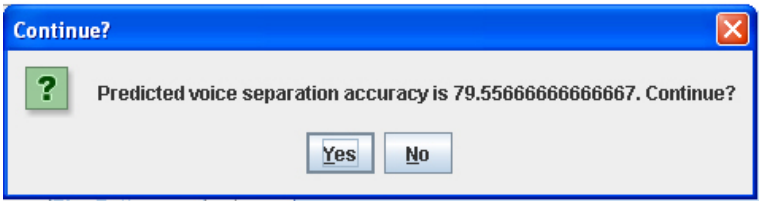
So, say you have an input piece for which a voice separation with the given parameters has created outputs in which there are no rests within the voices (where rests should be) and the notes within the voices jump large ranges between consecutive notes. The best alteration of parameters in this case would be to decrease the gap weighting and increase the pitch weighting.

Similarly, if a voice separation is produced where voices contain chords with very large ranges and many notes are often played in the same voice at the same time, the best alteration of parameters would be to increase both the chord and overlap weightings.

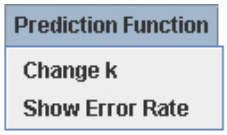
All the weightings default to 0.5 when you first load the tool.

Predictions

There is a prediction function written into the tool which tells you how accurate the voice separation is likely to be, as accuracies differ from piece to piece, within a given error margin, and allows you to choose whether or not to continue with analysis, given the expected accuracy:



The prediction function can be altered and accessed through the Prediction Function menu:



The error margin, previously mentioned, can be viewed by clicking on Show Error Rate. This will display a box showing the error rate which will apply to any predictions made:

This figure shows by what percentage the prediction might vary from the true accuracy rate. For example, if a prediction was given as 70% with the average error rate shown, 0.0827, the true value could range between 64.21% and 75.79%.

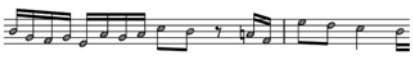
The accuracy rate, and therefore the overall average accuracy of the predictions, can be altered using the Change k menu option. This will display a box into which you can enter a new value of the prediction parameter k:

This value has a differing effect on the accuracy depending on the 'training set' used for prediction. The training set is fixed in this particular application so it is possible to change the value of k and see what effect it has on the accuracy rate, but the optimal value will remain as 3.

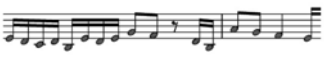
Pattern Extraction

The second main function of the tool is used to extract the most musically significant patterns out of the input piece. These patterns are the subject, answer and countersubject of a fugue (or similar form of contrapuntal music). The patterns extracted from the example above are shown below:

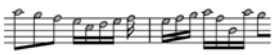
Subject



Answer

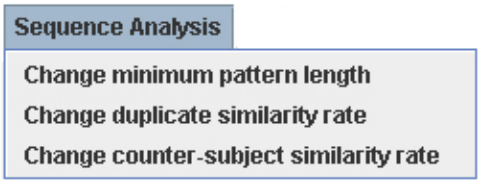


Countersubject

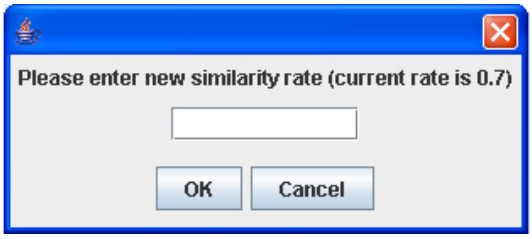


Parameters

There are 3 parameters for the pattern extraction function, which alter how the extraction process is carried out. These parameters can be found under the Sequence Analysis menu:



When any of these options are selected, a similar box to that shown above will appear in which you can enter the new value of the parameter:



When altered, these options will determine what patterns are outputted by the extraction function. Their influences are described below:

1. Minimum Pattern Length – determines the number of notes which must be present in a pattern before it can be considered significant.

2. Duplicate Similarity Rate – the threshold of similarity (as a percentage, i.e. 0.7 = 70%) *over* which a pattern must be in order to be considered the same as another pattern. This is used mainly for finding the answer pattern.

3. Countersubject Similarity Rate – the threshold of similarity *under* which a pattern must be in order to be considered different enough from the subject and answer patterns to be the countersubject.

So, say that pattern extraction for an input piece, given the current parameters, has not returned an answer – try reducing the duplicate similarity rate (it may have been that the answer is not as similar to the subject in that particular piece as is normally expected). This parameter is not used solely for the extraction of an answer pattern, so changing the value of this could alter the other patterns which are returned.

Similarly, if a countersubject has been returned which is incorrect and too similar to the other patterns returned, try increasing the countersubject similarity rate. This will cause re-analysis to look for patterns which less closely match the subject and answer in order to try and find the countersubject. This parameter can also be reduced to stop the function returning an incorrect countersubject when there is none in the input piece.

Finally, any incorrect patterns could be attributed to the minimum pattern length set. If the input piece has a much shorter/longer subject than

normal, the minimum pattern length should be decreased/increased in order to allow short patterns to be returned/force the function to search for longer patterns.

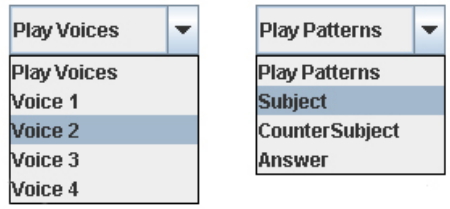
When the tool is first loaded, the default settings for these parameters are thus: minimum pattern length = 6, duplicate similarity rate = 0.7, countersubject similarity rate = 0.5.

Outputs

All the outputs created are available, in MIDI and PDF format, in the Outputs folder of the main Analyser project folder. This folder is cleared each time a new piece is sent for analysis, therefore if you want to keep any outputs from previous analyses, make sure you copy them out of the Outputs folder before running analysis again.

Audio

Once analysis has completed, it is possible to listen to the original file, the voices and patterns extracted and a version of the original file in which the extracted patterns have been highlighted (that is, accented in the music). All these files can be played by clicking the relevant button on the main interface screen. The voices and patterns must be selected from the relevant drop down menu:

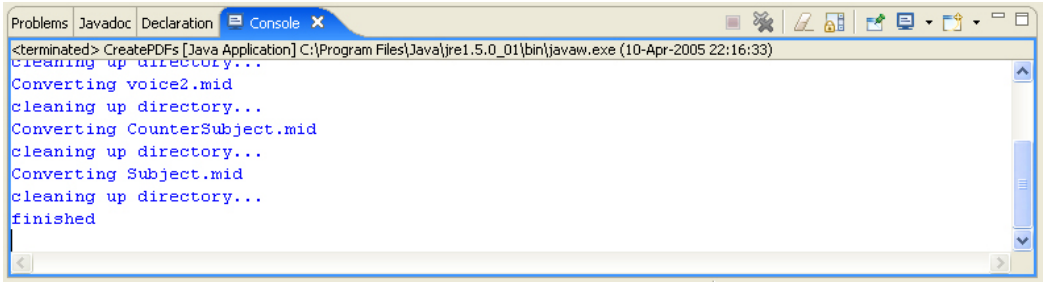


It is possible to play multiple outputs at once, and any currently playing outputs can be stopped by clicking the 'Cancel' button.

Visual

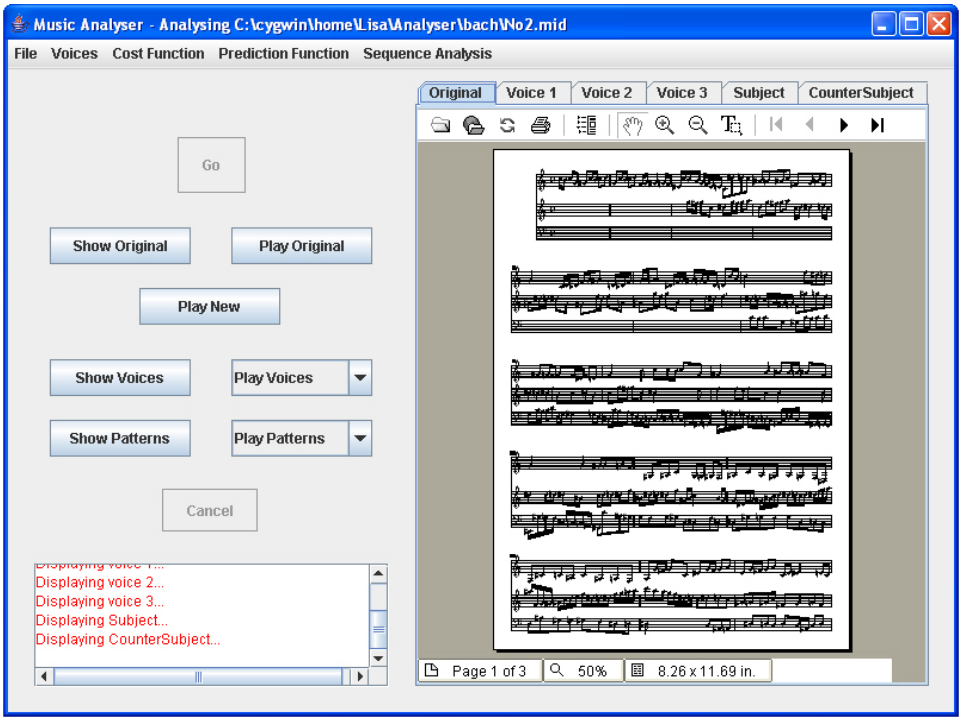
The visual outputs are slightly harder and more time consuming to create. The creation cannot be done directly from the main interface, therefore you must run a supplementary program to create the outputs which can then be viewed through the main interface. While the visual outputs are being created, you must not close the main interface, otherwise the outputs will not be able to be viewed (unless you access them directly from the Outputs folder).

You can run the program to create visual outputs by selecting the "CreatePDFs" Run Configuration in Eclipse. This program will keep you updated, through Eclipse, as to how output creation is progressing, displaying "finished" when the process has completed:



PDF versions of the original file, the separated voices and the extracted patterns can be viewed in the output viewing area by clicking on the relevant buttons on the main interface.

The outputs will appear under different tabs through which you can view at your leisure:



Questionnaires

Tool for Analysis of Contrapuntal Music

3rd Party User Questionnaire

Name: Simon Lewis

Your Musical Knowledge Level: Beginner / Intermediate / Expert

Your Computing Knowledge Level: Beginner / Intermediate / Expert

---

How helpful did you find the User Guide?

Excellent / Good / Average / Poor

Comments: Didn't understand musical terms

How would you assess the visual appearance of the tool?

Excellent / Good / Average / Poor

Comments:

How easy was the tool to use?

Excellent / Good / Average / Poor

Comments: Interface was fine but musical content not understood

How would you rate the functions available in the tool?

Excellent / Good / Average / Poor

Comments:

Were the available parameters useful?

Excellent / Good / Average / Poor

Comments: Showed improvement, bit trial and error process.

How good was the speed of analysis?

Excellent / Good / Average / Poor

Comments:

How useful were the tool's outputs?

Excellent / Good / Average / Poor / Don't know

Comments:

How accurate were the tool's Outputs?

Excellent / Good / Average / Poor / Don't know

Comments:

Tool for Analysis of Contrapuntal Music

3rd Party User Questionnaire

Name: Ashley Bowles

Your Musical Knowledge Level: Beginner / (Intermediate) / Expert

Your Computing Knowledge Level: Beginner / Intermediate / (Expert)

How helpful did you find the User Guide?

Excellent / (Good) / Average / Poor

Comments: Would probably be confusing to less musical people.

How would you assess the visual appearance of the tool?

(Excellent) / Good / Average / Poor

Comments:

How easy was the tool to use?

(Excellent) / Good / Average / Poor

Comments:

How would you rate the functions available in the tool?

Excellent / (Good) / Average / Poor

Comments: Highlighting relevant sections of fugue would be good too.

Were the available parameters useful?

(Excellent) / Good / Average / Poor

Comments:

How good was the speed of analysis?

(Excellent) / Good / Average / Poor

Comments:

How useful were the tool's outputs?

Excellent / (Good) / Average / Poor / Don't know

Comments:

How accurate were the tool's Outputs?

Excellent / Good / (Average) / Poor / Don't know

Comments: The file I chose came out with a low accuracy.

Tool for Analysis of Contrapuntal Music

3rd Party User Questionnaire

Name: Olivia Donnelly

Your Musical Knowledge Level: Beginner / Intermediate / **Expert**

Your Computing Knowledge Level: **Beginner** / Intermediate / Expert

---

How helpful did you find the User Guide? **Excellent** / Good / Average / Poor

Comments:

How would you assess the visual appearance of the tool? **Excellent** / Good / Average / Poor

Comments:

How easy was the tool to use? Excellent / **Good** / Average / Poor

Comments:

How would you rate the functions available in the tool? Excellent / **Good** / Average / Poor

Comments:

Were the available parameters useful? **Excellent** / Good / Average / Poor

Comments:

How good was the speed of analysis? Excellent / **Good** / Average / Poor

Comments: A bit slow for some pieces

How useful were the tool's outputs? Excellent / **Good** / Average / Poor / Don't know

Comments: Good time saver

How accurate were the tool's Outputs? Excellent / **Good** / Average / Poor / Don't know

Comments: Varied between pieces